A System for Prototyping Optical Architectures

by

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

December, 1995

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

A System for Prototyping Optical Architectures

Volume I: Concepts & Design

BY

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

December 1995

UMI Number: 1377985

UMI Microform 1377985
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

A System for Prototyping Optical Architectures

by

MUHAMMED ATIF MEMON

A Thesis Presented to the

FACULTY OF COLLEGE OF GRADUATE STUDIES

Department of Information and Computer Science

King Fahd University of Petroleum and Minerals

Dhahran — 31261, **Saudi Arabia**

December 20, 1995

Copyright © 1996 Atif Muhammed Memon

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **Muhammed Atif Memon** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee

Dr. Subbarao Ghanta (*Thesis Advisor*)

Dr. Mohsen Guizani (*Member*)

Dr. Muslim Bozyigit (*Member*)

Dr. Muhammed Al-Mulhem (*Member*)

Dr. Muhammed Al-Mulhem
(*Department Chairman*)

Dr. Ala H. Al-Rabeh
(*Dean, College of Graduate Studies*)

20/12/95
Date

THIS THESIS IS

DEDICATED TO

MY PARENTS

FROM WHOM I LEARNT

THE TRUE MEANING OF

HARD WORK.

Acknowledgement

I would like to thank my parents whose constant efforts, encouragement and hard work made it possible for me to reach this stage in life.

Thank is due to all my teachers in schools, colleges, and universities whose dedication made this work possible.

I thank my thesis advisor, Dr. Subbarao Ghanta, for his advice, support and encouragement throughout this research work. I thank Dr. Mohsen Guizani for his constant help before, during and after the thesis work.

Special thanks to Asjad M. T. Khan for always being there to solve all my system related problems.

I would also like to thank the members of my committee Dr. Mohammed Al-Mulhem and Dr. Muslim Bozyigit for their help and advice.

I thank Dr. Yilmaz Akyildiz and Dr. Bassel Arafeh who helped me develop interest in optical computing.

I would like to thank my friends Purushothaman, Abdullah, Khursheed and Al-Mutlaq for their constant help.

Several parts of the thesis were influenced by contributions from various sources. Credit is due to Yilmaz Akildiz for the excellent review on *Mathematica*, Adel Lafi for the example architectures, Nabeel Mosli for the review on object-oriented programming paradigm, Roman Maeder for the *Mathematica* 3D-graphics shapes, Wolfram Inc. for a copy of WRILatex, KACST for supporting part of the work in the form of a research project, Bahaa E. A. Saleh and Malvin Carl Teich for the review on the theories of light, and David F. Rogers and J. Alan Adams for basics on graphics.

My stay at Saudi Arabia was made more fun by staying with Ashiq Hussain, Mansoor Ali Baig, Mohammad Ibrahim, Sabih Al-Sayed, Saleem Parvez, Asim Qayyum, Kamran Hussain and Amir Farooqi. I thank all of them for their support.

In the end I would like to thank my brother, Kashif, my sister, Sadaf, and my brother-in-law, Imran, for taking care of things back home during my stay here.

I also thank Dr. Baluch and his family, Mr. Jamaluddin Unar and his family, Dr. Mohsen Guizani and his family and Mr. Safiullah Faizullah and his family for providing with an environment in which it was difficult to be homesick.

Contents

Abstract (English) **xlvii**

Abstract (Arabic) **xlviii**

Preface **1**

I Design of a CAD System for Optical Architectures 1

1 Introduction **2**

1.1 Introduction to Optical Computing 2
1.1.1 What is Optical Computing? 3
1.1.2 Requirements for Future Computers 3
1.1.3 Desired Trend for Computer Evolution 4
1.1.4 Advantages Offered by Optics 5
1.1.5 Applications of Optics 7
1.1.6 General Problems with Optical Computing 12
1.2 The Problem 14
1.2.1 Objectives of the Work 16
1.3 CAD Systems 17
1.4 Review of Existing Tools 18
1.5 Hardware Description Languages 22
1.6 Paradigms Supported by $\mathcal{OHDL}$ 23
1.7 Functional Programming 24
1.7.1 Motivation for Adopting Functional Programming 24
1.7.2 λ - Calculus 27
1.7.3 Functional Programming - The Paradigm 28

1.8 Fundamentals of Object-Oriented Programming 37
1.8.1 Objects 38
1.8.2 Classes 40
1.8.3 Inheritance 41
1.8.4 Advantages of the Object-Oriented Paradigm 41
1.9 Rule Based Systems 43
1.10 Constraint Handling in Visual and Symbolic Languages 44
1.10.1 Role of Constraints in Geometric Modelling 45
1.10.2 Constraint Specification Techniques 46
1.10.3 Constraint Satisfaction Techniques 47
1.11 Optical Architectures 47
1.12 Simulation 53
1.12.1 Discrete-Event Simulation 54
1.12.2 Continuous Simulation 54
1.13 Analyses of Optical Architectures 54
1.13.1 Different Approaches 56
1.14 Code Generation and Generators 57
1.15 Geometry and 3D Graphics 58
1.15.1 Solid Modeling 58
1.15.2 Basic Geometric Transformations 59
1.16 Outline of the Thesis 61
1.17 Summary 64

2 A Tour of OptiCAD **65**
2.1 Introduction 65
2.2 A Problem Definition 66
2.3 A Specification for Trip-Flop 66
2.4 Instantiation of Components 69
2.5 Layout of the Architecture 70
2.5.1 Placement 70
2.5.2 Orienting Components 73
2.6 Viewing - Camera Position 76
2.7 Generation of a Simulator, Delay Analyzer and Power Analyzer for the Architecture 79
2.8 Simulator Execution - Results of Ray Tracing 79
2.9 Power and Delay Analysis 79
2.10 Documentation and Document Generation 85

2.11 A Methodology for the Design of Optical Architectures 85
2.11.1 The User's Template for Architecture Specification 86
2.12 Summary 87

3 Top Level View of OptiCAD 98
3.1 Introduction 98
3.2 Roles 99
3.2.1 User's/Architect's View 99
3.2.2 Overall System 104
3.2.3 Role of a Domain Expert 104
3.2.4 Custom Simulators/Analyzers 106
3.2.5 Role of a Language Expert 107
3.2.6 Overall System Architect 109
3.3 Overall CAD System 111
3.4 Interfaces 112
3.4.1 with User 112
3.4.2 with Domain Expert 113
3.4.3 with Language Expert 114
3.4.4 with Outside World Tools 114
3.4.5 with the Implementation Platform 114
3.4.6 with Databases 115
3.5 Supporting Utilities/Tools 115
3.5.1 *Mathematica* Platform 115
3.5.2 Systems Programming 116
3.5.3 String Operations 116
3.5.4 Databases 116
3.5.5 Translators 116
3.5.6 Dependency Analyzers 117
3.6 Module Level Description of *OptiCAD* 117
3.6.1 Actual Simulation 123
3.6.2 Databases 124
3.6.3 Examples 125
3.6.4 Geometry 126
3.6.5 Graphics 126
3.6.6 History 127
3.6.7 Init 127
3.6.8 Misc 128

3.6.9 Optical Components . . . 128
3.6.10 Optimization . . . 129
3.6.11 Scratch . . . 130
3.6.12 Simulator . . . 130
3.6.13 Templates . . . 130
3.6.14 Translators . . . 131
3.6.15 Utilities . . . 131
3.7 Summary . . . 133

4 The Hardware Description Language 134
4.1 Introduction . . . 134
4.2 Requirements for HDLs . . . 135
4.3 Flow-Chart for the Description of an Architecture . . . 138
4.4 Operators . . . 141
4.5 Data Types . . . 141
4.6 Constants, Parameters, Variables and Units . . . 144
4.7 Expressions . . . 144
4.8 Assertions . . . 144
4.9 Constructs . . . 145
4.10 Components/Objects and Attributes . . . 146
4.11 Component Libraries and Instantiation . . . 148
4.12 Constraints . . . 153
4.13 Directives . . . 155
4.14 Support for Abstraction . . . 161
4.15 Reporting Errors/Warnings . . . 161
4.16 Summary . . . 162

5 Placement 163
5.1 Introduction . . . 163
5.2 Constraints . . . 164
5.2.1 Role of Constraints in Geometric Modelling . . . 165
5.2.2 Constraint Specification Techniques . . . 166
5.2.3 Constraint Satisfaction Techniques . . . 170
5.3 Bounding Box of a Component/Assembly . . . 172
5.4 Bounding Box Placement . . . 176
5.4.1 Coordinate Systems . . . 177
5.4.2 A Formulation of Equations from Position Specifications . . . 178

5.4.3 The Full Set of Equations for Trip-Flop Architecture 179
5.4.4 Solutions to the Position Constraints 181
5.4.5 The Completely Positioned Trip-Flop Architecture 181
5.5 Orientation of Bounding Box 185
5.5.1 Representation Schemes 185
5.5.2 Orientation Representation 187
5.5.3 Orientation Constraints and Equation Formulation 187
5.5.4 The Full Set of Constraints for Trip-Flop Architecture 202
5.5.5 Solutions to the Orientation Constraints 203
5.5.6 Placed and Oriented Architecture 203
5.6 Handling Over/Under-Constrained Specifications 209
5.7 Displaying the Placed and Oriented Components 214
5.7.1 Displaying Primitive Components - *draw[]* Function 214
5.7.2 2D Graphics 215
5.7.3 3D Graphics 222
5.7.4 Generic 3D Graphics Shapes 226
5.7.5 An Algorithm for Displaying Hierarchical Architectures 226
5.8 Summary 234

6 Modelling of Optical Components 235
6.1 Introduction 235
6.1.1 Concept of a Model 236
6.1.2 Motives for Model Building 236
6.1.3 What is Modelling? 237
6.1.4 A Modelling Approach for Optical Components 237
6.2 Theories of Light 239
6.2.1 Ray Optics 241
6.2.2 Wave Optics 256
6.2.3 Beam Optics 267
6.2.4 Fourier Optics 279
6.2.5 Electro-Magnetic Optics 284
6.3 Representation of Information 287
6.3.1 Motivation 287
6.3.2 Attributes 287
6.3.3 Discussion 288
6.4 Component Models 289
6.4.1 Structural Model of a Laser 290

6.4.2 Behavioral Model of a Laser 302
6.5 Vendor Supplied Information & Catalog Data 304
6.6 Summary 306

7 **Simulation** **307**
7.1 Introduction 307
7.1.1 Motivation for the Analysis of Optical Architectures and Systems 308
7.1.2 Preliminaries for Simulation 310
7.1.3 Simulation Models and Their Classification 314
7.1.4 Discrete Event Simulation 315
7.2 A Discrete Event Simulator for Digital Optical Architecture 317
7.2.1 Global (System Level) vs. Local (Component Level) Simulation 317
7.2.2 Events and Event Handling 319
7.2.3 Messages and Their Handling 320
7.2.4 Geometric Calculations for Next Component Determination . 321
7.2.5 The Algorithm for Determination of the Next Component During Simulation 322
7.3 Ray Tracing 324
7.4 Additional Issues 324
7.4.1 Geometric Calculations 325
7.4.2 Representation of Light 327
7.4.3 Cache - *Trading Space for Time* 327
7.4.4 Hybrid Simulation 328
7.4.5 Hierarchical Simulation - *Trading Time for Space and Compactness* 329
7.4.6 Ray Tracing as the Basis for the Computation of Performance Measures 330
7.4.7 Distributed/Parallel Simulation 331
7.5 A Template for a Simulator 331
7.6 An Example Simulation of Trip-Flop 332
7.7 Summary 332

8 **Power and Delay Analyses** **336**
8.1 Introduction 336
8.1.1 Need for Power and Delay Analyses 337
8.1.2 Power Analysis 339

8.1.3 Delay Analysis 341
8.2 Signal Flow Graph Construction 342
8.2.1 Simulation Produces Reference Points 342
8.2.2 Construction of Signal-Flow Graph 346
8.3 Path Enumeration 350
8.4 Power Analysis 351
8.5 A Template for a Power Analyzer 356
8.6 Delay Analysis 358
8.7 A Template for a Delay Analyzer 358
8.8 Limitations of the Current Analyses 362
8.9 Other Analyses 364
8.10 Possible Improvements to the Current Analysis 368
8.10.1 Hierarchical Analysis 369
8.11 Alternative Approaches 370
8.11.1 Paraxial Approximation and Matrix Analysis 370
8.11.2 Jones Calculus 371
8.11.3 Additional Analysis Techniques 371
8.12 Summary 372

9 Code Generation **373**
9.1 Introduction 373
9.1.1 Objectives 376
9.2 Previous Work 377
9.3 A Template Driven Code Generator 377
9.4 Splicing and its Restriction to Semantic Preserving Substitutions . . . 383
9.4.1 Examples/Applications 383
9.5 Summary 385

10 Conclusion **387**
10.1 Contribution of the Thesis 387
10.2 Synopsis of the Thesis 389
10.3 Limitations of the *OptiCAD* System in its Current Form 392
10.4 Possible Extensions and Future Work 392
10.5 Summary 397

II Example Optical Architectures 398

11 Michelson Setup for the Banyan Network 399

11.1 Introduction 399
11.2 Optical Implementation 400
11.2.1 Approach 400
11.2.2 Operation 401
11.3 OHDL Description of the Architecture 402
11.3.1 Initialization of the Simulator 402
11.3.2 Architecture Information 402
11.3.3 Components instantiation 403
11.3.4 Placement Constraints 403
11.3.5 Orientation Constraints 405
11.3.6 Layout 405
11.4 Conclusion 407

12 Optical Comparator Unit 408

12.1 Introduction 408
12.2 Optical Implementation 409
12.2.1 Approach 409
12.2.2 Operation 410
12.3 OHDL Description of the Architecture 410
12.3.1 Initialization of the Simulator 410
12.3.2 Architecture Information 412
12.3.3 Components instantiation 412
12.3.4 Placement Constraints 412
12.3.5 Orientation Constraints 413
12.3.6 Layout 414
12.4 Conclusion 414

13 Mechanically Adjustable Shifter Module 416

13.1 Introduction 416
13.2 Optical Implementation 417
13.2.1 Approach 417
13.2.2 Internal parameters and Constraints 417
13.2.3 Operation 418
13.3 OHDL Description of the Architecture 419
13.3.1 Initialization of the Simulator 419

13.3.2 Architecture Information 419
13.3.3 Components instantiation 419
13.3.4 Placement Constraints 419
13.3.5 Orientation Constraints 420
13.3.6 Layout 420
13.4 Conclusion 422

14 Optical Edge Detector **423**
14.1 Introduction 424
14.2 Optical Implementation 425
14.2.1 Approach 425
14.2.2 Internal parameters and Constraints 427
14.2.3 Operation 427
14.3 OHDL Description of the Architecture 433
14.3.1 Initialization of the Simulator 433
14.3.2 Architecture Information 433
14.3.3 Components instantiation 433
14.3.4 Placement Constraints 434
14.3.5 Orientation Constraints 435
14.3.6 Layout 436
14.4 Conclusion 436

15 Optical Serial Adder **438**
15.1 Introduction 438
15.2 Optical Implementation 439
15.2.1 Approach 439
15.2.2 Operation 440
15.3 OHDL Description of the Architecture 443
15.3.1 Initialization of the Simulator 443
15.3.2 Architecture Information 443
15.3.3 Components instantiation 444
15.3.4 Placement Constraints 445
15.3.5 Orientation Constraints 448
15.3.6 Layout 450
15.4 Conclusion 452

16 Perfect Shuffle Interconnection 453
16.1 Introduction 453
16.2 Optical Implementation 454
16.2.1 Approach 454
16.2.2 Operation 455
16.3 OHDL Description of the Architecture 456
16.3.1 Additional Definitions for the Component Library 456
16.3.2 Initialization of the Simulator 456
16.3.3 Architecture Information 457
16.3.4 Components instantiation 457
16.3.5 Placement Constraints 457
16.3.6 Orientation Constraints 458
16.3.7 Layout 458
16.4 Conclusion 458

17 All Optical Trip-Flop 460
17.1 Introduction 460
17.2 Optical Implementation 461
17.2.1 Approach 461
17.2.2 Operation 461
17.3 OHDL Description of the Architecture 462
17.3.1 Initialization of the Simulator 462
17.3.2 Architecture Information 462
17.3.3 Components instantiation 462
17.3.4 Placement Constraints 464
17.3.5 Orientation Constraints 466
17.3.6 Layout 466
17.4 Conclusion 468

18 The Correlator Associator 469
18.1 Introduction 469
18.2 Optical Implementation 470
18.2.1 Approach 470
18.2.2 Operation 470
18.3 OHDL Description of the Architecture 471
18.3.1 Initialization of the Simulator 471
18.3.2 Architecture Information 471

18.3.3 Components instantiation 472
18.3.4 Placement Constraints 472
18.3.5 Orientation Constraints 473
18.3.6 Layout 473
18.4 Conclusion 473

19 The Correlator Detector Module **475**
19.1 Introduction 475
19.2 Optical Implementation 476
19.2.1 Approach 476
19.2.2 Operation 476
19.3 OHDL Description of the Architecture 477
19.3.1 Initialization of the Simulator 477
19.3.2 Architecture Information 477
19.3.3 Components instantiation 478
19.3.4 Placement Constraints 478
19.3.5 Orientation Constraints 479
19.3.6 Layout 479
19.4 Conclusion 479

20 Optical Thresholder Unit **481**
20.1 Introduction 481
20.2 Optical Implementation 482
20.2.1 Approach 482
20.2.2 Operation 482
20.3 OHDL Description of the Architecture 484
20.3.1 Initialization of the Simulator 484
20.3.2 Architecture Information 484
20.3.3 Components instantiation 484
20.3.4 Placement Constraints 484
20.3.5 Orientation Constraints 485
20.3.6 Layout 486
20.4 Conclusion 486

21 Optical Memory Architecture **488**
21.1 Introduction 488
21.2 Optical Implementation 489
21.2.1 Approach 489

21.2.2 Operation 489
21.3 OHDL Description of the Architecture 490
21.3.1 Initialization of the Simulator 490
21.3.2 Architecture Information 490
21.3.3 Components instantiation 490
21.3.4 Placement Constraints 491
21.3.5 Orientation Constraints 491
21.3.6 Layout 492
21.4 Conclusion 492

22 Optical Vector Operation **494**
22.1 Introduction 494
22.2 Optical Implementation 495
22.2.1 Approach 495
22.2.2 Internal parameters and Constraints 497
22.2.3 Operation 497
22.3 OHDL Description of the Architecture 499
22.3.1 Initialization of the Simulator 499
22.3.2 Architecture Information 499
22.3.3 External Parameters 500
22.3.4 Components instantiation 500
22.3.5 Placement Constraints 501
22.3.6 Orientation Constraints 502
22.3.7 Layout 502
22.4 Conclusion 502

23 The Shifter Module **505**
23.1 Introduction 505
23.2 Optical Implementation 506
23.2.1 Approach 506
23.2.2 Operation 506
23.3 OHDL Description of the Architecture 508
23.3.1 Initialization of the Simulator 508
23.3.2 Architecture Information 508
23.3.3 Components instantiation 508
23.3.4 Placement Constraints 508
23.3.5 Orientation Constraints 509

23.3.6 Layout 510
23.4 Conclusion 510

24 $System_1$ of AT&T Switching Fabrics 512
24.1 Introduction 512
24.2 Optical Implementation 513
24.2.1 Approach 513
24.2.2 Operation 513
24.3 OHDL Description of the Architecture 514
24.3.1 Initialization of the Simulator 514
24.3.2 Architecture Information 514
24.3.3 Components instantiation 515
24.3.4 Placement Constraints 516
24.3.5 Orientation Constraints 516
24.3.6 Layout 517
24.4 Conclusion 517

25 Beam Combiner Unit 519
25.1 Introduction 519
25.2 Optical Implementation 520
25.2.1 Approach 520
25.2.2 Operation 520
25.3 OHDL Description of the Architecture 523
25.3.1 Initialization of the Simulator 523
25.3.2 Architecture Information 523
25.3.3 Components instantiation 524
25.3.4 Placement Constraints 524
25.3.5 Orientation Constraints 526
25.3.6 Layout 526
25.4 Conclusion 526

26 Crossover Interconnection Network 528
26.1 Introduction 528
26.2 Optical Implementation 529
26.2.1 Approach 529
26.2.2 Operation 530
26.3 OHDL Description of the Architecture 530
26.3.1 Initialization of the Simulator 530

26.3.2 Architecture Information 532
26.3.3 Components instantiation 532
26.3.4 Placement Constraints 532
26.3.5 Orientation Constraints 533
26.3.6 Layout 534
26.4 Conclusion 534

27 Passive Beam Combiner **536**
27.1 Introduction 536
27.2 Optical Implementation 537
27.2.1 Approach 537
27.2.2 Operation 537
27.3 OHDL Description of the Architecture 537
27.3.1 Initialization of the Simulator 537
27.3.2 Architecture Information 539
27.3.3 Components instantiation 539
27.3.4 Placement Constraints 539
27.3.5 Orientation Constraints 541
27.3.6 Layout 541
27.4 Conclusion 541

28 Spot Array Generator **543**
28.1 Introduction 543
28.2 Optical Implementation 544
28.2.1 Approach 544
28.2.2 Internal parameters and Constraints 545
28.2.3 Operation 545
28.3 OHDL Description of the Architecture 546
28.3.1 Initialization of the Simulator 546
28.3.2 Architecture Information 546
28.3.3 Components instantiation 546
28.3.4 Placement Constraints 546
28.3.5 Orientation Constraints 547
28.3.6 Layout 547
28.4 Conclusion 549

29 $System_2$ of AT&T Switching Fabrics **550**
29.1 Introduction 550
29.2 Optical Implementation 551
29.2.1 Approach 551
29.2.2 Operation 551
29.3 OHDL Description of the Architecture 553
29.3.1 Initialization of the Simulator 553
29.3.2 Architecture Information 553
29.3.3 Components instantiation 553
29.3.4 Placement Constraints 554
29.3.5 Orientation Constraints 555
29.3.6 Layout 555
29.4 Conclusion 555

30 Integrated Spot Array Generator **557**
30.1 Introduction 557
30.2 Optical Implementation 558
30.2.1 Approach 558
30.2.2 Internal parameters and Constraints 558
30.2.3 Operation 558
30.3 OHDL Description of the Architecture 560
30.3.1 Initialization of the Simulator 560
30.3.2 Architecture Information 560
30.3.3 Components instantiation 560
30.3.4 Placement Constraints 561
30.3.5 Orientation Constraints 562
30.3.6 Layout 562
30.4 Conclusion 562

31 $System_3$ of AT&T Switching Fabrics **564**
31.1 Introduction 564
31.2 Optical Implementation 565
31.2.1 Approach 565
31.2.2 Operation 565
31.3 OHDL Description of the Architecture 567
31.3.1 Initialization of the Simulator 567
31.3.2 Architecture Information 567

31.3.3 Components instantiation . . . 567
31.3.4 Placement Constraints . . . 568
31.3.5 Orientation Constraints . . . 569
31.3.6 Layout . . . 569
31.4 Conclusion . . . 569

32 1 × 3 BPG Interconnect for Banyan Connection 571
32.1 Introduction . . . 571
32.2 Optical Implementation . . . 572
32.2.1 Approach . . . 572
32.2.2 Operation . . . 572
32.3 OHDL Description of the Architecture . . . 574
32.3.1 Initialization of the Simulator . . . 574
32.3.2 Architecture Information . . . 574
32.3.3 Components instantiation . . . 574
32.3.4 Placement Constraints . . . 575
32.3.5 Orientation Constraints . . . 575
32.3.6 Layout . . . 576
32.4 Conclusion . . . 576

33 Lossy Beam Combiner 578
33.1 Introduction . . . 578
33.2 Optical Implementation . . . 579
33.2.1 Approach . . . 579
33.2.2 Operation . . . 579
33.3 OHDL Description of the Architecture . . . 581
33.3.1 Initialization of the Simulator . . . 581
33.3.2 Architecture Information . . . 581
33.3.3 Components instantiation . . . 582
33.3.4 Placement Constraints . . . 582
33.3.5 Orientation Constraints . . . 583
33.3.6 Layout . . . 583
33.4 Conclusion . . . 583

34 Spot Array Generation 585
34.1 Introduction . . . 585
34.2 Optical Implementation . . . 586
34.2.1 Approach . . . 586

34.2.2 Operation 586
34.3 OHDL Description of the Architecture 588
34.3.1 Initialization of the Simulator 588
34.3.2 Architecture Information 588
34.3.3 Components instantiation 588
34.3.4 Placement Constraints 589
34.3.5 Orientation Constraints 590
34.3.6 Layout 590
34.4 Conclusion 590

35 Single Stage of Banyan Network **592**
35.1 Introduction 592
35.2 Optical Implementation 593
35.2.1 Approach 593
35.2.2 Operation 593
35.3 OHDL Description of the Architecture 595
35.3.1 Initialization of the Simulator 595
35.3.2 Architecture Information 595
35.3.3 Components instantiation 595
35.3.4 Placement Constraints 595
35.3.5 Orientation Constraints 596
35.3.6 Layout 596
35.4 Conclusion 598

36 *System*$_4$ of AT&T Switching Fabrics **599**
36.1 Introduction 599
36.2 Optical Implementation 600
36.2.1 Approach 600
36.2.2 Operation 600
36.3 OHDL Description of the Architecture 601
36.3.1 Initialization of the Simulator 601
36.3.2 Architecture Information 601
36.3.3 Components instantiation 602
36.3.4 Placement Constraints 602
36.3.5 Orientation Constraints 603
36.3.6 Layout 603
36.4 Conclusion 603

37 Pupil Division Beam Combination **605**
37.1 Introduction 605
37.2 Optical Implementation 606
37.2.1 Approach 606
37.2.2 Operation 606
37.3 OHDL Description of the Architecture 608
37.3.1 Initialization of the Simulator 608
37.3.2 Architecture Information 608
37.3.3 Components instantiation 608
37.3.4 Placement Constraints 609
37.3.5 Orientation Constraints 609
37.3.6 Layout 610
37.4 Conclusion 610

38 *System$_5$* of AT&T Switching Fabrics **612**
38.1 Introduction 612
38.2 Optical Implementation 613
38.2.1 Approach 613
38.2.2 Operation 613
38.3 OHDL Description of the Architecture 615
38.3.1 Initialization of the Simulator 615
38.3.2 Architecture Information 615
38.3.3 Components instantiation 615
38.3.4 Placement Constraints 615
38.3.5 Orientation Constraints 616
38.3.6 Layout 616
38.4 Conclusion 618

III An Implementation of OptiCAD in *Mathematica* **619**

39 *Mathematica* **620**
39.1 What is *Mathematica*? 621
39.2 Structure of *Mathematica* 622
39.3 *Mathematica* as an Interactive Calculator 628
39.4 *Mathematica*'s Packages 628
39.5 *Mathematica* for Symbolic Computation 630
39.6 Programming in *Mathematica* 630

39.7 *Mathematica* as a Visualization System 632
39.8 *Mathematica* for Distributed Computation 632
39.9 Integration and Interaction with Other Software Systems 633
39.10 *Mathematica* as an Exploratory Tool for Scientists & Engineers . . . 634
39.11 *Mathematica* as a Multi-Media System 635
39.12 *Mathematica* as an Education Tool 636
39.13 *Mathematica* Interfaces & Notebooks 636
39.14 Advanced Issues of *Mathematica* 637
39.15 Limitations of *Mathematica* 637
39.16 Standard Package Categories & Their Functionalities 638

40 Global Variables & System Initialization **642**

41 Generic & Catalog Component Databases **644**

42 Icons for Optical Components **648**

43 draw[self] for different optical components **652**

44 Simulator **659**
Support functions for OHDL 660
Implementation Support Functions 660
Constraint Generation 662
Support Functions for OHDL 666
Global Variables 666

45 Results of Ray Tracing **671**

46 Power & Delay Analyses **677**
Initialization 678
Analysis – Implementation 679
Power Analysis - Implementation 680
Delay Analysis - Implementation 684
Analyses of Trip-Flop Architecture 686
Loading the Simulator Generated Data 686
Signal-Flow Graph 687
Power Analysis 690
Delay Analysis 690

47 FrameWork of Simulator **691**
RuleProcessing 692
Geometry 693
Init 697
Utilities - Debug 697
Cache 697
Initialization 713

48 Code Generation **724**

49 3D Graphics Operations on Mathematica 3D graphics Objects **727**

50 Architecture Translation to Chapter **730**

IV Supporting Implementation in *Mathematica* **740**

51 Units used in the system **741**

52 Dependency Generation **743**

53 Dependency Information Processing **750**

54 Predicates for Geometry **754**

55 Commonly used Geometry Functions **757**
Ray Intersecting with a Boundary between Two Media 760
Trace of the Function Using an Example 762

56 Miscellaneous Graphics Routines **767**

57 2D Graphics Operations **769**
2D Graphics Primitives 770
Transformation of Points 771
Transformation of Straight Lines 773
Rotation 774
Transformation Matrices for Different Primitive 2D-Operations 775

58 3D Graphics Operations **786**
Transformation Matrices for Different Primitive 3D-Operations 790
Local Scaling 791
Overall Scaling 793
Shearing 795
Rotation Around the Coordinate Axes 797
Reflection Through Planes 803
Translation 805
Rotation Around an Arbitrary Axis in Space 807
Reflection Through an Arbitrary Plane in Space 810
Orienting an Object in Space 813

59 Notebook Translation **816**

60 Option Processing **825**

61 Rewrite Rules **827**

62 Finite Precision Comparisions **829**

63 General String Manipulation Routines **831**

64 General Systems Programming Routines **838**

65 Generation of the Reference Manual **849**
List of All User Defined Functions in a Context. 850
List of All Usage::Function Messages 850
Table of All USES Context Dependencies 851
List of All Contexts 851
All Contexts TeX Generation 852
Table of All USED-IN Context Dependencies 853
Generating TeX File for Context Dependencies 853
Table of All USES (Function, Context) Dependencies 855
Table of All USED-IN (Function, Context) Dependencies 856
Generating the TeX Chapter for Usage Messages. 856

V Reference Manual for OptiCAD 862

66 Introduction to the Reference Manual **863**

66.1 Introduction 863
66.2 Purpose of the Reference Manual 864
66.3 List of All Contexts in *OptiCAD* 864
66.4 Dependencies Among Contexts 864
66.5 Reference for User Defined Functions 865

67 List of All Contexts in OptiCAD **866**

67.1 Databases'Dependencies' 866
67.2 Databases'FunctionsToContexts' 867
67.3 Geometry'Geometry' 867
67.4 Geometry'Predicates' 868
67.5 Graphics'Graphics3D' 868
67.6 Graphics'Misc' 869
67.7 OpticalComponents'draw' 869
67.8 OpticalComponents'Icons' 869
67.9 Simulator'Simulator' 870
67.10 Translators'NotebookToPackage' 872
67.11 Translators'TransformMma3D' 873
67.12 Utilities'OptionProcessing' 873
67.13 Utilities'RealArithmetic' 874
67.14 Utilities'StringOperations' 874
67.15 Utilities'SystemsProgramming' 875

68 Dependencies Among Contexts **878**

69 Reference for User Defined Functions **883**

69.1 Functions Starting with A 884
69.1.1 AddObject[] 884
69.1.2 AddOrientationConstraint[] 885
69.1.3 AddOrientationConstraints[] 886
69.1.4 AddPlacementConstraint[] 887
69.1.5 AddPlacementConstraints[] 888
69.1.6 attributeOf[] 889
69.1.7 axisQ[] 890
69.2 Functions Starting with B 891

69.2.1 backupAndDelete[] 891
69.2.2 baseName[] 892
69.2.3 boundingBoxEquations[] 893
69.3 Functions Starting with C 894
69.3.1 cleanTree[] 894
69.3.2 clearDirectoryInfo[] 895
69.3.3 clearTreeIndex[] 896
69.3.4 CloseAllStreams[] 897
69.3.5 colinearQ[] 898
69.3.6 ComputeOrientations[] 899
69.3.7 ComputeOutputPosition[] 900
69.3.8 ComputePositions[] 901
69.3.9 cone[] 902
69.3.10 ConvexLens[] 903
69.3.11 coordinateRules[] 904
69.3.12 coordinateRulesToQuadruples[] 905
69.3.13 createExamplesNotebook[] 906
69.3.14 createFunctionsFile[] 907
69.3.15 createPackage[] 908
69.3.16 cuboid[] 909
69.3.17 cuboidtolines[] 910
69.3.18 cylinder[] 911
69.4 Functions Starting with D 912
69.4.1 definitionCell[] 912
69.4.2 dependencyContexts[] 913
69.4.3 derivedFileNames[] 914
69.4.4 directoryQ[] 915
69.4.5 dirName[] 916
69.4.6 disk[] 917
69.4.7 displayObject[] 918
69.4.8 draw[] 919
69.4.9 drawDriver[] 920
69.5 Functions Starting with E 921
69.5.1 edgesToVertices[] 921
69.5.2 ellipsoid[] 922
69.5.3 equationOfPlane[] 923
69.5.4 existFileQ[] 924

69.5.5 extractFunctionLines[] 925
69.5.6 extractFunctionNames[] 926
69.6 Functions Starting with F 927
69.6.1 fileQ[] 927
69.6.2 FilterOptions[] 928
69.6.3 frontToken[] 929
69.6.4 functionNameToUsage[] 930
69.6.5 functionPairs[] 931
69.6.6 functionsFileNameQ[] 932
69.6.7 functionsFileNameToContextName[] 933
69.7 Functions Starting with G 934
69.7.1 GenerateArchitecture[] 934
69.7.2 generateContextDependencyDatabase[] 935
69.7.3 GenerateDerivedFiles[] 936
69.7.4 generateDirectoryInfo[] 937
69.7.5 generateTree[] 938
69.7.6 generateTreeIndex[] 939
69.7.7 generateUsageDatabase[] 940
69.7.8 geQ[] 941
69.7.9 getAllFunctionNames[] 942
69.7.10 getDirectoryInfo[] 943
69.7.11 GetObject[] 944
69.7.12 GetObjects[] 946
69.7.13 gtQ[] 947
69.8 Functions Starting with H 948
69.8.1 HalfWavePlate[] 948
69.8.2 handleDefaults[] 949
69.8.3 hasTokenQ[] 950
69.8.4 hologram[] 951
69.9 Functions Starting with I 952
69.9.1 initialize[] 952
69.9.2 inputCellToPairs[] 953
69.9.3 interferenceFilter[] 954
69.10 Functions Starting with L 955
69.10.1 laser[] 955
69.10.2 lensTwo[] 956
69.10.3 leQ[] 957

69.10.4 lineQ[] . . . 958
69.10.5 listToString[] . . . 959
69.10.6 ltQ[] . . . 960
69.11 Functions Starting with M . . . 961
69.11.1 mirror[] . . . 961
69.12 Functions Starting with N . . . 962
69.12.1 nBlanks[] . . . 962
69.12.2 nChars[] . . . 963
69.12.3 neQ[] . . . 964
69.12.4 nonAlphaNumericASCIISet[] . . . 965
69.12.5 nonDefinitionCells[] . . . 966
69.12.6 normalize[] . . . 967
69.12.7 notebookNameToContextName[] . . . 968
69.12.8 notebookNameToExamplesFileName[] . . . 969
69.12.9 notebookNameTofunctionsFileName[] . . . 970
69.12.10 notebookNameToPackageFileName[] . . . 971
69.12.11 notebookQ[] . . . 972
69.13 Functions Starting with O . . . 973
69.13.1 obtainDependencies[] . . . 973
69.13.2 obtainDependenciesForFile[] . . . 974
69.13.3 OHDLFileStatus[] . . . 975
69.13.4 OHDLFileType[] . . . 976
69.13.5 OHDL$initialize[] . . . 977
69.13.6 operateByIndex[] . . . 978
69.13.7 operationToMatrix[] . . . 979
69.13.8 orient[] . . . 980
69.13.9 orientationOf[] . . . 981
69.13.10 orientBoundingBox[] . . . 982
69.14 Functions Starting with P . . . 983
69.14.1 PickFirst[] . . . 983
69.14.2 PickLast[] . . . 984
69.14.3 PlanoConvexCylindricalLens[] . . . 985
69.14.4 pointInPlane[] . . . 986
69.14.5 pointOnThePlaneQ[] . . . 987
69.14.6 positionOf[] . . . 988
69.15 Functions Starting with Q . . . 989
69.15.1 quadruplesToCoordinateRules[] . . . 989

69.15.2 QuarterWavePlate[] 990
69.15.3 quote[] 991
69.16 Functions Starting with R 992
69.16.1 rayAtBoundary[] 992
69.16.2 reGenerateDerivedFiles[] 994
69.16.3 removeDerivedFiles[] 995
69.16.4 removeHead[] 996
69.16.5 removePattern[] 997
69.16.6 removePatterns[] 998
69.16.7 removeQuotedStrings[] 999
69.17 Functions Starting with S 1000
69.17.1 SEED[] 1000
69.17.2 seedgraphicsobject[] 1001
69.17.3 selfDependencyQ[] 1002
69.17.4 ShowArchitecture[] 1003
69.17.5 showState[] 1004
69.17.6 simulate[] 1005
69.17.7 slicedCylinder[] 1006
69.17.8 slicedTruncatedCone[] 1007
69.17.9 SLM[] 1008
69.17.10 split[] 1009
69.17.11 splitPathName[] 1010
69.17.12 splitRepeated[] 1011
69.17.13 splitter[] 1012
69.17.14 SSEED[] 1013
69.17.15 stand[] 1014
69.17.16 suffixQ[] 1015
69.18 Functions Starting with T 1016
69.18.1 tokenize[] 1016
69.18.2 transformMma3D[] 1017
69.18.3 transform3D[] 1018
69.18.4 translateNotebook[] 1019
69.18.5 truncatedCone[] 1021
69.19 Functions Starting with U 1022
69.19.1 unixLikeFind[] 1022
69.19.2 uses[] 1024
69.20 Functions Starting with V 1025

69.20.1 vectorAtAnAngle[] 1025
69.20.2 ViewArchitecture[] 1026
69.21 Functions Starting with W 1028
69.21.1 wavePlate[] 1028
69.21.2 writeTo[] 1029
69.22 Functions Starting with X 1030
69.22.1 xAxisQ[] 1030
69.23 Functions Starting with Y 1031
69.23.1 yAxisQ[] 1031
69.24 Functions Starting with Z 1032
69.24.1 zAxisQ[] 1032

Bibliography **I**

Vita **VIII**

List of Tables

4.1 Arithmetic Operators. 142
4.2 Relational Operators. 142
4.3 Logical Operators. 142
4.4 Miscellaneous Operators. 142
4.5 Prefix Operators/Functions. 143
4.6 Supported Data Types 143
4.7 The Grammar for Expressions in $\mathcal{OHDL}$. 145

5.1 The Solution to the Position Constraints. 181
5.2 The Solution to the Orientation Constraints. 209

6.1 The Finite State Machine for Laser. 303

8.1 Component Numbers and Their Types. 344
8.2 Component Numbers and Their Positions. 344
8.3 Some Important Attributes of Events/Messages from a Simulation Log File. 345
8.4 All Paths of 3 <= *length* <= 6 for the Signal Flow Graph of Trip-Flop Architecture. 352

68.1 Contexts and their Dependencies. 879
68.2 Contexts and their Dependencies (contd ...). 880
68.3 Contexts and their Dependents. 881
68.4 Contexts and their Dependents (contd ...). 882

69.1 The Functions and Their Contexts Making Use of **AddObject**. . . . 884
69.2 The Functions and Their Contexts Used by **AddOrientationConstraint**. 886

69.3 The Functions and Their Contexts Making Use of **AddOrientationConstraint**. 886
69.4 The Functions and Their Contexts Used by **AddOrientationConstraints**. 886
69.5 The Functions and Their Contexts Used by **AddPlacementConstraint**. 887
69.6 The Functions and Their Contexts Making Use of **AddPlacementConstraint**. 887
69.7 The Functions and Their Contexts Used by **AddPlacementConstraints**. 888
69.8 The Functions and Their Contexts Making Use of **attributeOf**. . . . 889
69.9 The Functions and Their Contexts Making Use of **axisQ**. 890
69.10 The Functions and Their Contexts Used by **backupAndDelete**. . . 891
69.11 The Functions and Their Contexts Making Use of **backupAndDelete**.891
69.12 The Functions and Their Contexts Used by **boundingBoxEquations**.893
69.13 The Functions and Their Contexts Making Use of **boundingBoxEquations**. 893
69.14 The Functions and Their Contexts Used by **cleanTree**. 894
69.15 The Functions and Their Contexts Used by **clearDirectoryInfo**. . . 895
69.16 The Functions and Their Contexts Making Use of **clearDirectoryInfo**. 895
69.17 The Functions and Their Contexts Used by **clearTreeIndex**. 896
69.18 The Functions and Their Contexts Used by **ComputeOutputPosition**. 900
69.19 The Functions and Their Contexts Making Use of **ComputeOutputPosition**. 900
69.20 The Functions and Their Contexts Used by **ComputePositions**. . . 901
69.21 The Functions and Their Contexts Used by **cone**. 902
69.22 The Functions and Their Contexts Used by **ConvexLens**. 903
69.23 The Functions and Their Contexts Making Use of **ConvexLens**. . . 903
69.24 The Functions and Their Contexts Making Use of **coordinateRulesToQuadruples**. 905
69.25 The Functions and Their Contexts Used by **createExamplesNotebook**. 906
69.26 The Functions and Their Contexts Making Use of **createExamplesNotebook**. 906
69.27 The Functions and Their Contexts Used by **createFunctionsFile**. . 907

69.28 The Functions and Their Contexts Making Use of **createFunctionsFile**. . . . 907
69.29 The Functions and Their Contexts Used by **createPackage**. 908
69.30 The Functions and Their Contexts Making Use of **createPackage**. . 908
69.31 The Functions and Their Contexts Used by **cuboid**. 909
69.32 The Functions and Their Contexts Making Use of **cuboid**. 909
69.33 The Functions and Their Contexts Used by **cylinder**. 911
69.34 The Functions and Their Contexts Making Use of **cylinder**. 911
69.35 The Functions and Their Contexts Used by **definitionCell**. 912
69.36 The Functions and Their Contexts Making Use of **definitionCell**. . 912
69.37 The Functions and Their Contexts Used by **dependencyContexts**. 913
69.38 The Functions and Their Contexts Used by **derivedFileNames**. . . 914
69.39 The Functions and Their Contexts Making Use of **derivedFileNames**.914
69.40 The Functions and Their Contexts Used by **directoryQ**. 915
69.41 The Functions and Their Contexts Making Use of **directoryQ**. . . . 915
69.42 The Functions and Their Contexts Making Use of **dirName**. 916
69.43 The Functions and Their Contexts Making Use of **disk**. 917
69.44 The Functions and Their Contexts Used by **displayObject**. 918
69.45 The Functions and Their Contexts Used by **draw**. 919
69.46 The Functions and Their Contexts Making Use of **draw**. 920
69.47 The Functions and Their Contexts Used by **drawDriver**. 920
69.48 The Functions and Their Contexts Making Use of **drawDriver**. . . . 920
69.49 The Functions and Their Contexts Used by **equationOfPlane**. . . . 923
69.50 The Functions and Their Contexts Making Use of **equationOfPlane**.923
69.51 The Functions and Their Contexts Making Use of **existFileQ**. . . . 924
69.52 The Functions and Their Contexts Used by **extractFunctionLines**. 925
69.53 The Functions and Their Contexts Making Use of **extractFunctionLines**. . . . 925
69.54 The Functions and Their Contexts Used by **extractFunctionNames**.926
69.55 The Functions and Their Contexts Making Use of **extractFunctionNames**. . . . 926
69.56 The Functions and Their Contexts Making Use of **fileQ**. 927
69.57 The Functions and Their Contexts Making Use of **FilterOptions**. . 928
69.58 The Functions and Their Contexts Used by **frontToken**. 929
69.59 The Functions and Their Contexts Making Use of **frontToken**. . . . 929
69.60 The Functions and Their Contexts Used by **functionNameToUsage**.930
69.61 The Functions and Their Contexts Used by **functionPairs**. 931

69.62 The Functions and Their Contexts Making Use of **functionsFileNameQ**. . . . 932
69.63 The Functions and Their Contexts Making Use of **functionsFileNameToContextName**. . . . 933
69.64 The Functions and Their Contexts Used by **GenerateArchitecture**. 934
69.65 The Functions and Their Contexts Making Use of **GenerateArchitecture**. . . . 934
69.66 The Functions and Their Contexts Used by **generateContextDependencyDatabase**. . . . 935
69.67 The Functions and Their Contexts Making Use of **generateContextDependencyDatabase**. . . . 936
69.68 The Functions and Their Contexts Used by **GenerateDerivedFiles**. 936
69.69 The Functions and Their Contexts Making Use of **GenerateDerivedFiles**. . . . 937
69.70 The Functions and Their Contexts Used by **generateDirectoryInfo**. 937
69.71 The Functions and Their Contexts Making Use of **generateDirectoryInfo**. . . . 938
69.72 The Functions and Their Contexts Used by **generateTree**. 939
69.73 The Functions and Their Contexts Used by **generateTreeIndex**. . 939
69.74 The Functions and Their Contexts Making Use of **generateTreeIndex**. . . . 940
69.75 The Functions and Their Contexts Used by **generateUsageDatabase**. 940
69.76 The Functions and Their Contexts Making Use of **generateUsageDatabase**. . . . 941
69.77 The Functions and Their Contexts Used by **geQ**. 941
69.78 The Functions and Their Contexts Used by **getAllFunctionNames**. 943
69.79 The Functions and Their Contexts Making Use of **getAllFunctionNames**. . . . 943
69.80 The Functions and Their Contexts Used by **getDirectoryInfo**. . . . 944
69.81 The Functions and Their Contexts Making Use of **getDirectoryInfo**. 944
69.82 The Functions and Their Contexts Used by **GetObject**. 944
69.83 The Functions and Their Contexts Making Use of **GetObject**. . . . 945
69.84 The Functions and Their Contexts Used by **GetObjects**. 946
69.85 The Functions and Their Contexts Making Use of **gtQ**. 947
69.86 The Functions and Their Contexts Used by **HalfWavePlate**. 948
69.87 The Functions and Their Contexts Making Use of **HalfWavePlate**. 948
69.88 The Functions and Their Contexts Used by **handleDefaults**. 949

69.89 The Functions and Their Contexts Making Use of **handleDefaults**. 949
69.90 The Functions and Their Contexts Making Use of **hasTokenQ**. . . . 950
69.91 The Functions and Their Contexts Making Use of **hologram**. 951
69.92 The Functions and Their Contexts Used by **inputCellToPairs**. . . . 954
69.93 The Functions and Their Contexts Used by **interferenceFilter**. . . 954
69.94 The Functions and Their Contexts Making Use of **interferenceFilter**.955
69.95 The Functions and Their Contexts Used by **laser**. 955
69.96 The Functions and Their Contexts Making Use of **laser**. 956
69.97 The Functions and Their Contexts Making Use of **lensTwo**. 956
69.98 The Functions and Their Contexts Making Use of **leQ**. 957
69.99 The Functions and Their Contexts Making Use of **lineQ**. 958
69.100 The Functions and Their Contexts Making Use of **listToString**. . . 960
69.101 The Functions and Their Contexts Making Use of **mirror**. 961
69.102 The Functions and Their Contexts Used by **nBlanks**. 962
69.103 The Functions and Their Contexts Making Use of **nBlanks**. 962
69.104 The Functions and Their Contexts Making Use of **nChars**. 963
69.105 The Functions and Their Contexts Making Use of **neQ**. 964
69.106 The Functions and Their Contexts Used by **nonDefinitionCells**. . 966
69.107 The Functions and Their Contexts Making Use of **nonDefinitionCells**. 966
69.108 The Functions and Their Contexts Used by **normalize**. 967
69.109 The Functions and Their Contexts Making Use of **notebookNameToContextName**. 968
69.110 The Functions and Their Contexts Making Use of **notebookNameToExamplesFileName**. 969
69.111 The Functions and Their Contexts Making Use of **notebookNameTofunctionsFileName**. 970
69.112 The Functions and Their Contexts Making Use of **notebookNameToPackageFileName**. 971
69.113 The Functions and Their Contexts Making Use of **notebookQ**. . . 972
69.114 The Functions and Their Contexts Used by **obtainDependencies**. 973
69.115 The Functions and Their Contexts Making Use of **obtainDependencies**. 974
69.116 The Functions and Their Contexts Used by **obtainDependenciesForFile**. 974
69.117 The Functions and Their Contexts Making Use of **obtainDependenciesForFile**. 975

69.118 The Functions and Their Contexts Used by **OHDLFileStatus**. . . 975
69.119 The Functions and Their Contexts Making Use of **OHDLFileStatus**. 976
69.120 The Functions and Their Contexts Making Use of **OHDLFileType**. 976
69.121 The Functions and Their Contexts Used by **operateByIndex**. . . . 978
69.122 The Functions and Their Contexts Making Use of **operateByIndex**. 978
69.123 The Functions and Their Contexts Used by **operationToMatrix**. . 979
69.124 The Functions and Their Contexts Making Use of **operationToMatrix**. 980
69.125 The Functions and Their Contexts Making Use of **orientationOf**. . 981
69.126 The Functions and Their Contexts Making Use of **PickFirst**. . . . 983
69.127 The Functions and Their Contexts Used by **PickLast**. 984
69.128 The Functions and Their Contexts Used by **PlanoConvexCylindricalLens**. 985
69.129 The Functions and Their Contexts Making Use of **PlanoConvexCylindricalLens**. 985
69.130 The Functions and Their Contexts Making Use of **pointInPlane**. . 986
69.131 The Functions and Their Contexts Used by **pointOnThePlaneQ**. 987
69.132 The Functions and Their Contexts Making Use of **pointOnThePlaneQ**. 988
69.133 The Functions and Their Contexts Making Use of **positionOf**. . . . 988
69.134 The Functions and Their Contexts Making Use of **quadruplesToCoordinateRules**. 989
69.135 The Functions and Their Contexts Used by **QuarterWavePlate**. . 990
69.136 The Functions and Their Contexts Used by **quote**. 991
69.137 The Functions and Their Contexts Making Use of **quote**. 992
69.138 The Functions and Their Contexts Used by **reGenerateDerivedFiles**. 994
69.139 The Functions and Their Contexts Used by **removeDerivedFiles**. 995
69.140 The Functions and Their Contexts Making Use of **removeDerivedFiles**. 995
69.141 The Functions and Their Contexts Making Use of **removeHead**. . 996
69.142 The Functions and Their Contexts Used by **removePattern**. . . . 997
69.143 The Functions and Their Contexts Making Use of **removePattern**. 998
69.144 The Functions and Their Contexts Used by **removePatterns**. . . . 998
69.145 The Functions and Their Contexts Making Use of **removeQuotedStrings**. 999
69.146 The Functions and Their Contexts Used by **SEED**. 1000

69.147 The Functions and Their Contexts Making Use of **SEED**. 1000
69.148 The Functions and Their Contexts Used by **seedgraphicsobject**. . 1001
69.149 The Functions and Their Contexts Making Use of **seedgraphicsobject**. 1001
69.150 The Functions and Their Contexts Used by **selfDependencyQ**. . . 1002
69.151 The Functions and Their Contexts Making Use of **selfDependencyQ**. 1003
69.152 The Functions and Their Contexts Used by **ShowArchitecture**. . 1003
69.153 The Functions and Their Contexts Used by **simulate**. 1005
69.154 The Functions and Their Contexts Used by **slicedCylinder**. 1006
69.155 The Functions and Their Contexts Making Use of **slicedCylinder**. 1006
69.156 The Functions and Their Contexts Making Use of **slicedTruncatedCone**. 1007
69.157 The Functions and Their Contexts Used by **SLM**. 1008
69.158 The Functions and Their Contexts Making Use of **SLM**. 1008
69.159 The Functions and Their Contexts Making Use of **split**. 1009
69.160 The Functions and Their Contexts Making Use of **splitRepeated**. . 1012
69.161 The Functions and Their Contexts Making Use of **splitter**. 1012
69.162 The Functions and Their Contexts Used by **SSEED**. 1013
69.163 The Functions and Their Contexts Making Use of **SSEED**. 1013
69.164 The Functions and Their Contexts Used by **stand**. 1014
69.165 The Functions and Their Contexts Used by **suffixQ**. 1015
69.166 The Functions and Their Contexts Making Use of **suffixQ**. 1015
69.167 The Functions and Their Contexts Used by **tokenize**. 1016
69.168 The Functions and Their Contexts Used by **transformMma3D**. . 1017
69.169 The Functions and Their Contexts Making Use of **transformMma3D**. 1018
69.170 The Functions and Their Contexts Used by **transform3D**. 1018
69.171 The Functions and Their Contexts Making Use of **transform3D**. . 1019
69.172 The Functions and Their Contexts Used by **translateNotebook**. . 1020
69.173 The Functions and Their Contexts Making Use of **translateNotebook**. 1021
69.174 The Functions and Their Contexts Used by **truncatedCone**. . . . 1021
69.175 The Functions and Their Contexts Making Use of **truncatedCone**. 1022
69.176 The Functions and Their Contexts Used by **unixLikeFind**. 1022
69.177 The Functions and Their Contexts Making Use of **unixLikeFind**. . 1023
69.178 The Functions and Their Contexts Used by **uses**. 1025
69.179 The Functions and Their Contexts Making Use of **uses**. 1025

69.180 The Functions and Their Contexts Used by **ViewArchitecture**. . . 1026
69.181 The Functions and Their Contexts Making Use of **ViewArchitecture**. 1027
69.182 The Functions and Their Contexts Used by **wavePlate**. 1028
69.183 The Functions and Their Contexts Making Use of **wavePlate**. . . . 1028
69.184 The Functions and Their Contexts Used by **writeTo**. 1029
69.185 The Functions and Their Contexts Making Use of **writeTo**. 1029
69.186 The Functions and Their Contexts Making Use of **xAxisQ**. 1030
69.187 The Functions and Their Contexts Making Use of **yAxisQ**. 1031
69.188 The Functions and Their Contexts Making Use of **zAxisQ**. 1032

List of Figures

1.1 Illustration of AND, OR and Fan-out Capability 10
1.2 Two Different Ways of Realizing Vector-Matrix Multiplication. 11
1.3 Point Object 39
1.4 Example of an Inheritance Hierarchy 42

2.1 The Trip-Flop Architecture - Logical View 68
2.2 Snapshot of the Trip-Flop Architecture Without Orientations 72
2.3 The Fully Oriented Trip-Flop Architecture 75
2.4 A View of the Fully Oriented Trip-Flop Architecture from Camera Position {1, 1, 1} 76
2.5 Another View of the Fully Oriented Trip-Flop Architecture from Camera Position {1, 1, -1} 77
2.6 Projections of the Architecture Without Orientations 78
2.7 Event 1: Laser's Fired Beam Hits the Beam Splitter 80
2.8 Event 5: Beam Splitter's Output Leaves the System 80
2.9 Event 10: One Component of the Beam Splitter's Output Strikes the Mirror 81
2.10 Event 15: Beam Splitter's Output Strikes Another Beam Splitter . . 81
2.11 Event 20: Beam Splitter's Output Strikes Another Beam Splitter . . 82
2.12 Event 25: Laser's Fired Beam Hits the Beam Splitter 82
2.13 Power Decay Along Path {2, 8, 13} 83
2.14 Power Decay Along Path {2, 4, 1, 13} 83
2.15 Power Decay Along Path {2, 3, 7, 8, 9, 13} 83
2.16 Accumulated Delay Along Path {2, 8, 13} 84
2.17 Accumulated Delay Along Path {7, 8, 9, 13} 84
2.18 User's View of the System 88
2.19 User Template 89

2.20 User Template (continued) 90
2.21 User Template (continued) 91
2.22 User Template (continued) 92
2.23 User Template (continued) 93
2.24 User Template (continued) 94
2.25 User Template (continued) 95
2.26 User Template (continued) 96
2.27 User Template (continued) 97

3.1 The Overall System View – at Requirement Analysis Time 100
3.2 The *OHDL* View After Separating the Role of the User. 103
3.3 The Overall System View after Carving Out the Role of the Domain Expert. 105
3.4 The Current Shape of the *OptiCAD* System after the Birth of the Language Expert. 108
3.5 The *OptiCAD* System with Additional Emerging Roles. 110
3.6 The Major Modules in the *OptiCAD* System. 118
3.7 The Files that Make Up the *OptiCAD* System. 119

4.1 Flow-Chart for the Description of an Architecture 139
4.2 The Structure of the *OptiCAD* Library 150
4.3 The Different Levels of Component Instantiation 151

5.1 A Flow-Graph for Converting Temperature from One Scale to Another 167
5.2 A Component with its Reference Point, Reference Vector and Bounding Box. 174
5.3 Different Views of Two Components – Laser and Convex Lens with Their Bounding Boxes, Reference Points and Reference Vectors. . . . 175
5.4 The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, -1, 1} 182
5.5 The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, 1, 1} 183
5.6 The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, 1, -1} 184
5.7 A Point is Uniquely Determined by its Three Coordinates in 3-D Space 186
5.8 The Degrees of Freedom of an Object in 3-D Space 188
5.9 Rotating an Object Around its Principal Axis. 189
5.10 Rotation Around the z-axis 190

5.11 Rotation Around the z-axis 191
5.12 Rotation Around the z-axis 192
5.13 Rotation Around the y-axis 193
5.14 Rotation Around the y-axis 194
5.15 Rotation Around the y-axis 195
5.16 Rotation Around the Principal Axis 196
5.17 Rotation Around the Principal Axis 197
5.18 Rotation Around the Principal Axis 198
5.19 Random Orientations of the Laser Object 199
5.20 Random Orientations of the Laser Object 200
5.21 View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, -1, 1}. 204
5.22 View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, 1, 1}. 205
5.23 View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, 1, -1}. 206
5.24 A View of the Placed and Oriented Trip-Flop Architecture without Bounding Box from Camera Position {1, -1, 1} 207
5.25 Another View of the Placed and Oriented Trip-Flop Architecture without Bounding Box 208
5.26 Example of an Under-Constrained System 211
5.27 Example of an Over-Constrained System 212
5.28 Generic Shape – Cone 227
5.29 Generic Shape – Cuboid 227
5.30 Generic Shape – Cylinder 228
5.31 Generic Shape – Disk 228
5.32 Generic Shape – Double Helix 229
5.33 Generic Shape – Flat Cone 229
5.34 Generic Shape – Helix 230
5.35 Generic Shape – Moebius Strip 230
5.36 Generic Shape – Sliced Cylinder 231
5.37 Generic Shape – Sliced Truncated Cone 231
5.38 Generic Shape – Sphere 232
5.39 Generic Shape – Torus 232
5.40 Generic Shape – Truncated Cone 233
5.41 Generic Shape – Wire Cone 233

6.1 Different Types of Models and their Refinement 238
6.2 Ray Travelling from One Medium to Another 243
6.3 Reflection from a Planar Mirror 243
6.4 Focusing of Light by a Paraboloidal Mirror 244
6.5 Reflection from an Elliptical Mirror 244
6.6 Reflection of Parallel rays from a Concave Spherical Mirror 245
6.7 Ray Deflection by a Prism 246
6.8 Beam-Splitter Splits a Beam into Two Components 247
6.9 Beam-Merger Combines Two Beams into One 247
6.10 A Cube Beam-Splitter and its Operation 248
6.11 A Biconvex Spherical Lens 248
6.12 Thin Lens: (a) Ray Bending, (b) Image Formation. 249
6.13 Free-Space Propagation 251
6.14 Refraction at a Planar Boundary 252
6.15 Refraction at a Spherical Boundary 253
6.16 Transmission Through a Thin Lens 253
6.17 Reflection from a Plane Mirror 254
6.18 Reflection from a Spherical Mirror 254
6.19 Reflection of a Plane Wave from a Planar Mirror 260
6.20 Reflection and Refraction of a Plane Wave at a Dielectric Planar Boundary 261
6.21 Transmission of a Plane Wave Through a Transparent Plate 262
6.22 Transmission of a Plane Wave at an Angle θ Through a Transparent Plate 263
6.23 Transmission of a Plane Wave Through a Prism 263
6.24 Transmission of a Plane Wave Through a Thin Lens 264
6.25 A Double Convex Lens 265
6.26 A Thin, Transparent Plate with Periodically Varying Thickness . . . 266
6.27 Interferometers: (a) Mach-Zehnder Interferometer (b) Michelson's Interferometer, and (c) Sagnac Interferometer 268
6.28 Setup of Michelson's Interferometer using Catalog Components 269
6.29 Snapshot of the Setup of Michelson's Interferometer 270
6.30 Transmission of a Gaussian Beam through a Thin Lens 277
6.31 The $4-f$ Imaging System. If an Inverted Coordinate System is used in the Image Plane, the Magnification is Unity 282
6.32 The $4-f$ System Performs a Fourier Transform Followed by an Inverse Fourier Transform. 283

6.33 The $4 - f$ System Performs a Fourier Transform Followed by an Inverse Fourier Transform. 283
6.34 Examples of Object, Mask, and Filtered Image for Three Spatial Filters: (a) Low-pass Filter; (b) High-pass Filter; (c) Vertical-pass Filter. Black Shows No Transmittance While White Shows Full Transmittance285
6.35 The Base of the Laser 292
6.36 The Vertical Bar of the Laser 293
6.37 Combining the Vertical Bar and the Base 293
6.38 The Sliding Sleeve for the Vertical Bar 294
6.39 Positioning the Sleeve Over the Base 295
6.40 "Sliding" the Sleeve on the Vertical Bar 295
6.41 The Complete Stand for the Laser 296
6.42 The Laser Itself – Horizontal Cylinderical Shape 297
6.43 "Welding" the Sleeve to the Laser Bar 297
6.44 Positioning the Laser Bar over the Base 298
6.45 View of the Sleeve and Laser Combination with the Base 298
6.46 Snapshot of the Vertical Bar Perpendicular to the Laser 299
6.47 "Mounting" the Sleeve and Laser Combination over the Vertical Bar . 299
6.48 The Complete Laser Assembly 300
6.49 The Beam to be Emitted 301
6.50 The Laser "Emitting" the Beam 301
6.51 The Laser Assembly Complete with an Emitting Beam 302

7.1 Ways to Study a System 312
7.2 Event List/Handler, Components, Architectures, & Message Handler 318
7.3 A Template for a Simulator 332
7.4 System Configuration when $Clock = 1$ 333
7.5 System Configuration when $Clock = 5$ 333
7.6 System Configuration when $Clock = 10$ 334
7.7 System Configuration when $Clock = 15$ 334

8.1 The Trip-Flop Architecture with a Few Important Points Marked . . 340
8.2 The Trip-Flop Architecture with Numbered Components. 343
8.3 Three Components and their Corresponding Flow Graph. 346
8.4 The Signal Flow Graph with Vertex Labels 347
8.5 The Directed Signal Flow Graph without Vertex Labels 348
8.6 The Directed Signal Flow Graph with Vertex Labels 349

8.7 Signal Flow Graph Highlighting One Simple Path $\{3, 2, 8, 13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function . . . 353
8.8 Signal Flow Graph Highlighting One Simple Path $\{2, 3, 5, 6, 13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function . . . 354
8.9 Signal Flow Graph Highlighting One Simple Path $\{12, 3, 7, 8, 9, 13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function . . . 355
8.10 The Generic Power Analyzer Template. . . . 356
8.11 Generating a Power Analyzer for an Architecture by Using the Generic Power Analyzer Template. . . . 357
8.12 Signal Flow Graph Highlighting One Simple Path $\{2, 8, 9, 13\}$ and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function . . . 359
8.13 Signal Flow Graph Highlighting One Simple Path $\{3, 2, 4, 1, 3\}$ and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function . . . 360
8.14 Signal Flow Graph Highlighting One Simple Path $\{7, 8, 2, 4, 1, 3\}$ and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function . . . 361
8.15 The Generic Delay Analyzer Template. . . . 362
8.16 Generating a Delay Analyzer for an Architecture by Using the Generic Delay Analyzer Template. . . . 363

9.1 The General Framework for Code Synthesis . . . 379
9.2 A Template with an Embedded Action EA_1 . . . 380
9.3 Executing the Embedded Action EA_1 to Produce a New Template Containing EA_2 and EA_3 . . . 380
9.4 The Intermediate Result Produced After Evaluation of All the Embedded Actions . . . 381
9.5 The Final Code Produced for the Specific Architecture. . . . 382
9.6 An Example of a Template with an Embedded Statement. . . . 383
9.7 Recursive evaluation of the embedded statements. . . . 384
9.8 Infinite Recursion with Splicing. . . . 386

11.1 Michelson's Setup for the Banyan Network. . . . 401
11.2 3D-Layout of Michelson Setup for the Banyan Network . . . 406

12.1 The Schematic of the Optical Comparator. 411
12.2 3D-Layout of Optical Comparator Unit 415

13.1 The Schematic of the Shifter Module. 418
13.2 3D-Layout of Mechanically Adjustable Shifter Module 421

14.1 Block Diagram of the Architecture. 426
14.2 Schematic of the Edge Detector. 428
14.3 The Original Input Image. 429
14.4 The Output Showing the Edges. 430
14.5 The Vertical Edges. 431
14.6 The Horizontal Edges. 432
14.7 3D-Layout of Optical Edge Detector 437

15.1 A Serial Adder using Optical Look-Ahead. 441
15.2 3D-Layout of Optical Serial Adder 451

16.1 The Schematic for the Perfect Shuffle Architecture. 455
16.2 3D-Layout of Perfect Shuffle Interconnection 459

17.1 The Trip-Flop Architecture. 463
17.2 3D-Layout of All Optical Trip-Flop 467

18.1 The Schematic of the Correlator Associator. 471
18.2 3D-Layout of The Correlator Associator 474

19.1 The Schematic of the Correlator Detector. 477
19.2 3D-Layout of The Correlator Detector Module 480

20.1 The Schematic of the Thresholder. 483
20.2 3D-Layout of Optical Thresholder Unit 487

21.1 The Complete Optical Memory Architecture. 489
21.2 3D-Layout of Optical Memory Architecture 493

22.1 Logic Using S-SEED Devices. 496
22.2 The Vector Operation. 498
22.3 3D-Layout of Optical Vector Operation 503

23.1 The Shifter Module. 507
23.2 3D-Layout of The Shifter Module 511

24.1 Schematic of S-SEED Implementation. 514
24.2 The Complete Layout of $System_1$. 515
24.3 3D-Layout of $System_1$ of AT&T Switching Fabrics 518

25.1 Image Division Beam-Combination Unit. 522
25.2 3D-Layout of Beam Combiner Unit 527

26.1 The Optical Hardware of the Crossover Interconnect. 531
26.2 3D-Layout of Crossover Interconnection Network 535

27.1 The Passive Beam-Combination Unit. 538
27.2 3D-Layout of Passive Beam Combiner 542

28.1 Fourier-Plane Spot Array Generation. 545
28.2 3D-Layout of Spot Array Generator 548

29.1 The Schematic of $System_2$ of the AT&T Switching Fabrics. 552
29.2 3D-Layout of $System_2$ of AT&T Switching Fabrics 556

30.1 The Integrated Spot Array Generator. 559
30.2 3D-Layout of Integrated Spot Array Generator 563

31.1 The Schematic of $System_3$ of the AT&T Switching Fabrics. 566
31.2 3D-Layout of $System_3$ of AT&T Switching Fabrics 570

32.1 Pupil Plane Interconnection. (a) 1×3 BPG Interconnect. (b) Banyan Network using 1×3 Interconnect. 573
32.2 3D-Layout of 1×3 BPG Interconnect for Banyan Connection . . . 577

33.1 Lossy Beam Combination. 580
33.2 3D-Layout of Lossy Beam Combiner 584

34.1 Spot Array Generation. 587
34.2 3D-Layout of Spot Array Generation 591

35.1 Block Diagram of the Hardware Realizing a Single Stage of the Banyan Network. 594
35.2 3D-Layout of Single Stage of Banyan Network 597

36.1 The Block Diagram of the Six Stage Banyan Network. 601
36.2 3D-Layout of $System_4$ of AT&T Switching Fabrics 604

37.1 Pupil Division Beam Combination Unit. 607
37.2 3D-Layout of Pupil Division Beam Combination 611

38.1 $System_5$ of the AT&T Switching Fabric. 614
38.2 3D-Layout of $System_5$ of AT&T Switching Fabrics 617

39.1 Snapshot of *Mathematica* Session Illustrating its Computing Facilities 623
39.2 Snapshot of *Mathematica* Session Illustrating its Symbolic Computation Power 624
39.3 Snapshot of *Mathematica* Session Illustrating its Graphics Capabilities 625
39.4 *Mathematica* Front-End, Back-Ends and their Interaction. 626
39.5 Hierarchy of Virtual *Mathematica* Back-End Servers. 627

Thesis Abstract

Name: Atif Muhammed Memon

Title: A System for Prototyping Optical Architectures

Major Field: Computer Science

Date of Degree: December, 1995

Optical computing is the field that exploits various optical phenomena and components towards the realization of different special/general purpose computer systems. A systematic approach towards the design of these architectures is needed. This requires the development of sufficiently broad methodologies, standards for components and devices, and design tools/systems.

This thesis advocates a methodology for the development of optical architectures together with the design and implementation of a system (*OptiCAD*) for facilitating the design activity.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system. The second part contains several optical architectures designed and described using *OptiCAD*. The third part contains annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. These address issues such as component modeling, component libraries, simulator and different kinds of analyses. A large collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. Part V is a reference manual for the whole system.

Master of Science Degree

King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

December, 1995

خلاصة الرسالة

إسـم الطالب الكامل : محمد عاطف ميمن
عنوان الدراسـة : نظام للنمذجة الهندسية البصرية بالحاسب الآلي
التخصص العـام : علوم الحاسب الآلي
تاريخ منح الشهـادة : نوفمبر ١٩٩٥م

الحاسبات البصرية عبارة عن طرق تتبع للإستفادة من المميزات البصرية بحيث يمكن التوصل إلى نمذجة نظم الحاسوب البصري للإستعمالات العامة والخاصة .

فهناك حاجة ماسة لإيجاد نظام موحد لعدة محاولات في هذا المجال ، وهذا يتطلب تطوير الآراء والمعايير الكافية للمكونات والأجهزة اللازمة للتصميم بمساعدة الحاسب الآلي .

وفي هذا البحث سوف نحاول طرح فكرة تطوير نظم للتصميم بمساعدة الحاسب الآلي لتسهيل عملية التصميم البصري . وهذا التصميم سيطلق عليه اسم « التصميم البصري بمساعدة الحاسب الآلي » «OPTICAD» .

وينقسم هذا البحث إلى خمسة أجزاء : فالجزء الأول يحتوي على وصف للمفاهيم والمبادئ المتبعة في عملية التصميم . أما الجزء الثاني فقد احتوى على عدة بصريات صُممت وشُرحت باستعمال التصميم البصري بمساعدة الحاسب . والجزء الثالث يحتوي على برنامج مذكرات «Mathematica» الخاصة بالتصميم البصري بمساعدة الحاسب ، ويتعلق الجزء بالمكونات والنمذجة والمحاكاة والتحاليل . والجزء الرابع والأخير تم فيه تقديم مجموعة كبيرة من الأدوات التي تحتوي على هذه المذكرات . أما الجزء الخامس والأخير فيشتمل على مرجع كامل لهذا البحث .

درجة الماجستير في العلوم
جامعة الملك فهد للبترول والمعادن
الظهران - المملكة العربية السعودية

نوفمبر ١٩٩٥م

Preface

OptiCAD is a Computer Aided Design (CAD) system for designing optical architectures. It was designed and implemented to facilitate the design and verification of optical architectures.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system omitting the implementation details. It is however a complete document describing critical issues, design decisions and algorithms employed.

The second part contains several optical architectures designed and described using *OptiCAD*. These have been carefully chosen so as to illustrate most of the features of *OptiCAD*. This part essentially contains the user's view of the system.

The third part contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. Each notebook is presented as a self contained chapter. These include issues such as component modeling, component libraries, simulator and different kinds of analyses.

A rich collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. These utilities facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to be useful in other domains.

Reading through Parts III and IV is by itself expected to be an illustrative and interesting exercise in appreciating the power of functional programming in general and *Mathematica* in particular. The code is not necessarily efficient. Whenever there was a choice between efficiency and readability/clarity, the latter was preferred.

Part V is a reference manual for *OptiCAD* and the *OHDL* language.

Part I

Design of a CAD System for Optical Architectures

Chapter 1

Introduction

Chapter Abstract

This Chapter discusses basic concepts, mentions related work done in the area of CAD systems, defines optical computing, and explains the motivation for developing the OptiCAD system. It also outlines the structure of this thesis.

1.1 Introduction to Optical Computing

This section gives a brief introduction to the field of optical computing. It suggests some of the requirements of future computers and the role that optics can play in this regard. It also mentions some of the advantages offered by optics and its applications in computing.

1.1.1 What is Optical Computing?

Optical Computing is the study of all aspects related to the understanding of optical phenomena and component models so that they can be used to obtain integrated hardware/software solutions to problems.

1.1.2 Requirements for Future Computers

Computers are finding applications in an increasing number of areas. As the advantages of computers are realized, the demands on their capability continue to grow. The computers of the future will have to be designed so that they can cope up with these ever increasing demands [17].

Computers should be general purpose and flexible enough to run a wide variety of applications and algorithms in different languages. The design should allow for the incorporation of new languages and programming methodologies.

As the user's resources and needs grow, it should be possible to extend the capabilities of the computer. This will eliminate the need for discarding an older computer and purchasing a new, more powerful one. Problems such as software compatibility should not be an issue anymore.

The computer should be reliable in its operations. It should perform all calculations accurately. It should not fail completely in its operations. In case one of its components fail, the computer should continue to perform its operations. It should show graceful degradation. Although it will not be operating at its peak speed, continuation of operation is necessary in many vital applications. This will allow it to be used in critical applications involving real-time operations.

An important part of the computer is its software. This includes the operating

system and application software. The software environment should be efficient and easy to use.

Increased competition in today's market require all tools to be cost effective. The computer is one of the most important tools and hence it should also be cost effective.

In order to realize all the above requirements, the architecture of the computer is changed in the following ways.

- Reconfigurable to achieve fault tolerance.
- May need complex interconnection networks.
- Should have massive parallelism.
- The instruction set should have string manipulation for symbolic computation.

1.1.3 Desired Trend for Computer Evolution

In spite of the enormous computing power of today's computers, they cannot perform tasks that are considered simple by the human brain. One desirable goal is to make a computer outperform the brain in these aspects.

Instead of the tedious task of programming a computer, mechanisms should be incorporated into the computer so that it can learn from its surroundings and "experience". This will not only make the life of the programmer simpler, but will automatically cause changes in the program depending on the circumstances. Such a learning facility will require enormous interaction with the surroundings.

The trend is towards having simple processing elements connected together. This requires a massive interconnection network.

Associations play an important role in increasing the power of the brain over a computer. In fact, humans remember things from associations rather than addresses. Computers are increasingly using associative memories for many of their operations.

1.1.4 Advantages Offered by Optics

The shift towards parallel computing had special requirements that VLSI technology could not fully support. These requirements place a burden on the communications lines and interconnections between nodes of parallel machines. The *Von Neumann bottleneck* represents a fundamental problem in the architectures of serial and parallel computers. This occurs when information is constantly moving to and from a shared memory as it is needed by the processor(s) [59]. Such a problem is easily overcome by optical systems, because of the inherent parallelism that characterizes them.

Following are some of the advantages of optics:

Low Interference:

Since light is made up of photons which have no charge, two beams of light travelling in the same medium do not interfere. Hence millions of signals can travel through the same optical fiber.

Low Electro-magnetic Interference:

The electrons circulating in a conventional computer are susceptible to electro-magnetic interferences and field-disturbances (EMI) [82], and are likely to fail operating in radiation zones (such as nuclear reactors and nuclear battlefields). Optical computers, on the other hand, would continue to function undisturbed.

Parallelism:

One advantage of using light is the exploitation of spatial parallelism. This introduces computation in 3-D. Optical architectures usually tend to exploit 3-D as opposed to the 2-D layout of VLSI design.

Interconnectivity:

Light can travel through free space. Since there is no restriction of placing wires connecting one point to another, massive interconnectivity can be realized. Problems of fan-out are eliminated and broadcast can be easily achieved.

Speed:

Photons being energy do not have the disadvantage of resistance that electrons have to face. This ensures maximum speed of operation.

Size and Cost:

With recent developments in miniaturization of optical components, optical architectures tend to be smaller than their functionally equivalent electronic counterparts. This also leads to lower costs.

Communication:

One major advantage is in the fields of communication and interconnections. A single optical fiber is capable of TeraHertz bandwidth. This is equivalent to one million ethernet cables. Optics finds applications in interconnection networks. Since there is no interference between beams, millions of beams per square inch can coexist. Due to the reduced need for physical wires for carrying signals, reconfigurable networks are easily realizable.

Storage:

Another area where optics has already found application is in storage. Compact

disks (ROM), Write Once (WORM) and Read-Write (Magneto Optical) disks are already in common use. The full potential of optics has yet to be tapped. Even then, all of the mentioned media offer capacities in excess of $10GB$ range.

Holography:

Holography has found applications in both storage and computation. With storage capacities of more than several gigabytes in a space smaller than a coin, holographic elements are becoming increasingly popular. With associative memory search, it is possible to search thousands of words simultaneously.

1.1.5 Applications of Optics

Optical systems fall into either of the following two classes.

1. Special purpose analog systems for
   - image processing
   - signal processing.
2. General purpose digital optical computing systems.

A major portion of ongoing research is towards mimicking existing electronic computing elements (logic gates, memories, switches) using optical components. However, emphasis is recently being given to the special features of optical systems/devices. These devices allow the realization of functionalities which are very difficult to obtain with digital electronics, and/or computationally intractable in the digital domain. In other words, trends are emerging with shifted focus from *technology-driven* to *requirements-driven* methodologies.

Physicists are trying to develop, enhance, and invent optical components which can be used in optical architectures. Many of these new optical components are suitable for building computer systems. Suitability here is in terms of space, and potential for miniaturization. These components have ensured minimal power losses and hardly noticed beam deviation within the experiments they constructed.

Recent advances in optical devices and miniaturization have led to the development of numerous architectures. Most of these architectures employ the inherent parallelism offered by light, low noise, minimum interference and careful problem formulation.

Following are some of the applications of optics:

Image Processing:

Image processing, which involves both pattern recognition and character recognition ([12, 24, 39]). In *image processing*, images are actually thought of in terms of optical signals and light beams. Image processing can be carried out by specific optical systems directly on the image with no need for sampling, quantization and the $\Omega(N^2)$ conventional algorithms that process it. These optical systems operate faster than their electronic counterparts.

Neural Networks:

Optical implementations of neural networks and expert systems ([60, 10, 86]).

Associative Memories:

Optical implementation of storage devices, random access or associative memories ([46, 28, 49]).

Logic Operations:

Optical mimicking of logical/electronic components, such as logical gates and Flip-Flops ([29, 34, 23]).

Arithmetic Operations:

Optical implementation of computational systems, such as subtraction/addition units, look-up tables, numerical base conversion systems, residue arithmetic systems ([56, 54, 19]). Functional capabilities of optics using free space optics are illustrated in Figure 1.1. Following are some of the functions that can be done using optics.

1. ANDing (Multiplication)
2. ORing (Addition)
3. Broadcast (Fan-out capability)

Matrix Operations:

Optical implementation of different matrix operations (see Figure 1.2) which are given a considerable attention ([33, 58, 61]).

Linear Algebra Systems:

Optical implementation of linear algebraic problem solving systems ([13, 27]).

Interconnection Networks:

Optical implementation of different interconnection algorithms, such as the perfect shuffle, Banyan network, Crossover network ([37, 36, 11]).

Numerical Computing:

A fair amount of work related to numerical computing and optical transform actions in general ([81, 59, 25, 62, 87, 14, 68]).

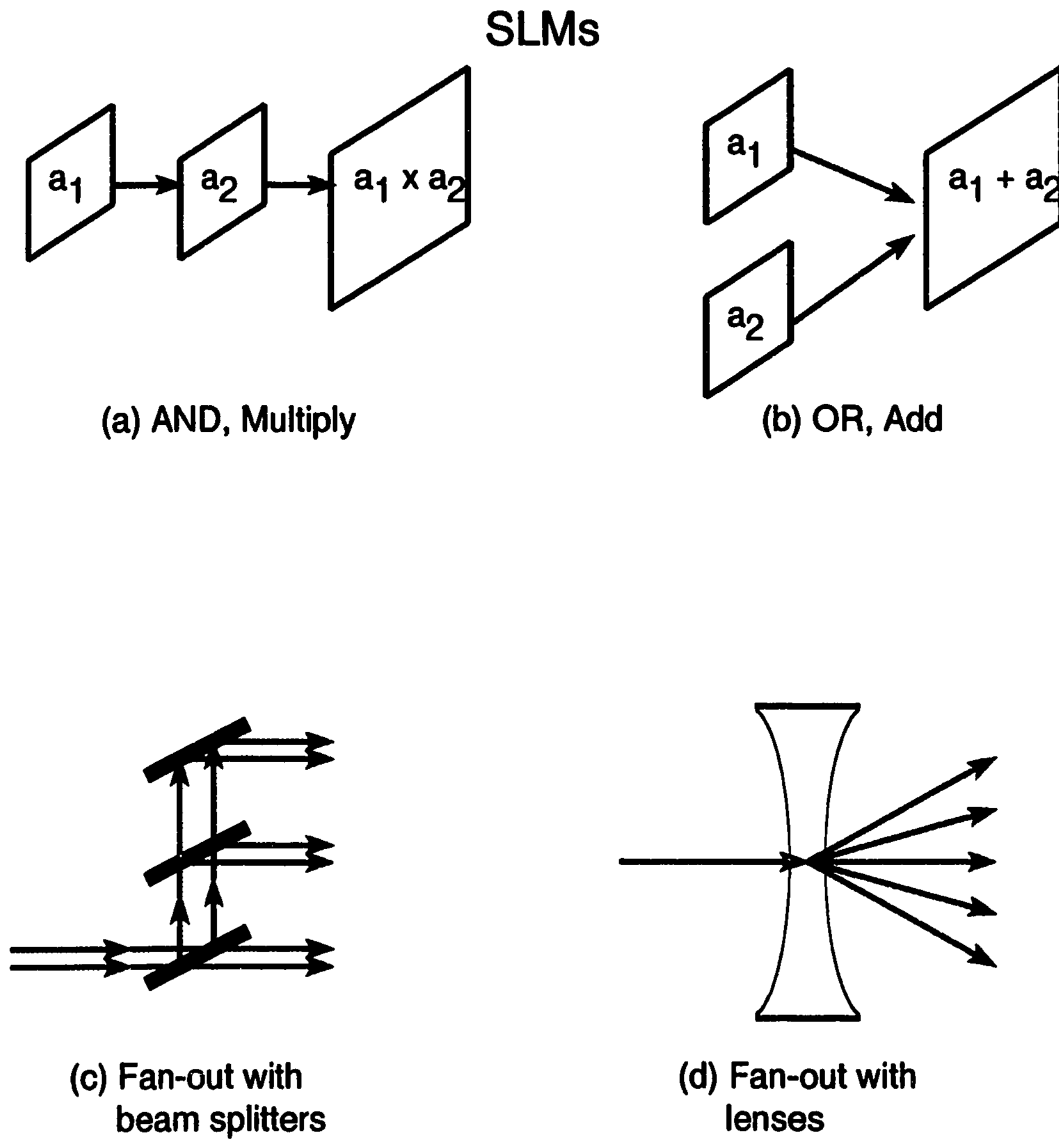


Figure 1.1: Illustration of AND, OR and Fan-out Capability

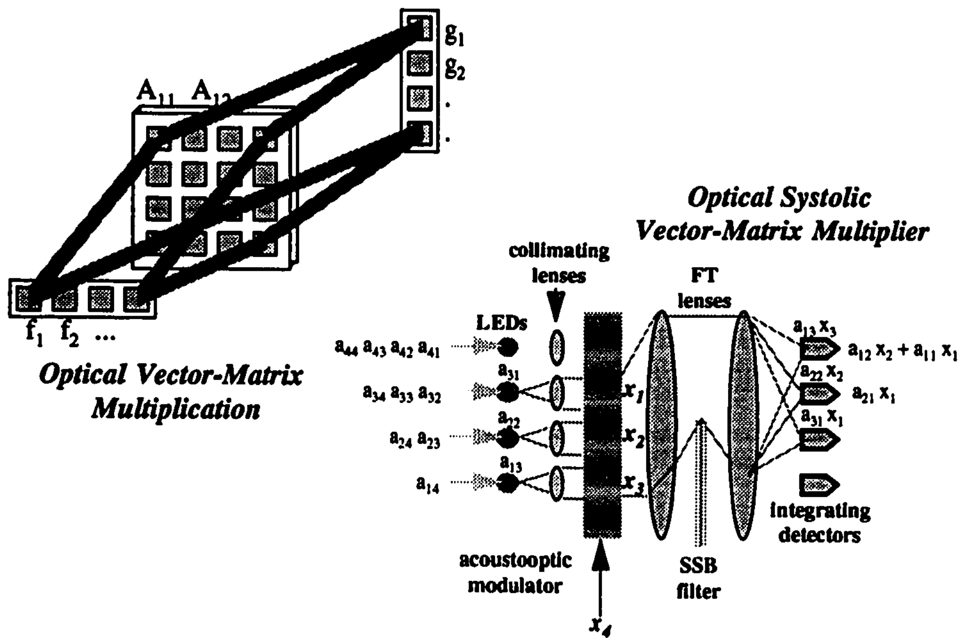


Figure 1.2: Two Different Ways of Realizing Vector-Matrix Multiplication.

1.1.6 General Problems with Optical Computing

Problems arise when comparing optical computers with digital electronic computers. Some of these are:

1. In optical computers, decisions are hard to realize. Signals in optical computers are light beams which cannot disappear all of a sudden by a decision making block of the system. Therefore, in order to effectively use optical computers/computing, multi-level decision making has to be flattened as much as possible.

2. The operations of optical systems are analog by nature, and hence are not very accurate. Attention must be paid to scaling and normalization issues when analog computations are carried out.

It has been found that one of the most time consuming phases of optical system design is the phase of setting up the architecture for testing. The goal of this setup is to establish the feasibility and to prove the correctness of the design. However, in practice this is not a simple task, and several difficulties arise. Some of these are:

1. *The manufacturing limitations of the different optical components.*

   Such limitations force designers to use the available components, regardless of the problems this might cause.

2. *The deviation defects of the light beam used as input to the experiment which might be caused by different factors such as thermal noise.*

   This makes it necessary for the architect to keep changing the positions (alignment) of the components of the experiment according to the direction of the incident beam.

3. *Technical problems due to failures of any component that is part of the experiment.*

   Locating a bad/malfunctioning component is a time consuming task.

4. *The serious problem of alignment/adjustment of the components constructing the design under experimentation.*

   This takes an enormous amount of time and effort. It is very difficult to ensure that the current alignment/adjustment of components will give the desired results. Instead, usually using these components at the specified positions gives a result very close to the desired one. This result has to be enhanced by continuous alignment/adjustment till it reaches the desired level of correctness and accuracy.

5. *Lack of CAD systems for optical architectures.*

   At this point there are no general purpose CAD systems for optical architectures. Existing tools are highly specialized for some application.

6. *Diversity and lack of standardization.*

   As the field is still evolving in several diverse fronts, there are few if any standards. This makes the task of keeping up with the field more difficult.

   Optical architecture design still suffers from *lack of proper design methodologies.* There are no set rules to guide an architect to come up with a design. It is usually done in an intuitive manner.

7. *Insufficient understanding for the automation of CAD systems analyses.*

One of the steps of designing an architecture is its testing. Testing usually involves performing some analyses and validating the results. As the understanding of optics increases, the analyses criteria change and new ones are introduced. Automation of these require a clear understanding of their meaning. Few common and well understood ones are delay analysis, power dissipation analysis, number of surfaces encountered and number of Fourier planes encountered. Others are more complex and less understood such as mechanical stability of an architecture.

1.2 The Problem

The problem that is taken up in this thesis is one of refining the formulation of requirements for an optical architect from the previous accounts [45, 3]. The major issue to tackle with is the design and implementation of an overall CAD system for optical architectures. Such a CAD system should allow the description of an architecture. The designer should be able to analyze the architecture and obtain simulation results. Design of such a system has the following requirements:

1. *A reasonable front-end/language making it natural and easy to describe an architecture.*

   The front-end should provide facilities that are similar to setting up an actual prototype. This includes the provision of optical components, placement and orientation of components, triggering of inputs and viewing the results on a screen.

2. *A layout system for automated layouts.*

Some of the placement issues in architectures are better spelled out as constraints that have to be satisfied in order for the architecture to work properly. The CAD system should allow for the specification of constraints and their enforcement. As a result, an automatic layout of the architecture should be produced satisfying all the spatial constraints.

3. *A visualization system to provide quick feedback.*

   A good visual output of the architecture will help uncover any faults in the specification. It is desirable to have multiple views from different camera positions. Projections can help in minimizing the alignment problems.

4. *Various component databases - to keep the voluminous data hidden from the user.*

   Every practical CAD system for optical architectures must have most of the commercially available components in the form of databases. These are provided by vendors in the form of catalogs. Inclusion of components and their computer models results in large databases. Details of these must be hidden from the user/architect. They should be provided with simple interfaces to the architect for use in the architecture.

5. *Architecture simulation support.*

   The signals and their behavior needs to be traced through each component. Sources and detectors must be identified and results of simulation presented clearly.

6. *Provision for conducting analysis.*

As a result of the simulation, various kinds of analyses (common ones being power and delay analyses) need to be done. The results of these analyses will be used to establish the feasibility and correctness of the architecture.

7. *Provision for libraries.*

   Most architectures make use of a number of smaller assemblies. These assemblies are themselves architectures designed and tested separately. They are treated as components in higher level architectures. This requires provision in the CAD system for libraries where assemblies can be stored and used.

1.2.1 Objectives of the Work

The objectives of the proposed work are:

1. *To understand the requirements of an Optical CAD system.*

   Before designing the CAD system it is important to have a detailed specification from the user or architect. It is important to separate the user's views from that of the library maintainer. The system should provide these two views and the interfaces needed.

2. *To design and implement an OptiCAD system.*

   The formal design of the complete system needs to be carried out. This requires separation of the subsystems and their interaction protocols. Each subsystem then needs to be designed to its lowest level. Further separation may be needed at each level. The CAD system designed and implemented as part of this thesis is henceforth referred to as *OptiCAD* and the language provided to the user as *OHDL*.

3. *To describe various architectures.*

   In order to test the correctness and usefulness of the system, it is necessary to play the role of an architect and describe a few architectures. The choice of these architectures must be carefully made so as to cover a wide spectrum. As these architectures are described, issues of generality and user-friendliness will surface. Another advantage of these descriptions will be a rich collection of components used in each architecture.

4. *To generate example specific simulators and analyzers.*

   Once some architectures have been described, specific simulators can be generated. These simulators upon execution will determine the correctness of the architecture. In the system design phase it will also help in determining bugs and design problems.

1.3 CAD Systems

Design is the course of action to change an existing state of affairs to a preferred one [71]. Basic characteristics of design are:

1. to design is to change
2. design begins with requirements
3. to design is to represent
4. evolutionary nature of design processes [18].

A design method is an explicitly prescribed procedure or set of rules which can be followed by the designer to produce a design. The most widely accepted design

paradigm is what is called *Analysis-Synthesis-Evaluation* paradigm. In this paradigm of design, a stage of analysis is followed by one or more synthesis stages and concluded by an evaluation stage.

Alternatively, the design process can be viewed as a sequence of transformations on behavioral, structural, and physical design representations, at various levels of abstraction. Computer based design automation (DA) tools enable designers to realize large and complex designs. They shorten design time, improve product quality (performance and reliability), and reduce product costs. The DA tools in the VLSI domain can be categorized into the following classes:

1. Design information and flow provides the foundation on which design automation systems are built.
2. Synthesis creates new representations and provides refinements to existing representations for objects being designed.
3. Analysis is to evaluate the consistency or correctness of design representations.
4. Validation is the informal and less rigorous correctness check for the equivalence of two design representations. Verification is to provide a formal process for demonstrating the equivalence of two design representations under all specified conditions.

1.4 Review of Existing Tools

There are a number of CAD tools for optical architectures. A few of the more popular ones have been surveyed and reviewed. A vast majority of them are meant for the

development of imaging systems/architectures. Specifically the following systems were considered: SOLORD, SOLSTAR, OSLO 2–5, CODE V, GEN II Plus, GUERAP V, and OPTEC.

The basic features of these tools are:

1. SOLORD, OSLO 2–5, CODE V, GEN II Plus, and OPTEC are essentially for the design of image forming optical systems; they are mostly used to build, analyze, and design lens-, mirror-, and prism-based systems.

2. They support different analyses techniques such as: wavefront tracing, statistical analysis, spot diagrams, zernekie analysis, gaussian beam propagation, ray tracing, partial coherence analysis, diffraction MTF, and RMS wavefront error analysis. CODE V also supports thermal and pressure environment analysis.

3. Many of these tools use some form of global optimization either by using Lagrange multipliers or simulated annealing.

4. To deal with component parameter tolerances most of these systems introduce an "error function". These error functions are considered during the optimization process.

5. To cope up with the complexity of having too many surfaces, they resort to non-sequential raytracing by which they aggregate a number of surfaces.

6. Most of these systems are interactive. They provide facilities to group together interactive commands as macros. This primitive macro facility is the basis of the language supported by these systems. Hence, these languages are an afterthought and consequently are inflexible.

7. These languages are not geared towards providing a unified view of the total optical system under development. The user of these tools is expected to know the various subsystems and the interaction among them; this restricts the utility of these tools.

8. Most of these tools provide access to databases of optical components from specific manufacturers.

9. Systems like GUERAP V are for the analysis of stray light and radiometry. Since it is outside the scope of this work, we omit the discussion of such systems.

10. SOLSTAR is a software tool for laying out optical architectures.

To summarize, the existing tools suffer from one or more of the following drawbacks:

1. Meant for the design of imaging systems only

2. Specification/Description language is not the major aspect of the system

3. Predominantly interactive systems

4. Limited facilities for constraint specification and handling

5. Lack of hierarchical modeling and analysis techniques

6. Limited number of different types of components that are made available to the system architects.

Following are other kinds of tools/methodologies that were not developed for imaging systems.

1. The HATCH system [63], which is a general tool to design fiber optic and waveguide switching based systems. It starts with a lumped delay design. It verifies the sequential behavior of these systems using logic simulation of idealized gates. Component delays then allow HATCH to produce fiber lengths for a distributed delay design. When loss and crosstalk specifications are added, HATCH identifies critical paths for insertion of signal restoring switches. The final design is then simulated with delay, loss and crosstalk specifications. This simulation produces logarithmically scaled plots of signal amplitudes versus time under worst case loss and crosstalk assumptions [40].

2. A methodology for the design of continuous dataflow synchronous systems [64]. It is more like an algorithm for solving delay constrained minimization problems. The simplicity of this algorithm gave it better performance. It was therefore included in the HATCH to carry out delay distribution [40].

3. A methodology for the design of digital optical computers was proposed in [57]. This methodology is described for arrays of optical logic gates interconnected in free space. It supports regularity in design and simplicity in the design process. It is a design technique based on programmable logic arrays (PLAs). The PLA is a general component that allows any function or group of functions to be implemented in a regular structure. The approach is to first generate all possible minterms, and then to select and combine the minterms that are needed to implement the functions.

4. An imperative style structured programming language for the description of optical architectures was designed and implemented in [45]. This work also contains models of numerous optical components and description of several ar-

chitectures.

5. Lens Lab 3-D [8] a system for designing optical systems and its commercial counterpart *Optica* can display arbitrary combination of lenses, mirrors and other optical elements in two and three dimensions, and compute the path of rays through the elements.

1.5 Hardware Description Languages

Conventional hardware i.e., electronic hardware has evolved over a period of time. There are design methodologies for the realization of a circuit that performs a given task. Tools have been developed to facilitate the task of hardware design. Some of these tools take a description of the intended hardware and perform some simulation. This description may be in the form of a graphical drawing or in the form of a textual language. These are the hardware description languages.

The hardware description languages provide facilities for simple and accurate description of the hardware. They are used by several tools use the description and provide domain specific analysis. The restriction to domain is because of the inflexibility of these tools. Implicit in their implementation is that the described hardware will be implemented in the electronic domain. The analyses are also fixed to electronics and results are produced as tables representing binary values.

Some of the commonly known hardware description languages are AHPL, VHDL [79], and BHDL.

Existing HDLs cannot easily be extended to other hardware such as optical hardware. This is because of some limitations in their basic design. These limitations are

mainly due to their application and design for digital electronic circuits.

Current HDLs are well suited for digital, discrete signals only. They cannot handle analog signals easily that are commonly used for processing in optics domain.

Electronic architectures and circuits are inherently 2-D. One of the major advantages of optics over electronics is the 3-D placement. Any CAD system for optical architectures must have support for 3D graphics, 3D visualization and 3D placement. This is missing in all tools meant for electronics.

The flow of signals in optics is different from a current flowing through a wire. These are mostly analog quantities flowing through components. Three different theories of light need to be supported.

1. ray propagation
2. beam propagation
3. wave theory

These theories attempt to explain the way in which light travels through an architecture. The component models must be designed to adapt to any of the three models.

1.6 Paradigms Supported by $\mathcal{OHDL}$

The designed CAD system makes use of a description language that has features common to the three programming/specification language paradigms – *functional*, *object oriented* and *rule based*. This following three sections provide an overview of these paradigms.

1.7 Functional Programming

Programming languages are based on paradigms or styles. The most widely used being the *imperative style.* The language designed and implemented in this thesis is based on the functional programming paradigm.

This subsection gives a review of the functional programming paradigm starting with the motivation for its adoption. Since functional programming finds its roots in λ - calculus, a brief overview of λ - calculus is given. Advantages of functional programming are discussed by means of examples. Some implementation issues are discussed followed by examples of how parallel applications can be developed using functional programming languages.

1.7.1 Motivation for Adopting Functional Programming

The programming power of the functional programming languages is generally not clear to the "non-functional" programmers. Exploitation of the capabilities of functional programming languages requires a thorough understanding of the paradigm.

Backus in his classical paper [4] describes the inadequacy of "conventional programming languages". The various "defects" described therein are summarized below.

1. Word-at-a-time style of programming.

2. Close coupling of semantics to state transitions.

3. Programming divided into statements and expressions.

4. No notion of reusability.

5. Lack of mathematical properties for reasoning about programs.

In order to choose a suitable programming language paradigm, it is important to specify some "requirements".

Mathematical: Proof of correctness of a program requires a sound mathematical model as its foundation.

State: Understanding the behavior of program is simplified if state transition is well defined and simple.

Abstraction: Problem representation in a program form can be greatly simplified if the underlying model allows a natural description of the problem without constraining the user with low level issues.

Every Programming Language has two main parts.

Framework which defines the overall rules of the system. For example, the FOR statement in *Pascal* is part of its framework. The language definition has fixed the semantics of the FOR statement and almost all other statements.

Changeable Parts whose existence and use is anticipated by the framework but whose particular behavior is not specified by it. For example the PROCEDURES, FUNCTIONS and LIBRARY ROUTINES of *Pascal* are its changeable parts.

Almost all conventional style languages have a large framework with very little flexibility left for changeable parts. What is needed is a minimal framework providing a great variety of powerful features entirely as changeable parts. Such a system could support many different features and styles without being changed itself.

The fundamental operation in Functional Programming is the application of functions to its arguments [78]. The top level program is a function which receives the program's input as arguments. The result of this function is the output of the program. This top level function is itself defined in terms of simpler functions. These simpler functions are themselves either primitives or defined in terms of yet simpler functions.

Functional programming consists of an algebra of programs/functions whose variables are functions and whose operations are combining forms. These combining forms are used to build higher level functions from primitive ones. These combining forms follow certain well understood algebraic laws. As a consequence it is relatively straightforward to obtain proofs of program correctness.

Since functional programs are viewed as an algebra, the execution reduces to successive transformation of states. Only one state transition occurs per major computation.

As opposed to imperative style languages, functional style languages do not have the assignment statement. This greatly simplifies program understanding. Values once given to variables can never be changed. Since a function is expected to return unique results for a given set of argument values, there is no notion of updating global variables within a function body. In fact functional programming is done without use of variables. A function body contains formal parameters which serve as place holders for values substituted as actual parameters.

Functional programs contain no side effects. This is due to the absence of global variables. Side effects are the major cause of bugs in imperative style languages. Since there are no side effects, a function can be computed at any time. This eliminates the need to explicitly define the execution order of the program.

Functions are used to express computational abstractions. The method used to carry out a computation is not given explicitly by the programmer. In fact no assumptions are made about the architecture of the underlying machine. Since the arguments of functions can be evaluated in any order, functional programs contain a vast amount of implicit parallelism [77]. Hence functional programs are more portable than conventional programs.

1.7.2 λ - Calculus

Theory Behind Definition and Manipulation of Functions

- How to define functions.

  Extension - to define a function as an explicit set of pairs, as in

  $f_1 \equiv \{<a,1>, <b,2>, <c,3>, <d,4>\}$.

  Comprehension - specify a function by giving its properties, as in

  $plus \equiv \lambda\ a,b \bullet a+b$.

- The general form of a **lambda expression** is

  $\lambda\ id_1, id_2, ..., id_n \bullet expr$

  where id_i are identifiers and $expr$ is some expression which may involve these identifiers. In the above, λ plays the role syntactically similar to that of a quantifier such as $\forall$ and $\exists$. This function has n arguments.

Example Functions

- $plus \equiv \lambda\ a,b \bullet a+b$.
- $square \equiv \lambda\ x \bullet x^2$.

- $successor \equiv \lambda\ n \bullet n + 1.$
- $predecessor \equiv \lambda\ n \bullet n - 1.$
- $even \equiv \lambda\ n \bullet divides(2, n).$
- $max \equiv \lambda\ x, y \bullet if\ x\ \geq\ y\ then\ x\ else\ y\ endif.$
- $Id \equiv \lambda\ x \bullet x$ – the identity function.

Typed λ Notation is useful when it is necessary to spell out the sets to which the function arguments must belong, as in

$successor \equiv \lambda\ n : N\ \bullet n + 1$

where N is the set of *Natural numbers.*

Generic Functions are those which may be applied to arguments belonging to any set as in *Id.*

Properties of functions may be defined by completely abstracting the members of the base set, as in

$successor \circ predecessor = Id$

1.7.3 Functional Programming - The Paradigm

Functional programming style is one of writing programs using expressions[5]. Since programs need to calculate a large number of values, an expression could be used to specify a whole collection of values. Hence an entire program is written as a single expression. The meaning of any of its parts can be easily understood without examining the entire program.

Comparison with Conventional Programming

Functional programming can be compared with conventional programming in a rudimentary way by means of an example. Consider the example of calculating the **Inner Product.**

Conventional Program The notation used for all imperative style programs will be *Pascal* style. The code for Inner-Product using an imperative style of programming will be the following.

```
c := 0;
For i := 1 to N do
        c := c + a[i] * b[i];
```

- The above code is inherently sequential.
- It does not convey the meaning of Inner Product, rather it spells out a mechanical way to compute it.
- One must trace the code mentally in order to understand its behavior.
- It is not general in that a part of the data i.e., N is responsible for the number of iterations.
- One major disadvantage is that the above code names its arguments i.e., it can only be used for vectors **a** and **b** and result must be stored in **c**.
- It also lacks hierarchy. It does not render itself for the construction of higher forms from simpler ones.

Functional Program In this thesis, the notation used for all functional style programs will be *Mathematica* style. The code for Inner-Product using an functional style of programming will be the following.

Function[{a, b}, Apply[Plus, MapThread[Times, {a, b}]]]

The above *Mathematica* code uses the following *functional forms*. These combine existing functions to form new ones. Detailed explanation is given in later sections.

Apply [f, {a1, a2, ...}] gives

f[a1, a2, a3, ...].

MapThread [f, {{a1, a2, ...}, {b1, b2, ...}, ...}] gives

{f[a1, b1, ...], f[a2, b2, ...], ...}.

- The functional style code for Inner-Product is simple, operates in two steps, and provides a definition of Inner-Product.
- It does not spell out the steps of computation; rather it defines the computation as an expression.
- No assumptions are made about the underlying machine.
- The code is hierarchical being built from simpler functions.
- It does not contain any bound named data. It can be applied to any pair of compatible vectors.

The above discussion shows that functional languages are more powerful compared to conventional languages. They can be used to write more concise, declarative programs at a much higher level of abstraction. The programmer need not worry about the lower machine dependent details. This helps the programmer to define the problem in a more natural manner.

Data Structures

Pairs

An object formed by grouping together several simpler objects is called a *Data Structure*[5]. The simplest data structure could be one grouping two objects. Such a data structure is called a *Pair*. The following are pairs.

[1, 2]

[thesis, report]

Indeed pairs need not contain only similar *Type* of objects. The following would equally qualify for pairs.

[year, 1970]

[2.45, 2]

Pairs can be used as actual arguments of dyadic[1] functions.

Tuples

Consider the following structure.

[article, book, thesis]

Although the above data structure could be thought of as formed by two applications of the pairing operation i.e.,

[1] a mathematical expression formed by addition or subtraction of dyads - PAIRS

[[article, book], thesis]

there is a major difference between the two. The difference is that

[article, book, thesis]

does not associate to the left. This object is *flat* with each object at the same *level.* Hence this is a new data structure with its own properties. It is called a *Triplet.* In general, *n-Tuples* may be formed by grouping together n objects.

Several operations may be defined over tuples, one of the most common is test-for-equality. General functions can be defined to carry out operations on tuples. For example given a tuple of the form

[Hours, Minutes, Seconds]

one could define a function **Convert** which takes a 3-tuple and returns the total number of seconds. Such a function could be defined as follows.

Convert = #1*3600 + #2*60 + #3 &

Application of the function is of the form.

Convert[20, 10, 5]

yielding

72605.

Lists

Most programs manipulate objects where the number of values is not known at the time of writing the program. Tuples cannot be used in these general cases. **Lists** are used in such cases.

In general a List is a grouping of an arbitrary number and types of objects. For instance,

{1, 2, 3, 4, a, b, thesis}

{book, {article, report}, 4, 5}

{ }

are some examples of lists.

Many useful operations are defined over lists. Examples of some common ones are as follows.

Append[{1, 23, 4}, {A, B, C}]

Prepend[1, {2, 3, 4}]

First[{2, 3, 4, 5}]

Rest[{2, 3, 4, 5}]

Functionals

In most functional programming languages, functions are treated as *first class citizens* i.e., they can be used in the most general manner. They can be passed as arguments

to other functions and can be returned as results of evaluating functions. Functions which yield functions as results are called *functionals.*

Allowing functionals gives functional programming its claimed power to build specification for computation quickly. Arbitrary functions can be defined/created and discarded dynamically as the programs execute. This comprises of the program's *changeable parts* discussed earlier.

Functionals can be used in data structures. Hence it is possible to have data structures of the following form.

{Sin[x], Front[Rest[List]], Plus[x, y]}

Add[Last[x], First[x]]

Programming style changes considerably after introducing the concept of dynamically created programs/functions. Special purpose functions need not be written at all. Instead the programming environment need only give the essential framework for the constructions and maintenance of functionals. In addition a library of *classes* of functions can be provided. Program writing would become creating instances of the function *Objects.*

Use of Functions Instead of Data Structures

The concept of functionals defined in the preceding section eliminates the need to maintain data in data structures. Instead functions defined dynamically could be used to store the data in a compact manner. This not only helps in simplifying the program considerably but also helps to abstract the data. All data accesses are in the form of function calls. Any storage of data would require the definition of a new function.

The notion of using functions instead of data structures is explained in subsequent sections with the help of examples.

Building Higher Level Functions

An important feature of higher level programming is that it allows abstractions at all levels. In particular it should be possible to build new abstractions from existing ones. These can be either functional abstractions or data abstractions.

Higher level functions may be defined by *partial application.* Partial application binds some, but not all of its arguments to values. The result is a function with fewer arguments, but the same number of results. The following benefits can be gained from partial applications.

1. The compiler may be able to perform transformations to produce better code. Of course, it is not always possible to simplify the function at compile time. However, partial simplification could be possible.

2. It allows better utilization of already existing code. Specific purpose functions can be defined in terms of existing general ones.

For example, consider the following function call to a general function.

Append[{0}, {1, 2, 3, 4}]

which appends the element *0* to the list {*1, 2, 3, 4*}. If such an operation is used regularly, it is desirable to define it as a new function of one argument by partial application to APPEND.

Append0 = Append[{0}, #1]

No Side Effects

The *assignment statement* is the most widely used statement in conventional language programs. This is also responsible for many undesirable effects. One of them is causing *side effects*. It changes the value of the variable on the left hand of the assignment. Thus the value of any variable at a certain point of the program execution is defined by the last executed assignment statement for this variable. Execution of any procedure that uses global variables is affected not only by the value of its parameters but also by the assignments to these global variables. Likewise, the effects of the procedure execution may include changes to the values of the global variables that have assignment statements in the procedure body.

Functional languages do not use assignment statements. In fact the lack of variables in functional programs and the absence of a state is an ideal platform for avoiding side effects. Any function once defined cannot be influenced by external forces to change its computation. It only returns the computation based on its parameters.

Absence of side effects in a program is useful for realizing its parallel execution. In conventional programs if there are parallel execution threads each containing assignments for a variable, then the final value of this variable will depend on the speed at which those threads are executed. Functional programs can show a larger amount of implicit parallelism due to:

1. Lack of imposition of restrictions in the form of control structures, as in imperative style languages.

2. Freedom from side effects.

Polymorphism

Functions provide a powerful means of computation because they abstract out the actual computation. For example the function

$$\textbf{Celcius} = (\# + 32) * 5/9 \ \&$$

captures the idea of *add* 32, *multiply by 5 and divide by 9* for every Fahrenheit by abstracting it as a formal parameter.

Polymorphism also provides an abstraction mechanism by capturing the *shape* of data structures by abstracting the *types* of their components. Hence, when *Lists* capture the idea of indefinitely long linear arrangements of objects, and functions that of indefinitely deep arrangements, polymorphism abstracts both these and allows the definition of a single function for them. The power of abstraction is that it allows us to recognize patterns of data and computation which occur repeatedly in programs and uses a single data type or function to represent or process every instance of that pattern.

1.8 Fundamentals of Object-Oriented Programming

In the objected-oriented programming paradigm [80], objects are the atomic units of encapsulation. Classes manage collections of objects, and inheritance structures collections of classes.

1.8.1 Objects

Objects partition the state of a computation into encapsulated chunks. Each object has an interface of operations that control access to an encapsulated state (Figure 1.3). The operations determine the object's behavior, while the state serves as a memory of past invocations that can influence future actions [80]. An object named Point with state variables (instance variables) x, y and four operations for reading and changing them can be defined as follows:

```
Point: object
    x:=0;
    y:=0;
    read-x():
        return x;
    read-y():
        return y;
    change-x(dx):
        x:=x+dx;
    change-y(dy):
        y:=y+dy;
```

By accepting messages only through read and change operations, the object Point protects its instance variables x, y from direct manipulation by other objects. The operations of an object share its state. Therefore, state changes by one operation can be seen by subsequently executed operations. For example, $read - x$ and $change - x$ share the variable x which is non-local to these operations although x is local to the object.

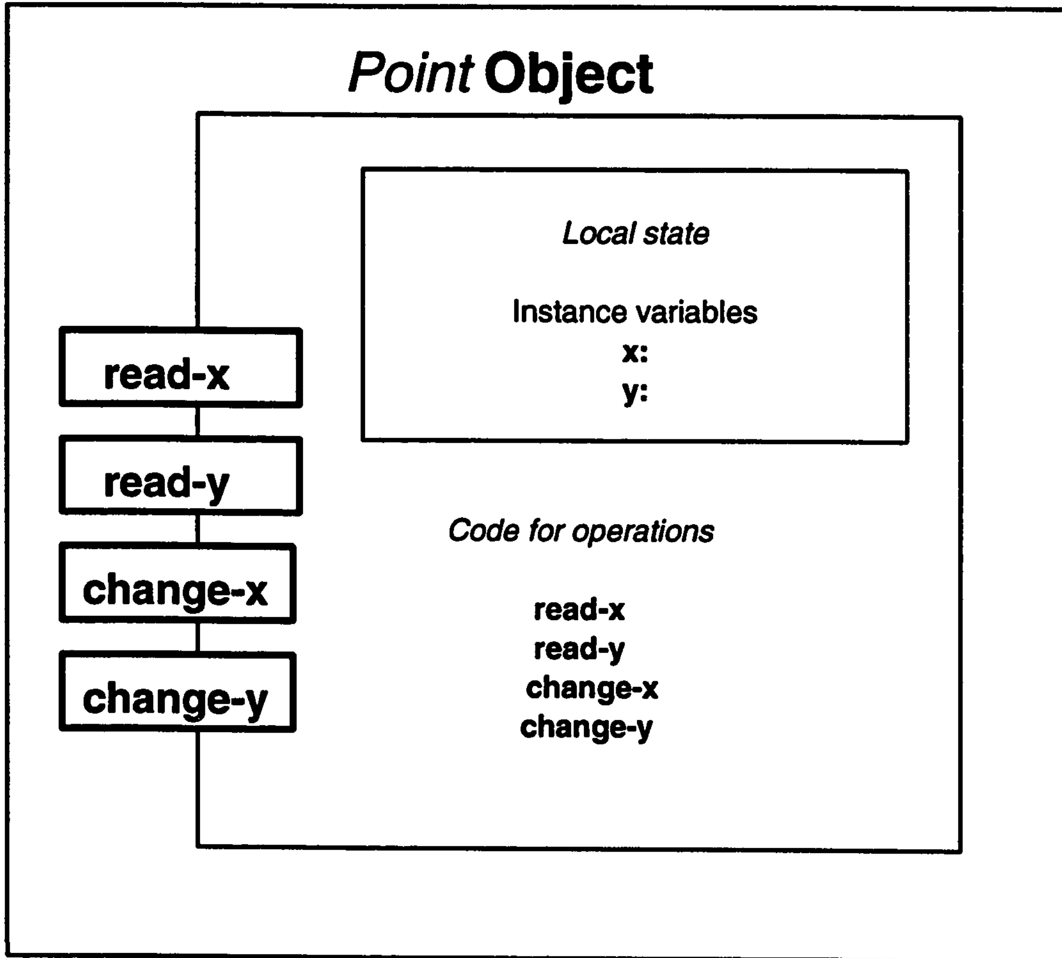


Figure 1.3: Point Object

1.8.2 Classes

Classes specify the behavior of collections of objects with common operations. For example, points with read and change operations may be specified by a class Point with two instance variables x, y and four explicit operations as follows:

```
Point: class
    x:=0;
    y:=0;
    read-x():
        return x;
    read-y():
        return y;
    change-x(dx):
        x:=x+dx;
    change-y(dy):
        y:=y+dy;
    create():
        return aNewPointID;
```

Classes specify the set of messages accepted by objects of the class. They define an encapsulation interface. Classes also serve as templates for creating objects with the specified interface and implementation behavior. For example, two instances $p1, p2$ of the class Point can be created as follows:

```
p1 := create Point;
p2 := create Point;
```

Each of the two point instances $p1, p2$ is an encapsulated chunk of state whose behavior is determined by the class operations.

1.8.3 Inheritance

Inheritance is a mechanism for sharing the code or behavior common to a collection of classes. It factors shared properties of classes into superclasses and reuses them in the definition of subclasses. Programmers can therefore specify incremental changes of class behavior in subclasses without modifying the already specified classes. Subclasses inherit the code of superclasses, to which they can add new operations and possibly new instance variables, and even override the behavior of an inherited procedure/function.

Figure 1.4 describes a class of mammals with persons and elephants as its subclasses. The class of persons has mammals as its superclass and students and males as its subclasses. Each instance John, Joan, Bill, Mark and Dumbo - has a unique base class. An instance's membership in more than one base class such as John's being both a student and a male, cannot be expressed.

1.8.4 Advantages of the Object-Oriented Paradigm

The object-oriented paradigm is useful for both the system user and developer. It lets the system user abstract the real world entities into objects in a natural manner. When the system is developed based on object-orientation as well, the user view maps easily into the system developer view. The simplicity of this mapping avoids a lot of the redundant overhead on the part of the developer and lets him work at a higher level of abstraction.

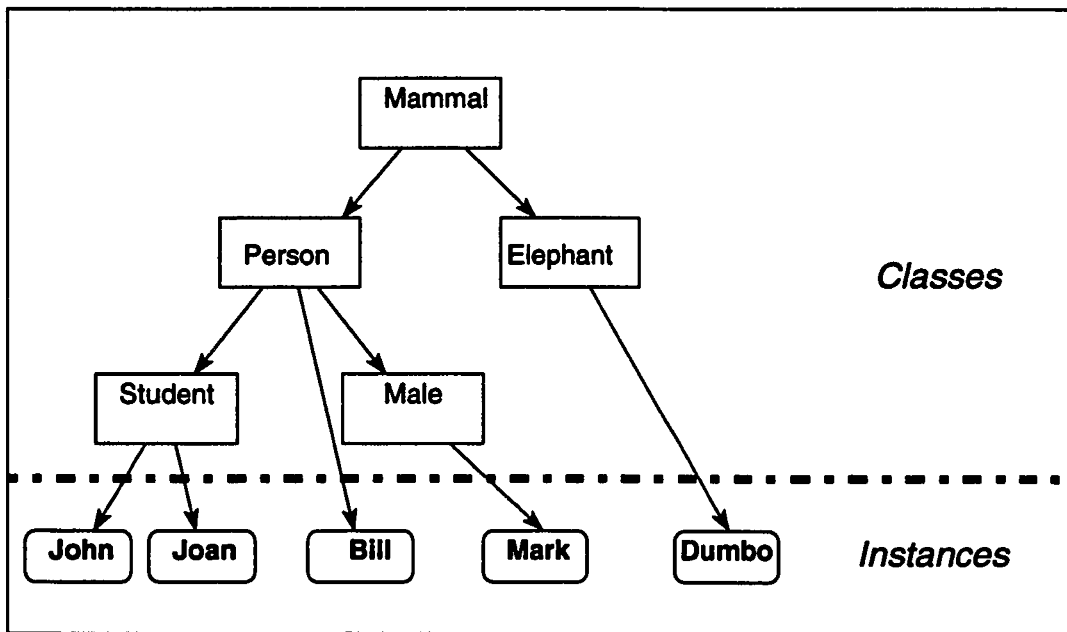


Figure 1.4: Example of an Inheritance Hierarchy

Other advantages of the object-oriented paradigm include: encapsulating object data and interfaces, and extending the development environment through inheritance, subclassing and code reuse.

Many people have anticipated the benefits of combining graphics with object-orientation [22], and few researchers have designed systems towards this. Wisskirchen [83] implemented the functionality of GKS and PHIGS in an objected-oriented fashion using Smalltalk. Other object-oriented systems include HOOPS [43] and Dore [44].

HOOPS (Hierarchical Objected-Oriented Picture System) allows application programmers to group primitives and attributes as objects. Once an object has been defined in this manner, it can then be used to form other objects providing a hierarchical construction capability.

The drawback of these systems is that they require the user to work at a low level to specify his requirements rather than allowing him to work at an abstract level suitable to the designer.

1.9 Rule Based Systems

Some systems take user specification in the form of a collection of rules. These follow a declarative paradigm. The combined effect of these rules defines the computation.

The user does not have to specify the steps of carrying out the computation, rather he has to describe the nature of the computation. The rules define the valid transformations that the domain of the computation allows.

Since the user does not specify the exact steps to realize the computation, an internal mechanism is needed. This mechanism should take the rules specified and must attempt to obtain the result following a certain order.

Rule based systems usually require the rules to be in a specific format. Rules must have a firing-precondition and an associated action part. Whenever a rule's precondition is satisfied, the associated action is taken. The built in mechanism must be capable of deciding which rules to fire and to resolve contentions among possibly many enabled rules.

Rule based systems are generally at a high level of abstraction and hence easier to use in many applications. The rules are easy to specify since the user can focus all his attention to a local context of one rule only, that too related to the problem at hand. As the user does not specify the execution sequence, these systems are easier to optimize.

One major drawback of rule based systems is their management in large scale systems. A flat rule base would be too cumbersome to handle in any practical sized system. Sophisticated, hierarchical management techniques must be employed in such cases.

1.10 Constraint Handling in Visual and Symbolic Languages

Both visual, graphical, geometric systems and symbolic systems depend heavily on some constraint specification and enforcement techniques.

The most natural way to spell out the placement of objects is in the form of constraints. The architect describes only those constraints that dictate the functionality and appearance of the architecture. Any placement satisfying all constraints is acceptable.

Constraints are restrictions on parameters/configurations so as to capture the desired relationships among them. These constraints may also be viewed as assertions about the desired shape/behavior of an optical component/architecture. The advantage of using constraints is that spatial/integrity relationships can be easily specified.

1.10.1 Role of Constraints in Geometric Modelling

Design is the process of making a sequence of decisions from among a number of alternatives. These decisions are made with the aim of configuring a system that is expected to meet a set of requirements. Design activity pervades all engineering fields. That is, design is an integral part of engineering.

The alternatives presented at every stage are usually due to the integrity, one has to preserve, in the domain under consideration. For example, choice of a material from several that exhibit a certain property.

The choices are restricted by fundamental properties and are governed by the laws of the domain. Integrity of the operations and the consistency of the design depends on the satisfaction of the restrictions inherent to the domain.

In the normal course of design in a particular domain, a designer is aware of only the integrity constraints that design has to fulfill. The design process is one of experimenting with numerous combinations of choices until all the integrity constraints of the domain are met. This process of exploring the design space (space of all possible design alternatives) is time consuming.

An alternative design strategy is to capture the semantics of the domain and its integrity rules as constraints and attempt to look for solution by automated techniques. This process is less time consuming and any solution found will meet the spelled out

requirements.

The problem of determining whether a set of constraints has a solution or not is in general undecidable. However, a restricted version of the problem wherein efficient search could be conducted would be of value in design support tools. Such tools capture some of the domain semantics and prune the design space guiding the designer through feasible space.

Hence tools/techniques are needed for the specification of domain constraints as well as their satisfaction for proposed alternative configuration.

1.10.2 Constraint Specification Techniques

There are numerous ways of spelling out constraints.

1. Mathematical equations relating different parameters/equations.
2. Undirected/BiDirectional labeled flow graphs or some such graphical modelling techniques such as Petri-nets.
3. Graphical Hold-Click-and-Specify paradigms.
4. Predicate calculus.
5. Rules.
6. Procedural.
7. Informal.

These are explained in detail in Section 5.2.2.

1.10.3 Constraint Satisfaction Techniques

Once a set of constraints has been specified, there are several methods of finding solutions. These methods depend on the way in which the constraints were specified.

1. Mathematical solution techniques for systems of simultaneous equations.
   (a) Iterative techniques
   (b) Genetic techniques
   (c) Search techniques
2. Graph models
   (a) Local propagation
   (b) Graph transformations

These are explained in detail in Section 5.2.3.

1.11 Optical Architectures

Numerous architectures are proposed in the literature. In this thesis, the following architectures[2] have been considered:

Michelson Setup for the Banyan Network

This architecture achieves the Banyan interconnection using only optical components [35].

[2]Each of these is presented as a Chapter in Part II.

An Optical Comparator Unit

This architecture gives the result of comparison of two input data. It has applications in building switches for interconnection networks.

Mechanically Adjustable Shifter Module

This architecture achieves the spatial shift of beams [52]. This is needed in applications where a beam must be directed to a certain output. The architecture exploits free-space interconnectivity of optics.

An Optical Edge Detector

This architecture exploits spatial parallelism, non-interference among intersecting beams and free-space connectivity to solve the real-time, variable-resolution edge detection problem. It has applications to image processing, computer vision and robotics.

Optical Serial Adder

This is an architecture of a serial adder that uses all-optical components. It employs the carry look-ahead algorithm in order to speed up the addition operation.

Perfect Shuffle Interconnection

This is an optical implementation of the perfect shuffle interconnection network [50]. It is constructed from classical optical components.

An All Optical Trip-Flop

The optical Trip-flop is a tri-state all-optical unit of memory. It can be used as a building block in higher order memory units like registers and latches. It has applications in designing optical buses.

The Correlator Associator

The correlator associator extracts an stored image matrix identified by the input image. It has applications in neural networks, storage retrieval, holography.

The Correlator Detector Module

The correlation detector (CD) represents the first layer of an associative memory two layer neural network. This architecture is used to correlate the input image with the stored patterns for an optical associative memory. The output of this subsystem is transferred to the opto-electronic thresholder of the associative memory neural-network [9].

Optical Thresholder Unit

The thresholder thresholds the input image employing an iterative scheme (attenuation). It serves as the connection between the first layer (correlator detector) and the second layer (correlator associator) in an optical memory architecture [9].

Optical Memory Architecture

The optical memory architecture is a feed-forward associative-memory neural network architecture [9]. It makes use of three architectures – the correlator associator, correlator detector and the thresholder.

Optical Vector Operation

This architecture implements S-SEED based optical logic gates. These gates are capable of performing optical logic functions such as NOR, OR, NAND and AND [47]. They can be used to implement complex switching building blocks such as 2×1 and 2×2 switching nodes. The inputs to these gates are

differential to avoid any critical biasing of the device. The complete architecture is constructed from off-the-shelf catalog components.

The Shifter Module

The shifter is a general purpose device with many applications. One application is the implementation of interconnections – straight and shifted. The shifter takes as input a pair of beams. The output is pairs of straight connected beams and crossed beams. This is the basic building block in many interconnection networks and is constructed by using only off-the-shelf optical components.

$System_1$ of AT&T Switching Fabrics

$System_1$ is the first of a series of five switching fabrics demonstrators constructed at AT&T. These systems demonstrate two-dimensional arrays of optical and optoelectronic devices which employ free-space interconnections. These demonstrators use S-SEED technology as the device platform. $System_1$ is the S-SEED based switching demonstrator.

Beam Combiner Unit

The beam combiner's basic functionality is to produce the combined image of the two super-imposed input images. It regenerates the desired signals out of clock to restore intensity and sharpness.

Crossover Interconnection Network

The crossover interconnect provides the connectivity between consecutive stages of a general optical multistage interconnection network (MIN) [38].

Passive Beam Combiner

The passive beam-combiner is actually a beam-combiner with the input laser

source replaced by an SLM to provide collimated laser source.

Spot Array Generator

Two dimensional arrays of SEED technology based switching nodes require an optical power supply to clock the devices [76]. The generation of 2D-arrays of uniform intensity spots requires two basic components. The first is a high power, single frequency, diffraction limited laser that can provide the appropriate power per pixel needed to meet the system speed requirements. The second component in a spot-array generator requires some mechanism to equally and uniformly divide the power from the laser and distribute it to the optical windows of the S-SEEDs. This architecture achieves the spot array generation.

$System_2$ of AT&T Switching Fabrics

This architecture is called $System_2$ of the AT&T switching fabrics [31]. It realizes a four-stage interconnection network.

Integrated Spot Array Generator

The spot array generator provides the optical power supply to clock devices [76]. These are generated as 2D-arrays of uniform intensity spots. This is done by using two basic components. The first is a high power, single frequency, diffraction limited laser that can provide the appropriate power per pixel needed to meet the system speed requirements. The second component in a spot-array generator provides a mechanism to equally and uniformly divide the power from the laser and distribute it to the optical windows of the S-SEEDs. This architecture achieves the spot array generation.

$System_3$ of AT&T Switching Fabrics

The availability of the integrated spot array generator allows changes and modifications to $System_2$ of the AT&T switching fabrics [31] resulting in an improved system – $System_3$.

1×3 BPG Interconnect for Banyan Connection

This architecture provides Banyan connectivity between two matrices.

Lossy Beam Combiner

The lossy beam-combination unit is used to combine several signals into a single signal.

Spot Array Generation

As systems become more sophisticated and use more components, there is a need for a spot array generator that provides a large number of equal power spots of light. This architecture redistributes the input optical power into 2048 equal power spots of light.

Single Stage of Banyan Network

This architecture is a single stage of the Banyan network. This single stage is modular i.e., it can be used to build systems of multiple stages of the Banyan network.

$System_4$ of AT&T Switching Fabrics

This architecture called $System_4$ achieves the functionality of a six-stage Banyan interconnection network. It is composed of a six stage 16×32 Banyan network (32 bit wide data path) with 1024 (32×32 array) AS-SEED per stage.

Pupil Division Beam Combination

The pupil division beam-combination unit is used to combine several signals into a single signal. This has the advantage of being less lossy than other approaches presented in the literature.

$System_5$ of AT&T Switching Fabrics

$System_5$ is the last of the five switching fiber demonstrators implemented at AT&T. It has increased functionality and temporal bandwidth.

1.12 Simulation

Simulation is the imitation of operations of various kinds of real-world facilities or processes with or without using a computer. If a computer is used then it will be called computer simulation. The facility or process under study/examination is referred to as a system. In order to study a system scientifically a set of assumptions must be made about its working. These assumptions constitute a *model.*

Since in the real world, relationships that compose the model are often complex, it is not possible to analyse the model *analytically.* Hence exact solutions to questions of interest are usually not available. Simulation permits the analysis of a model numerically. Data is generated to estimate the desired true characteristics of a model.

An optical architecture is a complex system containing continuously changing quantities. It is often not possible to exactly analyse the functionality of an architecture. One of the main reasons for this is the complex nature of light itself. For many components it is not known what attributes of light are changed. To obtain exact results, it is necessary to set up an actual prototype experiment. However, this

is very difficult and time consuming. A rough estimate about the functionality of an architecture can be obtained by simulation.

Of the different types of simulations, the ones of interest for this thesis are outlined next.

1.12.1 Discrete-Event Simulation

Discrete event simulation concerns the modelling of a system as it evolves over time. The state variables change instantaneously at separate points in time. These points in time define the events in the system. An *event* being an instantaneous occurrence that may change the state of the system.

1.12.2 Continuous Simulation

In this type of simulation the modelling is done for a system in which the state variables change continuously over a period of time. Usually continuous simulation models involve differential equations that give relationships for the rates of change of the state variables with time. If these equations are simple, they can be solved analytically, otherwise numerical-analysis techniques can be used to approximate the result.

1.13 Analyses of Optical Architectures

The main purpose of describing and simulating an architecture is studying the results of the simulation. These results will help in determining the feasibility of the architecture and making appropriate changes.

These could be based on spatial/temporal dimensions. Some important ones are as follows.

- Surfaces, Lens Passes, Image Planes
- Input Power
- Power Distribution in Space and Time

  input dependent

  input independent

- Delay Analysis in Space

  input dependent

  input independent

- Heat Dissipation in Space and Time
- Scatter in Space and Time
- Volume and Surface Analyses
- Throughput
- Maximum Operable Clock for Synchronous Architectures
- Hardware Cost
- Mechanical Stability
- Number of Fourier transforms
- Number of image multiplications

- Number of operations per seconds
- Degree of parallelism
- Maximum permitted signal to noise ratio
- Resolution (number of lines/pixels per unit length)
- Power
- Delay
- Intensity
- Polarization
- Distribution
- Crosstalk and noise
- Coherence
- Correlation

All of the above are considered in detail in Chapters 7 and 8.

1.13.1 Different Approaches

The analyses can be done according to one of the following models.

deterministic: If the model does not contain any probabilistic (i.e., random) components, then it is called *deterministic*. The output can be determined once the inputs and relationships have been specified.

Examples of deterministic modelling approaches are signal flow graph and system transfer function.

probabilistic: When the model contains some random components, then it is called probabilistic. The output of these models is random and can only be treated as an estimate of the true characteristics of the model. Most queueing systems are modeled using this approach.

1.14 Code Generation and Generators

The user description of the architecture is given in $\mathcal{OHDL}$. This description is translated to an internal form that is used for creating data structures and carrying out further translations.

Once the essential parts of the user description have been extracted, code can be synthesized for specific tasks. Code synthesis in a target language requires adhering to the syntax and semantics of that target language. A complete specification of the target language must be given to the code synthesizer. Apart from syntactic requirements, this involves keeping track of data structures to be used, type checking requirements, proper conversions and the programming paradigm of the target language.

A major concern for transforming specifications from one language to another is the preservation of their semantics/meaning.

Code generation plays a major role within the scope of this work. The specifications given by the user are at a very high level of abstraction. These cannot be directly taken for interpretation for efficiency considerations. It is much more feasible

to generate code for a particular aspect of the user description to a low level language which can then be efficiently executed. For example, if the architect wants to carry out several types of analyses (e.g. (1) Power, (2) Delay) on the architecture, specific code generated for only one aspect (e.g. Delay) of the architecture will be much more efficient in execution time.

Some common code generators are *lex* and *yacc* that are meant to generate code for lexical analyzers and parsers to be used in compilers/translators.

1.15 Geometry and 3D Graphics

3D-graphics is the basis for any spatial-placement system. Geometry forms the basis of 3D-graphics. This section briefly introduces the basics of geometry and 3D-graphics.

1.15.1 Solid Modeling

The building blocks that the proposed 3D graphics system uses are 3D solid components. The most widely used solid representations are *constructive solid geometry* (CSG) and *boundary representation.* CSG allows an object to be defined as a boolean combination of other objects such as intersection and union [26]. A *solid* is defined as a set of points in 3D space, with a surface separating points in the set from those not in the set. Given a number of solid objects (sets of points) in space, one can either combine them into composite objects (using the operation union), use only those points they have in common (set intersection) or use one solid to "carve out" another by subtraction (set difference).

In boundary representation a solid is defined as a set of surfaces that enclose a

space. The surfaces must join together to completely enclose the space, providing a well-defined inside and outside such that every point in space is either inside or outside the solid.

To display the edges of a 3D solid on a computer display the boundary representation is much easier to handle, whereas CSG is much easier for the user to handle than the boundary representation [41]. So, many existing systems employ CSG method for inputting an object, and automatically transform the CSG representation into the boundary representation to store the object internally [66].

1.15.2 Basic Geometric Transformations

Except for references to faces and edges, the only data that are stored in the boundary representation are the vertices in the 3D space, i.e., vectors of the form (x, y, z). In order to be able to view an object from different perspectives (angles) and to zoom in and out the particular object, one can apply three geometric transformations: translation, rotation and scaling. Transforming a geometric object is achieved by transforming all of its vertices.

A transformation of a vertex can be defined as a multiplication of the vertex with a corresponding transformation matrix. Then, we would be able to combine different transformation matrices by multiplying them. In order to represent a general transformation as a matrix multiplication, a vertex is required to be stored as a four-element, rather than a three-element, vector [26]. Then vertex v_i is represented as,

$$v_i = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix}. \tag{1.1}$$

The following matrix translates the vertex v_i a distance D_x along the $x-$axis, D_y

along the $y-$axis, and D_z along the $z-$axis:

$$T(v_i) = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix} \quad (1.2)$$

$$= \begin{bmatrix} x_i + D_x, & y_i + D_y, & z_i + D_z, & 1 \end{bmatrix}. \quad (1.3)$$

The following matrix scales the vertex v_i (as an endpoint) by S_x along the $x-$axis, S_y along the $y-$axis, and S_z along the $z-$axis:

$$S(v_i) = \begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.4)$$

$$= \begin{bmatrix} x_i * S_x, & y_i * S_y, & z_i * S_z, & 1 \end{bmatrix}. \quad (1.5)$$

The following matrix rotates the vertex v_i by the rotation angle θ about the $z-$axis:

$$\begin{bmatrix} x_i, & y_i, & z_i, & 1 \end{bmatrix} * \begin{bmatrix} cos\ \theta & sin\ \theta & 0 & 0 \\ -sin\ \theta & cos\ \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.6)$$

which gives:

$$\begin{bmatrix} x_i * cos\ \theta - y_i * sin\ \theta, & x_i * sin\ \theta + y_i * cos\ \theta, & z_i, & 1 \end{bmatrix}. \quad (1.7)$$

With suitable rotation matrices, one could achieve rotation of points (objects) around $x-$axis, $y-$axis or in general around a given line by a given angle. The reader is referred to [26] for further details.

Mathematica is a system for doing mathematics on a computer. *OptiCAD* is implemented using *Mathematica.* Chapter ?? gives the various features of *Mathematica.*

1.16 Outline of the Thesis

The objective of this thesis was the design and implementation of the *OptiCAD* system. This document describes the features of the system including a view of the user interface, the design issues and implementation details.

This thesis consists of four logical parts. Part I consists of algorithms, concepts, design decisions taken during the evolution of the *OptiCAD* system. It does not describe the implementation details of the system. Part I is made up of the following Chapters:

Introduction

This Chapter discusses basic concepts, mentions related work done in the area of CAD systems, defines optical computing, and explains the motivation for developing the *OptiCAD* system. It also outlines the structure of this thesis.

A Tour of *OptiCAD*

This Chapter presents the user's view of the *OptiCAD* system. *OptiCAD* is used to design an example optical architecture, a three state memory unit, called the Trip-Flop. The complete *OHDL* specification is given for the description of the Trip-Flop. Various analyses are carried out on the architecture and the results

are presented as graphs. The Chapter concludes by presenting a methodology for the design of optical architectures.

Top Level View of *OptiCAD*

A high level design overview of the *OptiCAD* system is presented in this Chapter. A development history of the system is given, outlining the roles of different players who interact with it. The Chapter also outlines the major subsystems and their interaction.

The Hardware Description Language

This Chapter describes *OHDL* – the specification language used by *OptiCAD*. It also describes the different constructs of the language and explains the function of the directives.

Placement

This Chapter presents an approach wherein the placement of optical components can be simplified by developing a 3D graphics system.

Modelling of Optical Components

This Chapter gives the notion of a component model. It outlines the different types of models and their uses and presents models of some optical components.

Simulation

This Chapter presents some issues related to simulation in general and to optical architectures in particular. It discusses various kinds of simulation techniques and presents a methodology for the simulation of optical architectures.

Power and Delay Analyses

This Chapter presents the role of analysis in designing optical architectures. It

outlines a technique for obtaining the power and delay analysis by using path enumeration.

Code Generation

This Chapter presents a technique for generating code to produce an independent tool that carries out a particular analysis on the specific architecture instead of simulating it in a general framework.

Conclusion

This chapter summarizes the work presented in this thesis. It highlights the specific contributions to the field of CAD software for optical systems, the limitations of *OptiCAD* in its current form, and outlines possible extensions and future work.

Part II (Chapters 11–38) is a collection of architectures that were designed and described using *OHDL* – the user interface language of *OptiCAD*. These architectures were mainly taken from the published literature. Some, however, were designed during the development stages of the system. All of these illustrate the expressive power of *OHDL* and the capabilities of *OptiCAD*.

Part III (Chapters 39–50) is a collection of *Mathematica* notebooks that constitute the implementation of the *OptiCAD* system. A Chapter on *Mathematica* begins this Part.

Part IV (Chapters 51-65) is a collection of general utilities that were developed to enhance the capabilities of the underlying implementation platform. Most of these can be applied to domains other than optics.

Part V (Chapters 66-69) provides a reference to the *OptiCAD* system.

1.17 Summary

This is the first chapter of this thesis and mostly outlines the basic concepts and previous work in the area.

Optical computing was defined and some of its applications were presented. A review of the previous and ongoing work in the area was presented. The motivation of this thesis and its objective was described. Some of the general problems with carrying out experiments using optical components were mentioned. The desired trend for computer evolution and the role of optics in this regard was presented. Some recent discoveries and their impact on today's computers was discussed.

An introduction to hardware description languages was given. Some programming language paradigms, namely functional, object-oriented and rule-based were described in detail. Constraint specification and satisfaction techniques were outlined. A small description of some of the architectures described later in the thesis was given. Different simulation techniques were described. The analyses requirements of optical architectures were outlined. The role of code generation and generators with respect to this thesis was introduced. Some concepts of geometry and 3-D graphics were presented. A brief sketch of the complete thesis was provided.

Chapter 2

A Tour of OptiCAD

Chapter Abstract

The user interface of the OptiCAD system is presented. An example architecture - the Trip-Flop, is designed using the system. The complete OHDL specification for the description of the Trip-Flop is given. Various analyses are carried out on the architecture and the results are shown as graphs. A methodology for the design of optical architectures is given.

2.1 Introduction

This Chapter shows the working of *OptiCAD* as seen by the user. An example of a design problem is considered. Subsequent sections show the various stages that the user must go through in order to describe his requirements and achieve the desired output. The output could be in the form of 3-D layout, a simulation and a specific analysis of the described architecture. The focus of this Chapter is on the features and capabilities of *OptiCAD*. The detailed study of the Trip-Flop and its design is

discussed in Part-II, at the price of some redundancy.

2.2 A Problem Definition

The problem of designing a tri-state memory element is considered in this Section. Specification of Trip-Flop architecture is not concerned with the details of the implementation. Also no assumption is made about the domain of the final solution to the tri-state element problem.

2.3 A Specification for Trip-Flop

The optical Trip-Flop is an all optical single unit memory. It has the capability to store any of three state values *Low, High* or *S*. Hence it functions as a tri-state flip flop (called Trip-Flop).

The Trip-Flop can be used as a building block for the realization of higher order memory units like registers and latches. Its three state behavior enables its use in buses where three levels are needed.

Input:

$\mathbf{i} \in \{None,\ Low,\ High,\ S\}$

Output:

$\mathbf{o} \in \{Low,\ High,\ S\}$

Function:

The Trip-Flop has a current configuration **c** $\in$ {*Low, High, S*}.

if (**i** ! = *None*) then

c $\leftarrow$ **i**

endif

return **c**

A schematic of the optical Trip-Flop is shown in Figure 2.1.

The configuration **c** of the Trip-Flop is defined using the following settings of the individual components. Component names are indicated in bold font.

$$\mathbf{c} = \mathit{Low} \triangleq \text{config}(\mathbf{if1}) = \text{transmitting \&\&}$$
$$\text{config}(\mathbf{if2}) = \text{transmitting.}$$
$$\mathbf{c} = \mathit{High} \triangleq \text{config}(\mathbf{if1}) = \text{reflecting \&\&}$$
$$\text{config}(\mathbf{if2}) = \text{transmitting.}$$
$$\mathbf{c} = S \triangleq \text{config}(\mathbf{if1}) = \text{don't care \&\&}$$
$$\text{config}(\mathbf{if2}) = \text{reflecting.}$$

Input: when it is *Low/High* it is p-polarized (circular).

when it is *S* it is s-polarized.

pbs1 splits circularly polarized input and reflects s-polarized input.

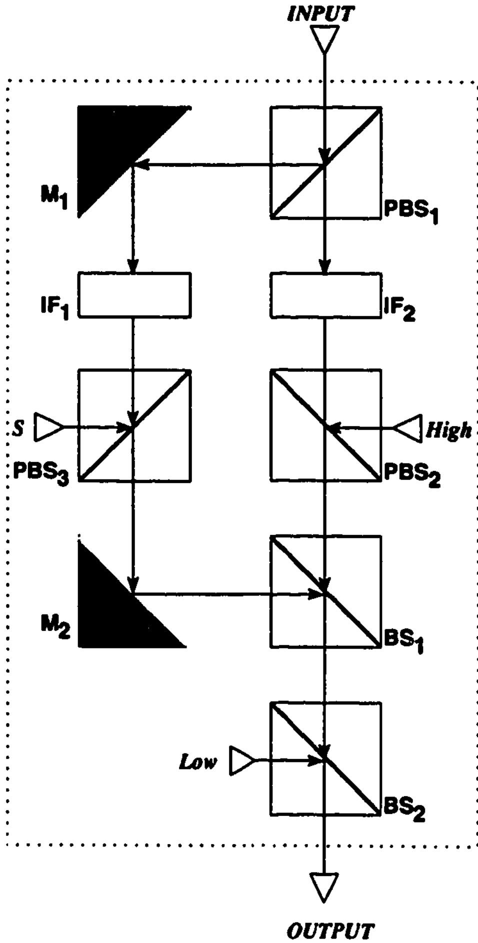


Figure 2.1: The Trip-Flop Architecture - Logical View

2.4 Instantiation of Components

Each optical component belongs to a generic class. It inherits all the properties of that class. Some attributes specific to that component are overridden. For example the class of beam splitters has two sub-classes namely polarizing and non-polarizing beam splitters. Further refinement could include the shape, size, operating temperature.

The two functions provided for component instantiation are

GetObject[ ]

This takes a component type and returns an instance of a component. Additional attributes such as catalog number may be provided as an argument. GetObject also provides additional mechanisms for specifying user defined attributes to the instantiated component.

GetObjects[ ]

This is a generalization of **GetObject[]**. This function returns multiple instances of similar components. In addition, an array of components with homogenous attributes can also be obtained.

For the Trip-Flop architecture, following are all the instantiations.

In[1]:=

```
{pbs1, pbs2, pbs3} = GetObjects[3, "PolarizingBeamSplitter", CatalogNumber ->
"PBS1"];

{if1, if2} = GetObjects[2, "InterferenceFilter", CatalogNumber -> "IF1"];

{m1, m2} = GetObjects[2, "Mirror", CatalogNumber -> "M1"];

{bs1, bs2} = GetObjects[2, "BeamSplitter", CatalogNumber -> "BS1"];

plh = GetObject["PulsedLaser", CatalogNumber -> "PL1", Intensity -> high];

pll = GetObject["PulsedLaser", CatalogNumber -> "PL1", Intensity -> low];

pls = GetObject["PulsedLaser", CatalogNumber -> "PL2", Intensity -> s];
```

2.5 Layout of the Architecture

The layout includes two phases. The first one is placing the components and the second one is orienting the placed components.

Spatial relationships are specified as constraints between components. $\mathcal{OHDL}$ provides several mechanisms for specifying spatial relationships as constraints.

2.5.1 Placement

These are the constraints related to maintaining distances between components. For example, it may be required that a component be x units away from another. Every component is assumed to have a bounding box. This is the smallest box that can enclose a component. A point (by default one of the corner points) on this bounding box is used for the position constraints.

Position constraints may be specified in any of the following ways.

default[ ]

The architect need not specify the exact coordinates of a component. A component placed at **default[]** will be assigned coordinates according to the satisfaction of the placement constraints. The architect may consider this component as placed. Any component may be placed relative to the **default[]** placed component.

Absolute[ ]

Although not a good practice, the architect may wish to specify the absolute coordinates for the placement of a component. An **absolute[]** placement requires the specification of the exact coordinates of the component in the architecture.

relativeTo[ ]

A convenient and natural way of specifying a component's position is with respect to that of another by giving the relative distances between them. This allows capturing the position constraints among components in a local context.

The following are placement constraints for Trip-Flop architecture.

In[2]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

In[3]:=

```
AddPlacementConstraint[default[pbs1]];
AddPlacementConstraints[{
 relativeTo[pbs1, m1, {-a, 0, 0}],
 relativeTo[m1, if2, {0, -b, 0}],
 relativeTo[if2, pbs3, {0, -c, 0}],
 relativeTo[pbs3, pls, {-f, 0, 0}],
 relativeTo[pbs3, m2, {0, -d, 0}],
 relativeTo[pbs1, if1, {0, -b, 0}],
 relativeTo[if1, pbs2, {0, -c, 0}],
 relativeTo[pbs2, plh, {g, 0, 0}],
 relativeTo[pbs2, bs1, {0, -d, 0}],
 relativeTo[bs1, m2, {-a, 0, 0}],
 relativeTo[bs1, bs2, {0, -c, 0}],
 relativeTo[bs2, pll, {-h, 0, 0}]}
];
```

In[4]:=

```
ComputePositions[];
```

AddPlacementConstraint(s)[] is the top level call provided to the user to specify placement constraints.

The directive **ComputePositions[]** uses the specified position constraints and component instantiation to compute the exact coordinates of each component. It also reports any errors for insufficient or infeasible constraints. Figure 2.2 shows the Trip-Flop architecture after all the components have been placed but not oriented.

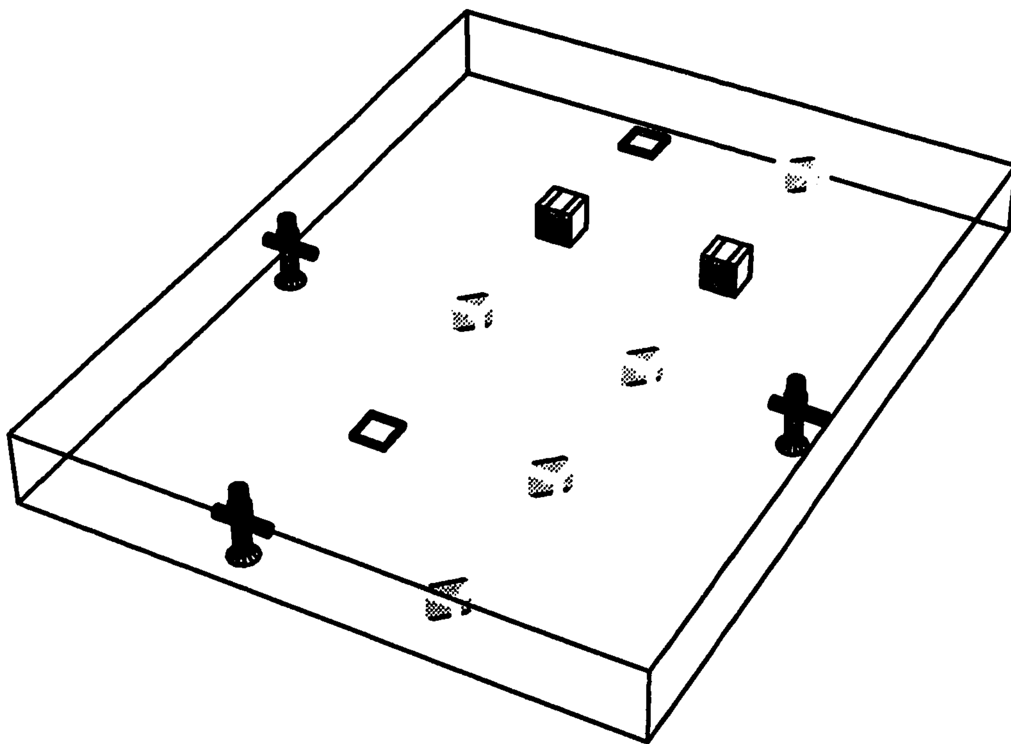


Figure 2.2: Snapshot of the Trip-Flop Architecture Without Orientations

2.5.2 Orienting Components

Orientation constraints specify the relationship between the angles of the components. For example, a component should be at an angle of $y°$ with another. Specifying orientation constraints is more difficult than placement constraints. One major difficulty is the formulation of a scheme that can uniquely orient an object in 3D space.

A set of constraints are provided by $\mathcal{OHDL}$ which make the orientation specification more natural.

orientationOf[ ]

This is used when the orientation of the component needs to be specified exactly in terms of angles with respect to the bounding box of the whole architecture.

parallelTo[ ]

Two components are parallel if they are placed with exactly the same orientation. Once one of the components have been oriented, all the others that are parallel to it can be oriented by using **parallelTo[]** constraint.

perpendicularTo[ ]

A laser oriented so that its beam strikes a lens face at an angle of 90° is perpendicular to that lens. This can be specified as a **perpendicularTo[]** constraint.

atAnAngle[ ]

Components that differ in only one angle in their orientation can be fixed by using the **atAnAngle[]** constraint.

relativeAngles[ ]

This constraint is useful in specifying the difference between the angles of two components.

The following are the orientation constraints for the Trip-Flop architecture.

In[5]:=

```
AddOrientationConstraints[{
 orientationOf[pl1, {angle[0], angle[0], angle[0]}],
 orientationOf[bs1, {angle[Pi/2], angle[0], angle[0]}]}
];

AddOrientationConstraints[{
 parallelTo[pl1, pls],
 parallelTo[pbs1, pbs3],
 parallelTo[if1, if2],
 parallelTo[pbs2, bs1],
 parallelTo[bs1, bs2],
 parallelTo[pbs3, pls],
 parallelTo[pbs1, if1]}
];

AddOrientationConstraint[perpendicularTo[bs1, pbs1]];

AddOrientationConstraints[{
 atAnAngle[m1, angle[45 degree], if2],
 atAnAngle[pls, angle[180 degree], plh]}
];

AddOrientationConstraints[{
 relativeAngles[m1, pbs1, {angle[45 degree], angle[90 degree], angle[0]}],
 relativeAngles[pbs3, m2, {angle[45 degree], angle[90 degree], angle[0]}]}
];
```

In[6]:=

```
ComputeOrientations[];
```

In[7]:=

```
ShowArchitecture[];
```

AddOrientationConstraint(s)[] is the call provided by $\mathcal{OHDL}$ for the addition of orientation constraints.

The directive **ComputeOrientations[]** uses the orientation constraints specified, it computes the exact angles of the component with respect to the bounding box of the architecture. Any unspecified orientation is assumed to be default i.e., parallel to the architecture's bounding box. It reports any errors for infeasible constraints.

Figure 2.3 shows one view of the Trip-Flop architecture after all the components have been positioned and oriented.

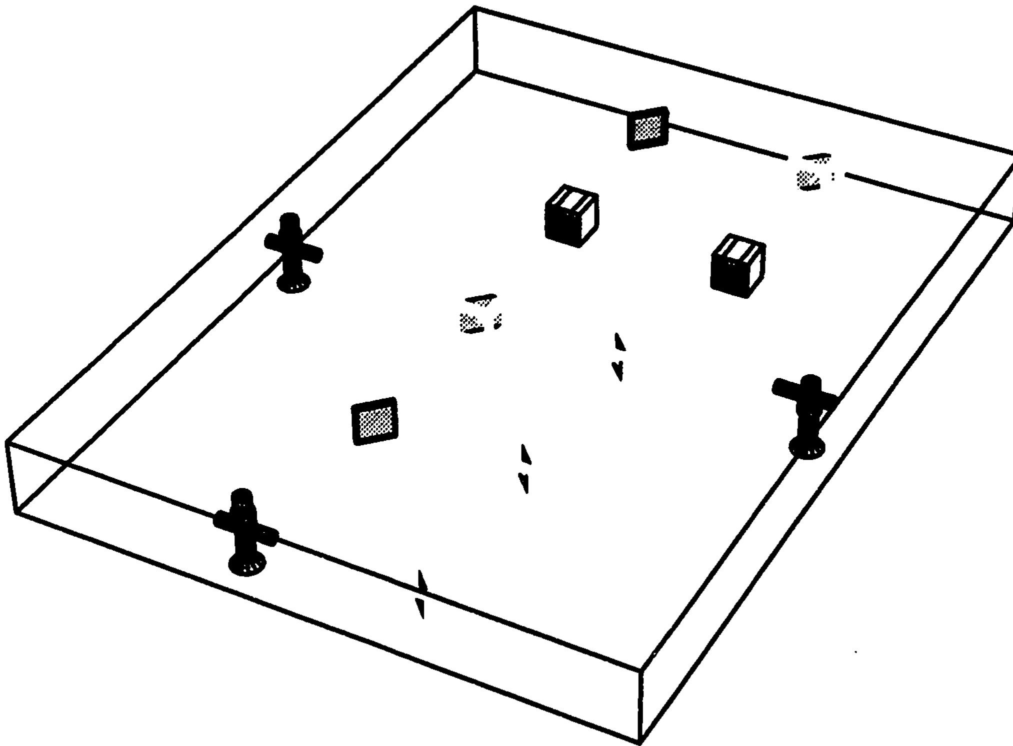


Figure 2.3: The Fully Oriented Trip-Flop Architecture

The **ShowArchitecture[]** directive generates the 3D layout of the architecture from the available description and presents it in the form of a graphics object that can be displayed.

2.6 Viewing - Camera Position

ViewArchitecture[] can be used to view the generated architecture from various camera positions. This is a desirable feature because some objects may obstruct others. Figures 2.4 and 2.5 show the Trip-Flop architecture from two different camera positions.

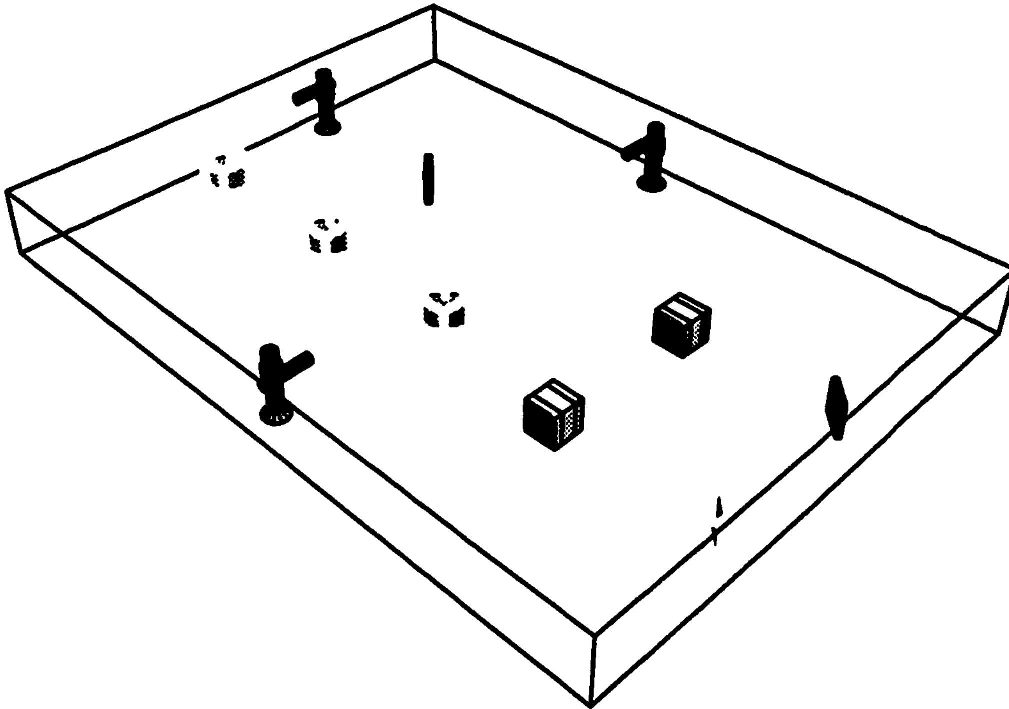


Figure 2.4: A View of the Fully Oriented Trip-Flop Architecture from Camera Position {1, 1, 1}

ViewArchitecture[] provides additional options for viewing, to obtain different kinds of projections. Figure 2.6 shows the Trip-Flop architecture with the components projected on the $x = 0$ and $y = 0$ planes.

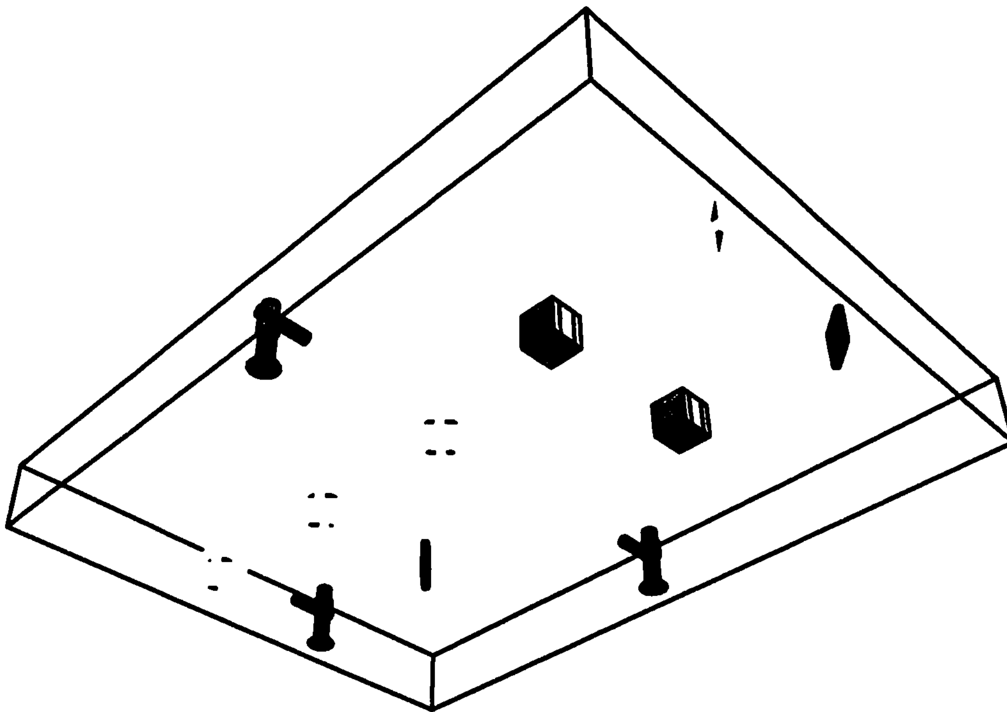


Figure 2.5: Another View of the Fully Oriented Trip-Flop Architecture from Camera Position {1, 1, -1}

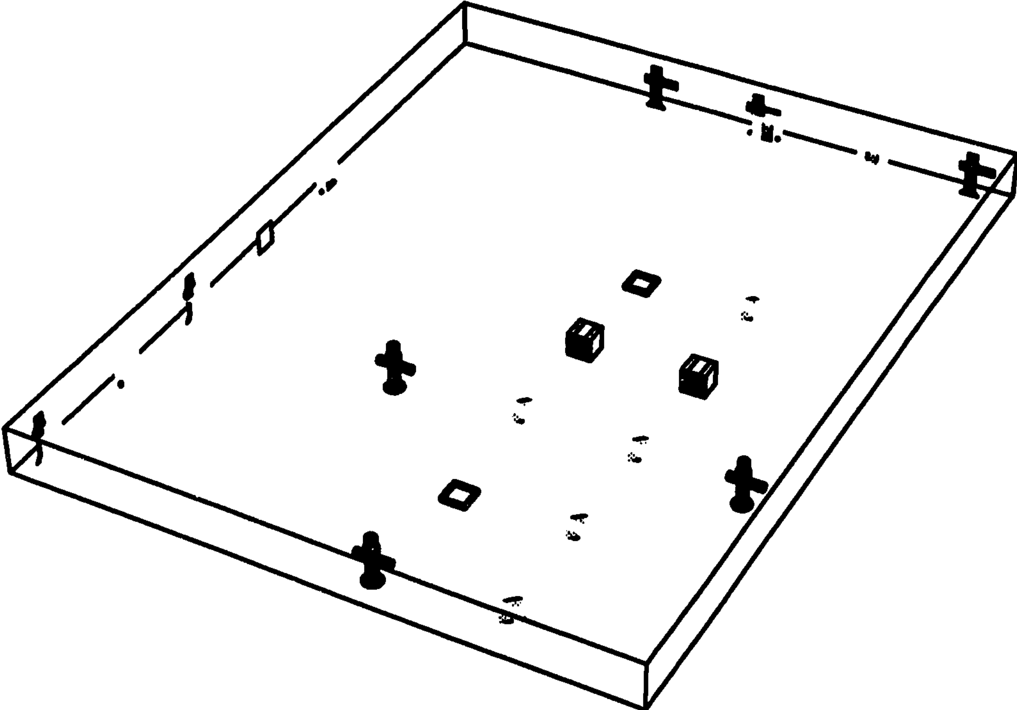


Figure 2.6: Projections of the Architecture Without Orientations

2.7 Generation of a Simulator, Delay Analyzer and Power Analyzer for the Architecture

Instead of simulating the entire architecture using the *OptiCAD* system itself with all its generality, specific purpose code can be generated in a specified target language using **GenerateSimulator[]**. Additional options to it include **Target Language**, **Output File Name**, **Simulator Type**. The interested reader is referred to Parts III and IV for implementation and Part V for a reference of these functions and their options.

2.8 Simulator Execution - Results of Ray Tracing

After the simulator has been generated it can be executed directly. The results of the simulations are given in Figures 2.7 through 2.12. They are the snapshots after distinct events referred to in the figure captions.

2.9 Power and Delay Analysis

After the simulation has been done, a variety of analyses can be obtained. Results of two common types - power and delay analyses have been shown in Figures 2.13 through 2.17 for different optical signal flow paths. The figure captions indicate the path under consideration. The numbers correspond to the component in the architecture.

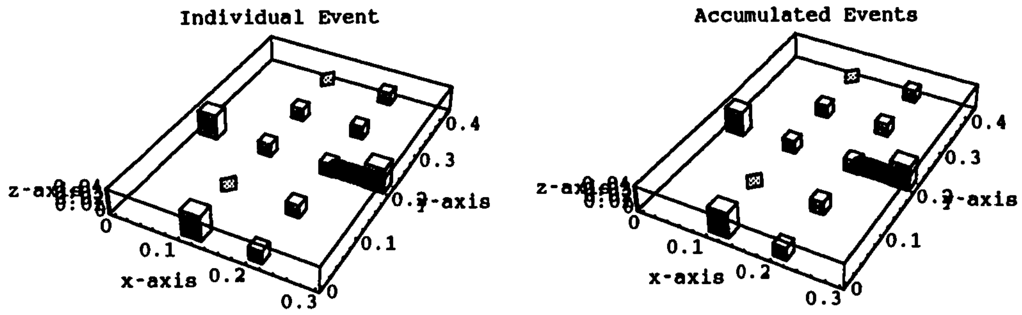


Figure 2.7: Event 1: Laser's Fired Beam Hits the Beam Splitter

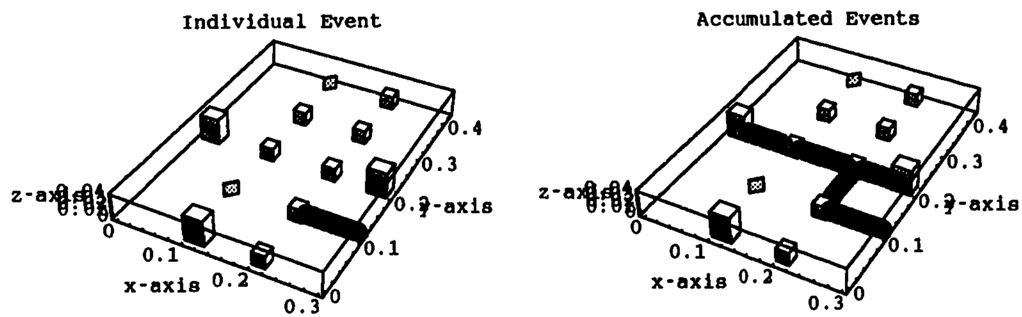


Figure 2.8: Event 5: Beam Splitter's Output Leaves the System

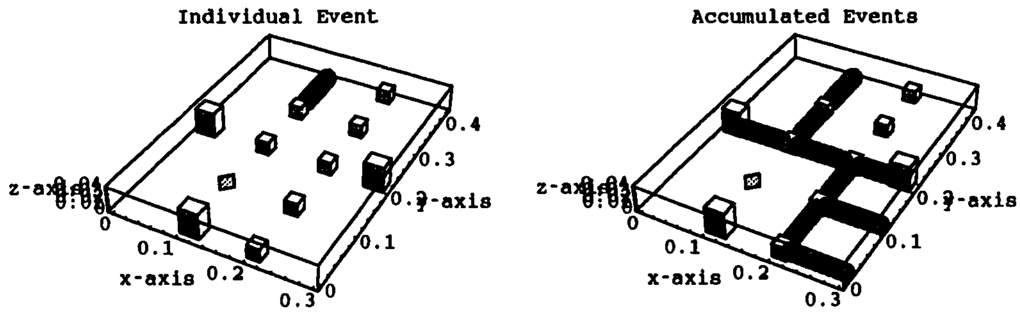


Figure 2.9: Event 10: One Component of the Beam Splitter's Output Strikes the Mirror

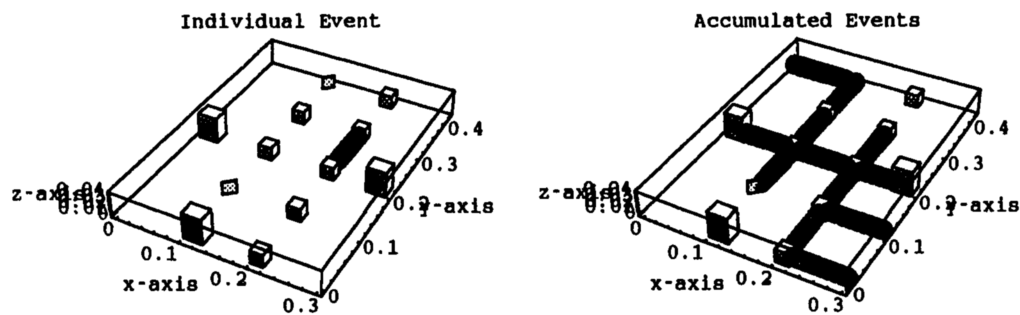


Figure 2.10: Event 15: Beam Splitter's Output Strikes Another Beam Splitter

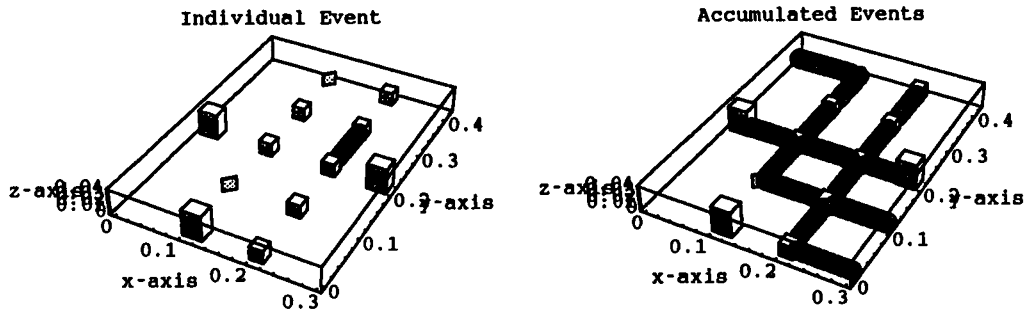


Figure 2.11: Event 20: Beam Splitter's Output Strikes Another Beam Splitter

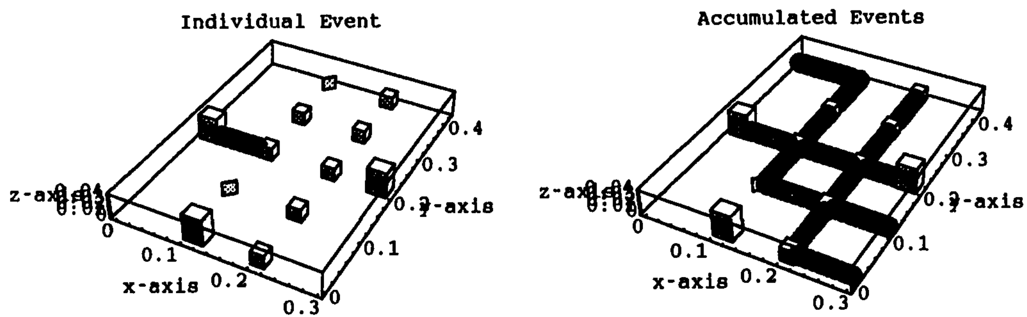


Figure 2.12: Event 25: Laser's Fired Beam Hits the Beam Splitter

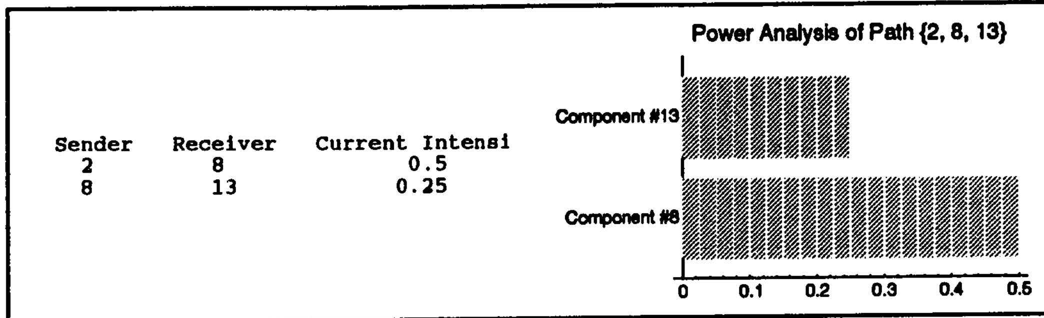


Figure 2.13: Power Decay Along Path {2, 8, 13}

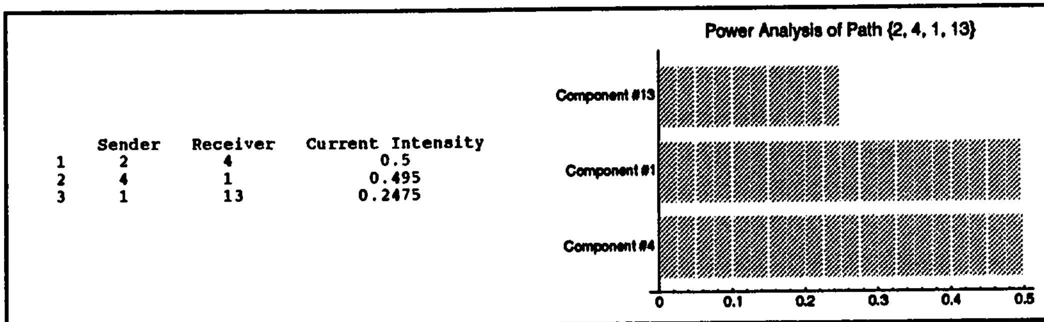


Figure 2.14: Power Decay Along Path {2, 4, 1, 13}

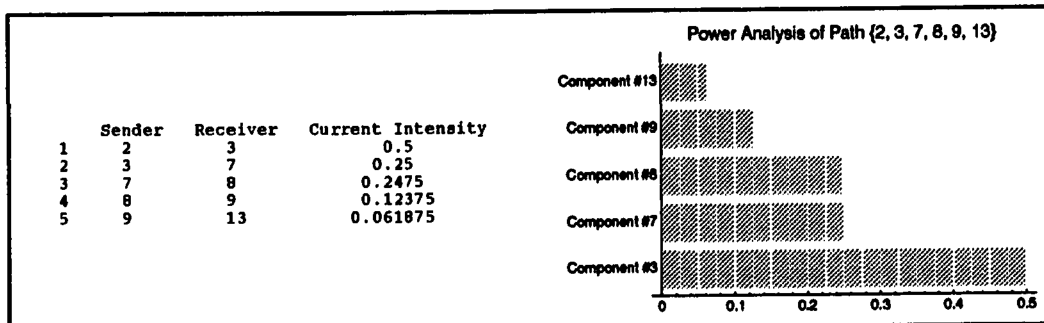
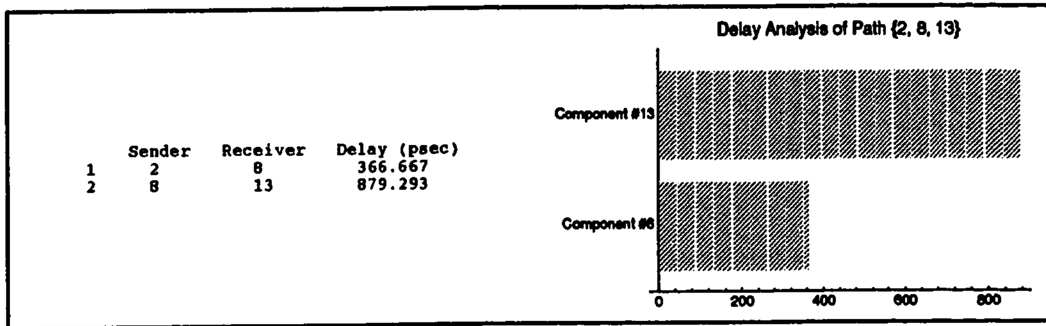
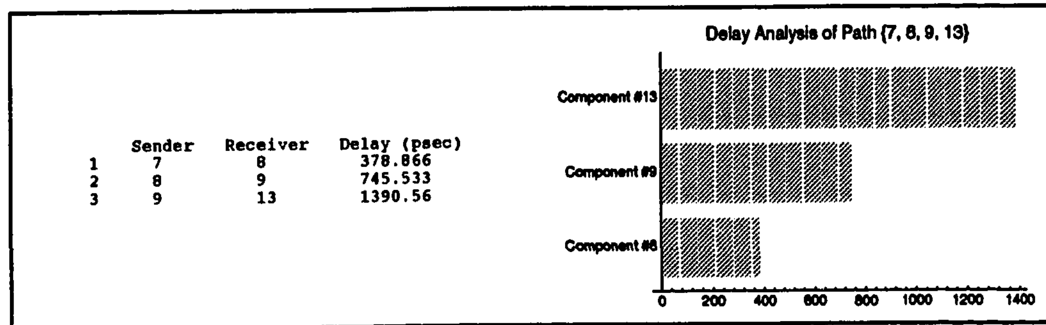


Figure 2.15: Power Decay Along Path {2, 3, 7, 8, 9, 13}



	Sender	Receiver	Delay (psec)
1	2	8	366.667
2	8	13	879.293

Figure 2.16: Accumulated Delay Along Path {2, 8, 13}



	Sender	Receiver	Delay (psec)
1	7	8	378.866
2	8	9	745.533
3	9	13	1390.56

Figure 2.17: Accumulated Delay Along Path {7, 8, 9, 13}

2.10 Documentation and Document Generation

Ideally the flow from specification to analysis is a sequential single pass. In practice however, the designer will have to repeat several steps and at times will have to go back to the previous steps to try alternative approaches. Once the desired solution is reached (simulation and analysis) and the results confirm the correctness of the solution according to specifications, the architect is expected to document various design decisions, alternatives and tradeoffs in the appropriate sections of the architecture template document.

Such a document is translated into a complete typeset document including the problem specification, approach, design, solution to the problem on hand. Chapters in Part II are a consequence of such automated production.

2.11 A Methodology for the Design of Optical Architectures

The steps outlined in the preceding Sections for coming up with a design for the Trip-Flop architecture can be summarized as a methodology. This is strictly from the end user's perspective of the system (see Figure 2.18). Many of the design, domain and language issues are well-hidden at this high level of abstraction. The designer/user/architect generally goes through the following steps in order to describe an architecture.

1. *Problem definition* – Identify the problem and obtain the problem statement.

2. *Specification of Inputs/Outputs* – Identify the number of inputs and outputs. An informal relationship between inputs and outputs is usually known at this stage.

3. *Architectural/Algorithmic solution to the problem* – Design an architecture or algorithm to solve the problem. As far as possible this solution should be independent of the implementation domain.

4. *Encoding of Inputs and Outputs as optical signals*

5. *Identification of optical components and their instantiation*

6. *Spatial constraints – placement and orientation*

7. *Identification of required analyses*

8. *Generating specific simulators/analyzers and their execution*

9. *Interpretation of results*

10. *Verification and Validation of design using results*

11. *Fine tuning the design until the desired architectural/algorithmic solution to the problem on hand is obtained*

12. *Filling in the sections on approach, introduction, etc. and generating a document describing the architecture and its solution.*

2.11.1 The User's Template for Architecture Specification

A standard template has been designed to facilitate the description of architectures (see Figures 2.19 through 2.27). All parts of the template may not be used for each

architecture.

2.12 Summary

The Chapter presented a user's view of the CAD system. The different steps of describing an architecture and analyzing it were presented. Specifically, the problem of designing a tri-state memory unit (Trip-Flop) was considered. The design consisted of the following steps:

1. Providing the complete specifications of the Trip-Flop architecture.
2. Instantiation of components to be used as building blocks in the architecture.
3. Layout of the architecture – placement and orientation issues.
4. Viewing the architecture from different camera positions.
5. Generating a simulator for the architecture.
6. Executing the generated simulator to obtain results of ray tracing.
7. Generation of delay/power analyzers for the architecture and their execution.
8. Power and delay analysis results in the form of graphs.

A methodology for the design of optical architectures was described in detail. A generic user's template for the specification of architectures was presented.

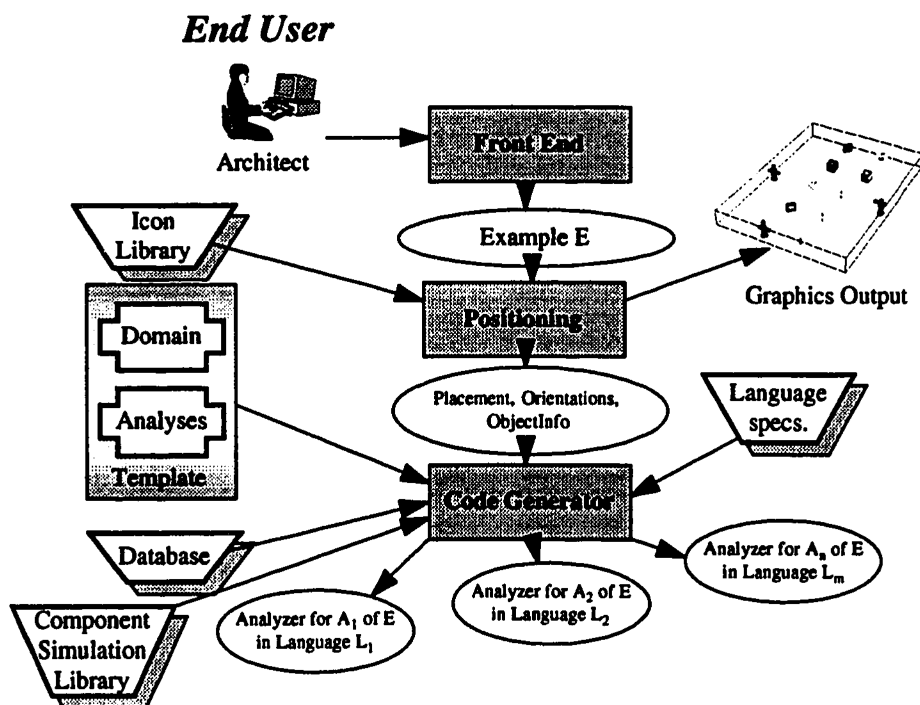


Figure 2.18: User's View of the System

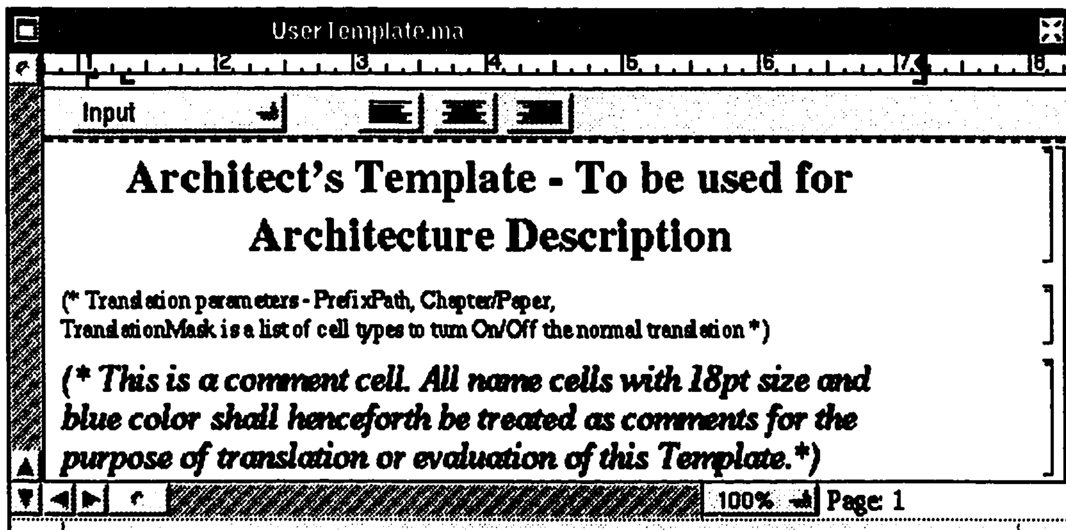


Figure 2.19: User Template

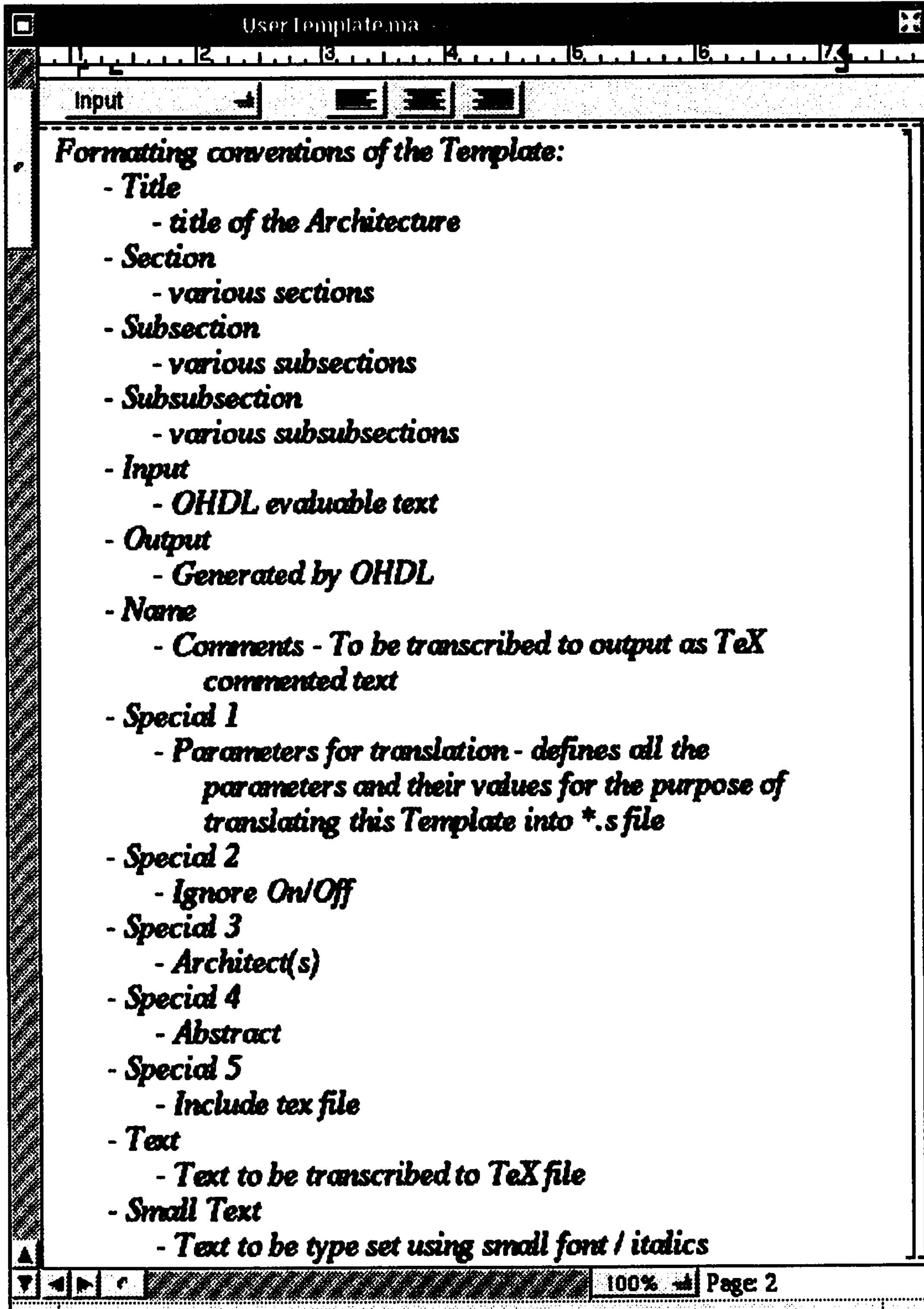


Figure 2.20: User Template (continued)

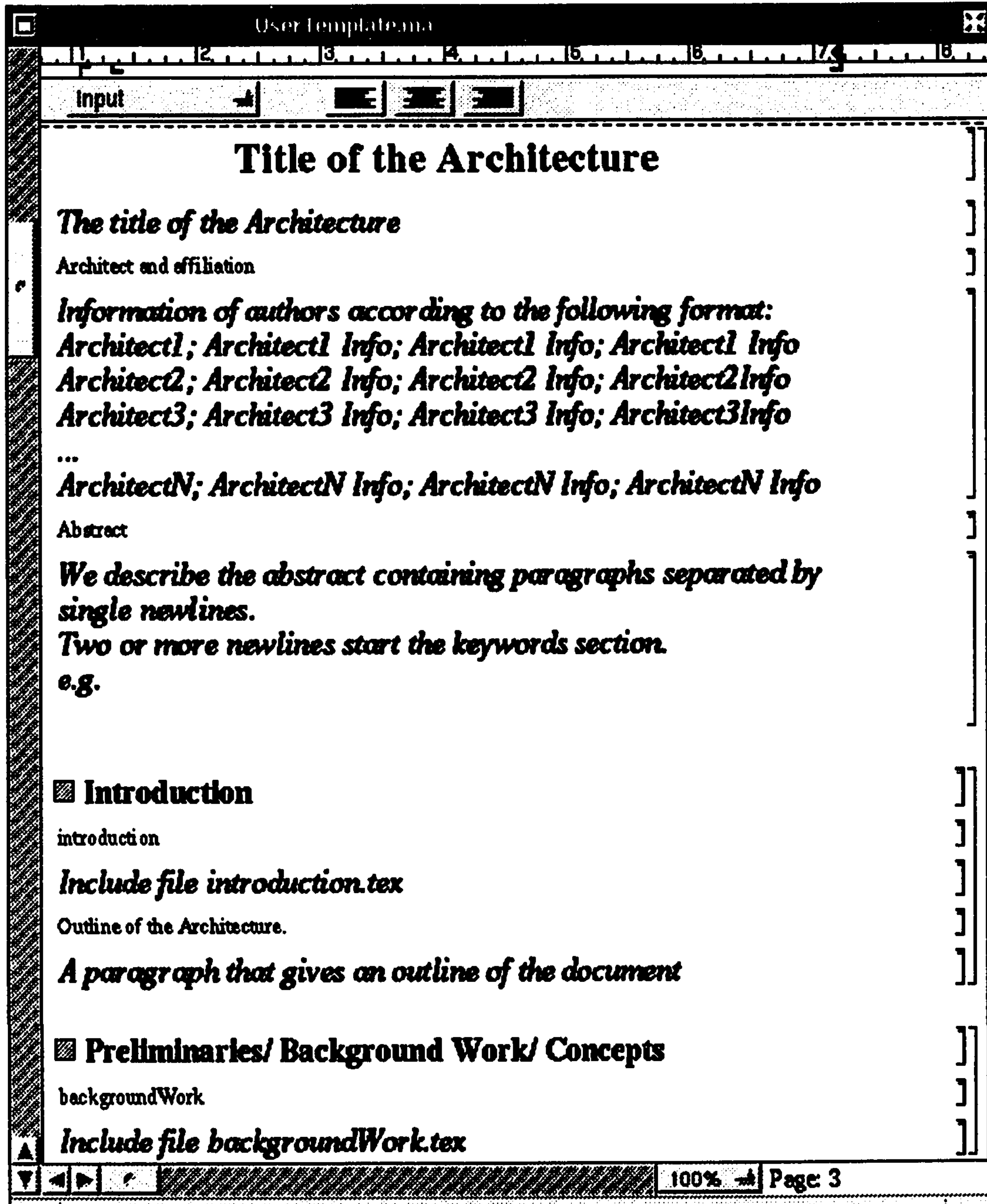


Figure 2.21: User Template (continued)

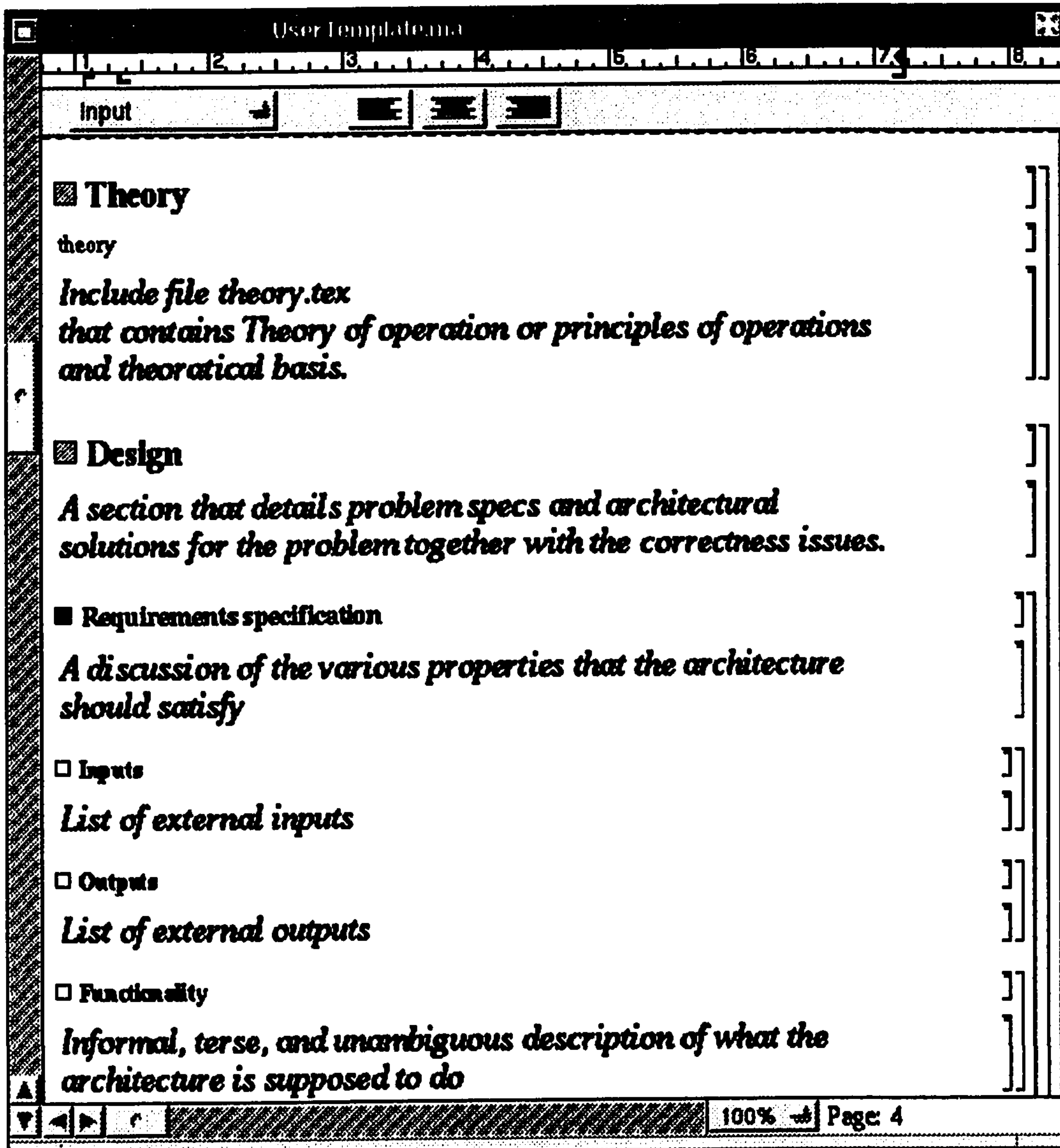


Figure 2.22: User Template (continued)

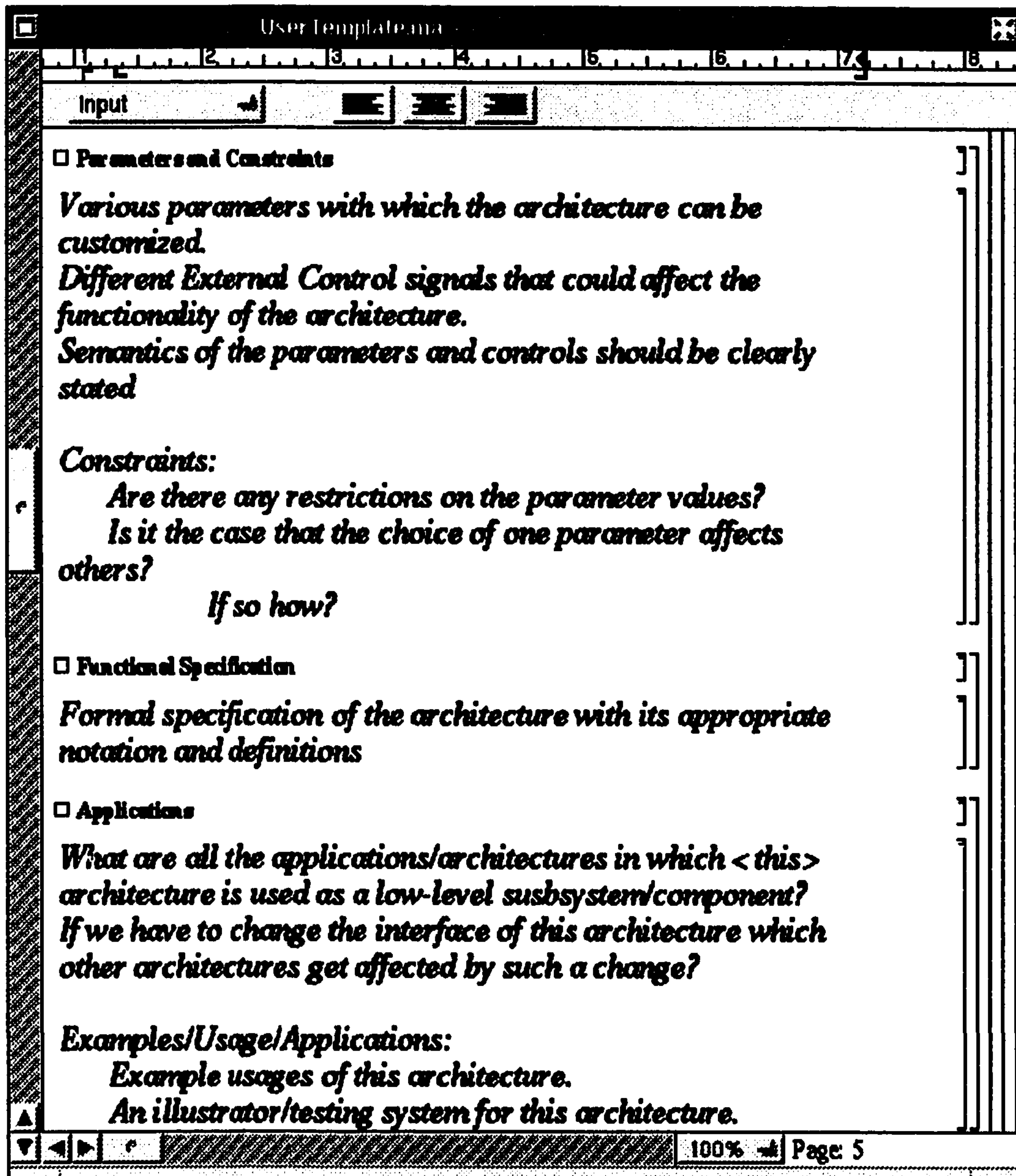


Figure 2.23: User Template (continued)

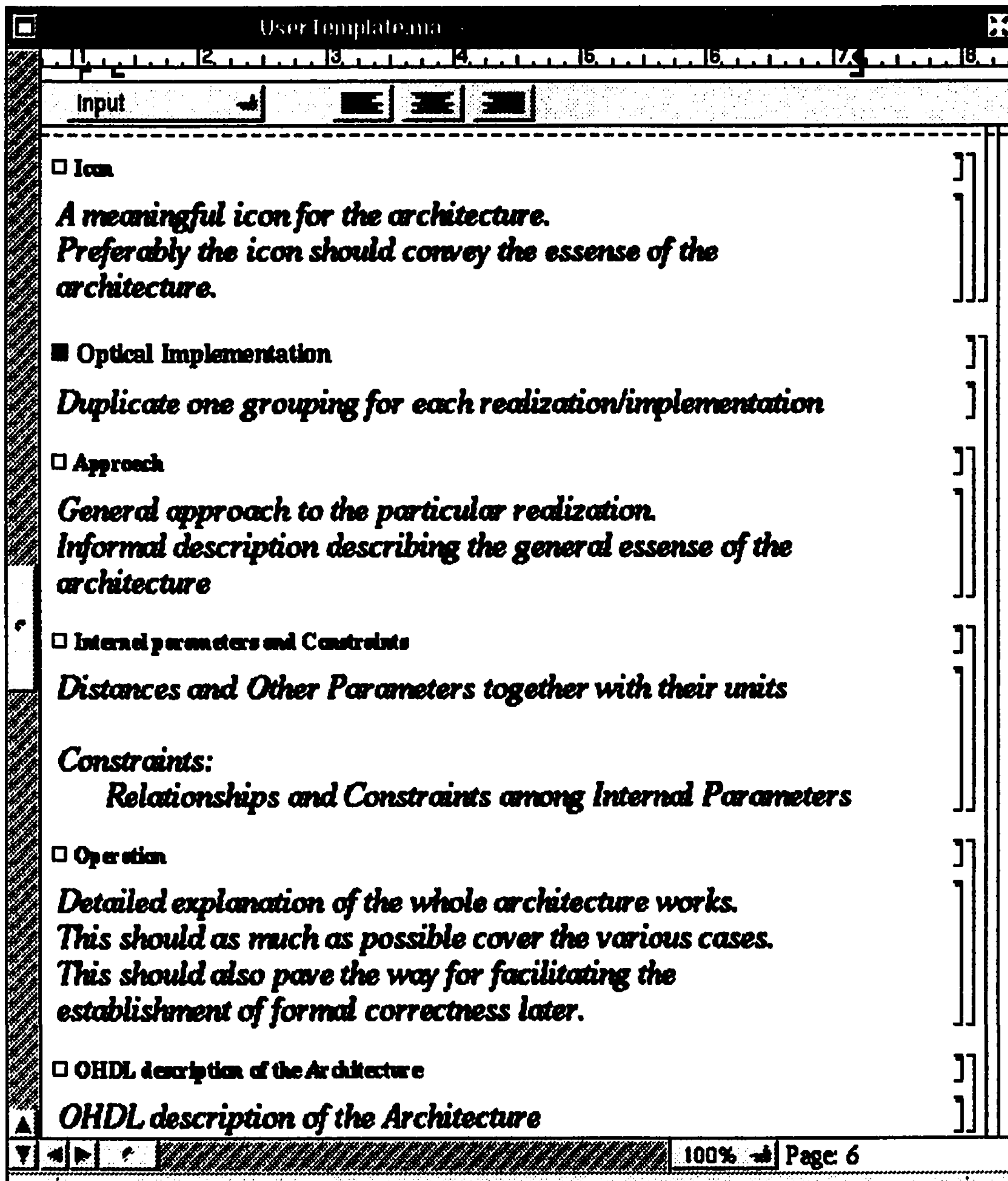


Figure 2.24: User Template (continued)

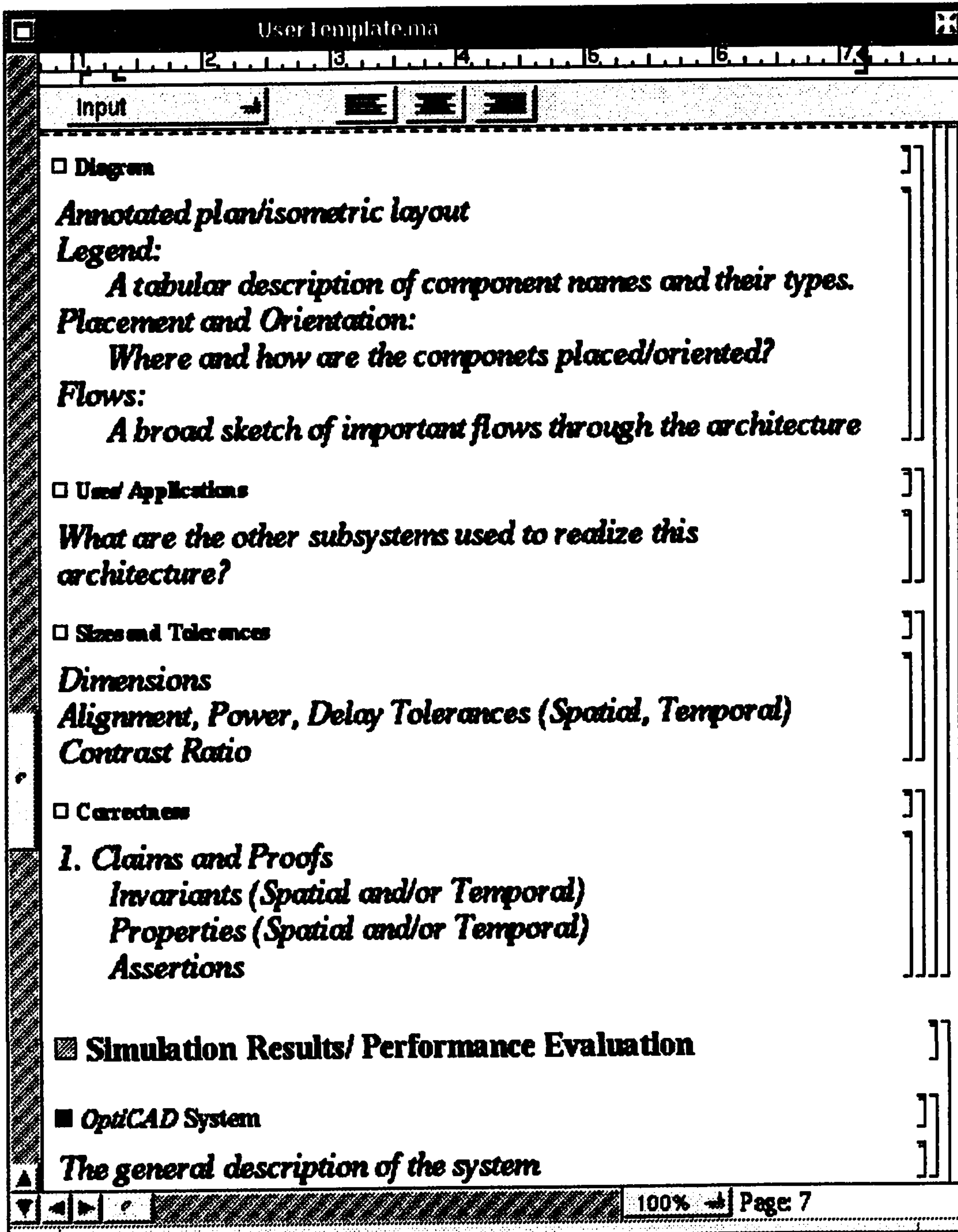


Figure 2.25: User Template (continued)

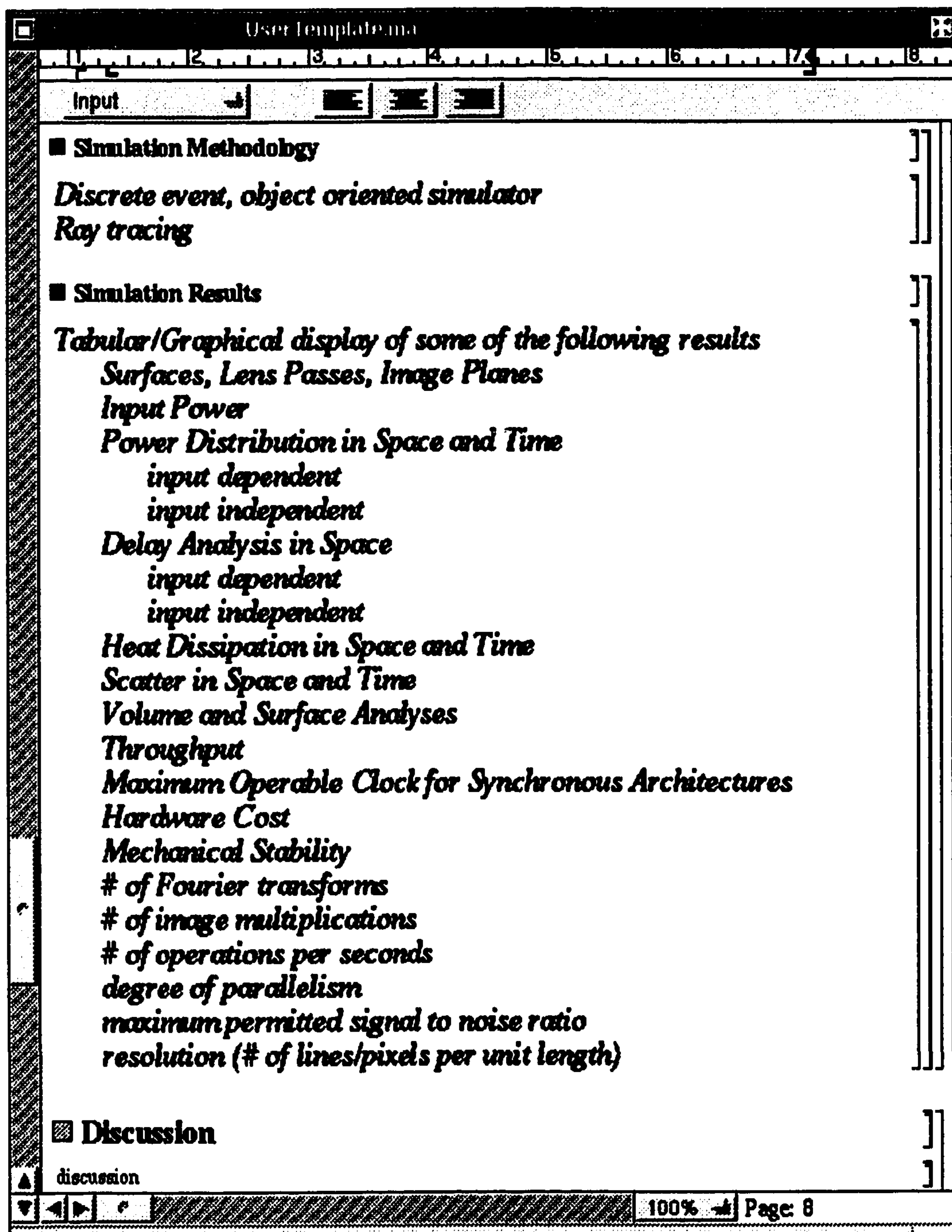


Figure 2.26: User Template (continued)

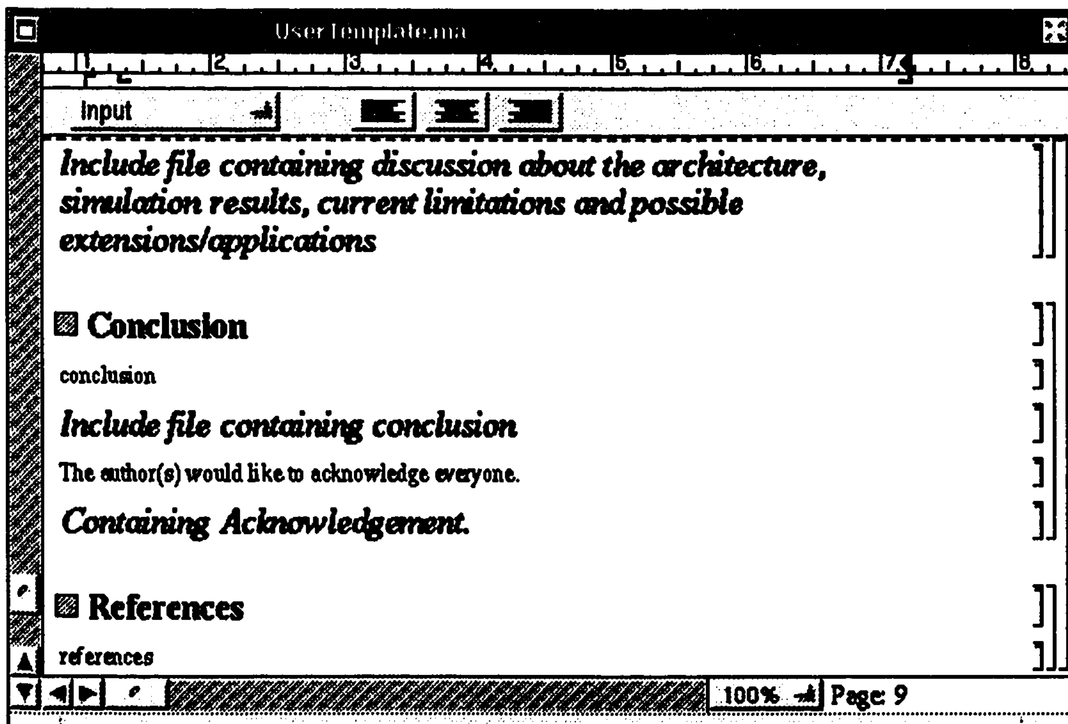


Figure 2.27: User Template (continued)

Chapter 3

Top Level View of OptiCAD

Chapter Abstract

A high level design overview of the OptiCAD system is presented in this Chapter. A development history of the system is given, outlining the roles of different players who interact with it. A brief sketch of the subsystems is provided.

3.1 Introduction

This Chapter presents a bird's eye view of the whole system. Such a broad view is expected to facilitate the comprehension of the detailed subsystems presented in the subsequent chapters and the parts by providing a blue print of the whole system in its current form. A glimpse of the development experience while developing *OptiCAD* is provided by outlining the evolutionary steps that led the system to its current state and the design decisions that were taken in the process.

3.2 Roles

Before implementing any software system, one has to come up with its detailed design. A prerequisite of a good design is familiarity with the problem being addressed in addition to knowledge about proven techniques and the ability to apply them to problem solving.

In order to appreciate the difficulties and problems that could be faced by a user (or architect) of a system, the designers of *OptiCAD* had to go through the experience of actually designing/describing some optical architectures in addition to studying some from the literature. The view of the complete system then was somewhat similar to the one shown in Figure 3.1.

The overall system designer would describe an optical architecture and view the graphical output. The system would then simulate the architecture and present the simulation results. For that purpose a number of architectures were collected from the literature and a few new ones were proposed. The reader is referred to Chapter 6 of [45], or for an updated and contemporary view, Part II of this thesis. Various prototypes were assembled and given to users who were not familiar with the *OptiCAD* system but were well versed with optical systems design otherwise.

3.2.1 User's/Architect's View

The feedback received through observation and comments allowed the formulation of user's requirements. These could be summarized in the following points.

- *Confirming to optics concepts and terminology - as known in the field of optical computing.*

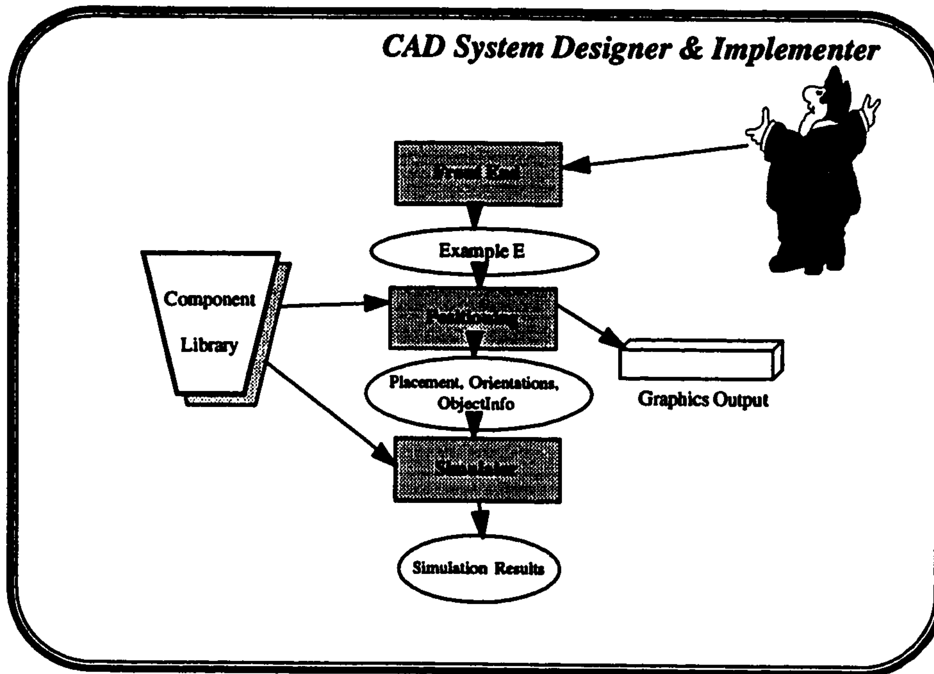


Figure 3.1: The Overall System View – at Requirement Analysis Time

The architect should not be burdened with the learning of yet another programming language. Hence the interface given to the architect should be simple to use. However, it should be general enough to allow description of arbitrary architectures.

- *Separation of the architecture description from the internal implementation details.*

  It is necessary to keep the user's view simple. No internal details should be visible at the user's front-end.

- *Extensive support for automating the tedious work.*

  Architecture description can become quite tedious [45]. Special attention is needed to reduce as much of the tedium as possible.

- *Support for vendor databases.*

  Since almost all architectures employ components provided by vendors, there should be support for incorporating vendor supplied component databases.

- *Support for automated 3D-layout generation.*

  Placement specification can be tricky in 3D. There should be sufficient support for automatic placement/layout.

- *Support for 3D-visualization.*

  A good 3D graphics support is necessary for visualization of the architecture. Bundled with routines for projecting on a 2D plane will allow a crude form of verification of placement and orientation.

- ***Support for constraint specification and enforcement.***

  The architect should be able to supply constraints that govern the functionality of the architecture. The system should have extensive support for constraint specification and enforcement. The expressive power should allow specification of a wide variety of constraints.

- ***Support for incremental and interactive development.***

  As optical architectures tend to be bulky, there must be support for incremental and interactive development.

- ***Mechanisms for adding new components to the system.***

  New optical components are emerging in the market every day. The system should provide mechanisms so that these new components can be readily incorporated. It should also allow for carrying out simulation on a wide variety of systems.

In addition to the above requirements, the following need to be hidden from the architect.

- The inner workings of the system including data structures.

- Detailed algorithms for placement, simulation, geometry, and component models.

Now the role of the user/architect became clearer (see Figure 3.2). The user would be shielded from the details of the system and will be given a clean interface to interact with $\mathcal{OHDL}$.

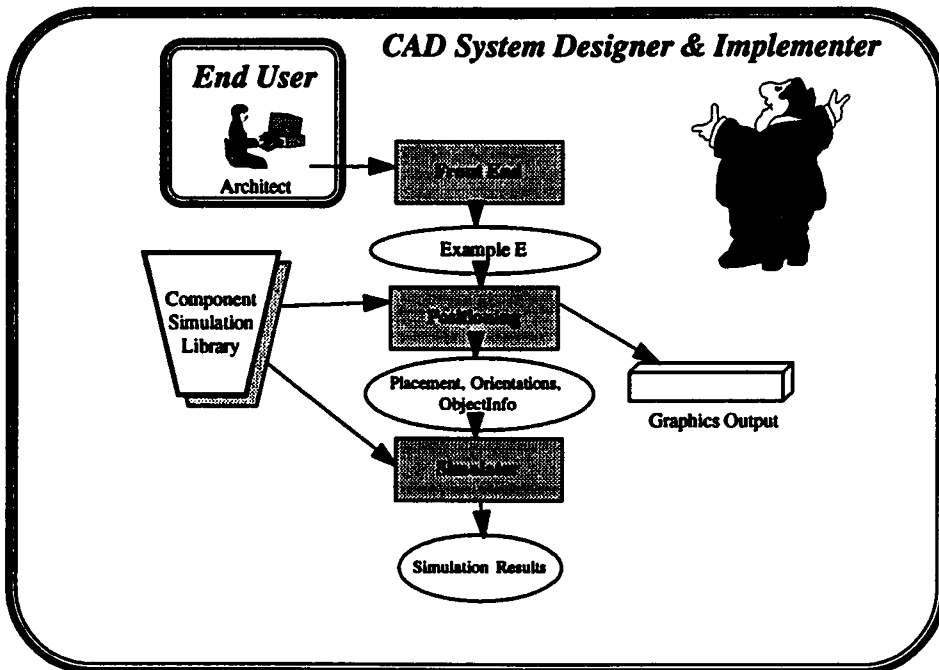


Figure 3.2: The $\mathcal{OHDL}$ View After Separating the Role of the User.

3.2.2 Overall System

As a prototype was being developed[1], it became clear that more information was needed about the way optics works or various components are supposed to work. In addition, a precise mathematical description of the geometry of the components is needed coupled with the knowledge of 3D graphics. A proper understanding of modelling of components' behavior in an algorithmic manner was required. It was a daunting task as it is difficult to find people who are simultaneously experts in modelling techniques, physics, mathematics and algorithms.

The role of a domain expert emerged who was to be primarily concerned with the issue of expressing his domain expertise (in this case expertise of optics, optical phenomena, components and the way they work and should work) in developing readily usable simulator models for optical components, optical systems and analysis techniques (see Figure 3.3).

The role of an overall system architect has also been sharpened by the factorization and elimination of the need to know about the domain expertise. What is now needed on the part of *OptiCAD* system architect is that of defining a reasonable set of utilities and provide those as mechanisms for ready use by the domain expert to augment the system with the domain specific models and knowledge.

3.2.3 Role of a Domain Expert

The domain expert's job is to express the comprehension of how optics works by developing component models, and analysis techniques at the component and system

[1]This took several forms and several attempts were made to implement it in various platforms like Lex/Yacc [45] and RenderMan [3] prior to the current implementation platform – *Mathematica*

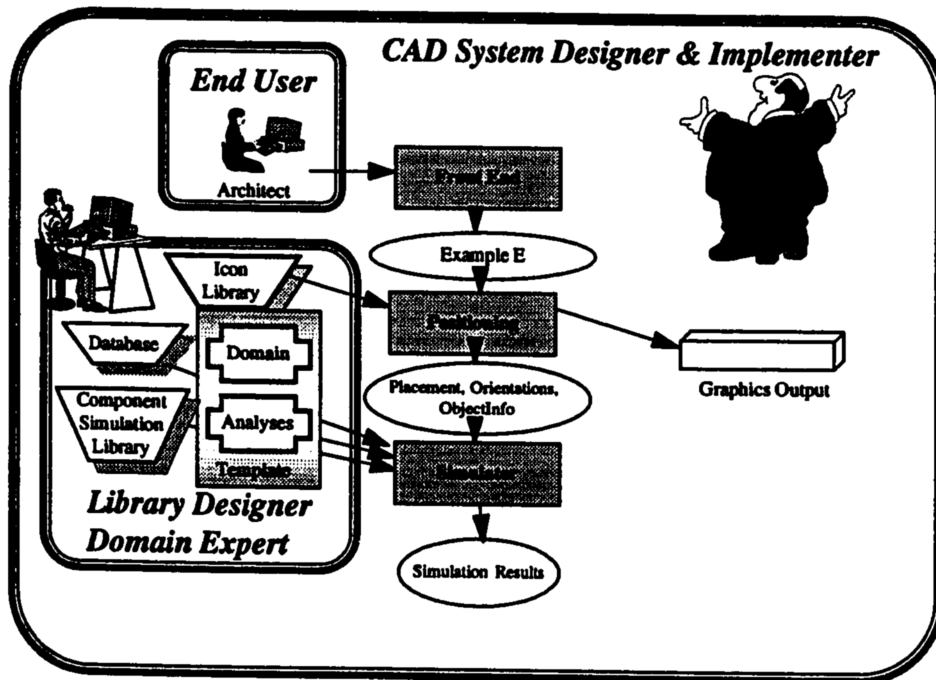


Figure 3.3: The Overall System View after Carving Out the Role of the Domain Expert.

level.

The domain expert's task is made easier by providing him with high-level mechanisms for creating geometric shapes, graphical modelling, creating and populating databases, component modelling and different analyses.

An optics domain expert complements the *OptiCAD* system with:

- a generic library of component models (both structural and behavioral).
- vendor specific databases for components.
- variety of analysis schemes for optical components and systems.

3.2.4 Custom Simulators/Analyzers

As optical architectures tend to be large i.e., they contain numerous components with complicated models, it becomes impractical to carry out simulation and analysis in a high-level language provided by *Mathematica*. An alternative is to develop the whole system in a lower level language (like C). However, this approach would not allow the use of the excellent prototyping/development environment offered by *Mathematica*. Moreover, even when using a low level language, it becomes impossible to perform simulation while carrying all the extra baggage of generality. This generality is however needed to simulate a wide variety of optical architectures.

A need was felt for writing architecture specific simulators in a low level compiled language (such as C) that would execute faster. This is now possible for every architecture. The requirements have now changed so that from the high-level architecture specifications supplied by the user, the *OptiCAD* system should generate a simulator. Such a simulator would contain information relevant to that particular architecture.

It should also be possible to generate this simulator in a wide variety of languages. This would increase the portability of the generated simulators.

Now the goal was to generate instance specific code in a number of target languages. This amounts to expertise in developing code generators for a number of languages which requires a thorough knowledge of the semantics of programming languages. The particular expertise required on the part of the *OptiCAD* system architect, prompted carving out another role for a language expert (see Figure 3.4).

3.2.5 Role of a Language Expert

Code-(as well as Tool-, Simulator-, and Analyzer-) generators need to consult a generic template and the instance specific information in order to generate a custom tool in a requested target language. The need for such a tool obviates the necessity to capture the semantics of a number of target languages, so that the desired custom made tool/ analyzer can be synthesized in the specified target language. The role of the language expert then is to codify the semantics of various target languages so as to extend the functionality of the *OptiCAD* system to be able to produce analyzers in the desired target languages.

To facilitate the task of semantics description, the following can be provided:

- A meta-language for describing the semantics of target languages.
- A representational language, **R**, in which the instance specific tool is internally represented in the *OptiCAD* system.
- Mechanisms that simplify the task of providing translational schemes from **R** to a certain target language **T**.

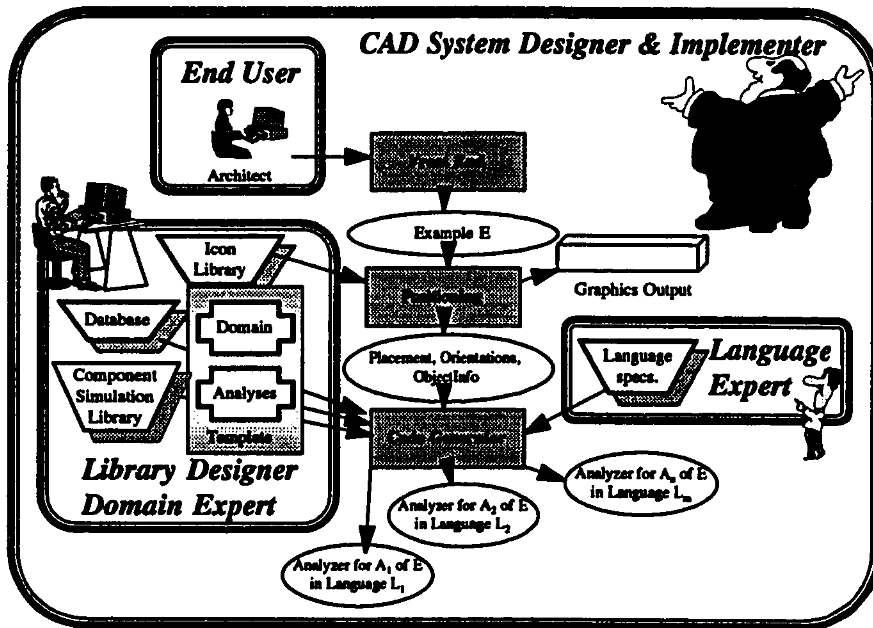


Figure 3.4: The Current Shape of the *OptiCAD* System after the Birth of the Language Expert.

The language expert will also provide the system with a protocol specification that will enable $\mathcal{OHDL}$ to generate distributed code if needed. This protocol specification will indicate all the necessary low-level routines needed to write parallel/distributed applications in that language.

3.2.6 Overall System Architect

The role of the overall system designer and architect is then to clearly define the interfaces expected by the end-user, domain expert, language expert, and other experts, design and realize various representational languages and meta-languages, and provide the necessary mechanisms demanded by various interfaces. Care must be exercised in seeing that an evolving system with diverse contributions from different types of experts will work together complementing each other and providing an extensible system.

The process of creating different roles is expected to continue as the system continues to grow and evolve (see Figure 3.5).

Even at this stage, two or three new subsystems can be seen emerging.

- *Geometric designer.*

  This role in expected to fulfil all the geometry needs of the system. Currently these are being provided as part of the general utilities and tools.

- *Layout decision making algorithm designer.*

  The layout generation from the constraint specification is currently done by using equation solving techniques. As more sophisticated techniques are needed, it becomes necessary to separate the role so as not to over burden the overall

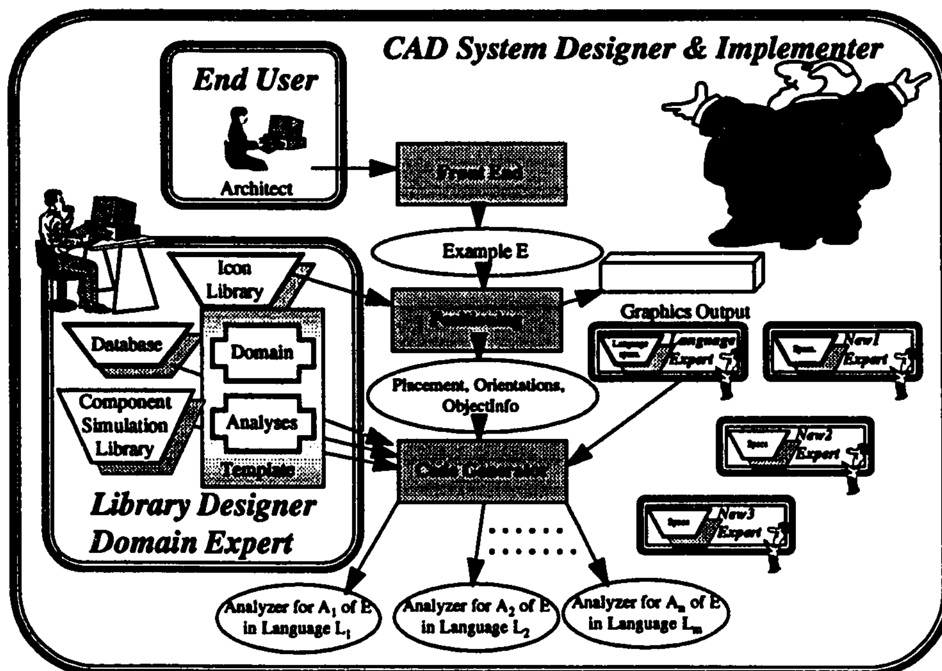


Figure 3.5: The *OptiCAD* System with Additional Emerging Roles.

system architect.

- *Constraint manager.*

  Once full constraint management is incorporated, there will definitely be a need for a separate role which will take care of all the different constraint specification, management and enforcement techniques.

3.3 Overall CAD System

With all the different roles identified, the system's current state is the one shown in Figure 3.4. The user's place is still the same with the front-end shielding the user from the internal details of the *OptiCAD* system. The domain expert is responsible for all domain related issues. A clean interface is provided to him so that he can define all the domain concepts at a relatively high level. The language designer's role is defined as of one providing the semantics of new languages and the related protocols. As of now only these three roles have been carved out from the original system. These were all handled once by the overall system architect.

The system operates in the following manner.

1. The user describes the architecture using the front-end. The user's focus is in exploring design alternatives and coming up with the "best" architectural or algorithmic solution to the problem at hand.

2. The domain expert provides the details of all optics related issues. These include graphical icons for the components, different analyses techniques, and component models. The domain expert need not be well versed with the task of designing optical architectures for specific problems.

3. Using the architecture, the system extracts all the necessary information and produces a 3D graphical layout. This is done after processing all the placement constraints.

4. The components' positioning, orientation information is carried to the code generator.

5. Using the specifications given by the language expert, the system couples them with the architecture specific issues and component models to produce specific analyzers. These analyzers are architecture specific generated in a specified target language. If needed the analyzer can be executed as a distributed application. Such synthesis of parallel/distributed applications is not part of the current implementation.

3.4 Interfaces

This section presents the various interfaces to the different players – the user, domain expert, and language expert.

3.4.1 with User

The user is provided an interface consisting of the following:

- A *Mathematica* style front-end that allows the specification of architectures.

- A very high level language $\mathcal{OHDL}$ that allows specification of constraints, placement of components and directives to view the architecture, generate the architecture, and generate different analyzers.

- An interactive environment that allows the incremental development of architectures. The environment allows the architect to build the architecture in a step by step manner, displaying intermediate results if needed. In case of an error, an erroneous constraint may be replaced.

- A graphical output in the form of postscript. This is handy for viewing components, architectures and simulation results.

3.4.2 with Domain Expert

The primary role of the domain expert is to plug into the system domain related information. The interface provided to him consists of:

- The language $\mathcal{OHDL}$ which he may use to create assemblies and add them to the component libraries. This is same as the one given to the user.

- A major task of the domain expert is to design component libraries. He must be provided with a clean interface using which he can add new components, delete old obsolete components and generally manipulate the libraries.

- Each component must have a shape associated with it for easier visualization of the complete architecture. A complete language for the design and specification of 3D graphics shapes is provided. A rich collection of generic shapes is also given.

- Extensive support for mathematical modelling of components and routines for equation solving facilitates the component modelling task.

- The analysis template creation task is made easier by providing support in the form of a meta-language for splicing.
- A component level simulator language.

3.4.3 with Language Expert

The language expert needs all the tools necessary to describe an arbitrary language's semantics and its related protocols. These are given in the form of:

- a meta-language for the specification of lexical, syntactic, semantic and translational aspects of programming languages.
- a high level language to describe the protocols.

3.4.4 with Outside World Tools

As the system evolves it becomes obvious that there is a need to link with libraries in other languages like C, FORTRAN. Routines are provided that can achieve this.

3.4.5 with the Implementation Platform

To minimize the dependencies on the implementation platform, a layer of utilities is provided. These utilities eliminate the need for system calls by giving the same functionality in the implemented system.

3.4.6 with Databases

As component models become more detailed, and their number increases, and as specifications are provided for additional target languages, there comes a point where *OptiCAD* must rely on commercial database management systems. These systems will maintain all the databases for *OHDL* and will communicate with it to provide results.

3.5 Supporting Utilities/Tools

In order to make *OptiCAD* self contained and portable, many utilities were developed during the design and implementation phases. These fall into the following categories.

3.5.1 *Mathematica* Platform

The whole *OptiCAD* system was developed in *Mathematica*. This choice was made specifically because of the high-level language provided by *Mathematica*. The rich collection of mathematical functions also makes implementation simpler. As the implementation progressed, the full capabilities of *Mathematica* became apparent. It is relatively easy to build higher and higher levels of abstraction by defining very general functions. The development time reduces drastically. Additional features such as graphics support eliminate the need to develop such a support from scratch. The interactive style of the *Mathematica* notebook coupled with the above mentioned features makes *Mathematica* the ideal prototyping environment.

3.5.2 Systems Programming

OptiCAD has extended the built-in capabilities of *Mathematica* by providing additional functionalities such as automatic generation of *Mathematica* packages from notebooks. All these require constant interaction with the underlying operating system. To minimize dependency on the operating system, a layer of routines was developed that mimics some of its functions. Some of them are discussed in later sections.

3.5.3 String Operations

Translation and code generation requires substantial support in the form of string operations. Although *Mathematica* provides the user with a rich collection of string manipulation routines, *OptiCAD* has supplemented these by defining numerous new functions that are more commonly used in code generation and translators.

3.5.4 Databases

Since all libraries are stored as databases, the basic database operations have to be provided on top of *Mathematica* support.

3.5.5 Translators

Automatic generation of *Mathematica* packages from notebooks, code generation and translation from one internal form to another require extensive support in the form of general translation routines. These routines are fairly general in the sense that they can be used to realize many translations relatively easily.

3.5.6 Dependency Analyzers

Dependency analysis is the basic operation in a wide variety of applications. All applications requiring some ordering of events must have dependency analysis to ensure correctness of operation. These include scheduling problems, finding critical paths, generating code, distributed application generation, code synthesis all of which are used by *OptiCAD* in one form or another. These dependency analyzers also find application in determining the context dependencies between automatically generated packages.

3.6 Module Level Description of *OptiCAD*

Figure 3.6 shows the different modules and a crude representation of their dependencies. An explanation of each of these is given below.

Figure 3.7 shows the files that make up the *OptiCAD* system. Theses are organized according to category.

The conventions followed while creating and maintaining these files are as follows.

1. Local variables

    - Abbreviation of a word by omitting its vowels.
    - If long, they are abbreviated by the first character of the word in the phrase.
    - All local variables start with a lowercase character.

2. All other variables (Global, within a package) will be spelled out fully.

    - Internal symbols/variables/functions start with a lowercase character.

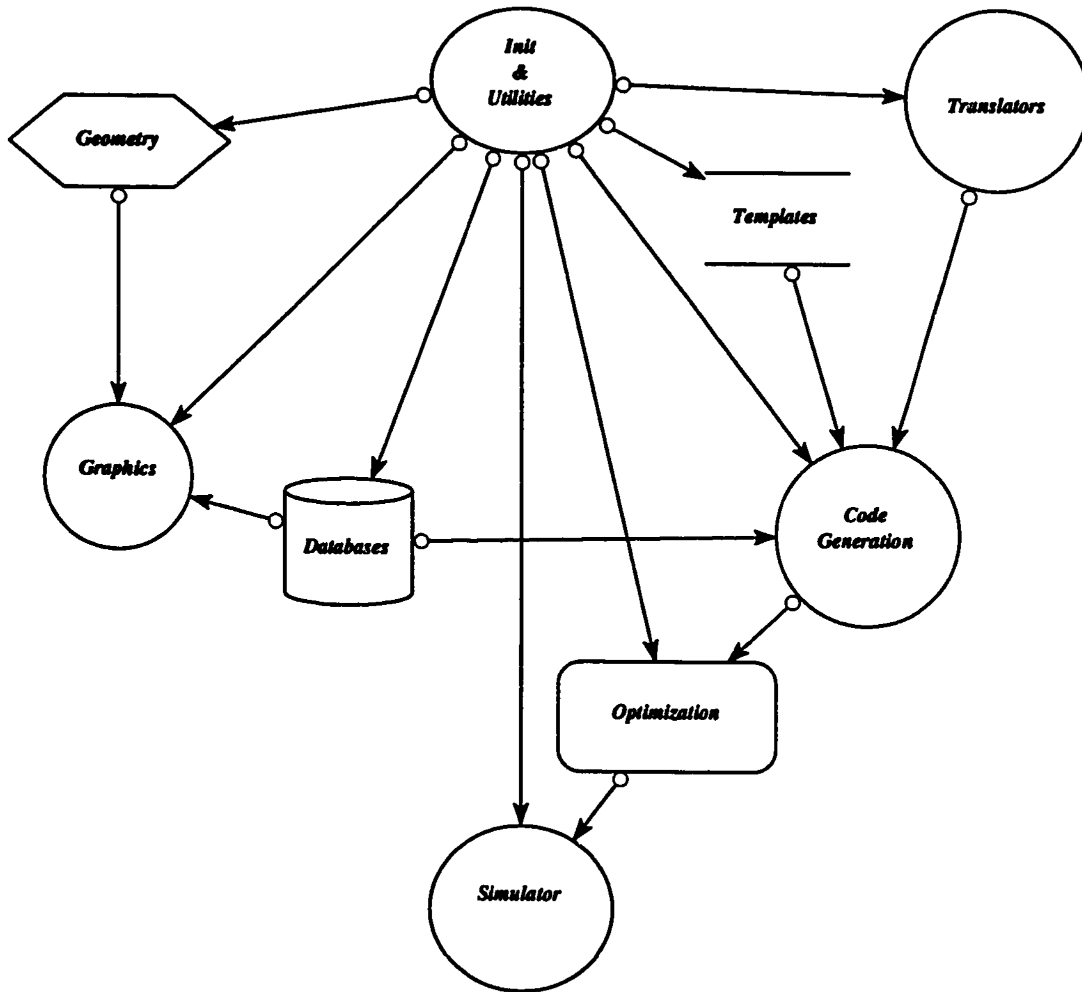


Figure 3.6: The Major Modules in the *OptiCAD* System.

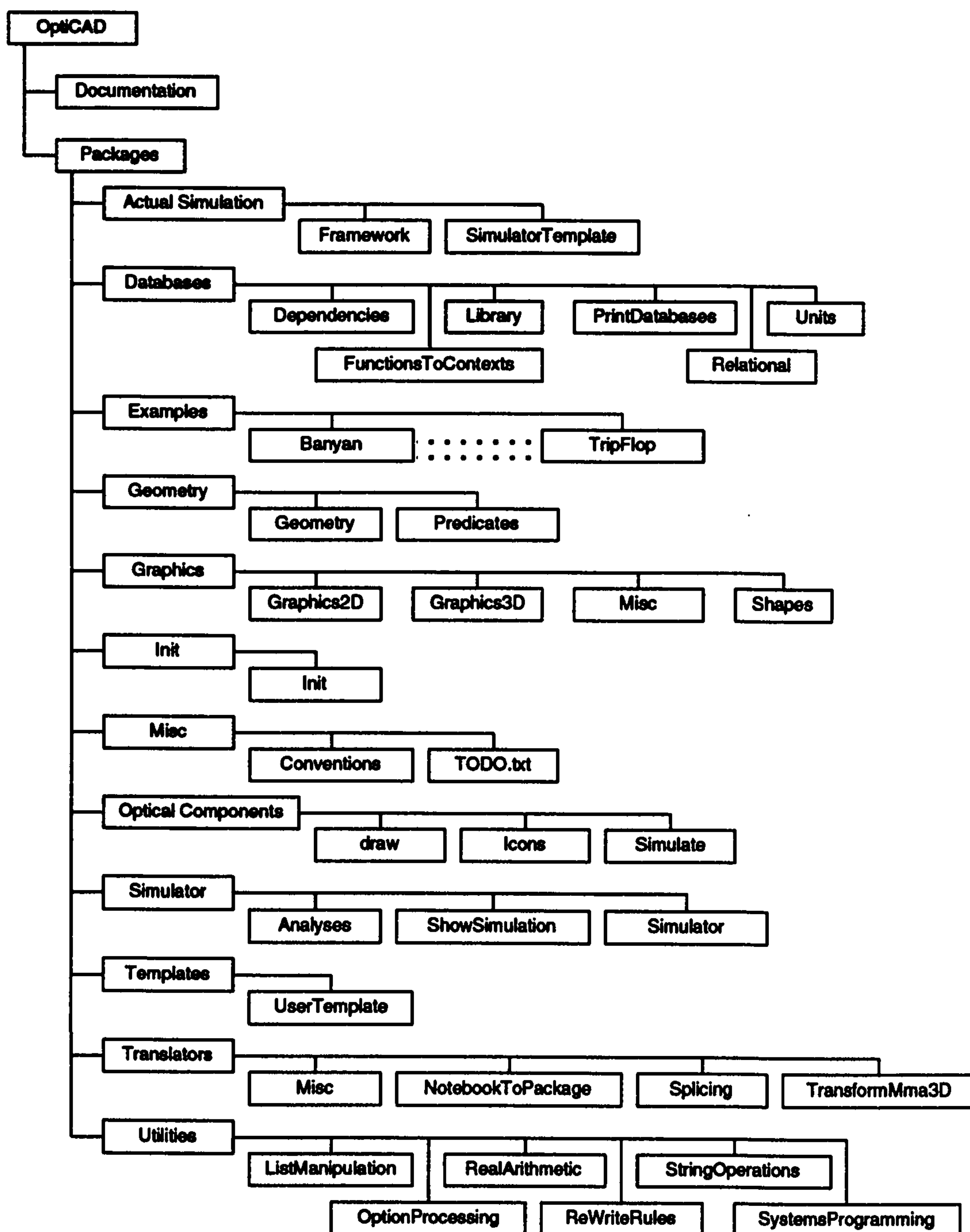


Figure 3.7: The Files that Make Up the *OptiCAD* System.

- exported symbols/variables/functions start with a capital letter.

3. All exported symbols will be protected.

The names to the files are given according to the following conventions:

For a Topic TT in a category CC

1. Context name

   `PrefixCC`TT (Prefix = *OptiCAD*)

2. Examples Notebook

   `CC/TT.Examples.m[ab]`

   End user's perspective – informally explains the functionality and illustrates the usage of the Notebook with examples.

3. Detailed Notebook

   `CC/TT.m[ab]`

   Designer/implementer view – annotated notebook detailing the problem definition, interface, data representation, design decisions/algorithms. It also includes representative examples, TODO list and limitations.

4. Tests Notebook

   `CC/TT.Tests.m[ab]`

   Test cases – rigorous tests that test the boundary conditions of the implementation. There are two types of test suites:

   (a) testing in the context of the loaded packages.

(b) testing in isolation with simulated interfaces to other packages.

In each of these categories, there are further refinements. The first one corresponds to the publicized interface only and all these tests should be passed for the package to be acceptable. The second one which is a super set of the first one can assume the internal representation of internal algorithms to create more tests. Some of these additional tests may also conduct performance comparison among alternative implementations in addition to functionality.

5. Package

   `CC/TT.m`

   Exported machine readable form – Packaged version of the implementation for inclusion in other notebooks/ packages. It is usually stripped of all documentation/examples. It should support different modes such as: DEBUG, TRACE, DUMMY and NORMAL. A function should be provided enabling the change of these modes. Default mode is NORMAL.

   - *DEBUG: displays the function names in the call sequence.*
   - *TRACE: displays the arguments, return values for the call sequence.*
   - *DUMMY: provides a dummy interface to the external environment.*
   - *NORMAL: provides the publicized functionality.*

6. ASCII file

   `CC/TT.notes`

   An ASCII file containing various notes, comments, extensions, documentation, features, limitations and known bugs.

For each category CC

1. Master.m

   `CC/Master.m`

   A package that includes the various topic packages of category CC.

2. Uses/UsedBy relationship file

   `CC/CrossReferences.txt`

   For all external functional references/ dependencies.

3. Readme file

   `CC/README.txt`

   One line description for the files in directory CC.

4. history file

   `CC/history.txt`

   Highlights of design decisions/enhancements/bug fixes etc.

5. To do file

   `CC/TODO.txt`

   List of additional functions/features that can be provided.

Files for the System

1. Readme file

   `README.txt`

   One line description for the categories.

2. history file

   `history.txt`

   Highlights of design decisions/ enhancements/ bug fixes etc.

3. To do file

   `TODO.txt`

   List of additional functions/features that can be provided.

The following sections outline the important files of *OptiCAD* and give a very brief description. The interested reader should consult Parts III and IV of this thesis for detailed code and examples.

3.6.1 Actual Simulation

Framework.ma

This notebook contains the complete code for the simulation of the Trip-Flop architecture. It was created automatically from the simulator template and the Trip-Flop architecture description. Some editing was done later.

Execution of this notebook will create a log file that will record all the events of the simulation. The animation of the simulation may be seen in the notebook Simulator/ShowSimulation.ma.

SimulatorTemplate.ma

This is the general template for a Simulator. The architecture specific code has been left out so that any architecture can be simulated by plugging in the right information. This can be done automatically by executing the code generator

that consults this template. It is self contained in all other aspects. Code has been duplicated from other notebooks to eliminate the problems involved in loading of packages.

3.6.2 Databases

AllFunctions.database

Contains a list of all the functions defined in *OptiCAD*. This list is useful in finding the context dependencies.

ContextDependencies.database

This is a table of all the context dependencies in the system. This is the basis for generating the *Mathematica* packages automatically.

Dependencies.ma

This notebook contains routines to extract dependencies from functions. These dependencies are later used (among other things) to find context dependencies in the generated packages.

FunctionDependencies.database

Gives a comprehensive list of the dependencies among different functions.

FunctionsToContexts.ma

This notebook assumes the availability of a raw functional dependency file. It processes this information to produce a context dependency file. This context dependency file is later used to generate **Needs[]** information in the generated packages.

FunctionUsage.database

Gives a usage message for each function implemented in the system. This is a good online reference manual for the system.

Library.ma

This notebook contains the two databases – one is the generic database which contains the attributes related to a component. The other database (vendor provided) contains fine tuned attributes as provided by the vendor.

PrintDatabases.ma

This notebook presents system related information such as Function Dependencies, Context Dependencies, Usage Messages. The result of this is further processed to produce the reference manual (Part V).

Relational.ma

Contains all the routines related to operations on relational databases.

Units.ma

This notebook contains all global units, constants and values used in the user interface. It should be loaded before loading the front end.

3.6.3 Examples

These are the example architecture description notebooks. More details are presented in Part II.

3.6.4 Geometry

Geometry.ma

This notebook contains the routines necessary for geometrical computations. They are extensively used by **Graphics3D**, **Simulation** and several other notebooks.

Predicates.ma

This notebook contains the necessary predicates for geometry. These are not only used by Geometry.ma but by several other notebooks.

3.6.5 Graphics

Graphics2D.ma

This notebook contains the necessary routines to display, position and orient graphics objects in 2D. Numerous operations are provided such as:

- translateBy[point[{m, n}]]
- rotateAround[angle[theta]]
- rotateAround[point[{m, n}], angle[theta]]
- rotateBy[angle[theta]]
- reflectThrough[line[point[{x1, y1}], point[{x2, y2}]]]
- scaleLocal[{xScale, yScale}]
- scaleOverall[scaleFactor]

Graphics3D.ma

This notebook contains the necessary code for 3D graphics operations. The various operations supported are:

- reflectThrough[plane[a, b, c, d]]

- rotateAround[line[point[{x1, y1, z1}], point[{x2, y2, z2}]], angle[theta]]
- translateBy[{l, m, n}]
- reflectThroughXY[ ]
- reflectThroughYZ[ ]
- reflectThroughXZ[ ]
- rotateAroundXBy[angle[theta]]
- rotateAroundYBy[angle[theta]]
- rotateAroundZBy[angle[theta]]
- shear[{b, c, d, f, g, i}]
- scaleOverall[scaleFactor]
- scaleLocal[{xScale, yScale, zScale}]

Misc.ma

This notebook contains miscellaneous routines for graphics. These are used by the other graphics notebooks.

Shapes.ma

This notebook contains the necessary code to realize generic 3D shapes. These can later be used to construct icons for components.

3.6.6 History

This directory contains files related to the evolution of *OptiCAD*. Details are omitted.

3.6.7 Init

Init.ma

This notebook should be loaded immediately after loading the Mathematica kernel. It will load the necessary definition files and declare global variables to be used by *OHDL*.

3.6.8 Misc

Contains all files related to *OptiCAD* but of miscellaneous nature.

3.6.9 Optical Components

BeamSplitterSimulator.maCell

This file contains the mathematical model of the beam splitter as a *Mathematica* front-end cell.

BoundingBoxSimulator.maCell

This file contains the mathematical model of the bounding box as a *Mathematica* front-end cell.

draw.ma

Each optical component has a draw[self] definition that it calls to render a 3D icon. This notebook serves as a database that provides these definitions.

GenericComponentSimulator.maCell

This file contains the mathematical model of a generic component as a *Mathematica* front-end cell.

Icons.ma

This notebook contains the necessary code to create icons for optical components. These are created by using the generic graphics shapes provided by other notebooks. The routines are general and the icons can be customized to a specific size, resolution and orientation.

InterferenceFilterSimulator.maCell

This file contains the mathematical model of the interference filter as a *Mathematica* front-end cell.

LaserSimulator.maCell

This file contains the mathematical model of the laser as a *Mathematica* front-end cell.

MirrorSimulator.maCell

This file contains the mathematical model of the mirror as a *Mathematica* front-end cell.

PolarizingBeamSplitterSimulator.maCell

This file contains the mathematical model of the polarizing beam splitter as a *Mathematica* front-end cell.

PulsedLaserSimulator.maCell

This file contains the mathematical model of the pulsed laser as a *Mathematica* front-end cell.

Simulate.ma

This notebook serves as a database of simulate[self] definitions. There is one entry for each component.

3.6.10 Optimization

This directory contains all notebooks related to optimization. This is related to the implementation of various optimization techniques such as simulated annealing,

genetic programming, etc. These are expected to be of use in optimizing architectures in later stages.

3.6.11 Scratch

All scratch files are kept in this directory.

3.6.12 Simulator

Analyses.ma

This notebook contains the routines for carrying out power and delay analysis on a log file generated by the simulator. This log file corresponds to one simulation of a particular architecture. It first creates a signal flow graph and then produces bar graphs for each simple path.

ShowSimulation.ma

This notebook reads the log file generated by the simulator and displays it graphically in the form of individual events and accumulated events. The architecture information is also read from a file.

Simulator.ma

This is an experimental notebook containing code from various other notebooks. This notebook provides the basic framework which was later used to design the simulator template.

3.6.13 Templates

Contains all templates such as simulator template and analysis template.

3.6.14 Translators

Misc.ma

This notebook contains translators generally used but not easy to classify.

NotebookToPackage.ma

This notebook in conjunction with many of the utility files, takes a *Mathematica* notebook and generates the corresponding *Mathematica* Package. It removes all the irrelevant information such as example sessions. In a more general setting it can be used to carry out a number of general translations. The notebook also contains various illustrative examples.

ArchitectureToChapter.ma

This notebook in conjunction with many of the utility files, takes an $\mathcal{OHDL}$ description of an architecture and generates the corresponding LaTeXdocument. This was used extensively to generate Part II of this thesis.

Splicing.ma

This notebook contains all the routines related to code generation based on templates. It is used by *OptiCAD* to generate analyzers and simulators.

TransformMma3D.ma

Contains generic routines for carrying out different transformations on 3D-graphics objects in general and *Mathematica* graphics objects in particular.

3.6.15 Utilities

ListManipulation.ma

A collection of list manipulation operations extending those of *Mathematica*.

OptionProcessing.ma

Contains many useful functions related to manipulating *Mathematica* options.

RealArithmetic.ma

Predicates for the relational operators == , ! = , >, < , <=, for the comparison of finite precision real numbers. These were developed in order to have a controlled complexity over the thorny problem of having to deal with finite precision computations.

ReWriteRules.ma

This notebook contains some of the most widely used functions for processing re-write rules. This was created primarily to minimize the dependence on *Mathematica*'s re-writing capabilities and confine the scope to a small group of functions. These can be reimplemented to reflect any changes if any are made in subsequent versions of *Mathematica*.

StringOperations.ma

Contains several useful functions related to string operations. This rich collection is used extensively for code generating, dependency analysis etc.

SystemsProgramming.ma

Implements some useful utilities as functions similar to those found in Unix. These were implemented to reduce dependency on the underlying operating system.

3.7 Summary

This Chapter presented the top-level design of the *OptiCAD* system. Four different views of *OptiCAD* were presented. These were the:

1. user/architect view,
2. domain expert's view,
3. language expert's view,
4. and the *OptiCAD* designer's view.

The top-level description of *OptiCAD* was given and its interfaces with user, architect, domain expert, language expert, outside world tools, implementation platform and databases was presented.

Chapter 4

The Hardware Description Language

Chapter Abstract

OHDL - the specification language provided by OptiCAD is described. The different constructs are described. Explanation of the different directives is provided.

4.1 Introduction

The user interacts with *OptiCAD* using a hardware description language for optical architectures (*OHDL*) – the specification language. This chapter gives an overview of the features of the language and serves as a first draft of the reference manual for this evolving language - *OHDL*.

The chapter first overviews the requirements for hardware description languages.

It then outlines the principle features of the language $\mathcal{OHDL}$.

4.2 Requirements for HDLs

Since the primary goal is to design a *specification* language, the immediate need is to have a high level language. This will facilitate the specification of the architecture at a relatively high level of abstraction and the architect need not bother about the low level issues related to languages. Such a facility will help the architect to concentrate mainly on the design aspects of the problem and the language will serve as tool for the evolution of the architecture.

In order to restrict too much generality, which tends to make languages bulky, $\mathcal{OHDL}$ should be domain specific. This allows incorporating certain domain related issues into the language itself which makes the end-user perspective of the language both simple, concise and easy-to-use.

Applicative rather than imperative style language is needed. This is because the end-user may not be an expert programmer, he needs to know only about the domain not details of imperative style programming.

Since the language is to be used for describing architectures which are assembled from components (objects), it is natural to allow a flavor of object-oriented paradigm. The architect should be able to instantiate components from libraries. The notion of grouping is necessary to allow creation of assemblies from a set of components.

Practical architectures tend to be bulky i.e., they involve many components. However, most of these architectures can be logically divided into modules which perform well defined tasks. These modules can be separately modeled and tested as individual architectures. The architectures can be instantiated and used in larger assemblies.

This introduces the need for supporting hierarchy. The language should provide mechanisms wherein the architect can describe individual assemblies, test them and add them to higher level component libraries. These components can be used in larger architectures where the low level details can remain hidden. Support for abstraction and encapsulation should be provided by the language.

Prototyping optical architectures requires tricky placement of components and spatial constraints need to be maintained throughout the setup of an experiment. This is one of the most time consuming phases of setting up an experiment involving optical components. Any optical architecture specification language should provide support for the addition and enforcement of spatial constraints. These must include position and orientation constraints. The specification of these constraints should be natural and enough support must be provided in order to realize any type of placement constraint. Extensive error reporting should aid in the verification and enforcement of these constraints.

Apart from placement constraints, the language should provide mechanisms for making assertions at different points in the architecture. These assertions include enforcement of some condition or checking of some attribute of light passing through a point in the architecture. Run time support must be provided to verify and check these assertions.

The description of an architecture will require extensive library support. These libraries will include among other things the optical components and assemblies. The architect should be able to add assemblies and components to these libraries. Multiple libraries should maintain components and assemblies. In order to provide additional flexibility to the architect, he should be able to override any attribute of a component. Accessing an object from the library requires traversal of a hierarchy. The highest

level is the **generic** components library. This is a class of generic optical components. The attributes indicate the generic properties of the components. The next level is the vendor specific library. This contains a collection of databases provided by vendors. Most of the generic attributes of components are fixed here to reflect the vendor provided products. At a higher level are the user-added assemblies. These are architectures, described and tested by the user, added as components in the library. At the lowest level is the facility provided to the user to specify any additional attribute or override any existing attribute. This is needed for custom made components.

The main purpose of designing *OptiCAD* is to aid the architect in designing, testing and verifying an architecture without actually setting up a prototype which could take much longer and could be expensive. This is because actual setup requires addressing complex placement issues which are time consuming. Also to verify the functionality of an entire system, a flat setup is needed. Due of the lack of hierarchy, this could be very difficult and time consuming to debug and verify. Hence *OptiCAD* is primarily designed and implemented to increase the productivity of the architect.

The requirement of this system is quite different from other systems. The described system is not going to be used as a software itself. It is written primarily to explore a problem and its possible solutions. Moreover, the requirements of the problem being solved may not be known in advance. A description may undergo several revisions before being perfected. Hence, what is needed by the system is a strong prototyping environment, not a software development system. The requirements of such a prototyping environment are discussed in [70, 30].

The language should allow the creation of multiple instances of objects/assemblies. Once an architecture has been described, it can be reused in another assembly. This requires the maintenance of libraries, definition of contexts and some mechanism for

switching from one context to another. These are all necessary to support reusability.

4.3 Flow-Chart for the Description of an Architecture

Figure 4.1 shows the steps followed when describing an optical architecture using *OptiCAD*.

The description starts with the identification of the external parameters to the architecture. These are external factors (not inputs) that will change some property of the architecture. Examples are specifying the refractive index of a material, number of components in an array component, resolution of the architecture, mechanical displacement of a component. These will be supplied as parameters at architecture instantiation time. Once an instance has been fixed, these cannot be changed. This is in contrast to inputs to the architecture which are dynamically changing.

The identification of inputs and outputs to the architecture are crucial in describing its functionality.

The architect must identify the components required to assemble the architecture. These components may either be generic optical components, vendor supplied products, pre-designed assemblies or user defined components. In any of those cases the architect must create instances of the components by supplying the necessary parameters or attribute values. If the architecture uses new components which are not available in the *OptiCAD* libraries, then these libraries must be updated by adding the specification of the components. This requires knowledge of the physical properties of the components.

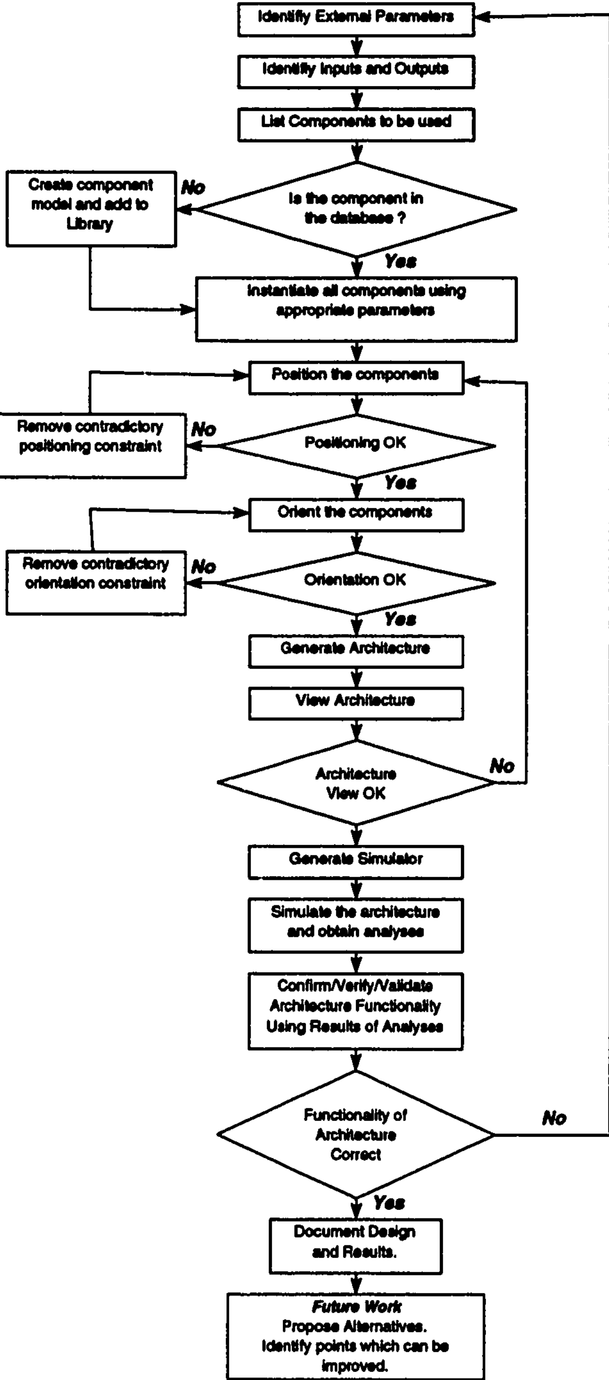


Figure 4.1: Flow-Chart for the Description of an Architecture

The next step is positioning the components in the assembly. The 3D placement freedom offered by optical components makes positioning complicated. This is even more difficult when dealing with large architectures. One approach that has been followed here is specifying the positional relationships as constraints. That is, the positional relationship between two components is given by the architect as a placement constraint. For example the distance between an SLM and a convex lens may be restricted to be equal to the focal length of the lens. This can be given as a placement constraint. This approach has the advantage of restricting the attention of the designer to a local context. However large the architecture may be, these individual relationships can be identified and treated in isolation.

If there are no contradictions in the positioning constraints given by the architect, the design can go to the next step otherwise proper warnings and error messages guide the designer to the source of the problem.

Another important and time consuming phase of the design is the orientation of the components. Optics involves careful orientation of all components. Examples of orientation relationships are (1) laser should hit the mirror at an angle of 30°, (2) the mirror should be aligned so that the beam coming from the interference filter placed at (x_1, y_1, z_1) and oriented at $(\theta_{x1}, \theta_{xy1}, \theta_{p1})$ reflects from the mirror and strikes the SLM placed at (x_2, y_2, z_2) and oriented at $(\theta_{x2}, \theta_{xy2}, \theta_{p2})$. Such relationships and more complicated ones are common in optical architectures. These are again given by the architect as constraints. Once all the orientation constraints have been specified, the exact coordinates and angles of each component can be computed. This phase is complete if there are no contradictions in the specification.

Once all the components are instantiated, positioned and oriented, the setup is complete. The architect can perform any of the following steps:

Generate the Architecture:

A 3D layout of the complete architecture along with the exact placement information is generated. This is done by considering all the positioning constraints.

View the Architecture:

Once the architecture has been generated, it may be viewed from different camera coordinates. Different projections can be obtained to facilitate the task of verifying the alignment. Other layout information such as floor area, volume of the architecture can also be obtained.

Generate a Simulator for the Architecture:

Various analyses on the architecture require generating different simulators and analyzers. Code generation techniques are used to obtain an architecture specific, analysis specific analyzer in a given target language.

Simulate the Architecture:

Rather than generating code for the architecture, *OptiCAD* is capable of simulating the architecture and obtaining various analyses reports.

4.4 Operators

The operators and functions supported by *OHDL* are summarized in Tables 4.1 through 4.5.

4.5 Data Types

OHDL supports numerous data types summarized in Table 4.6.

Operator	Name/Function	Prefix Equivalent
+	Addition	Plus[$expr_1$, $expr_2$]
−	Subtraction	Minus[$expr_1$, $expr_2$]
×	Multiplication	Times[$expr_1$, $expr_2$]
Space	Multiplication	Times[$expr_1$, $expr_2$]
/	Division	Divide[$expr_1$, $expr_2$]
^	Exponentiation	Power[$expr_1$, $expr_2$]

Table 4.1: Arithmetic Operators.

Operator	Name/Function	Prefix Equivalent
==	equal to	Equal[$expr_1$, $expr_2$]
! =	not equal to	UnEqual[$expr_1$, $expr_2$]
<	less than	Less[$expr_1$, $expr_2$]
≤	less than or equal	LessEqual[$expr_1$, $expr_2$]
>	greater than	Greater[$expr_1$, $expr_2$]
≥	greater than or equal	GreaterEqual[$expr_1$, $expr_2$]

Table 4.2: Relational Operators.

Operator	Name/Function	Prefix Equivalent
&&	AND	And[$LogicalExpr_1$, $LogicalExpr_2$]
||	OR	Or[$LogicalExpr_1$, $LogicalExpr_2$]
!	NOT	Not[$LogicalExpr$]

Table 4.3: Logical Operators.

Operator	Name/Function	Prefix Equivalent
=	assign to	Set[lhs, rhs]
{}	list construction	List[$elt_1, elt_2, ..., elt_n$]
− >	rule	Rule[lhs, rhs]

Table 4.4: Miscellaneous Operators.

Operator	**Name/Function**
Sin[], ArcCos[], Exp[], Log[], Mod[]	trigonometric/logarithmic
$expr_1$[args]	application of function $expr_1$
$If[booleanExpr, expr_1, expr_2]$	conditional statement function

Table 4.5: Prefix Operators/Functions.

Data Type	**Permitted Values**
Integers	$\$MinNumber < i < \$MaxNumber$
Real	$\$MinNumber < i < \$MaxNumber$
Complex	x + Iy where x, y are Real numbers
List	$\{elt_1, elt_2, ..., elt_n\}$
Set	$\{elt_1, elt_2, ..., elt_n\}$ where $elt_1 \neq elt_2 \neq ... \neq elt_n$
Array	var = $\{elt_1, elt_2, ..., elt_n\}$
Angle	angle[real number in radians]
Constraint	default[], relativeTo[], sameAs[], parallelTo[], perpendicularTo[], orientationOf[], relativeAngles[], ...
Rule	Rule[lhs, rhs] when applied to an expression replaces each occurrence of lhs by rhs
String	"sequence of characters"
Graphics	Graphics3D[sequence of graphics primitives]
Component	ComponentName[Attributes]

Table 4.6: Supported Data Types

4.6 Constants, Parameters, Variables and Units

Some useful constants have been defined in $\mathcal{OHDL}$. These include "SpeedOfLight", "RefractiveIndexOfGlass". They can be used during the description of an architecture. However, they cannot be modified at the user level.

The described architecture may be sensitive to some external parameters. These are different from inputs to the architecture. These will be supplied as parameters at architecture instantiation time. Once an instance has been fixed, these cannot be changed.

Variables can be used freely in the system as place-holders for values, objects, and sequence of objects. No declaration is needed. Reassignment to variables is possible during the description of the architecture which may cause the variables to change their data type.

Units such as $mm, cm, m, in, degree, radians$ are provided. Values may be given in the form of "30 *degree*", "6 *mm*".

4.7 Expressions

Expressions in $\mathcal{OHDL}$ are given in the format confirming to the grammar given in Table 4.7.

4.8 Assertions

Assertions supplied by the user are necessary in order to verify and validate the functionality of a design. The architect can, at different points in the architecture,

$$
\begin{array}{rcl}
<expr> & \rightarrow & <arith-expr> \\
 & | & <string-expr> \\
 & | & <boolean-expr> \\
<arith-expr> & \rightarrow & <arith-expr> + <term> \\
 & | & <arith-expr> - <term> \\
 & | & <term> \\
<term> & \rightarrow & <term> * <factor> \\
 & | & <term> / <factor> \\
 & | & <factor> \\
<factor> & \rightarrow & (<arith-expr>) \\
 & | & \text{Identifier} \\
 & | & \text{Integer} \\
 & | & \text{Real}
\end{array}
$$

Table 4.7: The Grammar for Expressions in $\mathcal{OHDL}$.

specify values, constraints on attributes of objects. These assertions are verified during simulation and analyses.

An example of an assertion is that in an architecture, the architect claims that for the design to work properly, the intensity of the light beam emerging from the beam splitter should be equal to half the intensity emerging from the laser. Such a constraint is best modeled as an assertion made at a point in the architecture.

4.9 Constructs

The specification given in $\mathcal{OHDL}$ is a set of constructs and directives. The constructs range from simple assignment statements to more powerful control statements.

The assignment falls into two categories. Simple assignment where single values are associated with place holders. These place holders (or simply variables), after the assignment statement, are associated with a data type. All operations allowed by the

data type become legal on the variable. More complex assignments associate one or more variables with structures. Example are assigning an array of components to a single variable, assigning multiple objects to an equal number of variables.

Control structures include sequencing operation. Multiple statements are separated by ";". Conditional statement provides a way of taking one of several alternate paths depending on the outcome of a decision. Repetition may be carried out by the iteration construct.

4.10 Components/Objects and Attributes

The basic building block of an architecture is the optical component. Each component is unique in the sense that it has some unique properties or attributes. Any prototyping system must capture all these properties and present them to the designer. These properties are modeled as attributes of the component.

Some of the common attributes are summarized below.

ComponentType:

The type of component e.g. beam splitter, laser, mirror. This describes the generic class to which the component belongs.

CatalogNumber:

The number assigned to the specific component by the vendors. The component may be referred to by this number. All the attributes as given by the vendor are associated to the component.

Size:

The size of the smallest enclosing bounding box of the component. Each com-

ponent has a bounding box associated with it. This box makes placement, orientation and other simulation issues easier to handle.

Material:

The material with which the component is made e.g. glass, crystal. This also associates other attributes such as refractive index, operating temperature with the component.

RefractiveIndex:

The refractive index of the material of the component. This attribute can also be specified by the architect to override any existing definition provided by the vendor. This is useful in specifying user defined components.

Reflectance:

The percentage of light reflected by the surface of the component. This is needed in carrying out analyses such as power loss.

Transmission:

The percentage of light allowed to pass through the component. This attribute accounts for the intensity of the signal as it reaches the output.

OutputPower:

Associated usually with sources such as lasers, the output power indicates the strength of the beam emitted by the laser.

BeamDiameter:

The diameter of the beam emerging from the laser. This attribute dictates the sizes of the components that receive this light.

BeamDivergence:

The angle at which the beam starts diverging.

PulseDuration:

Optical architectures usually use pulsed lasers because of power dissipation problems. **PulseDuration** specifies the time duration of this pulse.

Angle:

Used for such components that operate with a fixed angle. For example 45° mirrors.

4.11 Component Libraries and Instantiation

Each optical component belongs to a generic class. It inherits all the properties of that class. Some attributes specific to that component could be overridden. For example the class of beam splitters can be divided in a crude fashion into polarizing and non-polarizing beam splitters. Further refinement could include the shape, size, operating temperature. This hierarchy suggests that the library used to maintain these components should also be hierarchical.

The component library of *OptiCAD* is made of multiple levels as shown in Figure 4.2. At the highest level is the generic component database. This contains the most common attributes of the components. The next level is the vendor provided catalog database. This is derived directly from the off-the-shelf components available with specific vendors. The attributes of components are fixed as given in the catalogs. The next level is the user designed components library. This library contains a collection of all architectures that have been designed, tested and stored as assem-

blies to be used in higher level architectures. At architecture description time, some of the attributes of the components can be specified directly by the architect. This corresponds to custom designed components that differ in few critical aspects from off the shelf components.

Instances of components can be created when needed in the architecture. The instantiation process goes through the hierarchy of the libraries (see Figure 4.3).

The two functions provided for component instantiation are

GetObject[ ]

This takes a component type and returns an instance of a component. Additional attributes such as catalog number may be provided at this stage as options to GetObject[]. GetObject also provides additional mechanisms for specifying user defined attributes to the instantiated component.

GetObjects[ ]

This is a generalization of GetObject[]. This function returns instances of similar components. In addition, an array of components with homogenous attributes can also be obtained.

The component type passed as a parameter to GetObject(s)[] above can take the following values.

Generic Class of a Component

These include conventional types corresponding to the component classification. Examples are laser, filter, SLM.

Assertion

This is the assertion made by the architect at a certain point in space in the

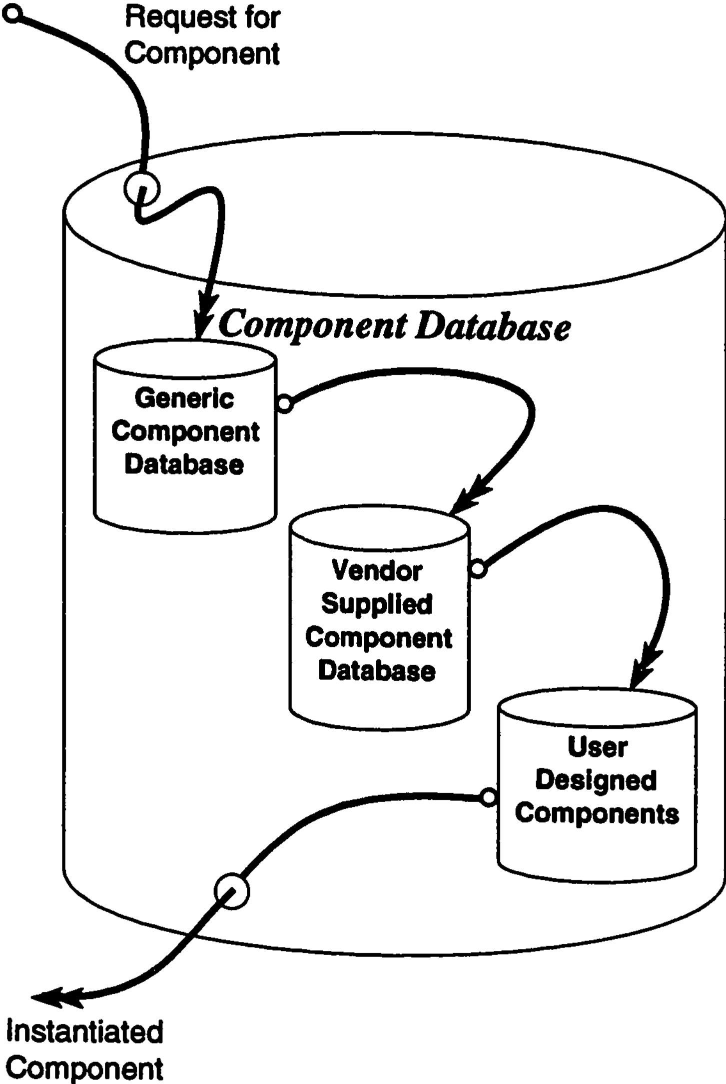


Figure 4.2: The Structure of the *OptiCAD* Library

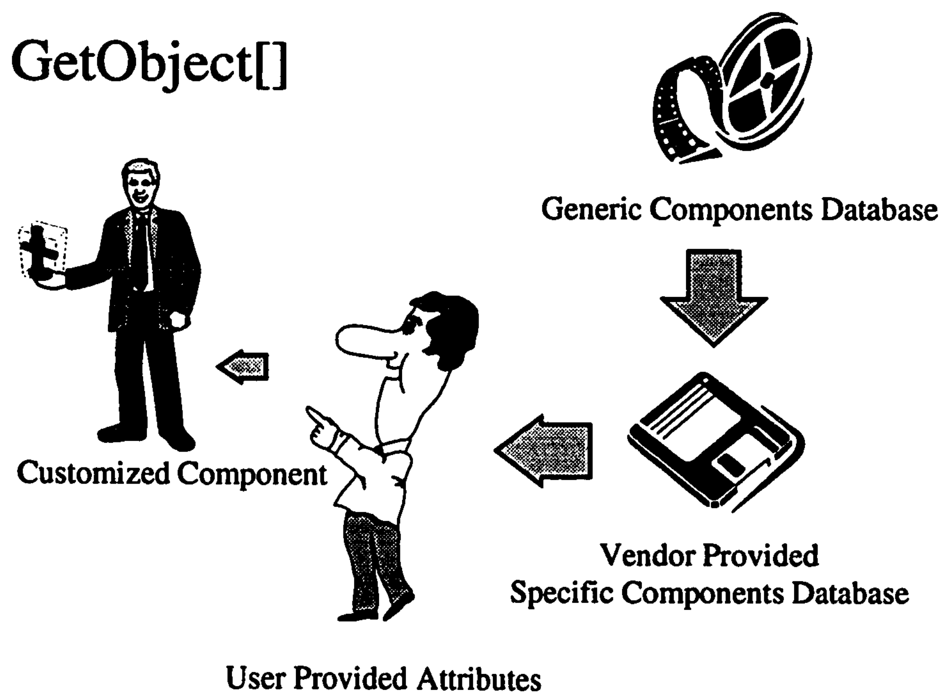


Figure 4.3: The Different Levels of Component Instantiation

architecture. This can be in the form of a boolean function or a predicate. During simulation, this predicate is tested for satisfaction. If at any time it is violated, an error is raised.

Assembly

Such a component type corresponds to an architecture that was described, tested and stored as a reusable component. Since multiple instances of the same assembly can be used in a single architecture, naming conflicts have to be resolved. A unique context name is composed from $< AssemblyPathName, Instance\# >$. All operations carried out on an instance are done by using this context name.

The assembly, being an architecture itself, has its own set of constraints, variables and definitions. These are loaded into the generated context at instantiation time.

At the architecture level, the assemblies are treated as single components. This is done to simplify the description process. During simulation, when a reference in the form of an incident signal is made to an assembly, the whole context of the simulation is changed. The new context is that of the assembly. Simulation is now done inside this assembly. When an output is generated, the context switches back to the higher level and the signal is shown as being emitted from a single component. Further details can be found in section 7.4.5

Associated with each instance of an assembly is a message dispatcher. This dispatcher is used by the architecture to communicate with the assembly. Each message dispatcher knows about the parent dispatcher i.e., the dispatcher of the higher level architecture, and all the components in the assembly. At instantiation time, the dispatcher is made aware of the size of the bounding box of

the assembly. It interprets messages incident to this bounding box, sends them inside the assembly for internal execution and returns the results back to the higher level dispatcher.

4.12 Constraints

One of the most important and time consuming phase of practically building a prototype from a design is the component placement. The components have all the degrees of freedom in 3D space. All architectures require maintenance of spatial relationships in order to realize correct functionality. When several copies of a smaller architecture are being used in a larger assembly, it becomes difficult to maintain exactly the same relationship for every architecture.

The spatial relationships can be specified as constraints between components. $\mathcal{OHDL}$ provides several mechanisms for specifying spatial relationships as constraints.

Placement Constraints:

These are the constraints related to maintaining distances between components. For example, it may be required that a component be x units away from another.

Every component is assumed to have a bounding box. This is the smallest box that can enclose a component. A point (by default one of the corner points) on this bounding box is used for the position constraints.

Position constraints may be specified in any of the following ways.

default[ ]

The architect need not specify the exact coordinates of a component. A component placed at default[] will be assigned coordinates according to

the satisfaction of the placement constraints. The architect may consider this component as placed. Any component may be placed relative to the default[] placed component.

Absolute[ ]

Although not a good practice, the architect may want to hard code the placement of a component. An absolute[] placement requires the specification of the exact coordinates of the component in the architecture.

relativeTo[ ]

A convenient and natural way of specifying a component's position w.r.t. another is by giving the relative distances between them. This allows capturing the position constraints among components in a local context.

Orientation Constraints:

These are about the relationships between the angles of the components. For example, a component should be at an angle of $y°$ with another. Specifying orientation constraints is more difficult than placement constraints. One major difficulty is the formulation of a scheme that can uniquely orient an object in 3D space. The details of this scheme are given in Chapter 5.

Several constraints are provided by $\mathcal{OHDL}$ which make the orientation specification more natural.

orientationOf[ ]

This is used when the orientation of the component needs to be specified exactly in terms of angles with respect to the bounding box of the whole architecture.

parallelTo[ ]

Two components are parallel if they are placed with exactly the same orientation. Once one of the components have been oriented, all the others that are parallel to it can be oriented by using parallelTo[] constraint.

perpendicularTo[ ]

A laser oriented so that its beam strikes a lens face at an angle of 90° is perpendicular to that lens. This can be specified as a perpendicularTo[] constraint.

atAnAngle[ ]

Components that differ in only one angle in their orientation can be fixed by using the atAnAngle[] constraint.

relativeAngles[ ]

This constraint is useful in specifying the difference between the angles of two components.

4.13 Directives

These are commands, statements that trigger an action, change the specification interpretation or generate an output. They differ from the specification of the architecture.

The supported directives are as follows:

1. Directives for specification of placement constraints.

   Centering(On)

   All the position/orientation constraints assume the centeroid of the components as the point of reference.

Centering(Off)

The point of reference for position/orientation constraints is reset to the default (one of the corner points).

2. Directives for triggering computations.

ComputePositions[ ]

Uses the specified position constraints and component instantiations to compute the exact coordinates of each component. Reports any errors for insufficient or infeasible constraints.

ComputeOrientations[ ]

Using the orientation constraints specified, it computes the exact angles of the component with respect to the bounding box of the architecture. Any unspecified orientation is assumed to be default i.e., parallel to the architecture's bounding box. Reports any errors for infeasible constraints.

3. Adding an assembly to the component library.

GenerateAssembly[ ]

Once an architecture has been described, its functionality tested and validated, it can be added to the component database. This directive saves the architecture description as a customizable and loadable package.

4. Interpretation directives.

GenerateArchitecture[ ]

This directive generates the 3D layout of the architecture from the available description and presents it in the form of a graphics object that can be displayed. GenerateArchitecture[] can take the following options.

SurfaceStyle:

The display surface style can be either a **WireFrame** or **Polygon[** *resolution* **]**.

HiddenSurfaceRemoval:

True (default) does not show the hidden surfaces.

False shows the hidden surfaces through the hiding surfaces.

2DView: Projection plane.

Projects the 3D view to a plane and presents as a 2-dimensional view.

3DView: **True** (default) shows the architecture as a 3D-object.

False projects the architecture on the $z = 0$ plane.

VolumeStatistics:

At this stage the sizes, orientations, positions of all the components in the architecture are known. The system can compute the volume of the complete architecture.

False (default) does not compute volume.

True computes volume.

ViewPoint: {x, y, z}.

Shows the architecture as seen from the camera point {x, y, z}.

HardwareCost:

Knowing the cost of all the components, the total hardware cost can be computed.

ViewArchitecture[ ]

GenerateArchitecture[] will generate the architecture as needed. Once an

architecture has been generated, it can be viewed using ViewArchitecture[] with some options/parameters.

ShowArchitecture[ ]

is a composite call to GenerateArchitecture[] followed by ViewArchitecture[]. If the generated architecture is already according to the user's specification, then the generation phase is omitted.

5. Simulation directives.

SimulateArchitecture[ ]

Using the description of the architecture, time of occurrence of external events and triggering of internal events, this directive simulates the entire architecture. The various options used during the simulation are as follows.

TraceSurfaces: True/False (default).

When **True**, keeps track of all the surfaces encountered by a beam travelling through the architecture.

ReportStrayRays: True (default)/**False**.

When **True**, reports any stray rays escaping from the bounding box of the assembly.

TraceFourierPlanes: True/False (default).

When **True**, keeps track of all the fourier planes (e.g. lenses) encountered by a beam during simulation.

CountOperations: True (default)/**False**.

When **True**, counts the number of operations performed during the simulation.

6. Directives for analysing the architecture.

AnalyzePower[ ]

Can analyse the power dissipated on

(a) a given path.

(b) a path of a given length.

(c) all paths.

Further options are provided to restrict the types of paths to be traced. These include the following.

VertexDisjointPathsOnly: True/False (default)

When **True**, traces only the vertex disjoint paths.

SimplePathsOnly: True (default)**/False**

When **True**, keeps track of paths without cycles (simple paths).

SummaryOfStatistics: True (default)**/False**

When **True**, produces a table of the important statistics.

DissipationReports: True/False (default)

When **True**, keeps track of the scattering of light at various points in the architecture.

AnalyzeDelay:

Can analyse the delay on

(a) a given path.

(b) a path of a given length.

(c) all paths

Additional options include

ThroughputAnalysis: True (default)/**False**

When **True**, measures the throughput of the architecture.

MaxOperableClockDetermination: True/False (default)

In synchronous architectures, the clock rate is usually determined from the delay analysis of various paths. The maximum clock rate is usually of interest as it gives an indication of the speed of the architecture. When this option is True, the maximum operable clock is determined.

7. Directives for generating code for the analyses of the architecture.

GenerateLayout[ ]

The 3D Layout description can be generated in a target language such as *postscript* or 3D-script of *Mathematica.*

GenerateSimulator[ ]

Instead of simulating the entire architecture using the system itself with all its generality, code for the architecture specific simulator can be generated in a specified target language.

GeneratePowerAnalyser[ ]

Comprehensive power analysis of an architecture is computationally expensive and can be time-consuming. It is desirable to generate efficient code in a low level language. This code when executed should give the power analysis of the architecture.

GenerateDelayAnalyser[ ]

Similar to power analyzer.

4.14 Support for Abstraction

OptiCAD allows designers to describe architecture, test them and add them to a library of assemblies. These assemblies can then be used in larger architectures as components. An assembly can be saved as a customizable and reloadable package in an assembly hierarchy. This involves providing a mechanism for resolving name conflicts. Multiple instances of a single assembly can be created within a single architecture. At simulation time, each assembly must have a message handler that can take messages from the higher level architecture, provide them to the assembly and return any resulting messages back to the calling architecture.

4.15 Reporting Errors/Warnings

The following error messages/warnings are generated by *OptiCAD*.

1. *Infeasible constraints.*

   The user supplied specifications were either inadequate or inconsistent.

2. *Violated assertions.*

   During the simulation, some assertion about the architecture was violated.

3. *Syntax errors.*

   The specifications were not according to the syntax of *OHDL*.

4. *Missing/Unknown component/assembly.*

   The "unknown component/assembly" error is raised when the component is not found in the system's library. "Missing component/assembly" error occurs

when some constraint/assertion has been made about a component that does not exist.

5. *Overlapping objects.*

   The position/orientation constraints may be such that some components overlap. This is undesirable and is reported.

6. *Escaping signal.*

   During simulation, some signal may escape from the architecture's bounding box. This may be desirable if the signal is intercepted by a higher level assembly and undesirable if it is due to some alignment problems.

4.16 Summary

This Chapter gives details of the hardware description language $\mathcal{OHDL}$ of *OptiCAD*.

It presented a flow-chart outlining the steps to follow when describing an architecture. $\mathcal{OHDL}$ definitions of operators, data types, constants, parameters, variables, units, expressions, assertions and constructs were described in detail. A description of components, their attributes, component libraries and their instantiation was presented. The constraints for placement and orientation of components were described in detail. Supporting directives for the language compiler were described. Support for reporting errors and warnings was discussed.

Chapter 5

Placement

Chapter Abstract

Orientation, placement and alignment of optical components or component assemblies is one of the most difficult phases of setting up a prototype. It is a tedious and error prone task. A substantial amount of effort in the design and verification of a 3D optical architecture goes into the issue of getting a correct layout. The layout of an architecture consists of fixing the positioning and orientation of all its components. This chapter presents an approach wherein the exploration of different placements can be simplified by developing a 3D graphics system.

5.1 Introduction

A hierarchical approach is employed to facilitate the end user's task of describing the design of the architecture. The architecture may be divided into logical parts where each can be designed and tested separately. These parts can then be treated as components, each performing a specified task. This simplifies the design of the

architecture. All the problems of debugging and verifying can be handled in a local context.

This Chapter discusses the importance of constraints in all engineering disciplines in general and in the design of optical architectures in particular. It then reviews the methods of specifying constraints. The methods of finding solutions satisfying a set of simultaneous constraints is discussed next. To simplify description, each component is assumed to have a local context. All details of the component are restricted to that local context. The notion of bounding box for each component and how it handles the local context of the component is discussed followed by a detailed discussion of how positioning and orienting an object can be realized as a constraint satisfaction problem. The specification of graphics (2D and 3D) primitives and operations is provided next. Most of the illustrations are based on the Trip-Flop architecture presented in Chapter 2.

5.2 Constraints

The most natural way to spell out the placement of objects is in the form of constraints. The architect describes only those constraints that dictate the functionality of the architecture. Any placement satisfying all constraints is acceptable.

Constraints are restrictions on parameters/configurations so as to retain the maintenance of desired relationships among them. These constraints may also be viewed as assertions about the desired shape/behavior of an optical component/architecture. The advantage of using constraints is that spatial/integrity relationships can be easily specified.

The following sections talk about the general issues related to constraints. Some

of these deal with the role of constraints in geometric modelling, how to satisfy them, how to verify their satisfaction, and how to enforce their satisfaction. It then presents some techniques for dealing with these issues.

5.2.1 Role of Constraints in Geometric Modelling

Design is the process of making a sequence of decisions from among a number of alternatives. These decisions are made with the aim of configuring a system that is expected to meet a set of requirements. Design activity pervades all engineering fields. That is, design is an integral part of engineering.

The alternatives presented at every stage are usually due to the integrity one has to preserve in the domain under consideration. An example could be the choice of a material from several that exhibit a certain property.

The choices are restricted by fundamental properties and are governed by the laws of the domain. Integrity of the operations and the consistency of the design depends on the satisfaction of the restrictions inherent to the domain.

In the normal course of design in a particular domain, a designer is aware of only the integrity constraints that design has to fulfill. The design process is one of experimenting with numerous combinations of choices until all the integrity constraints of the domain are met. This process of exploring the design space (space of all possible design alternatives) is time consuming.

An alternative design strategy is to capture the semantics of the domain and its integrity rules as constraints and attempt to look for solution by automated techniques. This process is less time consuming to the designer and any solution found will meet the spelled out requirements.

The problem of determining whether a set of constraints has a solution or not is in general undecidable. However, a restricted version of the problem wherein efficient search could be conducted would be of value in design support tools. Such tools capture some of the domain semantics and prune the design space guiding the designer through feasible space.

Hence tools/techniques are needed for the specification of domain constraints as well as their satisfaction for proposed alternative configuration.

5.2.2 Constraint Specification Techniques

There are numerous ways of spelling out constraints.

1. **Mathematical equations relating different parameters/equations.**

   These are commonly used to represent different laws. The equations make use of symbols representing constants and variables. Some common examples are Snell's law, laws of reflection/refraction, Maxwell's equations.

2. **Undirected/Bidirectional labeled flow graphs or other graphical variants like Petri-nets.**

   These are specifications representing cause-and-effect relationships. Availability of some parameters may cause triggering of some computation and the necessary output generated. These specifications are easily visualized as graphs with nodes corresponding to actions/computation and edges representing dependencies. Other graphs such as Petri-nets use specific notation (e.g. places, tokens) to represent additional features.

An example is that of temperature conversion between Centigrade and Fahrenheit. Figure 5.1 shows a flow graph that takes a temperature value as input and returns the converted value in the other scale.

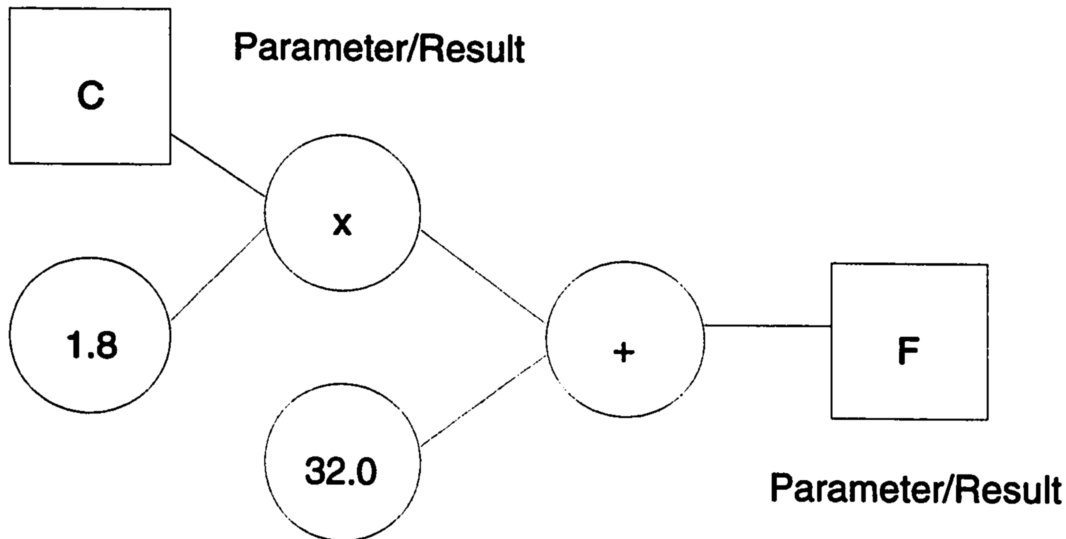


Figure 5.1: A Flow-Graph for Converting Temperature from One Scale to Another

3. **Graphical Hold-Click-and-Specify paradigms.**

   These are graphical specifications and are commonly used in graphical tools such as editors and front-ends. All specifications/relationships among objects are given graphically. The tools that employ these also have mechanisms to enforce the provided specifications.

   Common examples of graphical relationships/constraints are rubber-banding which restricts the operations to a local context, grouping which has the effect of making larger objects from smaller ones, attaching/gluing to preserve the spatial relationship between objects.

4. **Predicate calculus**

   It is a notation from formal logic used to specify complex functions. This notation is particularly useful in proving program correctness. Formal methods make use of techniques wherein a function is specified using predicate calculus and then the program to be tested is analyzed to determine whether the function it computes is in fact the one specified by the predicate calculus formula.

   Predicate calculus is particularly useful in providing high level specification for a function. The paradigm employed is essentially declarative as opposed to descriptive.

   For example two components C_1 and C_2 are to be aligned so that they are in the same plane i.e., their z-coordinates are equal. This constraint can be specified in predicate logic as follows:

   SameZPlane(C_1, C_2) if

   Location(C_1, $\{x_1, y_1, z_1\}$) and

   Location(C_2, $\{x_2, y_2, z_2\}$) and

   Equal(z_1, z_2).

5. **Rules.**

   Instead of specifying how a problem is to be solved, it is convenient to specify the rules allowed and search – preferably through automated techniques – for the solution in the solution space. Starting with an initial assumption that the values of the input variables meet certain constraints, the rules may be used to

deduce the constraints met by the values of other variables after execution of each program statement, until the results are obtained.

For example two components C_1 and C_2 are to be aligned so that they are in the same plane i.e., their z-coordinates are equal. This constraint can be specified by the following rule:

C_1 and C_2 are aligned if

(a) C_1 and C_2 are different components

(b) $\{x_1, y_1, z_1\}$ are the coordinates of C_1, $\{x_2, y_2, z_2\}$ are the coordinates of C_2, and $z_1 = z_2$.

6. **Procedural style.**

This is the imperative programming style of describing the method for finding the solution. In this method, the exact steps of carrying out the computation must be specified. In most cases, this is too detailed and low-level, often forcing the problem specification to be tied down to the implementation. Hence, it loses the global view of the problem.

For example to enforce the alignment of component C_1 to be at displacement $\{a, b, c\}$ relative to C_2, the following procedural approach must be followed.

(a) $x_1 = C_1.x$

(b) $y_1 = C_1.y$

(c) $z_1 = C_1.z$

(d) $x_2 = C_2.x$

(e) $y_2 = C_2.y$

(f) $z_2 = C_2.z$

(g) If (x_1 - x_2 < a) then $x_1 = x_2 +$ a;

(h) If (y_1 - y_2 < b) then $y_1 = y_2 +$ b;

(i) If (z_1 - z_2 < c) then $z_1 = z_2 +$ c;

(j) $C_1.x = x_1$

(k) $C_1.y = y_1$

(l) $C_1.z = z_1$

7. **Informal.**

   In this approach, specification is generally provided in a form close to natural language. It may be easy to specify but could be too ambiguous or too imprecise to execute. This arises from the ambiguities in natural language. A statement may have multiple meanings. Moreover, it is easy to give an incomplete specification. The design methodology, being informal, will not enforce a regular design.

   One example of an informal specification is: *Place the lens at a distance of 2mm from the laser.* This statement does not clearly state the position of either lens or laser. Numerous positions can be obtained from such a specification.

5.2.3 Constraint Satisfaction Techniques

Once a set of constraints has been specified, there are several methods of finding solutions. These methods depend on the way in which the constraints were specified.

1. Mathematical solution techniques for systems of simultaneous equations.

Iterative techniques – successive over-relaxation.

Numerous iterative techniques exist that follow this approach. The general approach is one of reducing the number of variables in each iteration. Once the value of one variable is found, it can be substituted in other relations to find values of others.

Genetic techniques

These are based on formulating the problem as a population of instances and evolving them through generations by using the appropriate combination/evolution techniques. After several generations they are expected to converge to acceptable solutions.

Search techniques

These techniques are concerned with searching the solution space for an acceptable solution. Criteria are applied at each point to determine the next search step.

2. Graph models

Local propagation

This method assumes the underlying representation to be a graph. Values, variables and operations form nodes. Operands and operators are joined by arcs. In this mechanism, known values are propagated along the arcs. When a node receives sufficient information from its arcs, it *fires*, i.e., calculates one or more values for the arcs that do not contain values and sends these new values out. These new values in turn cause other nodes to fire and so on.

Graph transformations

This technique uses rewrite rules to transform subgraphs of a constraint program into another graph. The rules are so designed as to maintain the semantics of the computation. Both the original program and the transformed one are *equivalent*.

3. Graphical

   These are restricted to linear relationships. They are inflexible as they disallow change when infeasible.

4. Procedural

Brute force

A step by step approach of executing each constraint to obtain the desired result.

5.3 Bounding Box of a Component/Assembly

The notion of associating a bounding box with a component was introduced during the evolution of *OptiCAD*. Initially the system was hard-coded to a few components. Whenever a new component was introduced, the whole system needed changes. The component models were also rudimentary with no notion of graphical representation nor of simulation. Out of necessity, a bounding box was defined for each component.

The bounding box masks the idiosyncrasies of each component with respect to their shapes. The need for a geometric bounding box increased with the consideration of object oriented geometric modelling.

Definition: The bounding box of a component is its smallest enclosing box.

With the continued development of new and more detailed simulation models, which emphasize on the behavior/functionality of the component rather than only the structure, the need for a virtual component has grown. The bounding box concept has been absorbed into the geometric/structural aspects of the component.

The essential idea of a bounding box is one of introducing a mechanism that permits abstraction. At the architecture level, all the components are treated as boxes. The details of geometry/functionality are all encased within each box. This factors out the complexity of the architecture in terms of design and analysis.

Addition of a new component has no effect on the total system. There is an additive cost of describing a component and making it available to the system. The multiplicative gains are through reuse and hierarchical support.

The notion of a bounding box has simplified the addition of new components and assemblies to the system. The architect can over a period accumulate a library of components with no change to the main system.

Since the bounding box is a regular shaped object, the issues of positioning, orienting and aligning are much easier. The bounding box of a component has a reference point and a reference vector. Their default positions and orientations are shown in Figure 5.2.

All position constraints are specified with respect to the reference point. This makes the specification of position constraints simpler as the component's shape need not play any role in these. All the orientation constraints are specified with respect to the reference vector. Figure 5.3 illustrates two objects and their reference points, vectors and bounding boxes.

The orientation of a component is done by specifying three angles. These angles are not the direction cosines of the reference vector. The steps in orienting an object

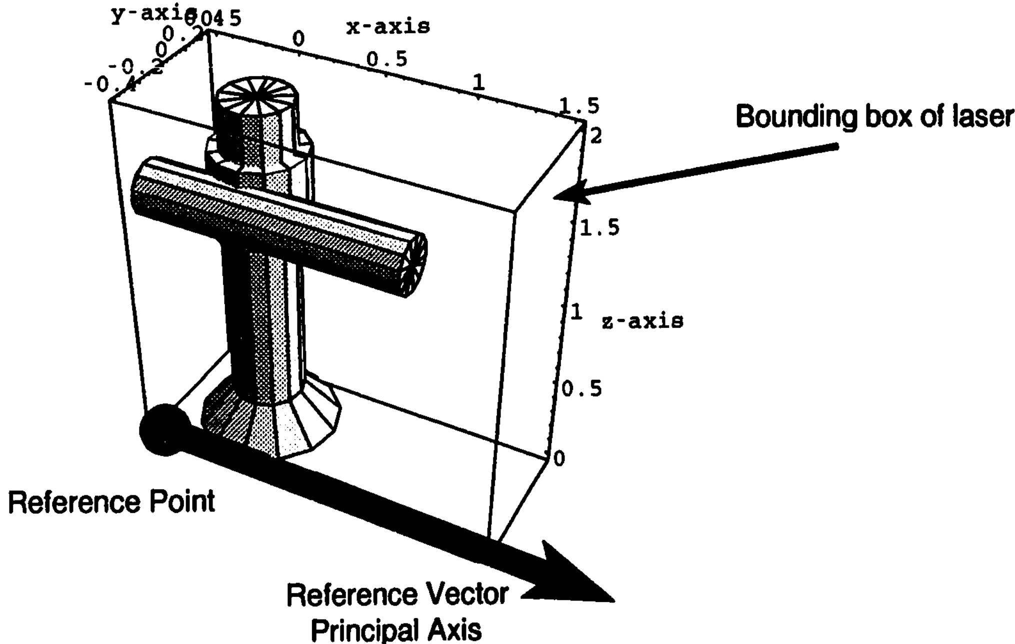


Figure 5.2: A Component with its Reference Point, Reference Vector and Bounding Box.

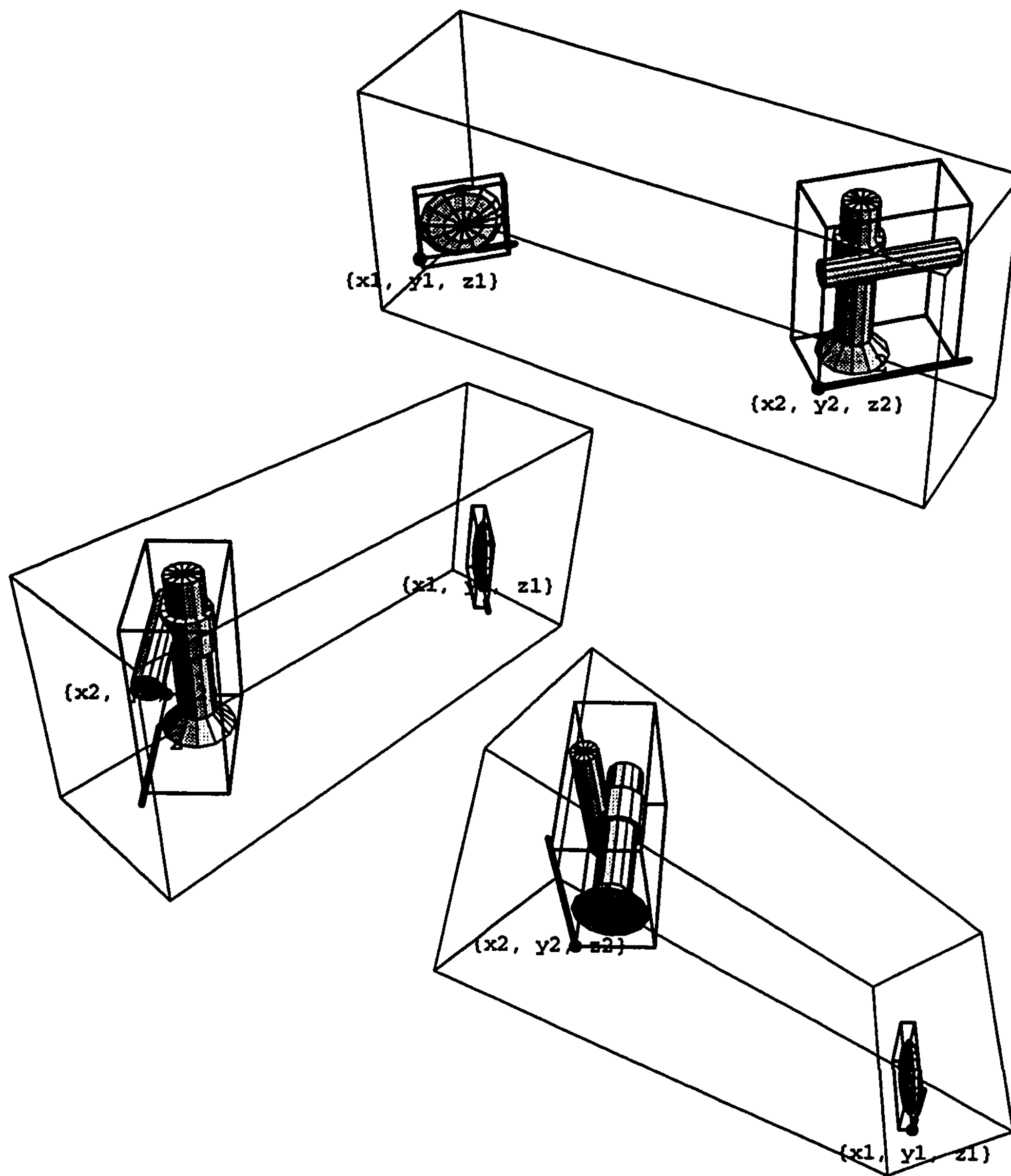


Figure 5.3: Different Views of Two Components – Laser and Convex Lens with Their Bounding Boxes, Reference Points and Reference Vectors.

by the angles {thetaX, thetaXY, thetaP} are as follows.

1. Rotate the object around its z-axis using the reference point as pivot by an angle of thetaX.

2. Rotate the object so that the reference vector makes an angle thetaXY with the xy-plane.

3. Rotate the object around its principal axis (reference vector) by an angle thetaP.

According to convention, positive angles are taken as anti-clockwise.

5.4 Bounding Box Placement

This section describes various possible ways in which objects may be oriented in 3D space. The main focus will be on how a component's bounding box can be oriented in space. This orientation specification must be unambiguous i.e., it must lead to a unique orientation.

Different coordinate systems are used to structure 3D space. The most common ones being rectangular, cylindrical and spherical. These and other coordinate systems are presented in Section 5.4.1. The placement constraint specification primitives presented in Chapter 4 are elaborated from an implementation point of view. For each of those positioning constraints, the formulated equations are shown in Section 5.4.2. The full set of generated position constraint equations is presented in Section 5.4.3. In Section 5.4.4, the tabular and graphical versions of the solution for the Trip-Flop architecture are presented.

5.4.1 Coordinate Systems

There are various coordinate systems. Each has its advantages and disadvantages in different applications. The most common coordinate system, i.e., the cartesian system, represents a point in 3D space as a triplet $\{x, y, z\}$. The same point can be represented in other coordinate systems. The formulae for representing a point $\{x, y, z\}$ in other coordinate systems are as follows [84].

Bipolar: $\{-\mathrm{Im}(2\,\mathrm{ArcCoth}(x + i\,y)), \mathrm{Re}(2\,\mathrm{ArcCoth}(x + i\,y)), z\}$

Bispherical: $\{-\mathrm{Im}(-2\,i\,\mathrm{ArcCot}(\sqrt{x^2+y^2} - i\,z)),$
$\mathrm{Re}(-2\,i\,\mathrm{ArcCot}(\sqrt{x^2+y^2} - i\,z)), ArcTan(x, y)\}$

Cartesian: $\{x, y, z\}$

Conical: $\{\sqrt{x^2+y^2+z^2},$
$$\frac{\sqrt{\frac{5x^2+4y^2+z^2+\sqrt{(-5x^2-4y^2-z^2)^2-16x^2(x^2+y^2+z^2)}}{x^2+y^2+z^2}}\,\mathrm{Sign}(y)}{\sqrt{2}},$$
$$\frac{\sqrt{\frac{5x^2+4y^2+z^2-\sqrt{(-5x^2-4y^2-z^2)^2-16x^2(x^2+y^2+z^2)}}{x^2+y^2+z^2}}\,\mathrm{Sign}(z)}{\sqrt{2}}\}$$

Cylindrical: $\{\sqrt{x^2+y^2}, ArcTan(x, y), z\}$

EllipticCylindrical: $\{\mathrm{Re}(\mathrm{ArcCosh}(x + i\,y)), \mathrm{Im}(\mathrm{ArcCosh}(x + i\,y)), z\}$

OblateSpheroidal: $\{\mathrm{Im}(\mathrm{ArcCosh}(\sqrt{x^2+y^2} + i\,z)),$
$\mathrm{Re}(\mathrm{ArcCosh}(\sqrt{x^2+y^2} + i\,z)), ArcTan(x, y)\}$

ParabolicCylindrical: $\{\frac{y}{\sqrt{-x+\sqrt{x^2+y^2}}}, \sqrt{-x + \sqrt{x^2+y^2}}, z\}$

Paraboloidal: $\{\frac{\sqrt{x^2+y^2}}{\sqrt{-z+\sqrt{x^2+y^2+z^2}}}, \sqrt{-z + \sqrt{x^2+y^2+z^2}}, ArcTan(x, y)\}$

ProlateSpheroidal: $\{\mathrm{Re}(\mathrm{ArcCosh}(i\sqrt{x^2+y^2}+z)),$
$\mathrm{Im}(\mathrm{ArcCosh}(i\sqrt{x^2+y^2}+z)), ArcTan(x,y)\}$

Spherical: $\{\sqrt{x^2+y^2+z^2}, ArcCos(\frac{z}{\sqrt{x^2+y^2+z^2}}), ArcTan(x,y)\}$

Toroidal: $\{\mathrm{Re}(2\,\mathrm{ArcCoth}(\sqrt{x^2+y^2}+i\,z)),$
$-\mathrm{Im}(2\,\mathrm{ArcCoth}(\sqrt{x^2+y^2}+i\,z)), ArcTan(x,y)\}$

For the remainder of the thesis we adopt the cartesian coordinate system.

5.4.2 A Formulation of Equations from Position Specifications

The position constraints are represented internally as a set of equations. Every constraint has its own mapping to equations. Some of the position constraints and their internal representation equations are presented next.

The constraint **AddPlacementConstraint[default[*pbs*1]]** produces the following equations:

$$pbs1.x == Translate[pbs1.x, Min[Allcomponents.x]] \quad (5.1)$$

$$pbs1.y == Translate[pbs1.y, Min[Allcomponents.y]] \quad (5.2)$$

$$pbs1.z == Translate[pbs1.z, Min[Allcomponents.z]] \quad (5.3)$$

Likewise the constraint

AddPlacementConstraint[absolute[*pbs*1, {0, 0, 0}]]

is translated to produce equations:

$$\{x_1 == 0, y_1 == 0, z_1 == 0\} \quad (5.4)$$

Finally the constraint

AddPlacementConstraints[{relativeTo[$pbs1, m1, \{-a, 0, 0\}$]}]

will result in:

$$\{-0.1 + x_1 == x_6, y_1 == y_6, z_1 == z_6\} \quad (5.5)$$

5.4.3 The Full Set of Equations for Trip-Flop Architecture

The complete position constraints for the Trip-Flop architecture translate to the following set of constraints.

1. $x_1 == 0$
2. $y_1 == 0$
3. $z_1 == 0$
4. $-0.1 + x_1 == x_6$
5. $y_1 == y_6$
6. $z_1 == z_6$
7. $x_6 == x_5$
8. $-0.1 + y_6 == y_5$
9. $z_6 == z_5$
10. $x_5 == x_3$
11. $-0.1 + y_5 == y_3$
12. $z_5 == z_3$
13. $-0.1 + x_3 == x_{12}$
14. $y_3 == y_{12}$
15. $z_3 == z_{12}$

16. $x_3 == x_7$
17. $-0.1 + y_3 == y_7$
18. $z_3 == z_7$
19. $x_1 == x_4$
20. $-0.1 + y_1 == y_4$
21. $z_1 == z_4$
22. $x_4 == x_2$
23. $-0.1 + y_4 == y_2$
24. $z_4 == z_2$
25. $0.1 + x_2 == x_{10}$
26. $y_2 == y_{10}$
27. $z_2 == z_{10}$
28. $x_2 == x_8$
29. $-0.1 + y_2 == y_8$
30. $z_2 == z_8$
31. $-0.1 + x_8 == x_7$
32. $y_8 == y_7$
33. $z_8 == z_7$
34. $x_8 == x_9$
35. $-0.1 + y_8 == y_9$
36. $z_8 == z_9$
37. $-0.1 + x_9 == x_{11}$
38. $y_9 == y_{11}$
39. $z_9 == z_{11}$

Variable (x)	Value	Variable (y)	Value	Variable (z)	Value
x_1	0.2	y_1	0.4	z_1	0.0
x_2	0.2	y_2	0.2	z_2	0.0
x_3	0.1	y_3	0.2	z_3	0.0
x_4	0.2	y_4	0.3	z_4	0.0
x_5	0.1	y_5	0.3	z_5	0.0
x_6	0.1	y_6	0.4	z_6	0.0
x_7	0.1	y_7	0.1	z_7	0.0
x_8	0.2	y_8	0.1	z_8	0.0
x_9	0.2	y_9	0.0	z_9	0.0
x_{10}	0.3	y_{10}	0.2	z_{10}	0.0
x_{11}	0.1	y_{11}	0.0	z_{11}	0.0
x_{12}	0.0	y_{12}	0.2	z_{12}	0.0

Table 5.1: The Solution to the Position Constraints.

5.4.4 Solutions to the Position Constraints

Solving the set of equations generated by the position constraints of the Trip-Flop architecture specification yields the values shown in Table 5.1.

5.4.5 The Completely Positioned Trip-Flop Architecture

The above results can be used to obtain a 3-D layout of the Trip-Flop architecture. Figures 5.4 through 5.6 show this layout from three different camera positions. The components are not properly oriented as the orientation information has yet to be incorporated.

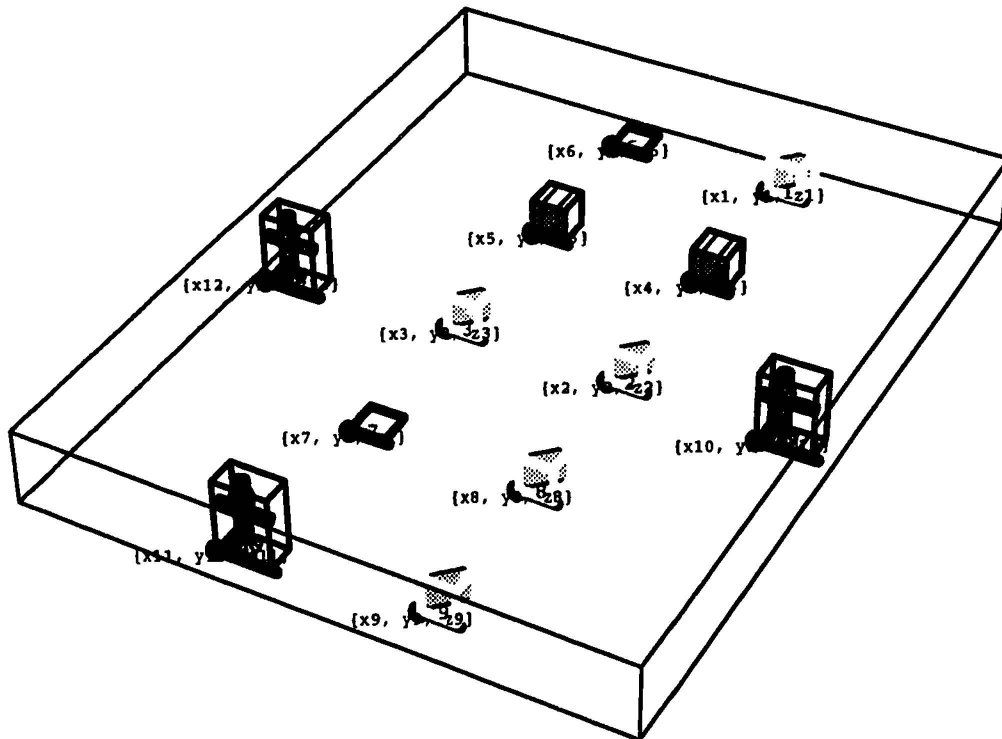


Figure 5.4: The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, -1, 1}

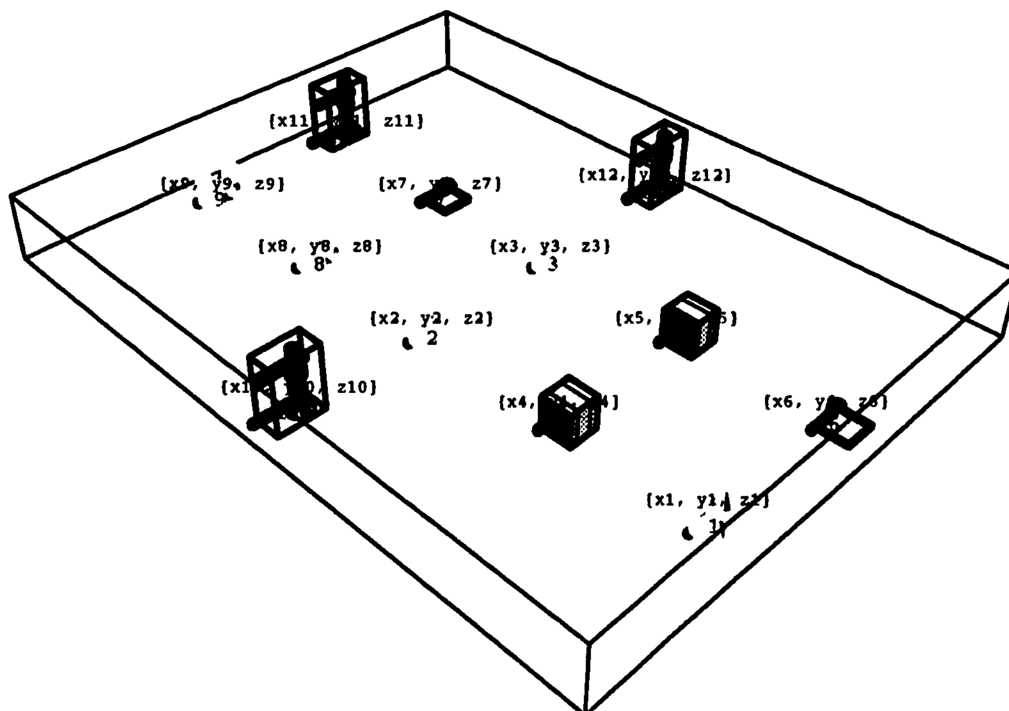


Figure 5.5: The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, 1, 1}

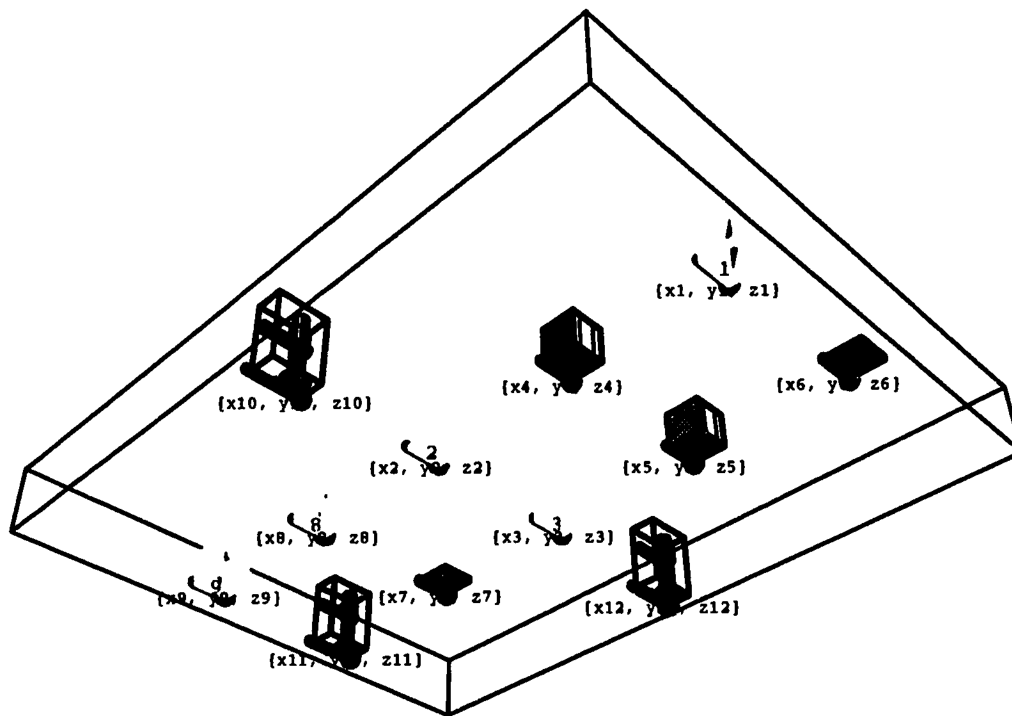


Figure 5.6: The Positioned Trip-Flop Architecture with Bounding Boxes: Camera Position {1, 1, -1}

5.5 Orientation of Bounding Box

Orientation of components/assemblies is their arrangement usually achieved by rotation around the reference point/vector or by reflection through a plane. Numerous optical architectures that employ bulk components rely on tricky placement and orientation for correct functionality. This section deals with the issue of orienting optical components in free space.

The next Subsection deals with fixing representation schemes for points, vectors and objects in 3D. A unique orientation representation is discussed in Section 5.5.2 that has a 1-1 mapping with the object's orientation. Chapter 4 listed several orientation constraints supported by $\mathcal{OHDL}$. They are illustrated in Section 5.5.3 with examples from Trip-Flop focusing on the implementation issues of translating the orientation specifications into mathematical equations. Section 5.5.4 continues with the combined set of equations for the Trip-Flop architecture. A solution to these is presented in Section 5.5.5.

5.5.1 Representation Schemes

A *point* in 3-D may be uniquely represented by the three coordinates i.e., the point can be fixed by giving the values of the vertical displacements (a, b, c) from the origin (see Figure 5.7).

A unit vector, at any instant, has a point of origin and a direction. It can be uniquely represented by giving the coordinates of its point of origin and the angles that it makes with at least two of the reference axes. The following steps obtain the vector from such a representation $\{(x_0, y_0, z_0), \theta_x, \theta_{xy}\}$

1. Take a unit vector parallel to one reference axis (say x-axis) at the point of

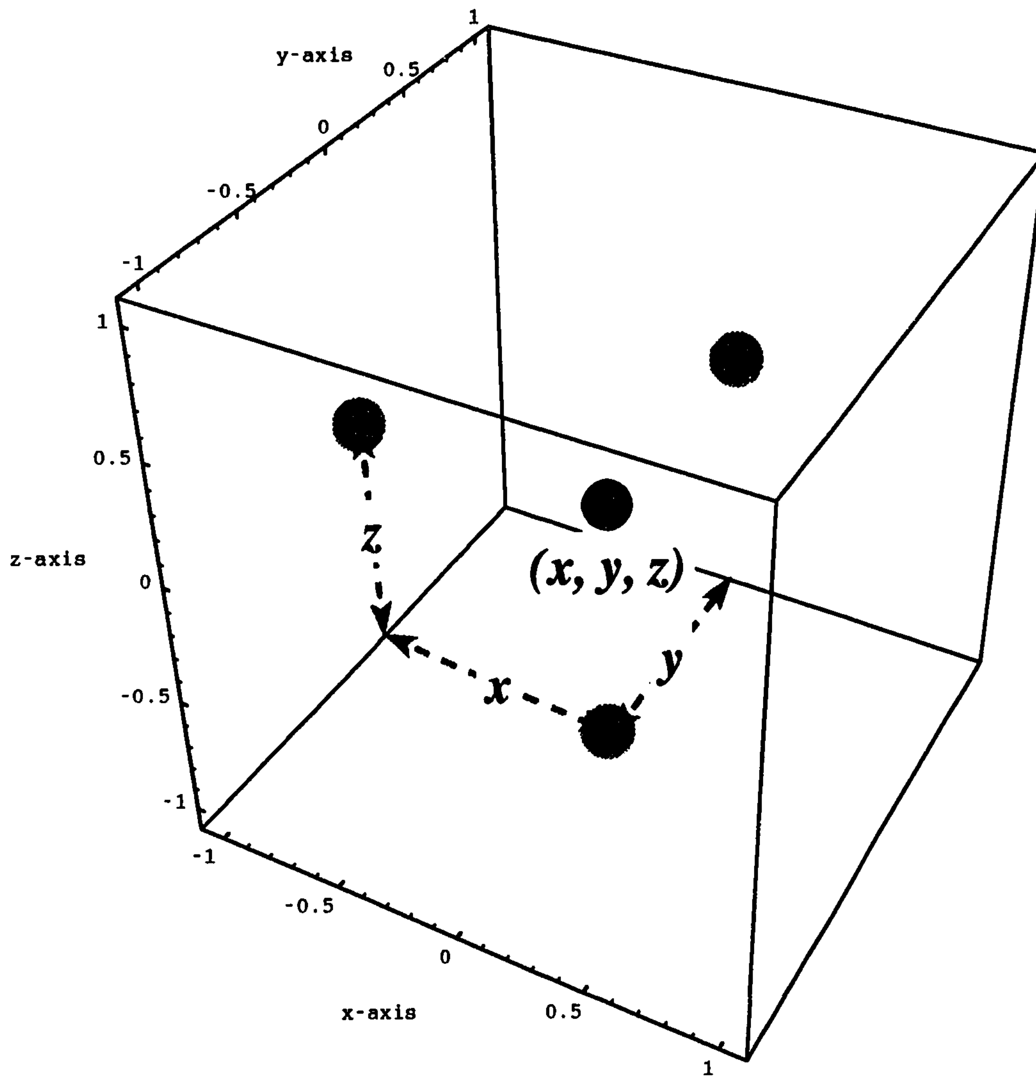


Figure 5.7: A Point is Uniquely Determined by its Three Coordinates in 3-D Space

origin of that vector i.e., (x_0, y_0, z_0).

2. Rotate the vector around the z-axis so that it makes an angle θ_x with the x-axis.

3. Rotate the vector in the plane perpendicular to xy-plane but containing a rotated vector of step 2, so that it makes an angle of θ_{xy} with the xy-plane.

5.5.2 Orientation Representation

Orienting an object in 3-D is more complex than placing a vector. This is because an object has more degrees of freedom as shown in Figure 5.8.

To simplify the task of orienting optical components, a reference vector is associated with each component. This is called the principal axis of the component. Once the principal axis is oriented, the only degree of freedom is that of rotating the object around it (see Figure 5.9).

This degree of freedom can be constrained by specifying the angle of rotation around the principal axis. Hence an object can be represented in 3-D space by $\{(x_0, y_0, z_0), \theta_x, \theta_{xy}, \theta_p\}$ where $\{(x_0, y_0, z_0), \theta_x, \theta_{xy}\}$ represent the principal axis orientation and θ_p specify the angle of rotation around this axis.

Complete rotations around the z-axis, y-axis, and the principal axis are shown in Figures 5.10 through 5.12, Figures 5.13 through 5.15 and Figures 5.16 through 5.18 respectively. Random orientations are shown in Figures 5.19 and 5.20.

5.5.3 Orientation Constraints and Equation Formulation

Orientation constraints are translated internally into a set of equations. The solution to this set of simultaneous equations will provide a satisfactory orientation.

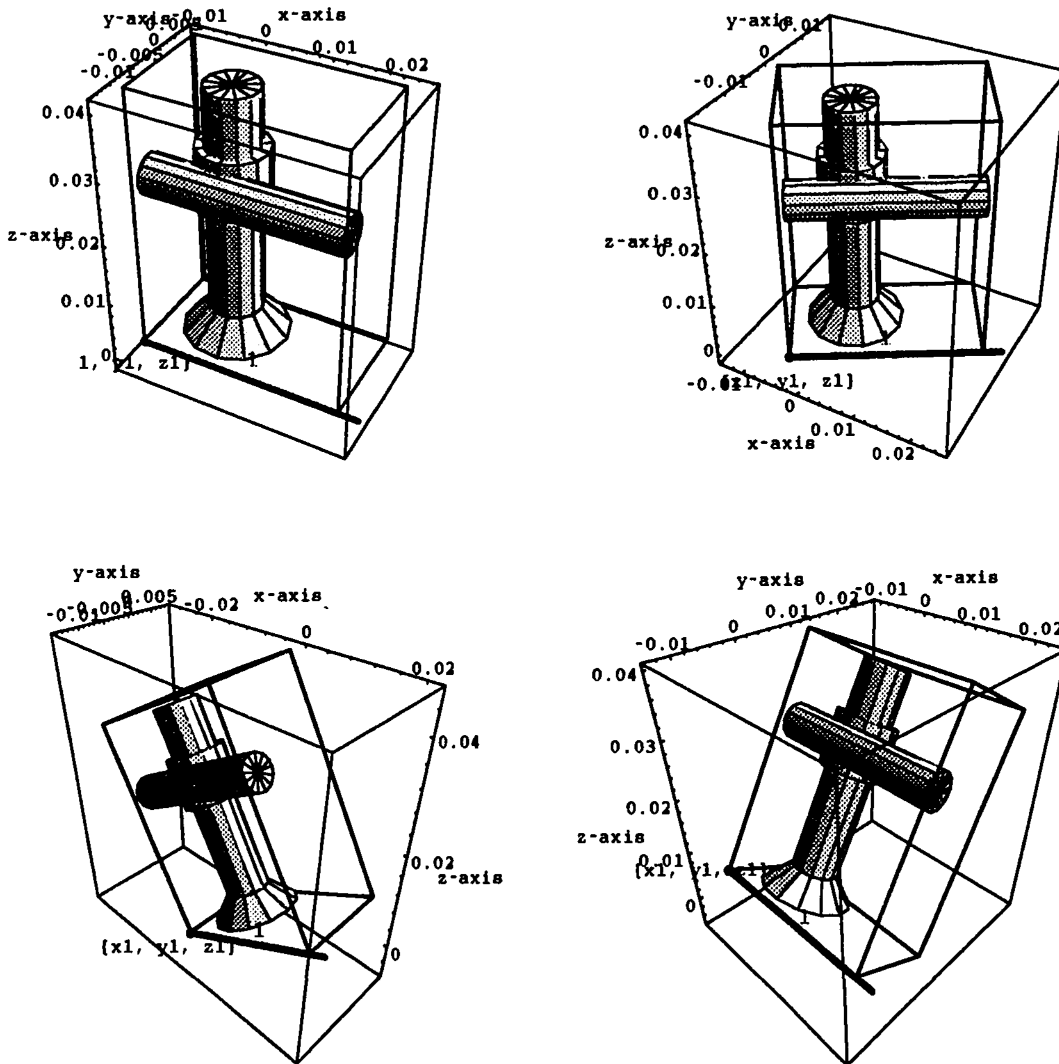


Figure 5.8: The Degrees of Freedom of an Object in 3-D Space

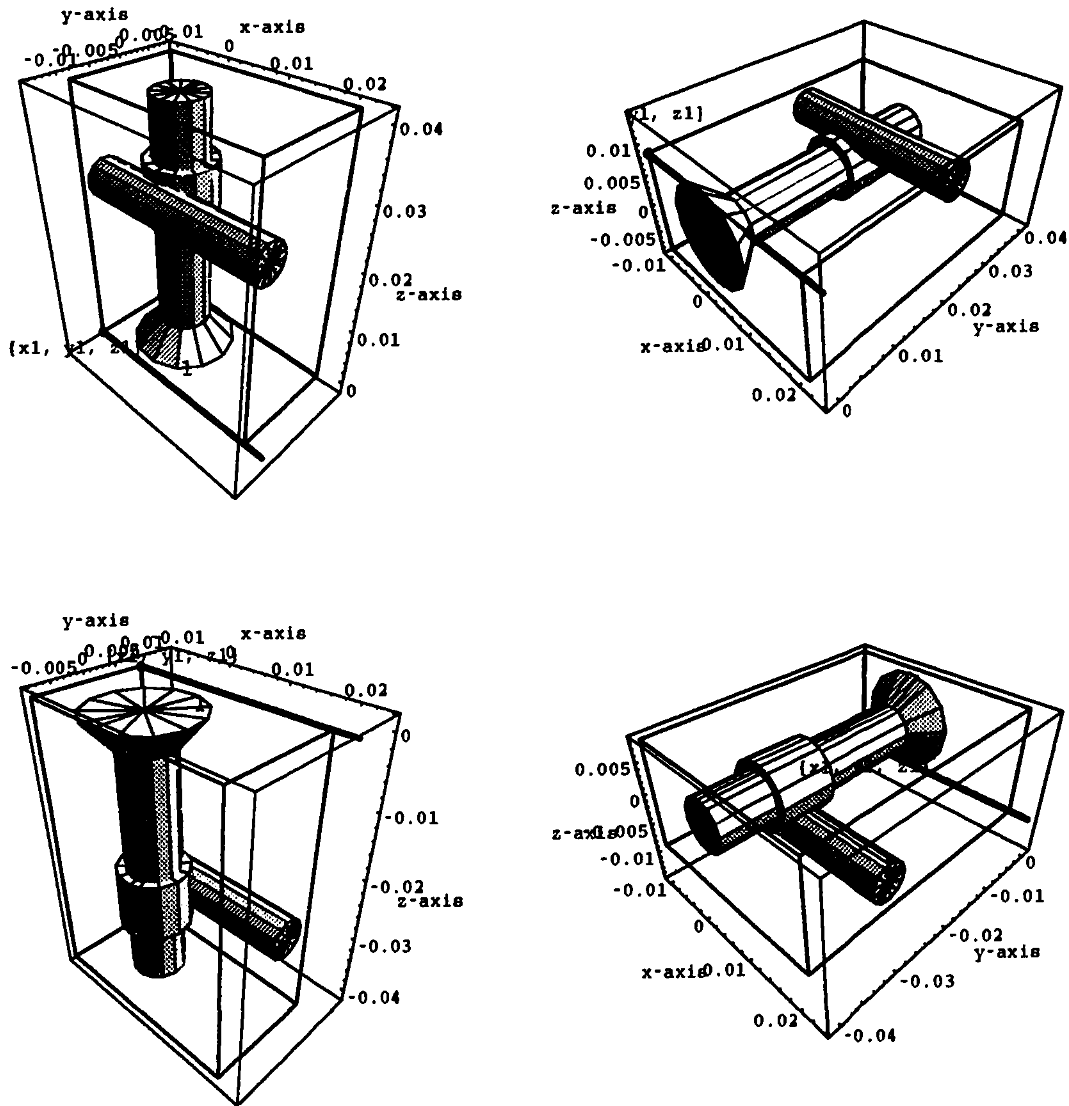


Figure 5.9: Rotating an Object Around its Principal Axis.

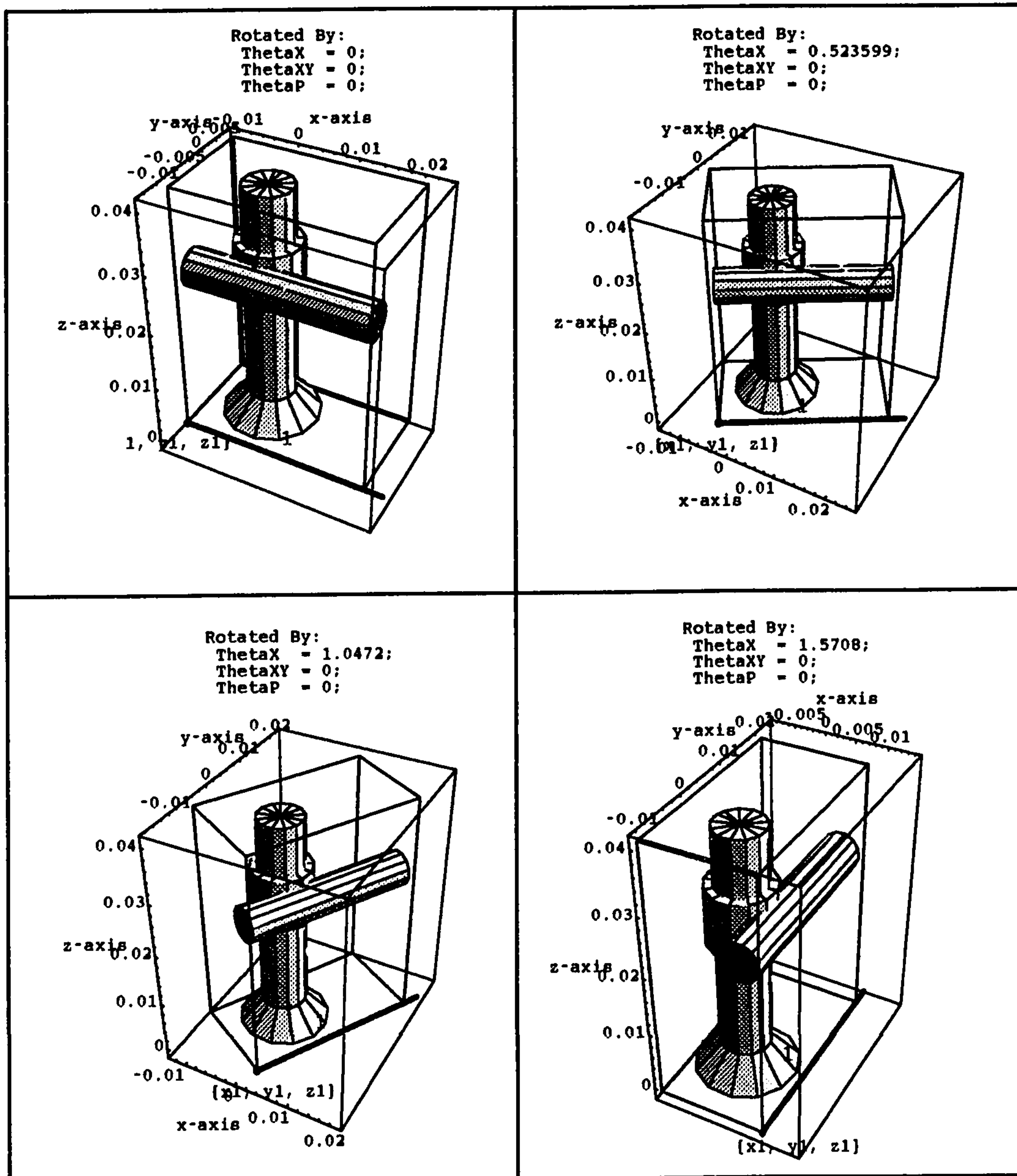


Figure 5.10: Rotation Around the z-axis

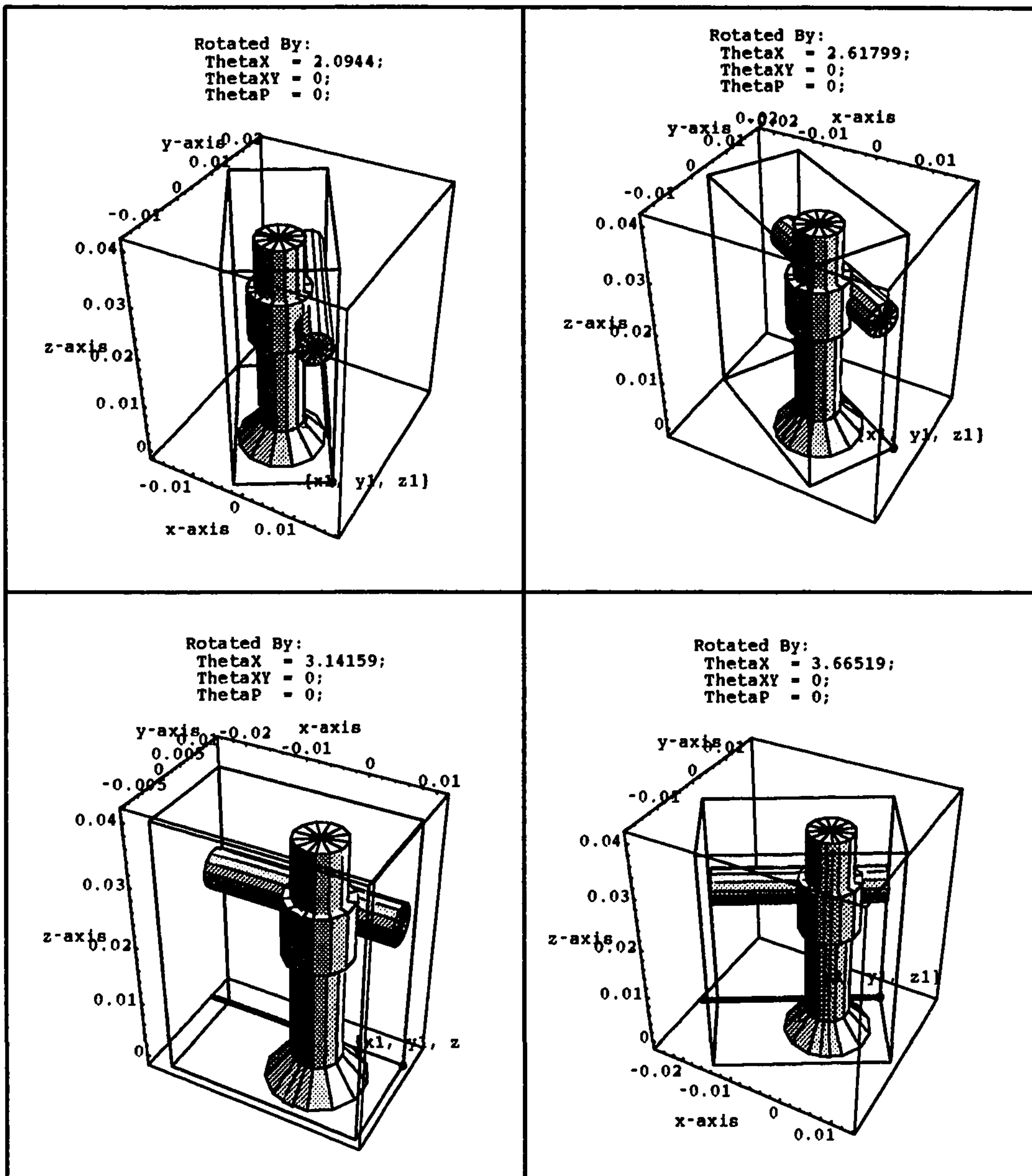


Figure 5.11: Rotation Around the z-axis

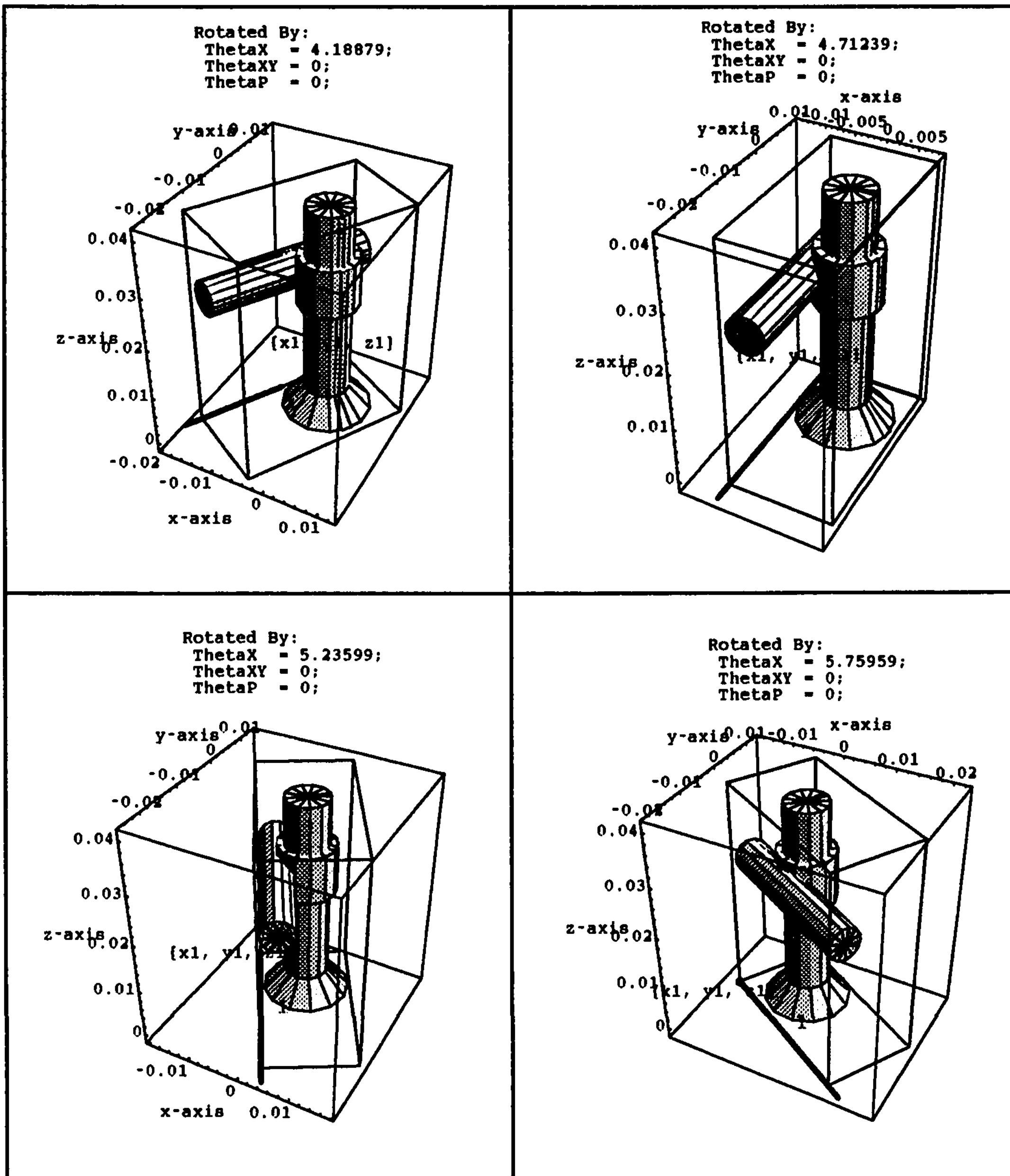


Figure 5.12: Rotation Around the z-axis

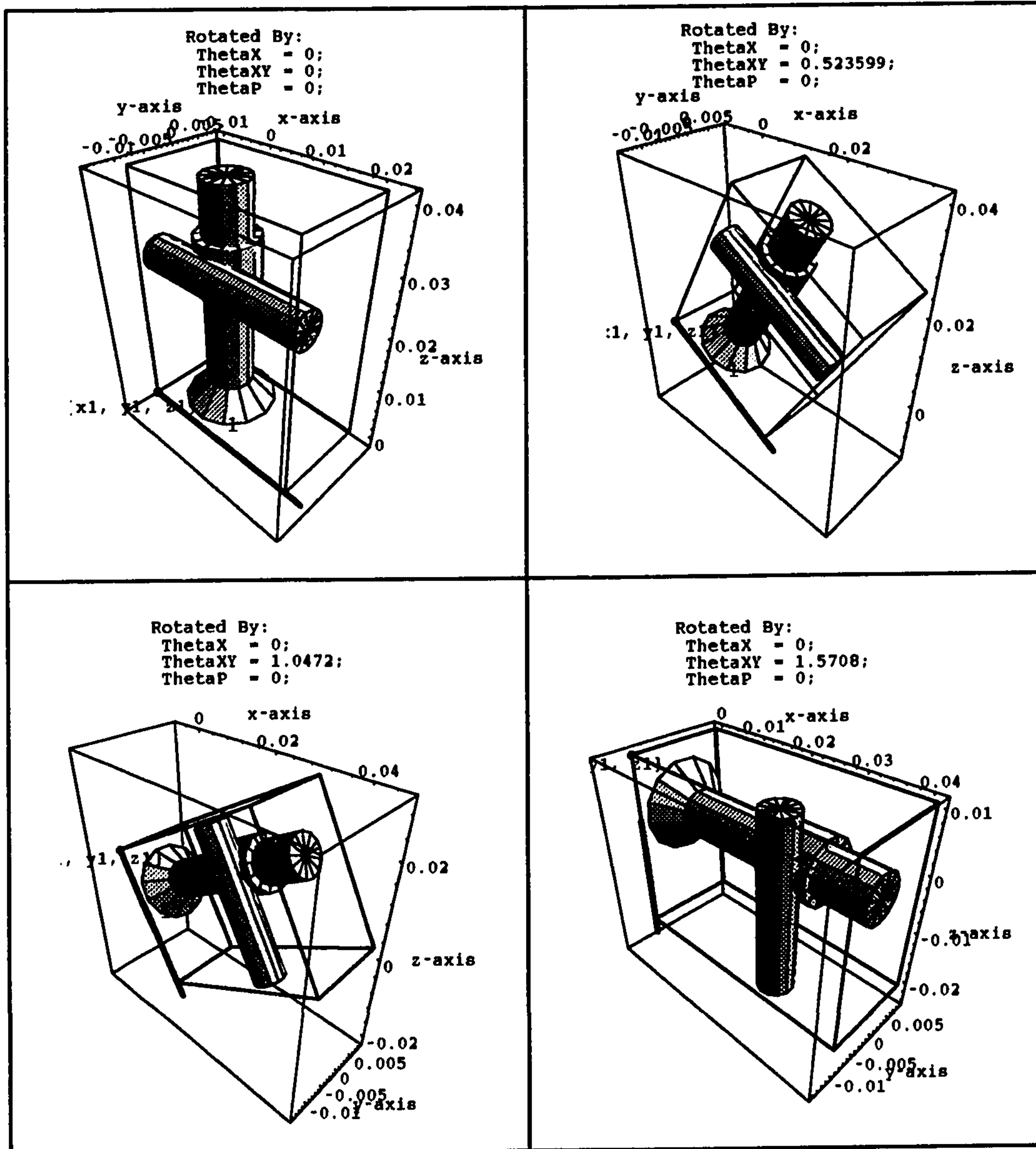


Figure 5.13: Rotation Around the y-axis

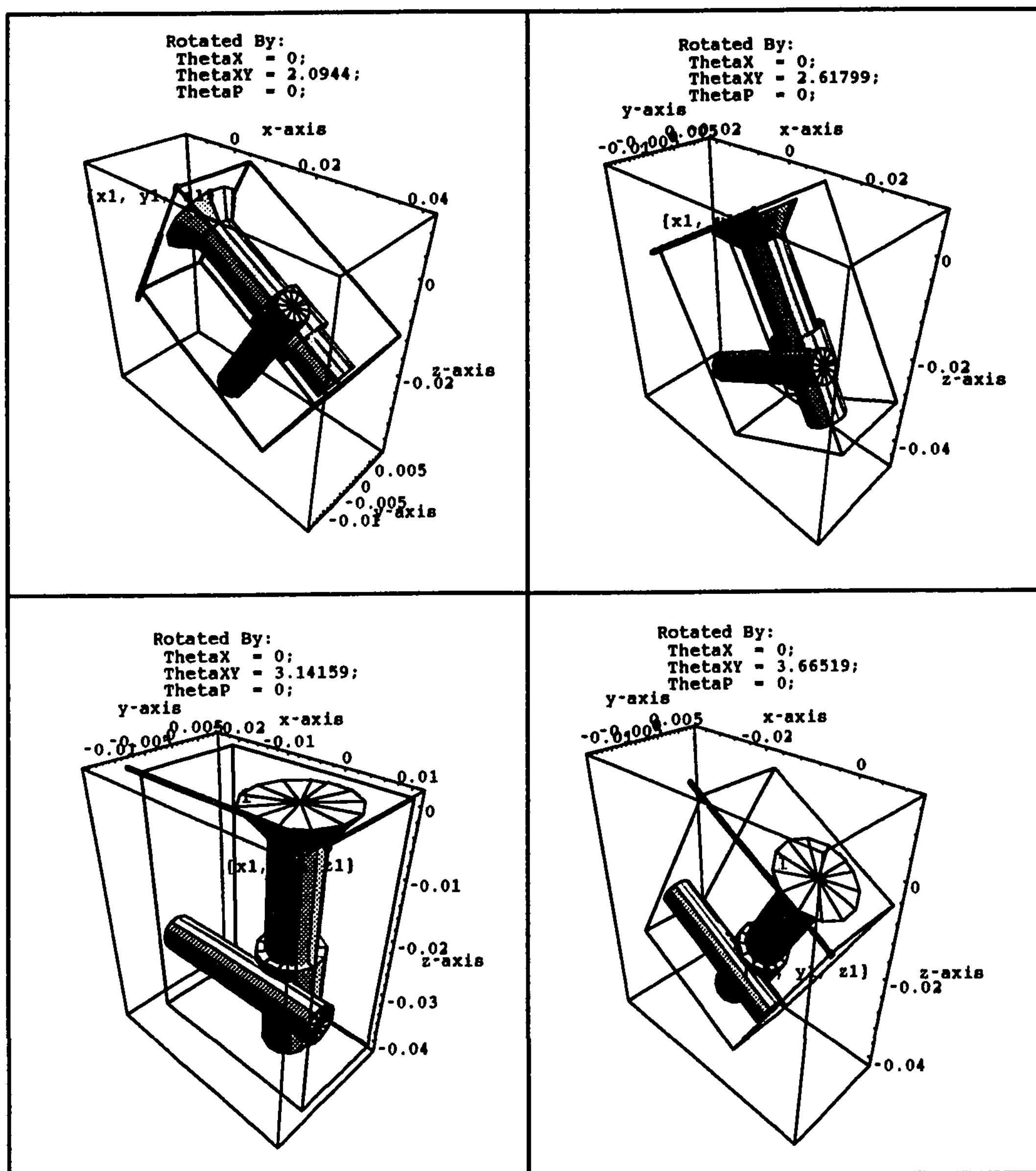


Figure 5.14: Rotation Around the y-axis

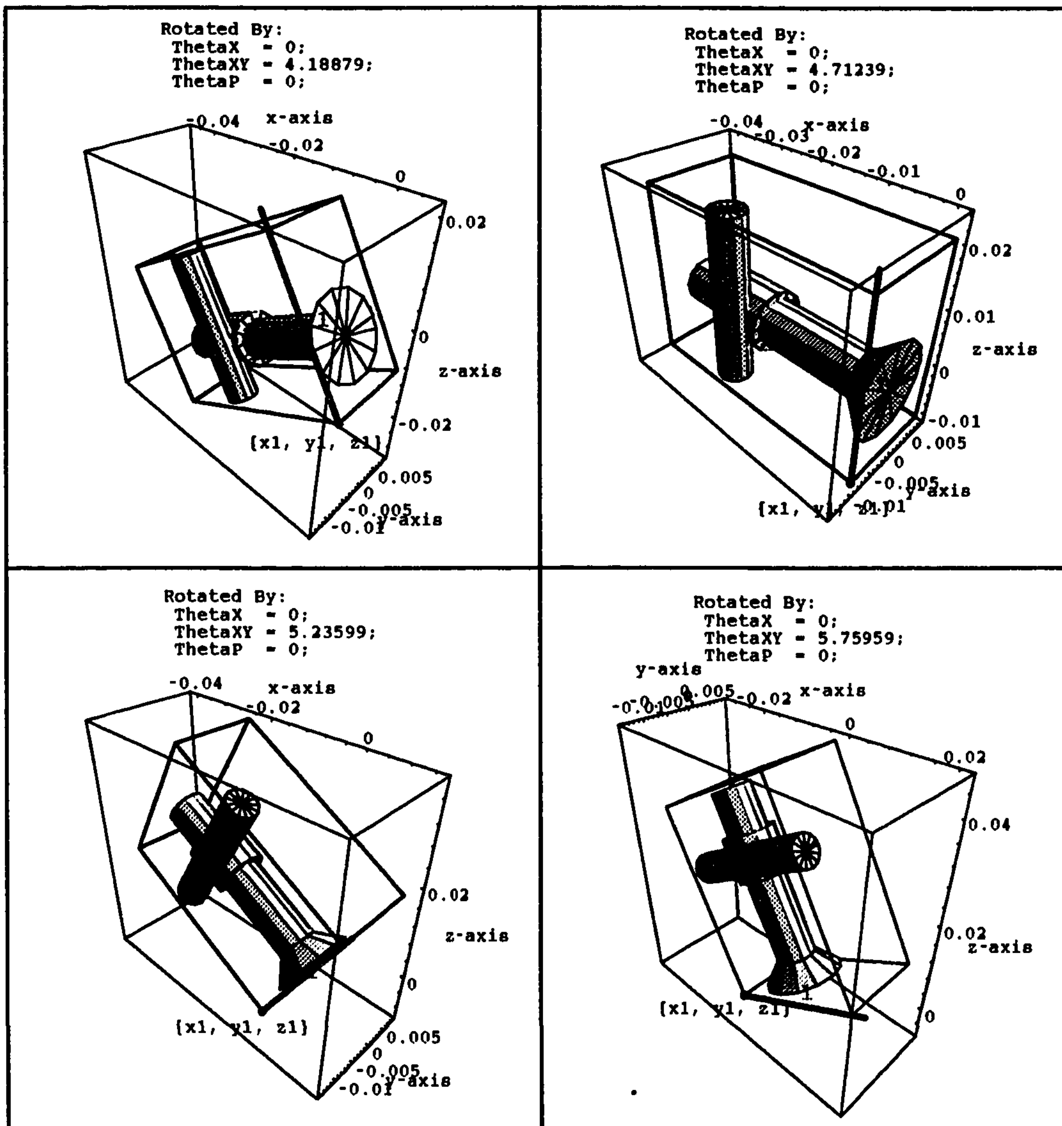


Figure 5.15: Rotation Around the y-axis

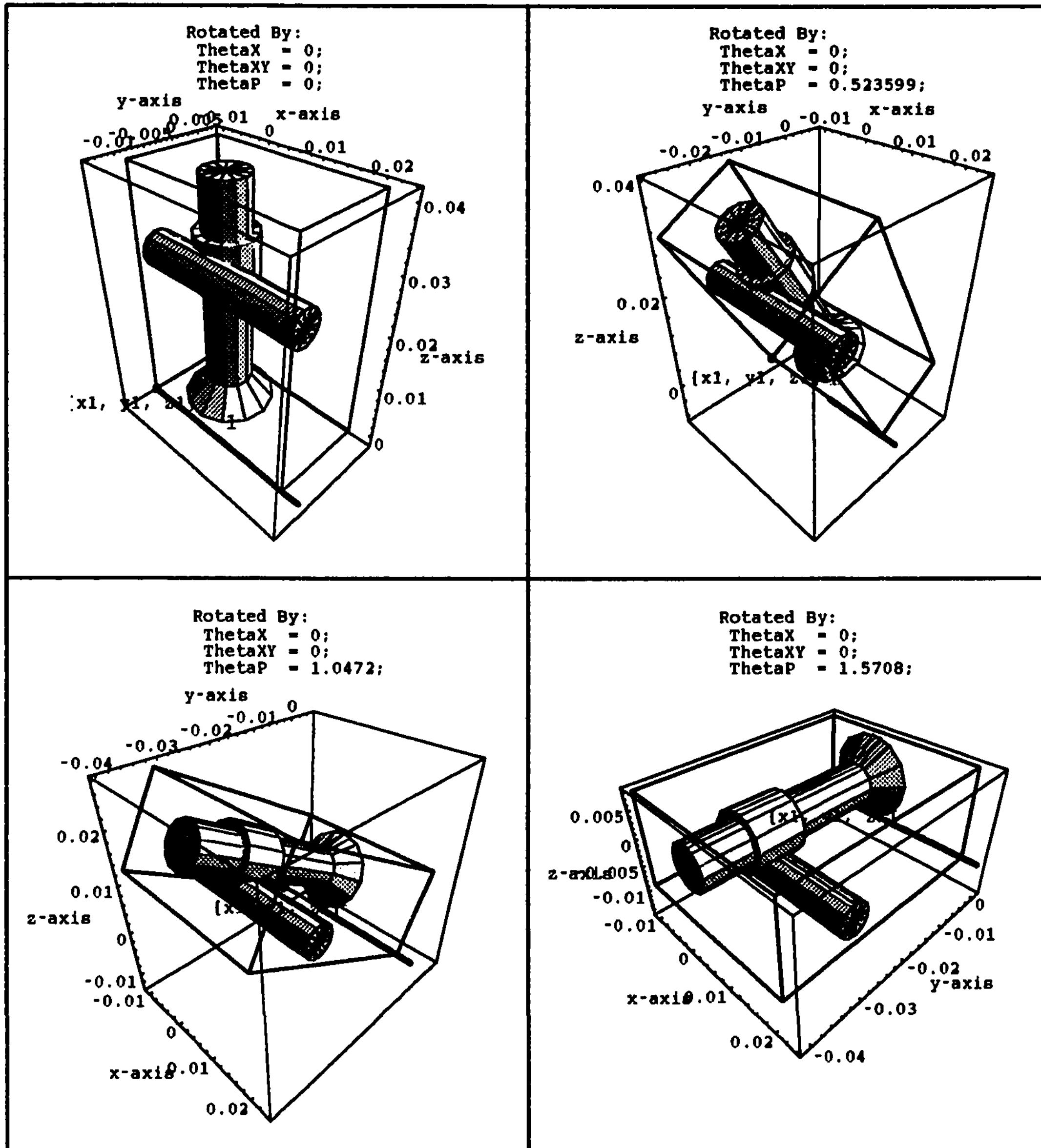


Figure 5.16: Rotation Around the Principal Axis

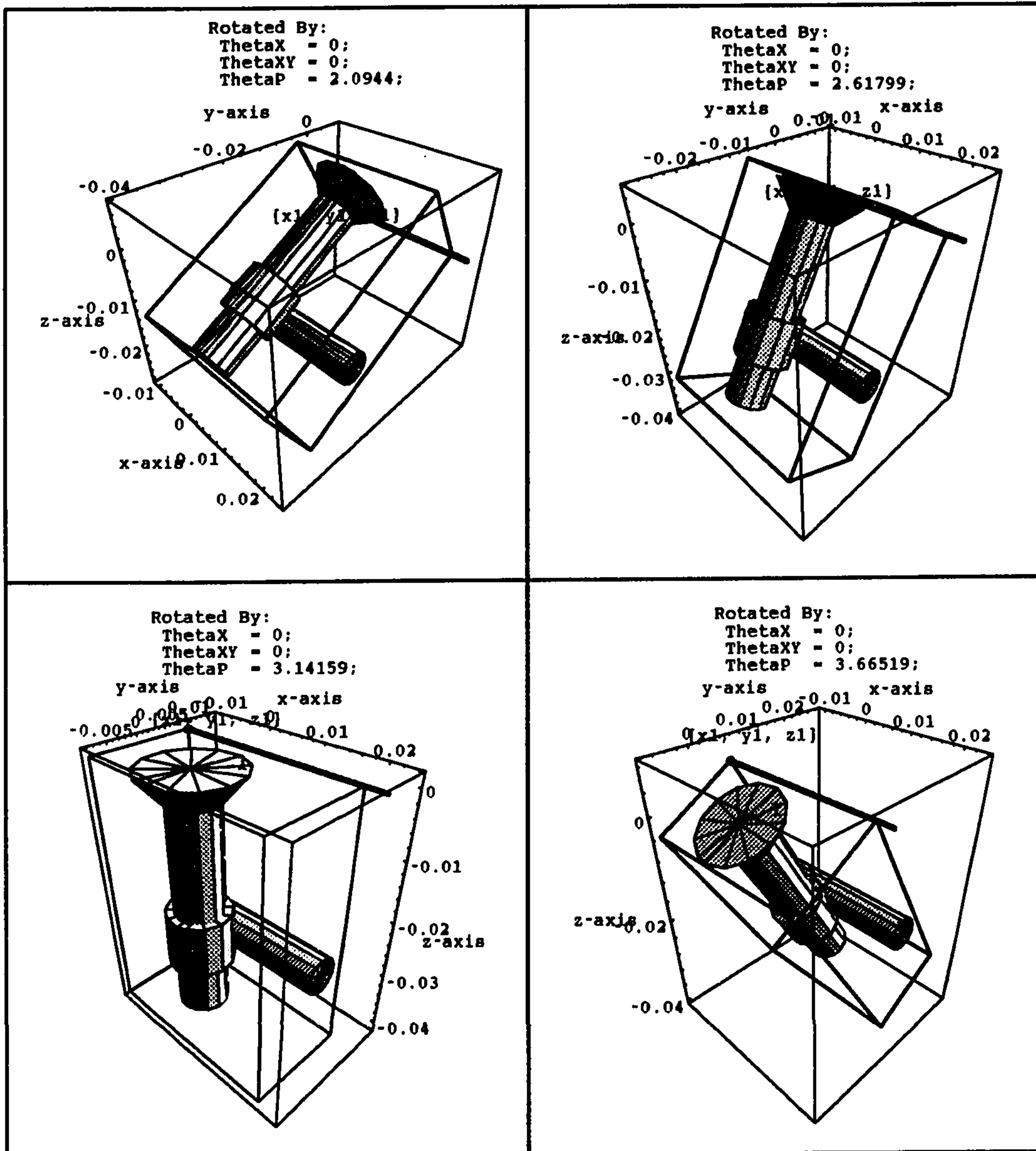


Figure 5.17: Rotation Around the Principal Axis

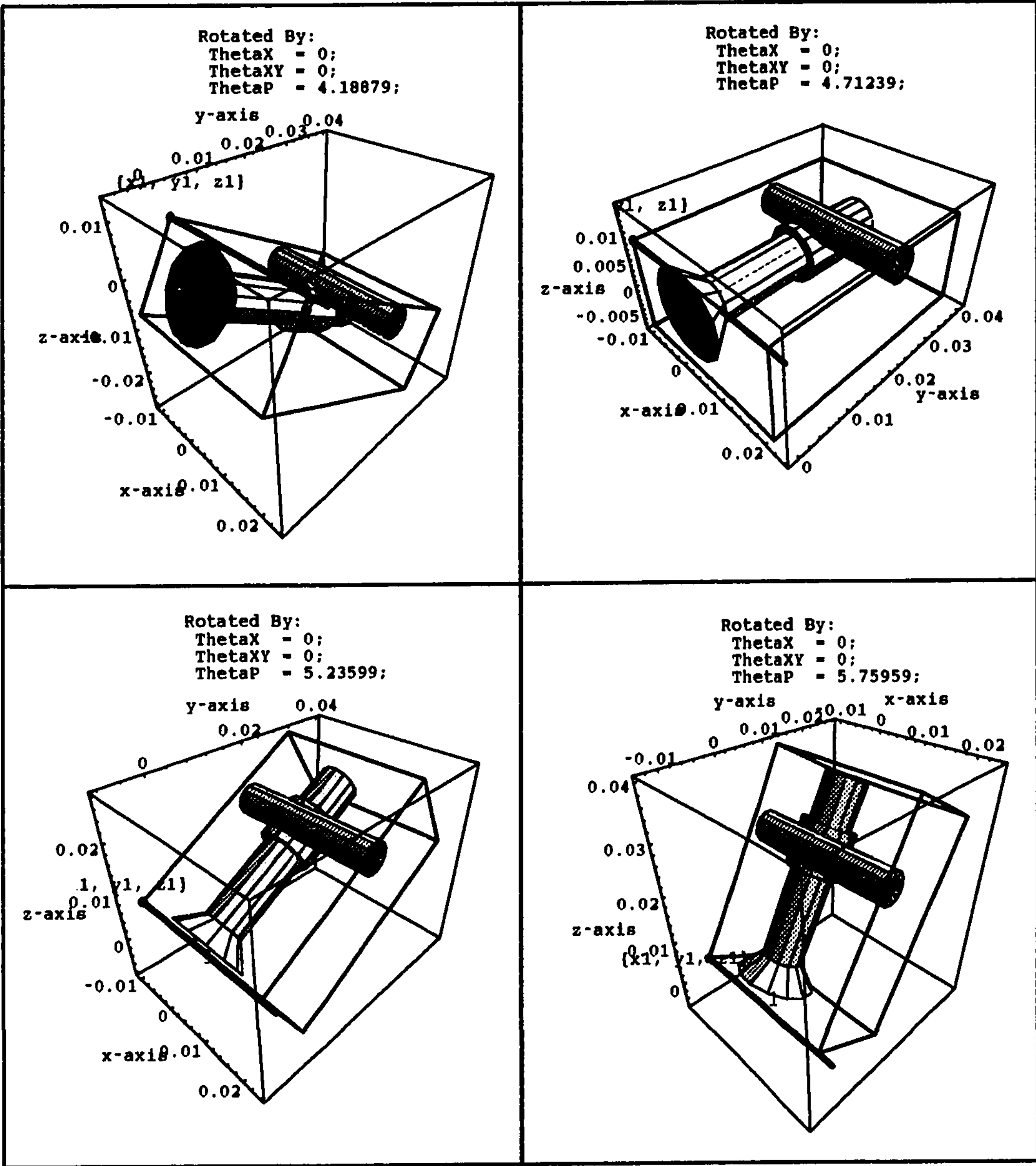


Figure 5.18: Rotation Around the Principal Axis

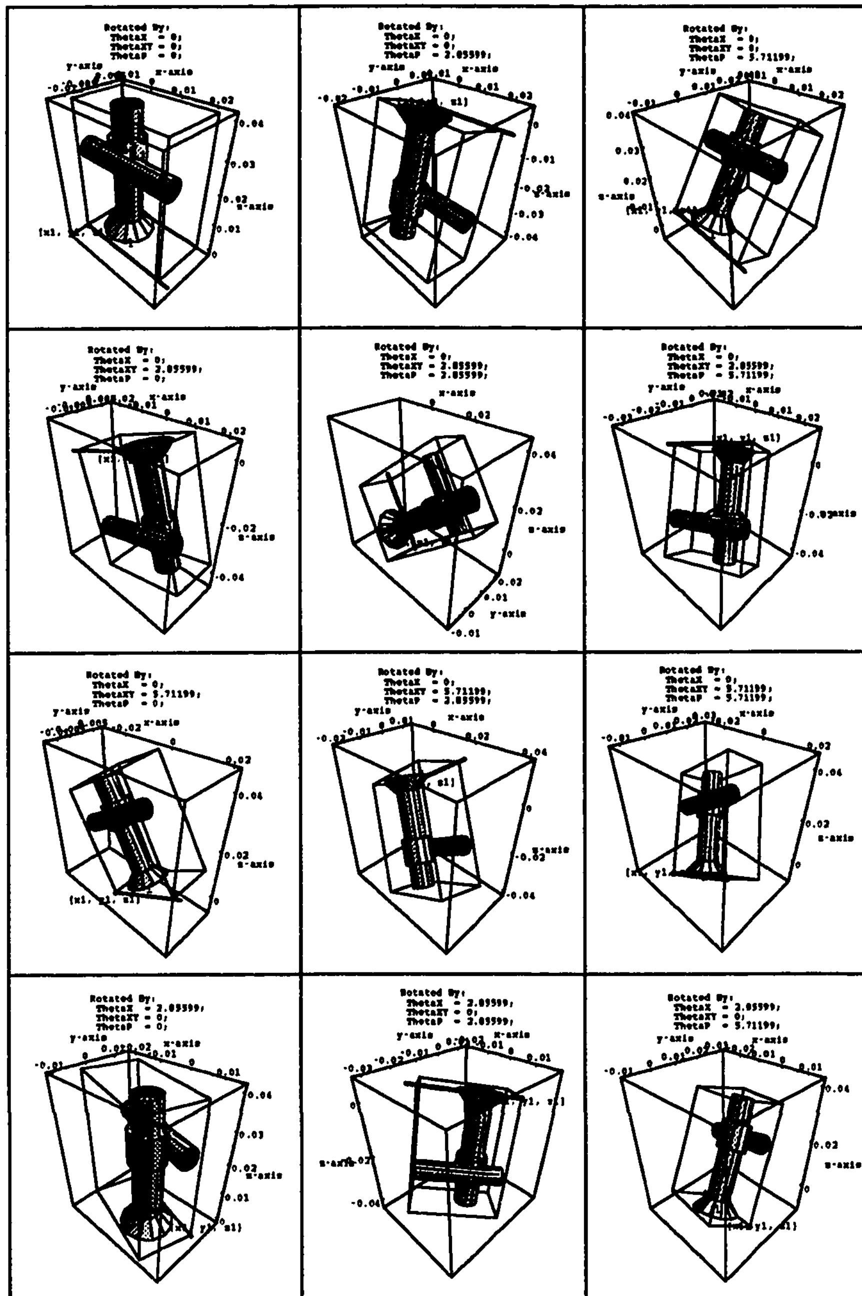


Figure 5.19: Random Orientations of the Laser Object

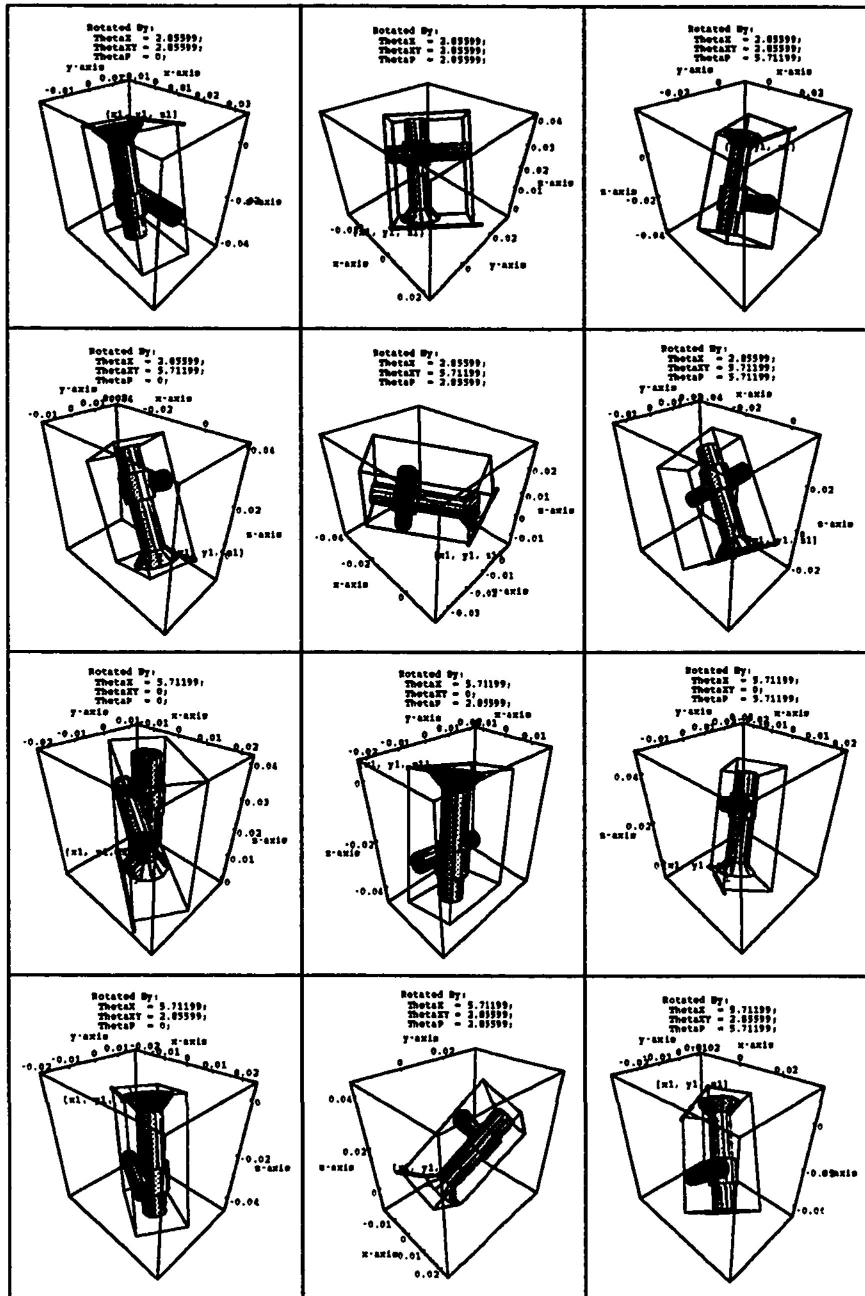


Figure 5.20: Random Orientations of the Laser Object

The orientation constraint **AddOrientationConstraints[{orientationOf[*bs*1, {angle[Pi/2], angle[0], angle[0]}]}]** produces the orientation equations:

$$\{r_8 == 1.5708, s_8 == 0, t_8 == 0\} \quad (5.6)$$

Likewise the orientation constraint

AddOrientationConstraint[parallelTo[*pbs*1, *if*1]]

produces the equations:

$$\{0 == r_4, 0 == s_4, 0 == t_4\} \quad (5.7)$$

The orientation constraint **AddOrientationConstraint[perpendicularTo[*bs*1, *pbs*1]]** results in:

$$\{r_8 == 1.5708\} \quad (5.8)$$

AddOrientationConstraints[{ atAnAngle[*m1*, angle[45 degree], *if2*] }] is translated to:

$$\{r_6 == 0.785398 + r_5\} \quad (5.9)$$

The constraint

AddOrientationConstraints[{ relativeAngles[pbs3, m2, {angle[45 degree], angle[90 degree], angle[0]}]}]

leads to the equations:

$$\{r_3 == 0.785398 + r_7, s_3 == 1.5708 + s_7, t_3 == t_7\} \quad (5.10)$$

The variables r, s and t correspond to the angles $\theta_x, \theta_{xy}, \theta_p$ respectively.

5.5.4 The Full Set of Constraints for Trip-Flop Architecture

The Trip-Flop orientation constraints generate the following equations.

1. $r_{11} == 0$
2. $s_{11} == 0$
3. $t_{11} == 0$
4. $r_8 == 1.5708$
5. $s_8 == 0$
6. $t_8 == 0$
7. $r_{11} == r_{12}$
8. $s_{11} == s_{12}$
9. $t_{11} == t_{12}$
10. $0 == r_3$
11. $0 == s_3$
12. $0 == t_3$
13. $r_4 == r_5$
14. $s_4 == s_5$
15. $t_4 == t_5$
16. $r_2 == r_8$
17. $s_2 == s_8$
18. $t_2 == t_8$
19. $r_8 == r_9$
20. $s_8 == s_9$
21. $t_8 == t_9$
22. $r_3 == r_{12}$

23. $s_3 == s_{12}$
24. $t_3 == t_{12}$
25. $0 == r_4$
26. $0 == s_4$
27. $0 == t_4$
28. $r_8 == 1.5708$
29. $r_6 == 0.785398 + r_5$
30. $r_{12} == 3.14159 + r_{10}$
31. $r_6 == 0.785398$
32. $s_6 == 1.5708$
33. $t_6 == 0$
34. $r_3 == 0.785398 + r_7$
35. $s_3 == 1.5708 + s_7$
36. $t_3 == t_7$

5.5.5 Solutions to the Orientation Constraints

Solving the equations of the previous section yields the values shown in Table 5.2.

5.5.6 Placed and Oriented Architecture

Incorporating the orientation information into the Trip-Flop layouts yields the new layouts shown in Figures 5.21 through 5.25. Some of these include the bounding boxes and principal axis.

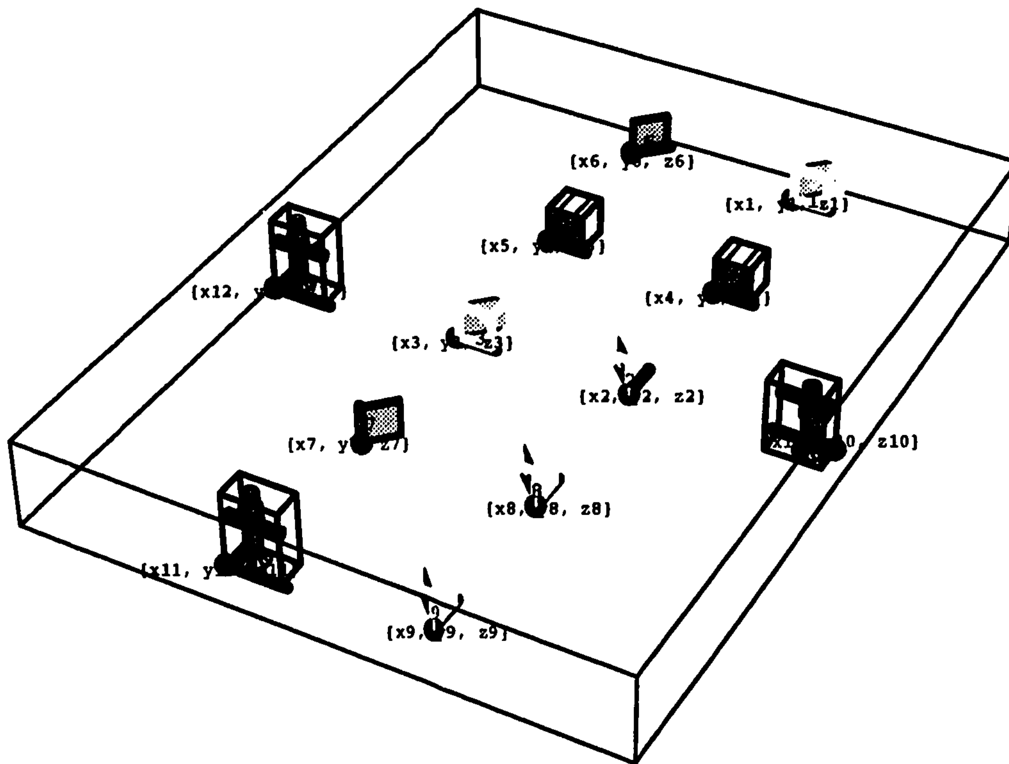


Figure 5.21: View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, -1, 1}.

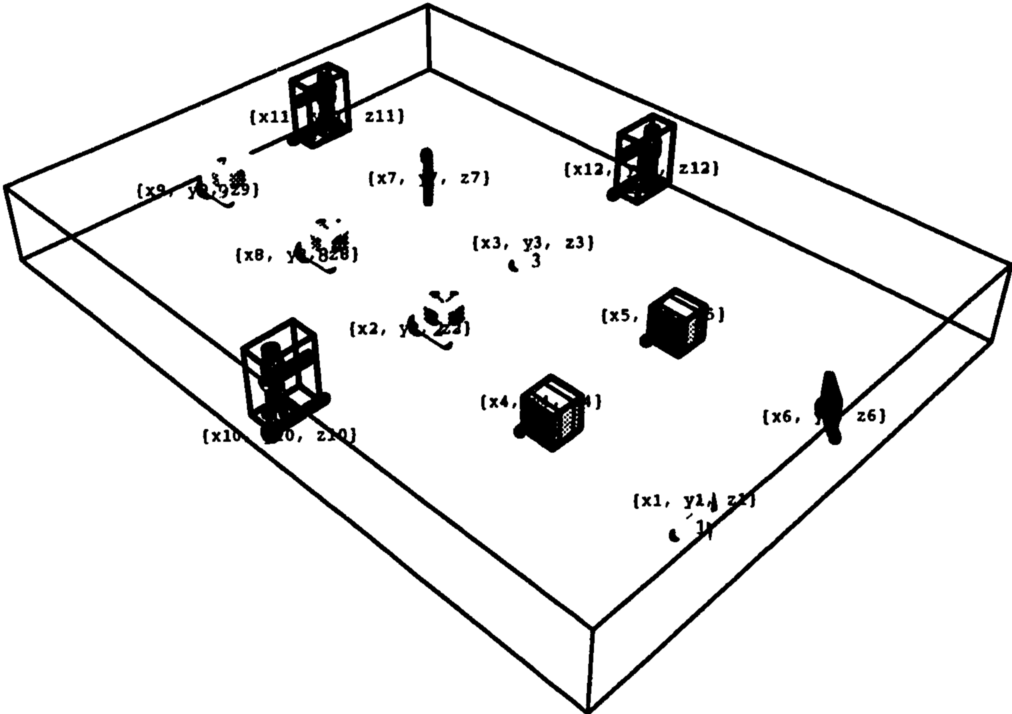


Figure 5.22: View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, 1, 1}.

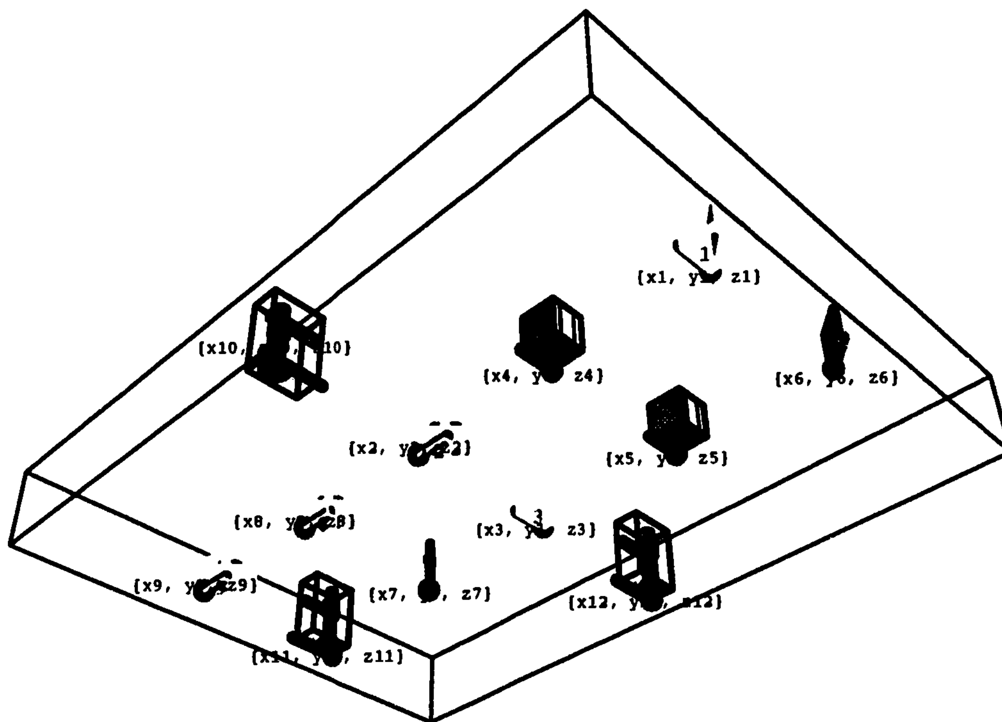


Figure 5.23: View of the Trip-Flop Architecture with Each Component Encased in its Bounding Box from Camera Position {1, 1, -1}.

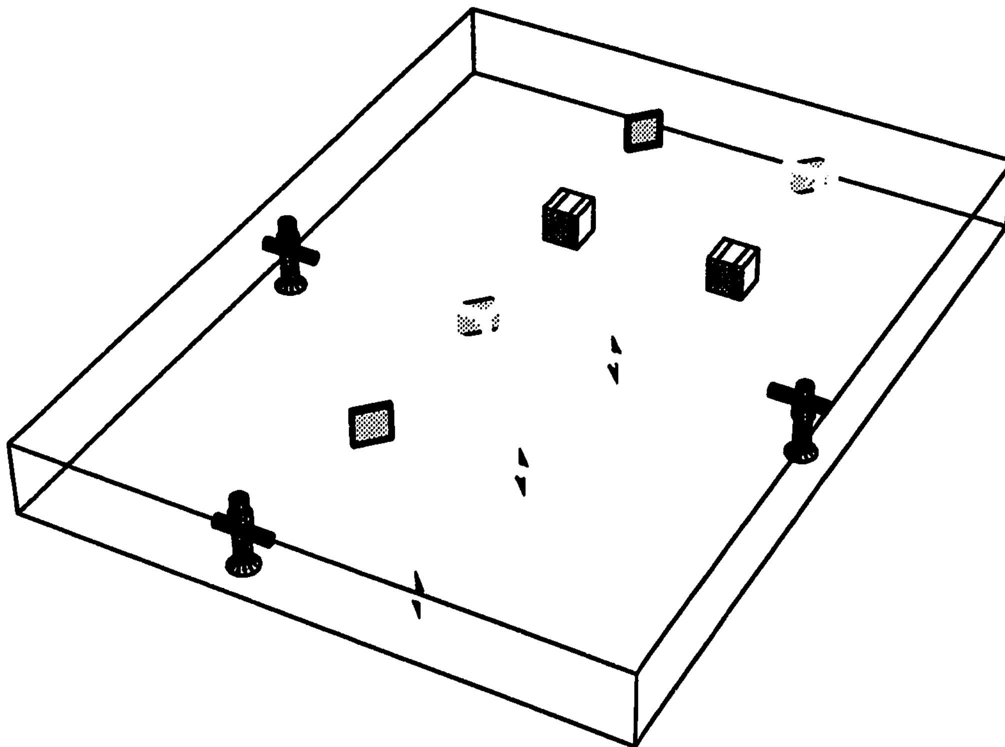


Figure 5.24: A View of the Placed and Oriented Trip-Flop Architecture without Bounding Box from Camera Position {1, -1, 1}

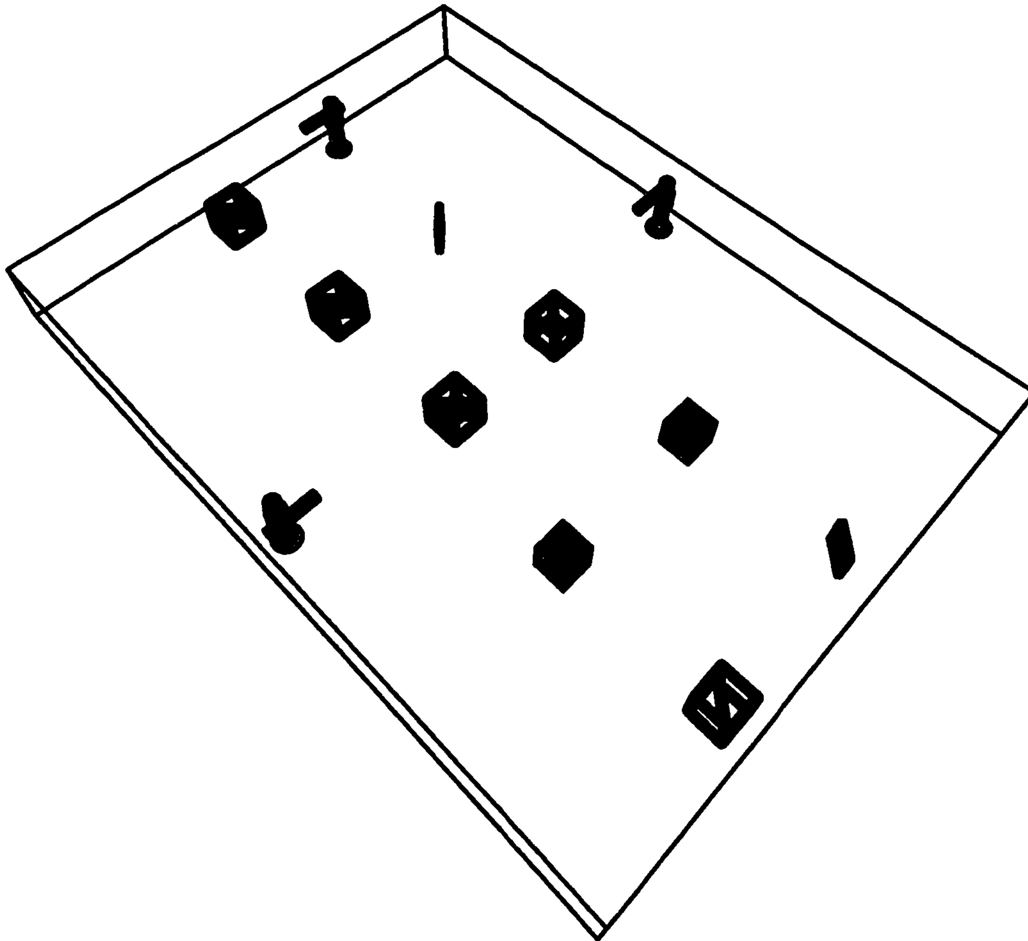


Figure 5.25: Another View of the Placed and Oriented Trip-Flop Architecture without Bounding Box

Variable (r)	Value	Variable (s)	Value	Variable (t)	Value
r_1	0.0	s_1	0.0	t_1	0.0
r_2	1.5708	s_2	0.0	t_2	0.0
r_3	0.0	s_3	0.0	t_3	0.0
r_4	0.0	s_4	0.0	t_4	0.0
r_5	0.0	s_5	0.0	t_5	0.0
r_6	0.785398	s_6	1.5708	t_6	0.0
r_7	−0.785398	s_7	−1.5708	t_7	0.0
r_8	1.5708	s_8	0.0	t_8	0.0
r_9	1.5708	s_9	0.0	t_9	0.0
r_{10}	−3.14159	s_{10}	0.0	t_{10}	0.0
r_{11}	0.0	s_{11}	0.0	t_{11}	0.0
r_{12}	0.0	s_{12}	0.0	t_{12}	0.0

Table 5.2: The Solution to the Orientation Constraints.

5.6 Handling Over/Under-Constrained Specifications

A common problem faced by an architect (a user of *OptiCAD*) is verifying whether the set of defined geometric constraints is complete and correct. There might be inconsistencies in the specification in the form of contradictory constraints and incomplete specifications.

Most of these constraint specification problems can be overcome by experience with designing a number of systems and following some simple design strategies. For example in specifying a position constraint between object x and y, it is a good idea to have already specified some position constraint for either x or y.

The current system does not provide enough support for the localization of an overly specified constraint system. It does however detect the presence of an inconsistency where a solution cannot be found.

Definition: When the resulting system has solutions in terms of one or more free variables, it is called an *under*-constrained system.

Under-constrained system is a result of an insufficient specification enabling a unique solution for the constraint equation. For example consider the system of four components shown in Figure 5.26.

Spatial relationships have been defined between components (C_1, C_2) and (C_3, C_4). However, no additional relationships have been defined. This results in an under constrained system.

Definition: When the resulting system cannot result in any feasible solution (i.e., solution space is ϕ), it is called an *over*-constrained system.

This is due to the presence of contradictory constraints. The system of four components shown in Figure 5.27 has a complete specification in the form of relationships marked by a, b, c, d, e and f. An additional relationship in the form of x introduces a contradiction in the specification and thus making it an over constrained system.

The user specified constraints need to be translated into an internal representation. The translation process must capture all the semantics of the high level specification.

In the current implementation, at any given level of abstraction in the architecture, all the specified constraints are translated into a set of simultaneous equations. These are solved by using a symbolic solution technique.

Under-specified systems result in solutions in terms of one or more independent variables. An attempt to "fix" the under-specified specification can be done by using any of the following techniques.

1. Pick any value from the domain of cartesian space for the independent variables. This will nullify the effect of under specification. Values for all the other variables can be found in terms of the newly assigned ones.

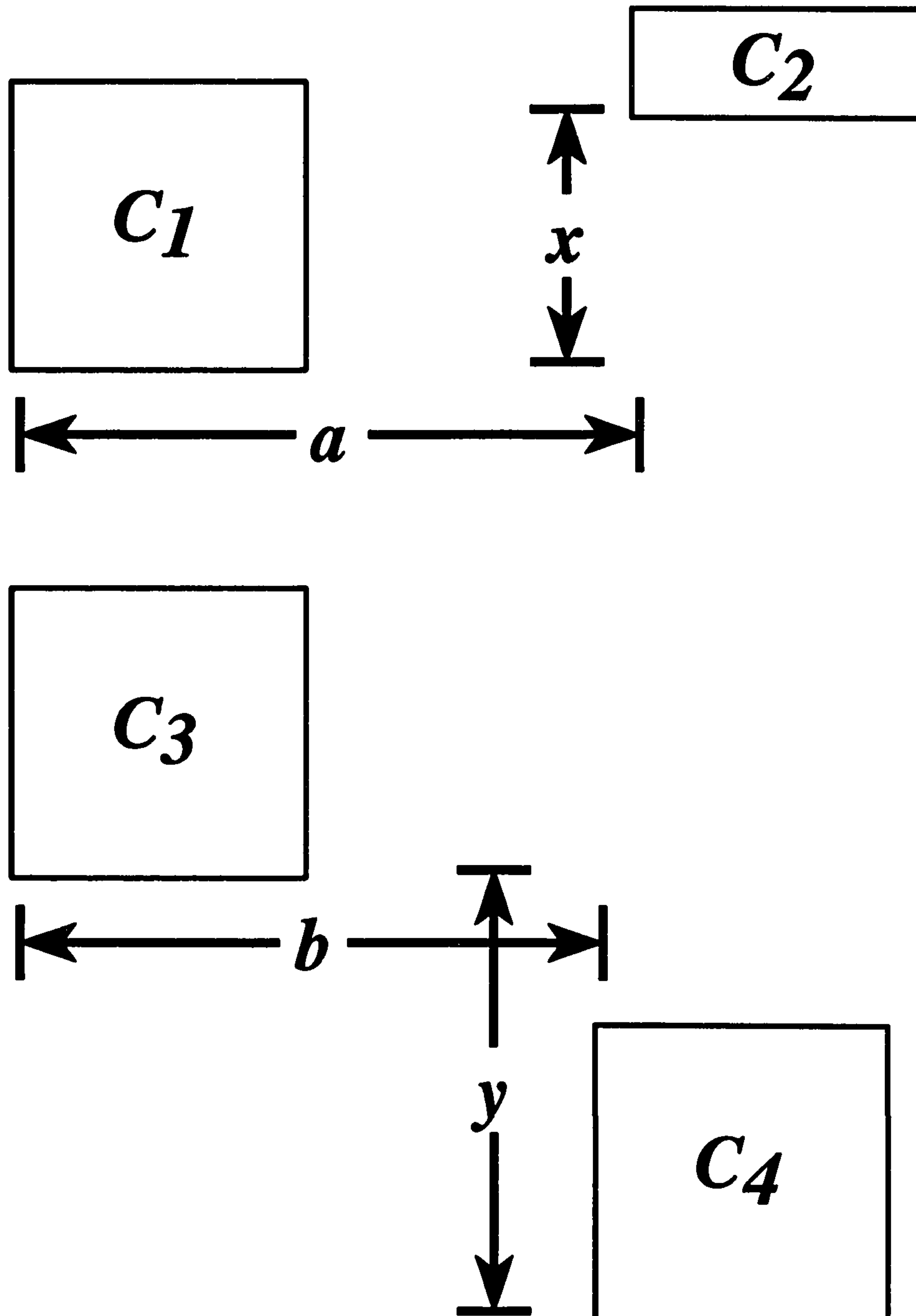


Figure 5.26: Example of an Under-Constrained System

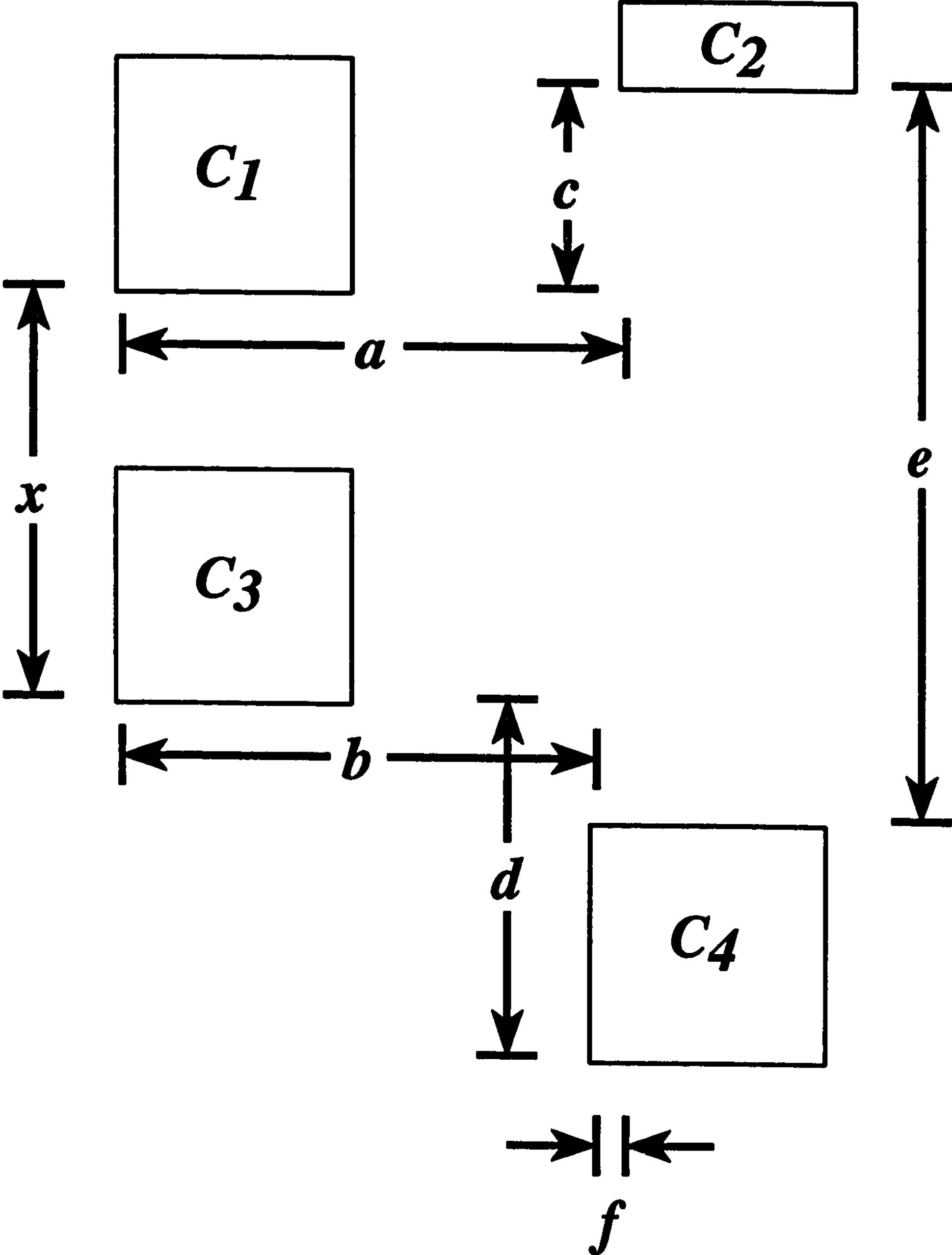


Figure 5.27: Example of an Over-Constrained System

2. Use more sophisticated measures/objectives to constrain the system. This is done by artificially adding more constraints. These constraints should not in any way change the functionality of the architecture, rather they should maintain its properties e.g. aspect-ratio.

3. Choose values for the independent variables so that the overall architecture is "optimized". This optimization could be in terms of minimizing the volume, reducing the clock speed of the architecture.

4. Decompose the existing architecture into smaller modules allowing the solution of smaller, individual problems.

The problem of over-specification arises from the existence of contradictory constraints. The removal of these "problematic" constraints may lead to a feasible solution. Several ways of handling over-constrained specifications are as follows.

1. The user enters each constraint one by one. The system can keep track of when the constraints become infeasible so that the user can identify the restricting constraint immediately.

2. When multiple constraints are given at once and it is realized that they are infeasible, then the system can start iteratively dropping one by one the constraints and stop when the remaining constraints can be satisfied. This could help in identifying the problem.

3. Divide the instance set into smaller manageable blocks and reformulate and solve for each block separately.

4. Resort to sophisticated techniques that keep the domain specific knowledge and try to eliminate the problematic constraints.

5.7 Displaying the Placed and Oriented Components

Earlier sections have dealt with automated ways of finding the places and orientations of components from a user provided high level specification.

Once the positions and orientations are fixed, the whole issue of 3D layout reduces to one of rendering graphics.

This section deals with the issue of displaying an architecture in 3D.

The approach is focused on providing scalable and object-oriented graphics. This further reduces the original problem to that of an incremental and extensible function **draw**[*component/assembly*]. Section 5.7.1 gives the details of such a function.

Section 5.7.2 deals with the provision of general purpose 2D graphics primitives and operations on 2D objects.

Section 5.7.3 further extends the 2D ideas into the more complex world of 3D space. 3D graphics primitives and general purpose operations on 3D objects are discussed.

A library of general purpose 3D graphics objects is presented in Section 5.7.4. This library contains readily instantiable objects. Such a library is also the basis for component modelling.

Section 5.7.5 describes a recursive algorithm that traverses a given hierarchy of components/assemblies, placing and orienting them as the traversal progresses.

5.7.1 Displaying Primitive Components - *draw[]* Function

The main purpose for the development of *OptiCAD* is to provide a system wherein an architect can describe his design and verify its functionality. It is impossible to

build every component into such a system. There is a clear need for an extensible system. The first step in this direction is the provision of mechanisms for maintaining a dynamic library of components.

Since there is a provision for the addition of new components, their shape needs to be described. This is provided in the form of a function **draw**[*component/assembly*]. This is an overloaded function, one definition for each component. Once a component is added to the component library, it is the responsibility of the component designer to create an icon and add it to the **draw[]** library.

If a component does not have an explicit **draw[]** associated with it, then the "default" **draw[]** definition is applied. This gives the component a cuboid icon of its size.

The function **draw[component/assembly]** can be called from any level of abstraction in the architecture hierarchy. It is responsible for returning a component's icon of the proper size.

The **draw[]** function of a component will return a customized icon for every instance of that component. For this it refers to some attributes of the component – reference point, vector, bounding box orientation, size. The default shape for the particular component is customized to the instance's requirements and the proper shape is produced.

5.7.2 2D Graphics

Although it is not enough to check an architecture by visual inspection, it is useful to examine it from different perspectives. This greatly facilitates the task of analyzing, synthesizing shapes and architectures.

Projections of the 3D architecture on to different planes requires the definition of 2D shapes and operations defined on those shapes.

OptiCAD has extensive support for 2D operations in the form of general routines. The complete details with algorithms are presented as a notebook in Part IV. This notebook contains the necessary routines to display, position and orient Graphics Objects in 2D. Numerous operations are provided such as:

- $translateBy[point[\{m,n\}]]$
- $rotateAround[angle[theta]]$
- $rotateAround[point[\{m,n\}], angle[theta]]$
- $rotateBy[angle[theta]]$
- $reflectThrough[line[point[\{x1,y1\}], point[\{x2,y2\}]]]$
- $scaleLocal[\{xScale, yScale\}]$
- $scaleOverall[scaleFactor]$

The notebook also discusses the following points.

Transformation of Points

Transformation of the point $\{x,y\}$ to $\{ax+cy, bx+dy\}$ can be accomplished by the transformation Matrix:

$$\mathbf{T} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{5.11}$$

Transformation of Straight Lines

A straight line can be represented by its two end points.

$\mathbf{L} = \{p_1, p_2\}$

Transformed line, L* according to the transformation T is [L . T]

The transformation of the line preserves:

1. **the midpoint according to [T].**
2. **parallelism between two lines.**
3. **Intersection of two lines.**

Transformations of Parallel Lines and Intersecting Lines

When a 2 × 2 matrix is used to transform a pair of *parallel lines*, the result is another pair of parallel lines. If the slopes of the two parallel lines are m_1 and m_2 where $m = m_1 = m_2$, then the slopes of the lines obtained after transformation will also be equal.

When a 2 × 2 matrix is used to transform a pair of intersecting straight lines, the result is also a pair of intersecting, straight lines. The point of intersection remains the same for the new intersecting lines.

Rotations

Any object represented as a list of vertex points can be rotated by multiplying it by a 2 × 2 matrix. More specifically, an object can be rotated by an angle of 90° in a counterclockwise direction by the transformation matrix:

$$[\mathbf{T}] = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \tag{5.12}$$

Similarly a 180° transformation is obtained by using the transformation:

$$[\mathbf{T}] = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \tag{5.13}$$

and a 270° rotation about the origin by using:

$$[\mathbf{T}] = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \tag{5.14}$$

In general, the transformation for a general rotation about the origin by an arbitrary angle θ is:

$$[\mathbf{T}] = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \tag{5.15}$$

Reflection

A pure two-dimensional rotation in the xy plane occurs entirely in the two-dimensional plane about an axis normal to the xy plane, a reflection is a 180° out into three space and back into two space about an axis in the xy plane. A reflection about the plane $y = 0$, the x-axis, is obtained by using the matrix:

$$[\mathbf{T}] = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{5.16}$$

Similarly, reflection about $x = 0$, the y-axis, is obtained by:

$$[\mathbf{T}] = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \tag{5.17}$$

A reflection about the line $y = x$ occurs for

$$[\mathbf{T}] = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{5.18}$$

and a reflection about the line $y = -x$ is given by:

$$[T] = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \tag{5.19}$$

Each of these matrices has a determinant that is identically -1. In general, if the determinant of a transformation matrix is identically -1, then the transformation produces a pure reflection.

Scaling

Scaling is controlled by the magnitude of the two terms on the primary diagonal of the matrix. If the matrix

$$[T] = \begin{pmatrix} s & 1 \\ 1 & s \end{pmatrix} \tag{5.20}$$

is used as an operator on the vertices of a polygon, a s times enlargement, or uniform scaling, occurs about the origin.

In general, if

$$[T] = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{5.21}$$

with $a = d$, $b = c = 0$, a uniform scaling occurs. If $a \neq d$, $b = c = 0$, a nonuniform scaling occurs. For uniform scaling, if $a = d > 1$, a uniform expansion occurs, i.e., the object gets larger. If $a = d < 1$, then a uniform compression occurs, i.e., the figure gets smaller.

Transformation Matrices for Different Primitive 2D Operations

Generalizing the 2×2 transformation matrix discussed earlier, the *general trans-*

formation matrix is now 3×3, i.e.,

$$[\mathbf{T}] = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ m & n & 1 \end{pmatrix} \tag{5.22}$$

where the submatrix

$$[\mathbf{T}] = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{5.23}$$

has exactly the same effect as revealed in previous discussions and m, n are the *translation factors* in the x and y directions respectively.

Summarizing the 2D-operations, following are some of the transformation matrices used for different 2D-transformations.

Translate by a Point $\{m, n\}$:

$$[\mathbf{T}] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ m & n & 1 \end{pmatrix} \tag{5.24}$$

Rotate Around the Origin by an Angle θ:

$$[\mathbf{T}] = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.25}$$

Rotate Around the Point $\{m, n\}$ by an Angle θ:

The transformation matrix is obtained as a result of the product of the three matrices:

1. Translate by the point $\{-m, -n\}$.
2. Rotate around the origin by the angle θ.
3. Translate by the point $\{m, n\}$.

Flip Around the x-axis:

$$[\mathbf{T}] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.26}$$

Flip around the y-axis:

$$[\mathbf{T}] = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.27}$$

Reflect through a straight line:

Assume that the straight line passes through the points $\{x_1, y_1\}$ and $\{x_2, y_2\}$. Calculate $c = \frac{x_1 \times y_2 - x_2 \times y_1}{x_1 - x_2}$ and $\theta = ArcTan\ ((y_2 - y_1)/(x_1 - x_2))$.

The transformation matrix is obtained as a result of the product of the following matrices:

1. Translate by the point $\{0, -c\}$.
2. Rotate around the origin by the angle $-\theta$.

3. Flip around the x-axis.

4. Rotate around the origin by the angle θ.

5. Translate by the point $\{0, c\}$.

Local Scaling:

The transformation matrix for scaling all x-coordinates by $xScale$ and y-coordinates by $yScale$ is:

$$[\mathrm{T}] = \begin{pmatrix} xScale & 0 & 0 \\ 0 & yScale & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.28}$$

Overall Scaling:

Transformation matrix for scaling all coordinates by $scaleFactor$ is:

$$[\mathrm{T}] = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & scaleFactor \end{pmatrix} \tag{5.29}$$

5.7.3 3D Graphics

To realize a geometric form of an architecture, extensive 3D graphics support is needed. This subsection is concerned primarily with specifying the supported preliminaries and operations that can be carried out on 3D graphics operations. Full details are included in Part IV of the thesis.

A point in 3D is represented as $\{x, y, z\}$. Transformations and matrices treat the matrix **T** as a geometric operator. An object **A** can be operated on by **T** to give **B** = **A T**

The interfaces of operations provided in the notebook are as follows.

1. **Reflection Through a Plane.**

```
reflectThrough[plane[a, b, c, d]]
```

2. **Rotate Around a Line with a Specified Angle**

```
rotateAround[
line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ],
angle[ theta ]
]
```

3. **Translation by an Amount.**

```
translateBy[ {l, m, n} ]
```

4. **Reflect Through the xy-plane.**

```
reflectThroughXY[ ]
```

5. **Reflect Through the yz-plane.**

```
reflectThroughYZ[ ]
```

6. **Reflect Through the xz-plane.**

```
reflectThroughXZ[ ]
```

7. **Rotate Around x-axis by an Angle θ**

```
rotateAroundXBy[ angle[θ] ]
```

8. **Rotate Around y-axis by an Angle θ**

```
rotateAroundYBy[ angle[θ] ]
```

9. **Rotate Around z-axis by an Angle θ**

```
rotateAroundZBy[ angle[θ] ]
```

10. **Shearing**

```
shear[{b, c, d, f, g, i}]
```

11. **Overall Scaling**

```
scaleOverall[ scaleFactor ]
```

12. **Local Scaling**

```
scaleLocal[ {xScale, yScale, zScale}]
```

A point $\{x, y, z\}$ in 3D space is represented as a four dimensional position vector i.e., $\{x', y', z', h\} = \{x, y, z, 1\}.[T]$.

The transformation from the homogenous coordinate system to ordinary coordinate system is: $\{x*, y*, z*, 1\} = \{x'/h, y'/h, z'/h, 1\}$.

Each operation has its corresponding transformation matrix. The generalized transformation matrix for 3D operations is as follows.

$$\mathbf{T} = \begin{pmatrix} a & b & c & p \\ d & e & f & q \\ g & i & j & r \\ l & m & n & s \end{pmatrix} \tag{5.30}$$

where,

1. The matrix
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & i & j \end{pmatrix} \tag{5.31}$$
produces a linear transformation like scaling, shearing, reflection and rotation.

2. The vector $\{l, m, n\}$ produces translation.

3. The vector
$$\begin{pmatrix} p \\ q \\ r \end{pmatrix} \tag{5.32}$$
produces a perspective transformation.

4. The value of $[s]$ produces overall scaling.

Every object has a bounding box. Initially, the bounding box is placed in the first octant with edges aligned to the coordinate axes. The bounding box has a reference point and a principal axis whose tail is the reference point. Every measure of the component is relative to the component's reference point. The principal axis is a unit vector in the x-direction of the local coordinate system.

The orientation is represented by $\{thetaX, thetaXY, thetaP\}$. The final orientation of the object is determined by the following steps.

1. The object is assumed to be in the first octant with its reference point at the origin.

2. Rotate the object around z-axis by an angle of thetaX. ($0 <= thetaX < 2\pi$).

3. Let L be a line passing through the origin in the xz-plane and perpendicular to the rotated principal axis.

4. Rotate the object around L by thetaXY ($0 <= thetaXY < 2\pi$). Hence thetaXY is the angle between the principal axis in its final orientation and the xy-plane.

5. Rotate the object around the principal axis by thetaP ($0 <= thetaP < 2\pi$).

The function **orient[]** will return the oriented object for a given object.

5.7.4 Generic 3D Graphics Shapes

Some of the generic shapes that have been constructed by using the graphic primitives to be used in designing icons for optical components are shown in Figures 5.28 through 5.41.

5.7.5 An Algorithm for Displaying Hierarchical Architectures

The algorithm for displaying optical architectures is as follows.

```
draw[object: instance of an architecture]

     If object is a primitive component then
                   draw[object]   <-- Primitive definition
     else
         for each object/component
     instance i in componentsOf[object] do
```

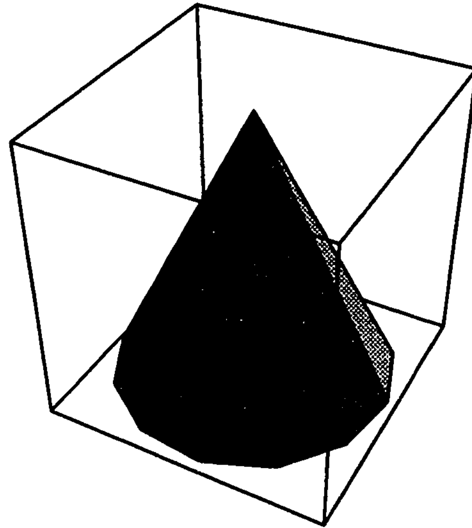


Figure 5.28: Generic Shape – Cone

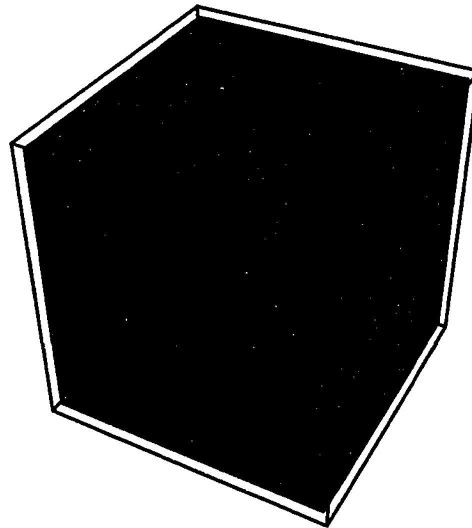


Figure 5.29: Generic Shape – Cuboid

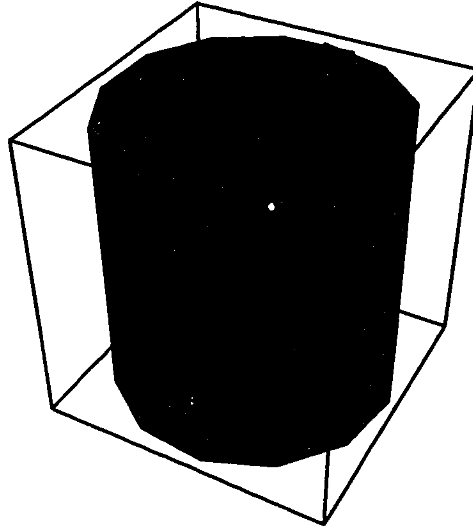


Figure 5.30: Generic Shape – Cylinder

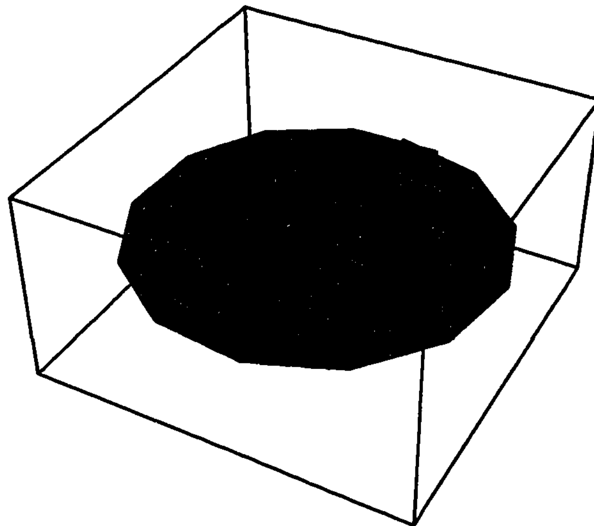


Figure 5.31: Generic Shape – Disk

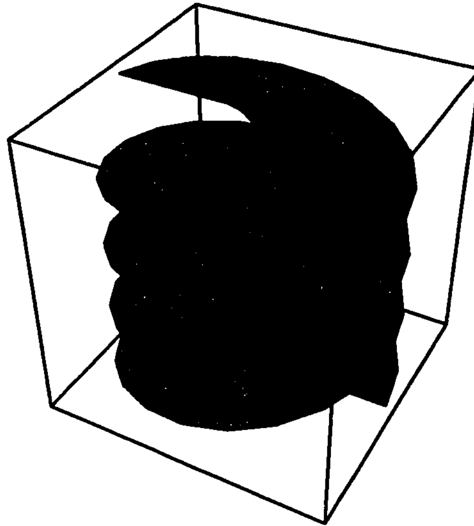


Figure 5.32: Generic Shape – Double Helix

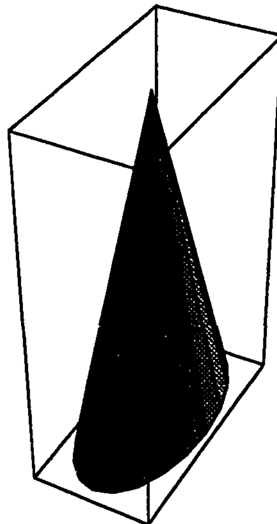


Figure 5.33: Generic Shape – Flat Cone

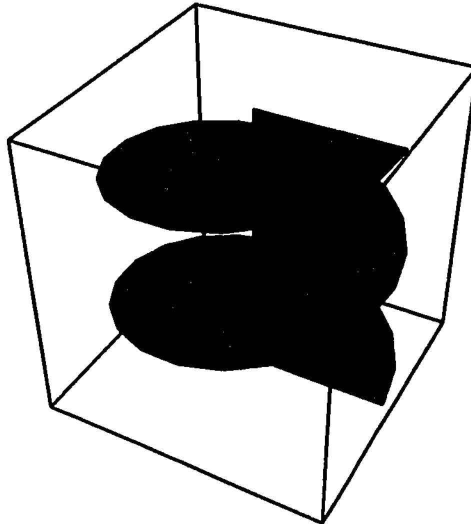


Figure 5.34: Generic Shape – Helix

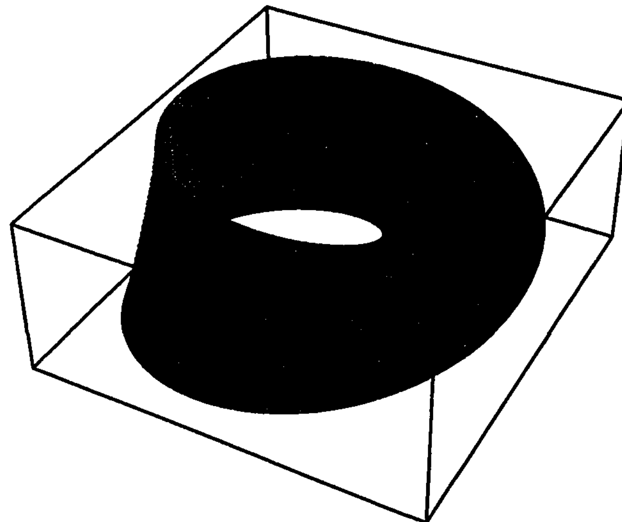


Figure 5.35: Generic Shape – Moebius Strip

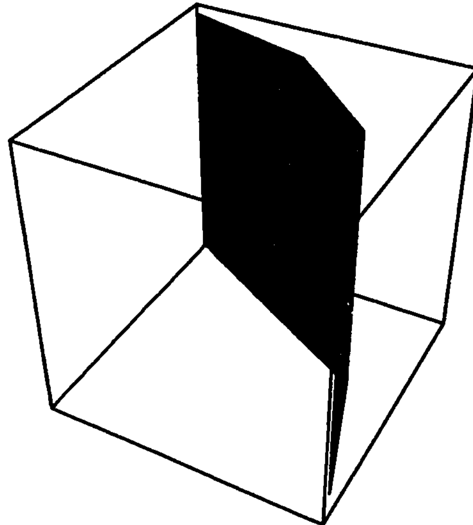


Figure 5.36: Generic Shape – Sliced Cylinder

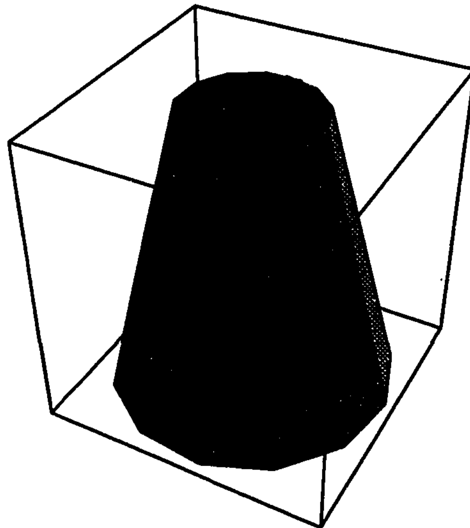


Figure 5.37: Generic Shape – Sliced Truncated Cone

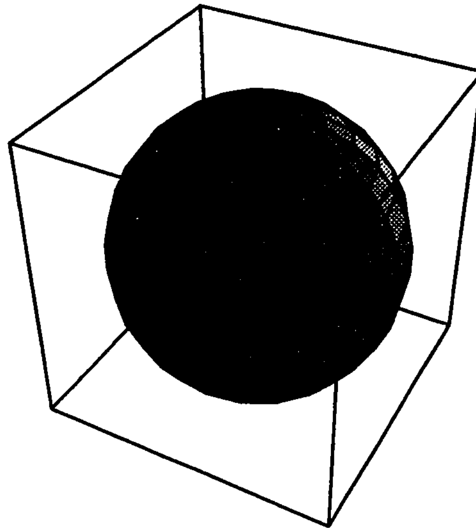


Figure 5.38: Generic Shape – Sphere

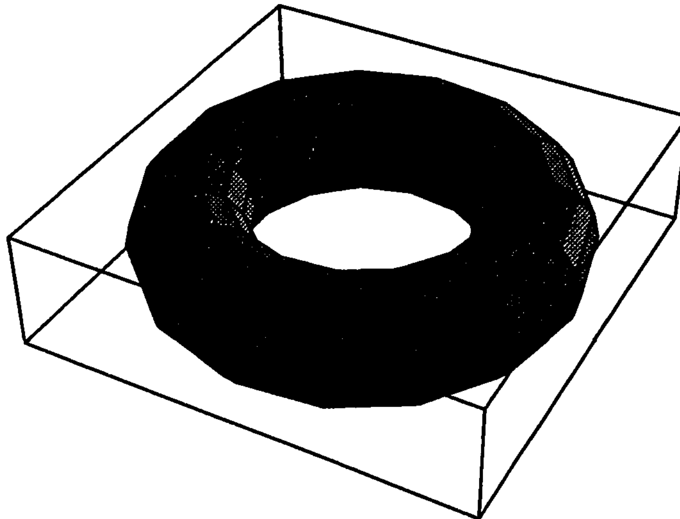


Figure 5.39: Generic Shape – Torus

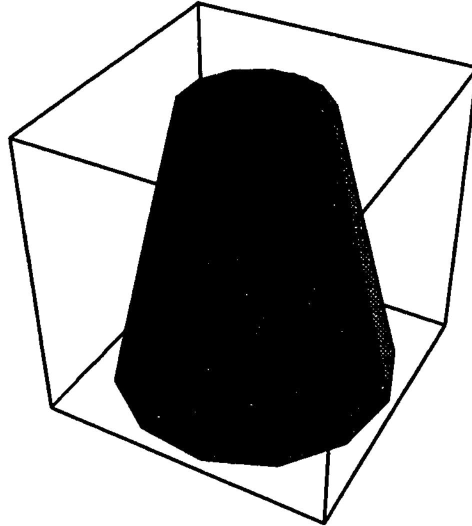


Figure 5.40: Generic Shape – Truncated Cone

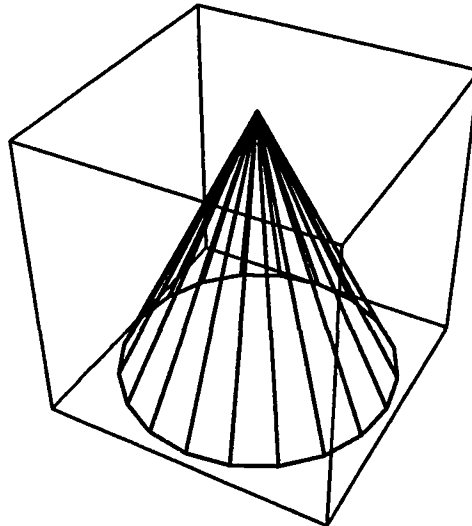


Figure 5.41: Generic Shape – Wire Cone

```
        - compute bounding box info for i;
        - compute orientation info for i
        - assemble an instance component ic;
        - draw[ic]
```

The above algorithm is recursive returning only if a primitive **draw[]** definition is encountered. Once **draw[]** is called for an architecture, recursive calls are made for each of the sub-assemblies and components. Further recursion is done until the primitive shapes are encountered. The final layout is assembled upon return of all calls.

5.8 Summary

Placement is one of the most important issues in building an optical architecture. This Chapter discusses some of the problems of placing and orienting optical components. The importance of constraints and their role in geometric modelling was presented. Various constraint specification and satisfaction techniques were outlined. The notion of bounding box of a component/assembly was introduced. Techniques for handling over/under constrained specifications were presented.

Bibliography

[1] Ghafoor A., Guizani M., and Sheikh S. "All-Optical Circuit-switched Multistage Interconnection Networks". *IEEE, Journal on Selected Areas of Communication*, 9(8):1218–1226, October 1991.

[2] Alferd V. Aho, Ravi Sethi, and Jeffery D. Ullman. *"Compilers: Principles, Techniques, and Tools"*. Addison-Wesley, 1986.

[3] Nabeel Al-Mosli. "Design & Implementation of a 3D-Graphics System: An Interactive Hierarchical Modelling Approach". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, June 1995.

[4] John Backus. "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs". *Communications of the ACM*, 21(8):613–641, August 1978.

[5] Roger Bailey. *"Functional Programming with HOPE"*. Ellis Horwood Series in Computers and Their Applications. Ellis Horwood ltd., 1990.

[6] Henri E. Bal and Dick Grune. *"Programming Language Essentials"*. Addison-Wesley, 1994.

[7] Partha P. Banerjee and Ting-Chung Poon. *"Principles of Applied Optics"*. Asken Associates Incorporated Publishers, 1991.

[8] D. Barnhard. "Lens Lab 3D". Technical report, 1993.

[9] Alain Bergeron, Henri H. Arsenault, and Denis Gingras. "Single-Rail Translation-Invariant Optical Associative Memory". *Applied Optics*, 34(2):352–357, January 1995.

[10] S. Bian, K. Xu, and J. Hong. "Near Neighbouring Neurons Interconnected Neural Network". *Optics Communications*, 76(3,4):199–202, May 1990.

[11] K. Brenner and A. Huang. "Optical Implementation of the Perfect Shuffle Interconnection". *Applied Optics*, 27(1):135–137, January 1988.

[12] L. Z. Cai and T. H. Chao. "Optical Image Subtraction using a LCTV SLM and White Light Imaging Grating Interferometer". *Journal of Modern Optics*, 37(6):1127–1138, 1990.

[13] H. J. Caulfield. "Improved Relaxation Processor for Parallel Solution of Linear Algebric Equations". *Applied Optics*, 29(20):2978–2981, July 1990.

[14] H. J. Caulfield, J. Shamir, J. E. Ludman, and P. Greguss. "Reversibility and Energetics in Optical Computing". *Optics Letters*, 15(16):912–914, August 1990.

[15] T. J. Cloonan et al. "Architectural Issues Related to the Optical Implementation of an EGS Network based on Embedded Control". *Optical Quantum Electronics*, 24:S415–S442, 1992.

[16] T. J. Cloonan and M. J. Herron. "Optical Implementation and Performance of One-Dimensional and Two-Dimensional Trimmed Augmented Data-Manipulator Networks for Multi-Processor Computer Systems". *Optical Engineering*, 28(4):305–314, 1989.

[17] McAulay Alastair D. *"Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers"*. John Wiley and Sons Inc., 1991.

[18] S. Dasgupta. "The Structure of Design Processes". *Advances in Computers*, 28:1–67, 1989.

[19] A. K. Datta, A. Basuray, and S. Mukhopadhyay. "Arithmetic Operations in Optical Computations Using a Modified Trinary Number System". *Optics Letters*, 14(9):426–428, May 1989.

[20] E. R. Davies. *"Machine Vision: Theory, Algorithms, Practicalities"*. Academic Press, 1990.

[21] L. Dron. "Multiscale Veto Model: A Two-Stage Analog Network for Edge Detection and Image Reconstruction". *International Journal of Computer Vision*, 11(1):45–61, August 1993.

[22] P. Egbert and W. Kubitz. "Application Graphics Modeling Support Through Object Orientation". *COMPUTER*, pages 84–90, Oct 1992.

[23] M. T. Fatehi. "Optical Flip-Flops and Sequential Logic Circuits Using LCLV". *Applied Optics*, 23(13):2163–2171, 1 July 1984.

[24] C. Ferrera and C. Vazquez. "Anamorphic Multiple Matched Filter for Character Recognition, Performance with Signals of Equal Size". *Journal of Modern Optics*, 37(8):1343–1354, 1990.

[25] M. A. Flavin and J. L. Horner. "Average Amplitude Matched Filter". *Optical Engineering*, 29(1):31–37, January 1990.

[26] D. Foley and A. van Dam. *"Fundamentals of Interactive Computer Graphics"*. Addison-Wesley, 1990.

[27] A. Ghosh and P. Paparao. "Matrix Preconditioning: A Robust Operation For Optical Linear algebra Operations". *Applied Optics*, 26(14):2734–2737, July 1987.

[28] G. R. Gindi, A. F. Gmitro, and K. Parthasarathy. "Hopfield Model Associative Memory with Nonzero Diagonal Terms in Memory Matrix". *Applied Optics*, 27(1):129–134, January 1988.

[29] K. Hara, K. Kojima, K. Mitsunga, and K. Kyuma. "Optical Flip-Flop Based on Parallel Connected Algaas/gaas PNPN Structure". *Optics Letters*, 15(13):749–751, July 1990.

[30] P. Henderson. "Functional Programming, Formal Specification, and Rapid Prototyping". *IEEE Transactions on Software Engineering*, SE-10(5):241–250, 1986.

[31] H. S. Hinton et al. "Free-Space Digital Optical Systems". *Proceedings of the IEEE*, 82(11):1632–1649, November 1994.

[32] Ellis Horowitz and Sartaj Sahni. *"Fundamentals of Computer Algorithms"*. Computer Science Press, 1978.

[33] H. Huang, L. Lilu, and Z. Wang. "Parallel Multiple Matrix Multiplication Using an Orthogonal Shadow Casting and Imaging System". *Optics Letters*, 15(19):1085–1087, October 1990.

[34] M. N. Islam. "All Optical Cascadable NOR Gate With Gain". *Optics Letters*, 15(8):417–419, April 1990.

[35] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[36] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[37] J. Jahns and B. A. Brumback. "Integrated Optical Shift and Split Model Based on Planar Optics". *Optics Communications*, 76(5,6):318–320, May 1990.

[38] J. Jahns and M. Murdocca. "Crossover Networks and their Optical Implementations". *Applied Optics*, 27(15):3155–3160, August 1988.

[39] J. Jahns and S. J. Walker. "Imaging with Planar Optical Systems". *Optics Communications*, 76(5,6):313–317, May 1990.

[40] H. F. Jordan. *"Digital Optical Computers at Boulder, Center for Optoelectronic Computing Systems"*. University of Colorado, Boulder, 1991.

[41] A. Kemper and M. Wallrath. "An Analysis of Geometric Modeling in Database Systems". *ACM Computing Surveys*, 19(1):47–91, March 1987.

[42] E. Kerbis, T. J. Cloonan, and F. B. McCormick. "An All-Optical Realization of a 2 × 1 Free-Space Switching Node". *IEEE Photon Technology Letters*, 2(8):600–602, August 1990.

[43] B. Kliewer. "HOOPS: Powerful Portable 3D-Graphics". *BYTE*, pages 193–194, July 1989.

[44] Kubota Pacific Computer Inc. *"Dore Reference Manual"*, 1991.

[45] A.O. Lafi. "The Design and Implementation of a Structured Programming Language for the Description of Optical Architectures". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, January 1992.

[46] P. Lalanee, J. Taboury, and P. Chavel. "A Proposed Generalization of Hopfields Algorithm". *Optics Communications*, 63(1):21–25, July 1987.

[47] A. L. Lentine et al. "Symmetric Self-Electrooptic Effect Device: Optical Set-Reset Latch, Differential Logic Gate and Differential Modulator/Detector". *IEEE Journal of Quantum Electonics*, 25(8):1928–1936, August 1989.

[48] John R. Levine, Tony Mason, and Doug Brown. *"Lex & Yacc"*. OReilly & Associates, Inc., 1992.

[49] S. Lin, L. Liu, and Z. Wang. "Optical Implementation of the 2D-Hopfield Model for a 2D-Associative Memory". *Optics Communications*, 70(2):87–91, Feburary 1989.

[50] A. W. Lohmann, W. Stroke, and G. Stucke. "Optical Perfect Shuffle". *Applied Optics*, 25:1530, 1986.

[51] Guizani M. "Picosecond Multistage Interconnection Network Architectures for Optical Computing". *Applied Optics*, 33(8):1587–1599, March 1994.

[52] M. A. Memon, S. Ghanta, and S. Guizani. "An Optical Architecture for Edge Detection". In *Seventh IASTED International Conference on Parallel and Distributed Computing and Systems.* Georgetown University, Washington D.C., 1995.

[53] F. L. Miller, J. Maeda, and H. Kubo. "Template Based Method of Edge Linking using a Weighted Decision". In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1808–1815, 1993.

[54] M. Mirsalehi and T. K. Gaylord. "Analytic Expressions for the Sizes of Logically Minimized Truth Tables for Binary Addition and Subtraction". *Applied Optics*, 29(23):3339–3344, August 1990.

[55] R. L. Morrison. "Symmetries that Simplify the Design of Spot Array Phase Gratings". *Journal of Optical Society of America A*, 1992.

[56] S. Mukhopadhyay. "An Optical Conversion System: From Binary to Decimal and decimal to Binary". *Optics Communications*, 76(5,6):309–312, May 1990.

[57] M. Murdocca. *"A Digital Design Methodology for Optical Computing".* The MIT Press, 1990.

[58] V. K. Murty. "Exact Parallel Matrix Inversion Using para-Hensel Codes with Systolic Processors". *Applied Optics*, 27(10):2022–2024, May 1988.

[59] J. A. Neff. "Major Initiatives for Optical Computing". *Optical Engineering*, 26(1):002–009, January 1987.

[60] M. Oita, J. Ohta, S. Tai, and K. Kyuma. "Optical Implementation of Large Scale Neural Networks Using a Time Division Multiplexing Technique". *Optics Letters*, 15(4):227–229, Feburary 1990.

[61] D. V. Pentelic. "Optical Computation of Determinants". *Optics Communications*, 64(5):421–424, 1987.

[62] D. Peri. "Optical Implementation of a Phase Retrieval Algorithm". *Applied Optics*, 26(9):1782–1785, May 1987.

[63] J.P. Pratt. *"HATCH Users Manual".* Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[64] J.P. Pratt and V.P. Heuring. "A Methodology for the Design of Continuous-Dataflow Synchronous System". Technical report, Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[65] M. E. Prise et al. "Design of an Optical Digital Computer". In W. Firth, N. Peyhambarian, and A. Tallet, editors, *Optical Bistability IV*, pages C2–15–C2–18. Les Editions de Physique, 1988.

[66] A. Requicha. "Representation for Rigid Solids: Theory, Methods and Systems". *ACM Computing Surveys*, 12(4):437–463, Dec 1980.

[67] Bahaa E. A. Saleh and Malvin Carl Teich. *"Fundamentals of Photonics"*. John Wiley & Sons Inc., 1991.

[68] J. M. Senior and S. D. Cusworth. "Wavelength Division Multiplexing in Optical Fiber Sensor Systems and Networks: A Review". *Optics and Laser Technology*, 22(2):113–126, 1990.

[69] Ravi Sethi. *"Programming Languages: Concepts and Constructs"*. Addison-Wesley, 1989.

[70] Ben A. Sijtsma. "Requirements for a Functional Programming Enviornment". In Rogardt Heldal, Carsten Kehler, and Philip Wadler, editors, *Functional Programming, Glasgow 1991*, pages 339–346. Springer-Verlag, 1992.

[71] H. A. Simon. *"Science of the Artificial"*. The MIT Press, second edition, 1981.

[72] WRI Staff. "*MathLink* External Communication in *Mathematica*". Technical report, WRI, 1991.

[73] WRI Staff. "The 3-Script File Format". Technical report, WRI, 1991.

[74] WRI Staff. "The *Mathematica* Compiler". Technical report, WRI, 1991.

[75] WRI Staff. "Guide to Standard *Mathematica* Packages". Technical report, WRI, 1993.

[76] N. Streibl. "Beam Shaping with Optical Array Generators". *Journal of Modern Optics*, 36(12):1559–1573, 1989.

[77] Boleslaw K. Szymanski, editor. *"Parallel Functional Languages and Compilers"*. ACM Press Frontier Series. Addison-Wesley, 1991.

[78] David A. Turner, editor. *"Research Topics in Functional Programming"*. The UT Year of Programming series. Addison-Wesley, 1990.

[79] U.S. Air Force. *"VHDL Users Manual"*, 1985.

[80] P. Wegner. "Dimensions of Objected-Oriented Modeling". *COMPUTER*, pages 12–21, Oct 1992.

[81] B. S. Wherret, S. Desmond Smith, F. A. P. Tooley, and A. C. Walker. "Optical Components for Digital Optical Circuits". *Future Generation Computer Systems*, 3:253–259, 1987.

[82] D. R. J. White, K. Atkinson, and J. D. M. Osburn. "Taming EMI in Microprocessor Systems". *IEEE Spectrum*, 22(12):30–37, December 1990.

[83] P. Wisskirchen. *"Object-Oriented Graphics"*. Springer-Verlag, 1990.

[84] S. Wolfram. *"Mathematica: A System for Doing Mathematics by Computer"*. Addison-Wesley, second edition, 1991.

[85] Amnon Yariv. *"Optical Electronics"*. Saunders College Publishing: HBJ College Publishers, 1991.

[86] E. Yee and J. Ho. "Neural Network Recognition and Classification of Aerosol Particle Distributions Measured with a Two-Spot Laser Velocimeter". *Optics Communications*, 74(5):295–300, January 1990.

[87] L. Zhang and L. Liu. "Incoherent Optical Implementation of 2D-Complex Discrete Fourier Transform and Equivalent 4-F System". *Optics Communication*, 74(5):295–300, January 1990.

A System for Prototyping Optical Architectures

Volume II: Concepts & Design

BY

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

December 1995

Preface

OptiCAD is a Computer Aided Design (CAD) system for designing optical architectures. It was designed and implemented to facilitate the design and verification of optical architectures.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system omitting the implementation details. It is however a complete document describing critical issues, design decisions and algorithms employed.

The second part contains several optical architectures designed and described using *OptiCAD*. These have been carefully chosen so as to illustrate most of the features of *OptiCAD*. This part essentially contains the user's view of the system.

The third part contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. Each notebook is presented as a self contained chapter. These include issues such as component modeling, component libraries, simulator and different kinds of analyses.

A rich collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. These utilities facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to be useful in other domains.

Reading through Parts III and IV is by itself expected to be an illustrative and interesting exercise in appreciating the power of functional programming in general and *Mathematica* in particular. The code is not necessarily efficient. Whenever there was a choice between efficiency and readability/clarity, the latter was preferred.

Part V is a reference manual for *OptiCAD* and the *OHDL* language.

Chapter 6

Modelling of Optical Components

Chapter Abstract

The notion of a component model is given. Different types of models and modelling is discussed. The use of component models and their use is provided. Some models of optical components are presented.

6.1 Introduction

The concept of a model and the activity of model building is fundamental to various engineering fields. In this Chapter, an attempt is made to compile information about optical components and propose models to render the *OptiCAD* system readily usable. In this Section, the concepts of model and model-building are presented. Section 6.2 reviews various theories of light. Section 6.3 discusses some information coding techniques. Section 6.4 presents some component models. Section 6.5 discusses vendor supplied components' models.

6.1.1 Concept of a Model

The term *model* refers to a structure or setup that has been purposely built to exhibit features and characteristics of some other objects or systems. Usually, only some of the features and characteristics will be retained in the model depending upon the use to which it is to be put.

Models may be *concrete* or *abstract*. Concrete models are physical models such as a model aircraft. They are generally used to build prototypes. Abstract models are mathematical i.e., algebraic symbolism is used to capture the internal relationships in the object being modelled.

Understanding of any system is necessary to maintain it, trouble shoot it, fix it, improve/adapt it to varying/changing environments. Understanding a system involves the identification of fundamental objects and grasping the relationships among them. Many engineering or applied fields fall back on mathematical techniques or models. Mathematical models – involves a set of mathematical relationships that may capture the semantics of real world phenomena/operations.

6.1.2 Motives for Model Building

The model building process consists of extracting entities of a system. This involves identifying the relationships between the different entities that make up the system. Revelation of these deeper relationships leads to better understanding of the overall system.

Simple and tractable models can be subjected to analyses that may or may not be possible/feasible/desirable on an actual system. Some analyses on actual systems may involve destroying some entities of the system. This is not feasible in many cases.

Since the model tends to be generally simpler than the actual system, it may be easier to carry out analysis on it.

Model building allows carrying out different experiments. The model can be subjected to different scenarios and the results can be compared with expected values. This kind of analysis (called *what-if analysis*) gives a better understanding of the overall system and allows prediction.

Since the model is general, all of the relationships between its entities need to be captured. The main focus of model building is on inherent relationships and not on data.

6.1.3 What is Modelling?

The activity of constructing one or more models for an object/system is called *modelling.*

Different models may offer different advantages for accuracy, robustness, cost, analytical tractability, simplicity, and theoretical basis. Usually, these features are at odds with each other. No single model can suit all different situations. So there is a need to come up with several different kinds of models so that they can be cross-validated and an appropriate model can be selected for application to the situation on hand.

6.1.4 A Modelling Approach for Optical Components

Every optical component will have:

- a set of attributes that take values from defined domains.

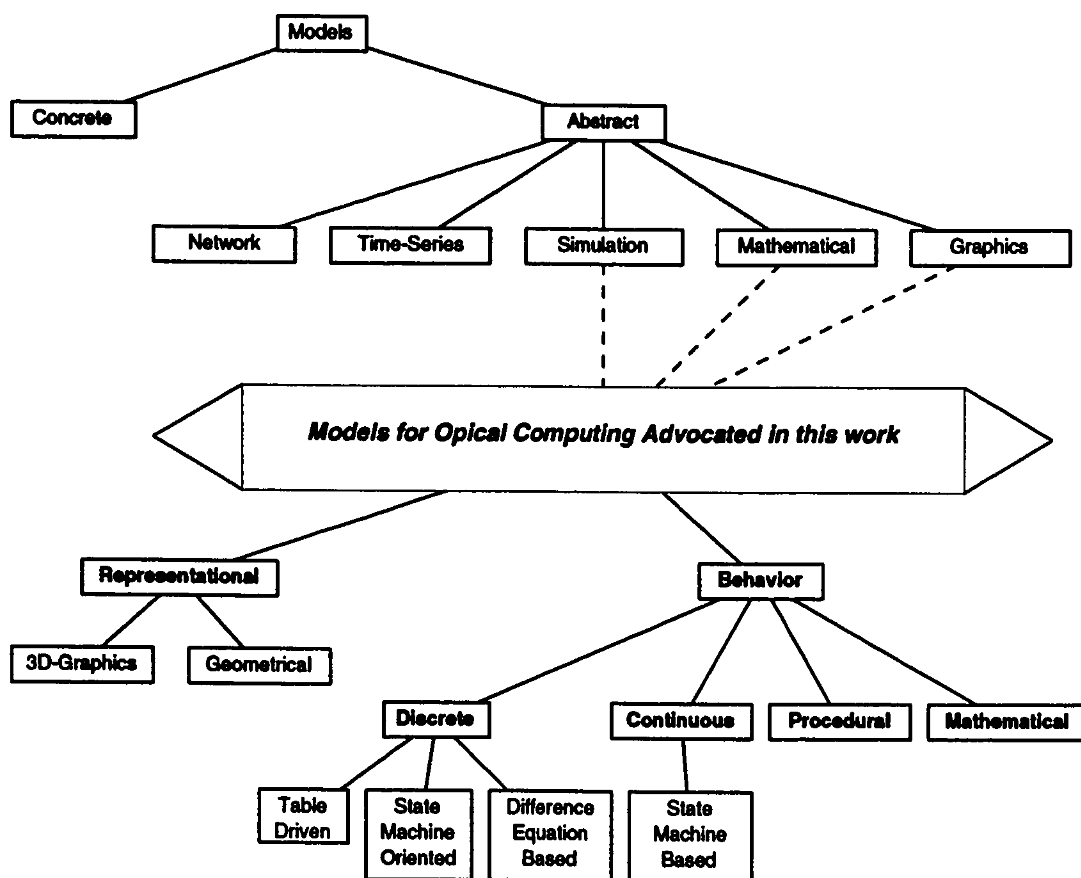


Figure 6.1: Different Types of Models and their Refinement

- constraints involving different attributes.

- a **representational model** which defines all aspects related to form, structural, graphical and geometrical aspects. This model is needed to represent the following features of a component.

  - 3D Graphics
  - Solid Modelling
  - Ray Tracing
  - Surface Modelling

- a **behavioral/functional model**. This could be either a mathematical models, state-oriented model, or procedural model derived from the fundamental principles of the theory of light. It may be continuous or discrete depending on the need and application.

Basic issues of representational modelling were covered in the placement Chapter using the 2D and 3D graphics operations and basic shapes. The 3D geometrical models for various components will be developed in this Chapter.

For each component so far as behavioral and functional modelling is concerned, either a set of mathematical equations or a state transition model will be presented. Section 6.4 presents an example of a model.

6.2 Theories of Light

Prior to embarking on modelling components' behavior, which is the study of interaction of light with matter, there is a need to understand the theories of light.

These theories attempt to explain different phenomena exhibited by light to varying degrees of satisfaction. Each theory has its own postulates and laws. Each theory strives to explain the nature of light. Increased sophistication of a theory leads to better understanding of the optical phenomenon exhibited by light.

Ray optics is the study of light when light is considered to be an idealized geometric line, obeying the laws of geometry. For this reason, it is also called geometric optics at times. Ray optics is insufficient to explain phenomena like diffraction.

Wave optics is a generalization of ray optics which can cope up with diffraction and interference.

Beam optics is better equipped for image processing applications.

The whole area of storing patterns and recalling them through holography is rooted in *fourier optics.*

A generalization of wave theory is *electro-magnetic optic theory* in which light is considered/treated like any other form of electro-magnetic radiation.

Most of this Section is a compilation from various sources [67, 7, 85]. The intent is to summarize the essence of each theory from the view point of a modeler (external view). The interested reader is referred to classic sources like [67, 7, 85] for more details and formal analyses (justification/internal view).

For each theory, we attempt to state the postulates of the theory, the laws of the theory and the mathematical model. Different components whose behavior can be explained by the theory are presented.

In its current form, the implemented (as well as modelled) components are mostly based on geometric aspects, and hence geometric optics. Future work includes extending the sophistication of models and the accuracy of the simulation. The next few subsections take up different theories in turn.

6.2.1 Ray Optics

Ray optics is the simplest of all the theories of light. Light travels in different media according to geometric rules. Ray optics is considered with the location and direction of light rays. It is useful in studying image formation.

Postulates of Ray Optics

- Light travels in the form of rays. The rays are emitted by light sources and can be observed when they reach an optical detector.
- An *optical medium* is characterized by a quantity $n \geq 1$, called the **refractive index**. The refractive index is the ratio of the speed of light in free space c_0 to that in the medium c. Therefore the time taken by light to travel a distance d equals $d/c = nd/c_0$. It is thus proportional to the product nd, known as the **optical path length**.

Principles and Laws

1. **Fermat's Principle**

   Light travels along the path of least time.

2. **Hero's Principle**

   In a homogenous medium, light rays travel in straight lines.

3. **Law of Reflection**

   The reflected ray lies in the plane of incidence. The angle of reflection equals the angle of incidence.

4. **Law of Refraction**

The refracted ray lies in the plane of incidence.

The angle of refraction (θ_1) and the angle of incidence (θ_2) and relation by *snell's law.*

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{6.1}$$

where n_1 and n_2 are the refractive indices of the two media respectively.

Two cases arise from the values of n_1 and n_2.

(a) *External refraction*: $(n_1 < n_2) \rightarrow$ bending closer to the normal.

(b) *Internal refraction*: $(n_2 > n_1) \rightarrow$ bending away from the normal.

Two subcases arise.

i. *refraction* if $\theta_1 < \theta_c$

ii. *Total internal reflection* – back into the same medium if $\theta_1 > \theta_c$

Here $\theta_c = ArcSin(\frac{n_1}{n_2})$ called the *critical angle* (or *Brewster angle*).

Mathematical Model

Propagation of light within a medium follows Fermat's principle:

$$\delta \int_A^B n(r)ds = 0 \tag{6.2}$$

Behavior of a ray at a boundary between two media follows two dictates of Snell's law applied to the tangential plane to the surface between the two media at the point of incidence.

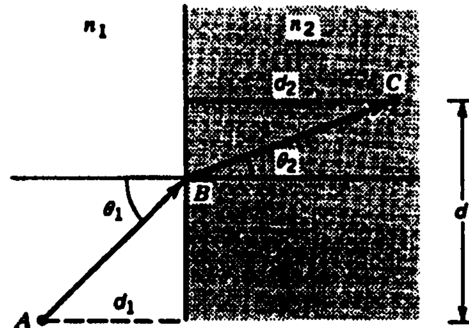


Figure 6.2: Ray Travelling from One Medium to Another

Components

Mirrors – characterized by the reflecting surface and its shape function.

A **planar mirror** reflects the rays originating from a point P_1 such that the reflected rays appear to originate from a point P_2 (see Figure 6.3) behind the mirror called the *image*.

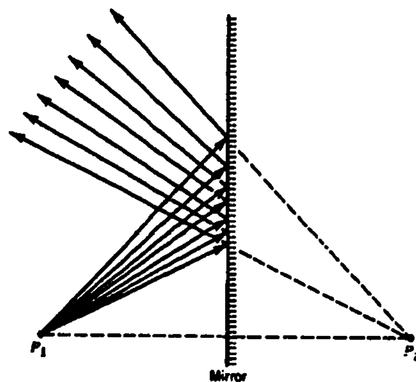


Figure 6.3: Reflection from a Planar Mirror

The surface of a **paraboloidal mirror** is a paraboloid of revolution. It has the property of focusing all incident rays parallel to its axis to a single point called the **focus**. The distance $\overline{PF} = f$ (seen in Figure 6.4) is called the **focal length**. Paraboloidal mirrors can also be used to make parallel beams of light from a single point source.

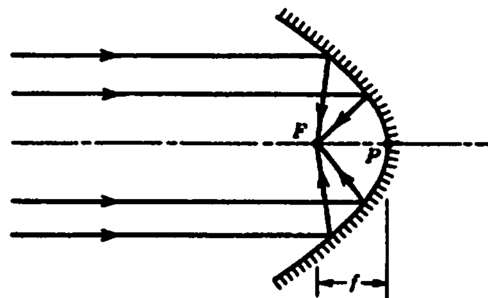


Figure 6.4: Focusing of Light by a Paraboloidal Mirror

An **elliptical mirror** reflects all the rays emitted from one of its two foci, e.g. P_1, and images them onto the other focus, P_2 as shown in Figure 6.5.

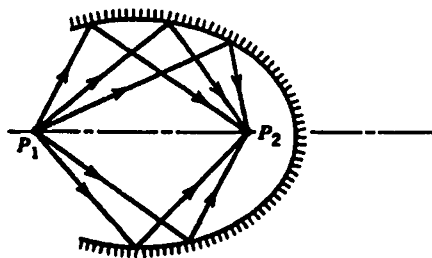


Figure 6.5: Reflection from an Elliptical Mirror

A **spherical mirror** has the property of reflecting parallel beams of light so that they meet the axis at different points. The envelope of the reflected beams

(shown as a dashed curve in Figure 6.6) is called the **caustic curve.**

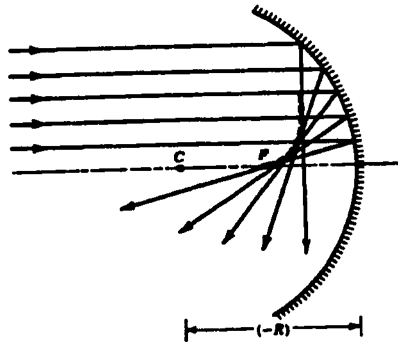


Figure 6.6: Reflection of Parallel rays from a Concave Spherical Mirror

Prisms

There are numerous types based on the kinds of cross-section geometry. A prism of apex angle α (as shown in Figure 6.7) and refractive index n deflects a ray incident at an angle θ by an angle θ_d given by the following relation.

$$\theta_d = \theta - \alpha + \sin^{-1}[(n^2 - \sin^2\theta)^{1/2}\sin\alpha - \sin\theta\ cos\ \alpha] \tag{6.3}$$

The above relation is obtained by using Snell's law twice at the two refracting surfaces.

When α is very small and θ is also very small the above equation transforms into the following approximation.

$$\theta_d \approx (n-1)\alpha \tag{6.4}$$

Beam-Splitters

A beam-splitter is an optical component that splits the incident light beam into

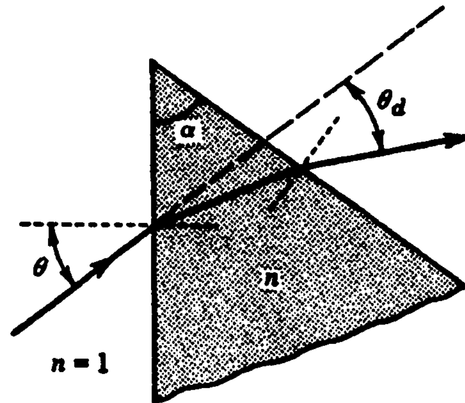


Figure 6.7: Ray Deflection by a Prism

a reflected beam and a transmitted beam (as shown in Figure 6.8). They are also used to combine two light beams into one (as shown in Figure 6.9).

The functionality of a beam-splitter can be realized in several ways. A **plate** beam-splitter can be constructed by depositing a semi-transparent film on a glass substrate. A **cube** beam splitter (shown in Figure 6.10) can be obtained by joining two right-angled prisms. The joint acts as a semi-transparent surface.

Lenses

A spherical lens is bounded by two spherical surfaces. It can be modelled by the radii R_1 and R_2 of its two surfaces, its thickness Δ, and the refractive index n of the material as shown in Figure 6.11.

As Figure 6.12(a) shows, a ray crossing the first surface at height y and angle

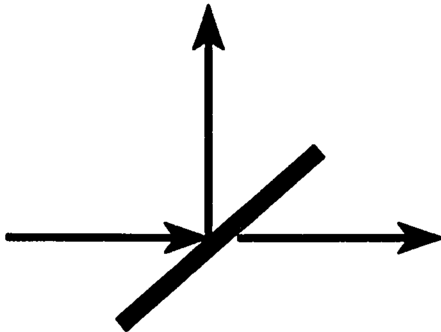


Figure 6.8: Beam-Splitter Splits a Beam into Two Components

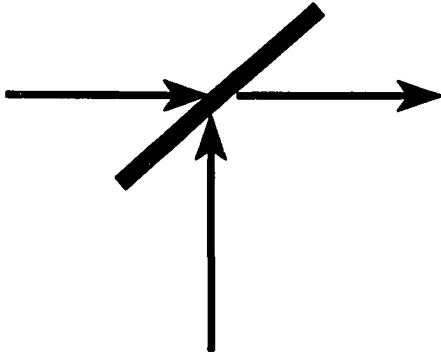


Figure 6.9: Beam-Merger Combines Two Beams into One

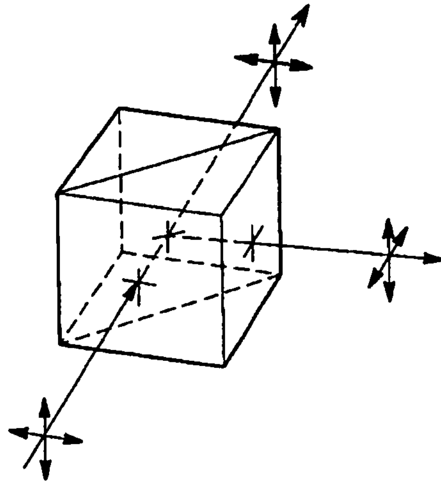


Figure 6.10: A Cube Beam-Splitter and its Operation

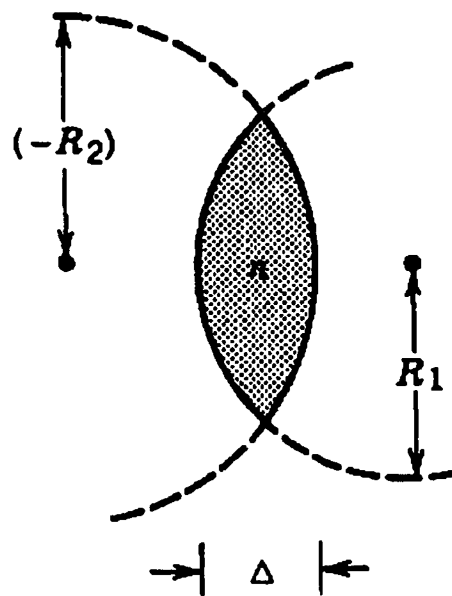


Figure 6.11: A Biconvex Spherical Lens

θ_1 with the z axis is traced by applying

$$\theta_2 \approx \frac{n_1}{n_2}\theta_1 - \frac{n_2 - n_1}{n_2 R} y \tag{6.5}$$

at the first surface to obtain the inclination angle θ of the refracted ray, which is extended to meet the second surface. The previous relation is once more applied to obtain θ_2 after refraction from the second surface.

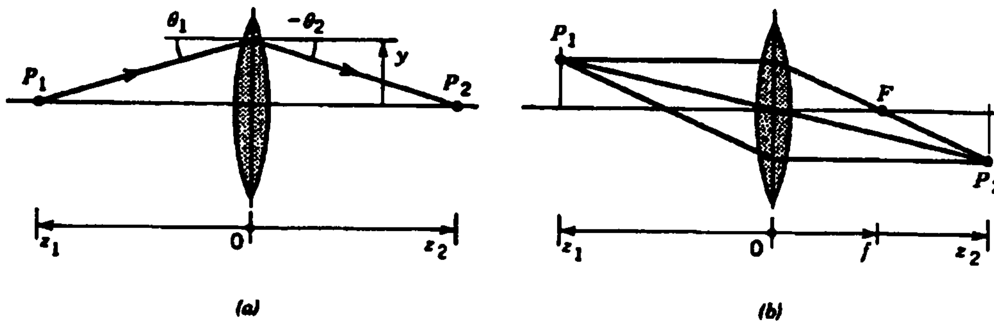


Figure 6.12: Thin Lens: (a) Ray Bending, (b) Image Formation.

Assuming that the lens is thin will greatly simplify matters. Some results are of importance.

- The angles of the refracted and the incident rays are related by the following equation.

$$\theta_2 = \theta_1 - \frac{y}{f} \tag{6.6}$$

where f is called the **focal length** and is given by the relation

$$\frac{1}{f} = (n-1)(\frac{1}{R_1} - \frac{1}{R_2}) \tag{6.7}$$

- As seen in Figure 6.12(b), all rays originating from a point $P_1 = (y_1, z_1)$ meet at a point $P_2 = (y_2, z_2)$ where the following relations hold.

$$\frac{1}{z_1} + \frac{1}{z_2} = \frac{1}{f} \tag{6.8}$$

and

$$y_2 = -\frac{z_2}{z_1} y_1 \tag{6.9}$$

Fibers

Graded index materials have a refractive index that varies with position in accordance with a continuous function $n(r)$. These materials are often manufactured by adding impurities of controlled concentrations.

In a graded index (**GRIN**) medium, optical rays follow curved trajectories instead of straight lines. A *graded index slab* is a slab of material whose refractive index is uniform in the x and z direction but varies continuously in the y direction. Since the nature and concentration of doping in these slabs can be controlled, they can be made to behave like any of the conventional optical components such as lens, prism.

A *graded index fiber* is a glass cylinder with a refractive index n that varies as a function of the radial distance from its axis. The doping is done so that a ray that enters the fiber remains confined to it so that the fiber serves as a light guide.

Analysis Techniques

Matrix Optics

Matrix optics is used to model optical systems made out of ray optic components,

under paraxial approximation. The system can be treated as a collection of matrices, one for each surface. A ray is modelled as an ordered pair – (*position*, *orientation*).

As the ray propagates through surfaces/components, the output ray can be given as the product of input ray and the paraxial transformation matrix of the component under consideration.

Ray transfer matrices of some simple components are given below:

Free-Space Propagation

Since rays travel in free-space along straight lines, a ray travelling a distance d is altered in accordance with $y_2 = y_1 + \theta_1 d$ and $\theta_2 = \theta_1$ (see Figure 6.13). The ray transfer matrix is therefore $M_{FreeSpacePropagation}$.

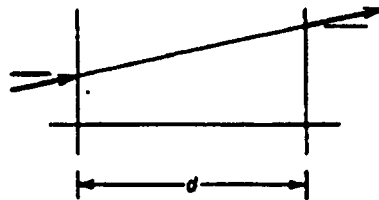


Figure 6.13: Free-Space Propagation

$$M_{FreeSpacePropagation} = \begin{pmatrix} 1 & d \\ 0 & 1 \end{pmatrix} \quad (6.10)$$

Refraction at a Planar Boundary

At a planar boundary between two media of refractive indices n_1 and n_2, the ray angle changes in accordance with Snell's law $n_1 \sin\theta_1 = n_2 \sin\theta_2$. For small

values of θ_1, the approximation $n_1\theta_1 = n_2\theta_2$ is obtained. The transfer matrix is given as $M_{RefractionPlanar}$.

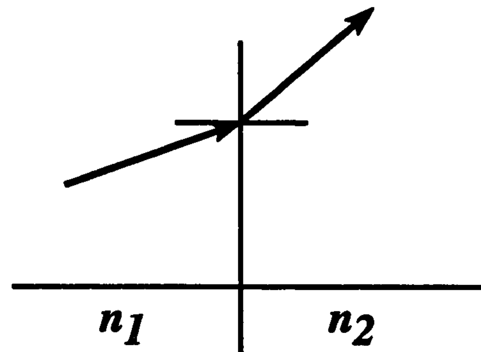


Figure 6.14: Refraction at a Planar Boundary

$$M_{RefractionPlanar} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{n_1}{n_2} \end{pmatrix} \tag{6.11}$$

Refraction at a Spherical Boundary

The ray transfer matrix is $M_{RefrSph}$.

$$M_{RefrSph} = \begin{pmatrix} 1 & 0 \\ -\frac{(n_2-n_1)}{n_2 R} & \frac{n_1}{n_2} \end{pmatrix} \tag{6.12}$$

Transmission Through a Thin Lens

The ray transfer matrix is $M_{TransThinLens}$.

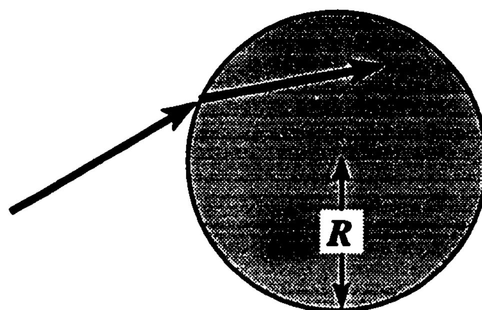


Figure 6.15: Refraction at a Spherical Boundary

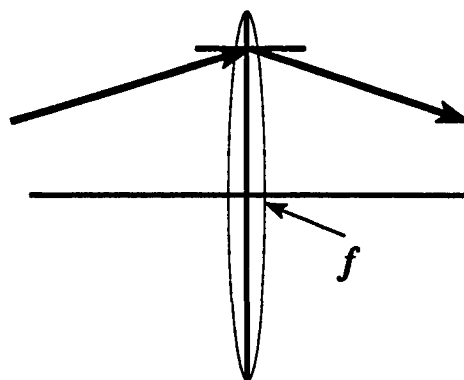


Figure 6.16: Transmission Through a Thin Lens

$$M_{TransThinLens} = \begin{pmatrix} 1 & 0 \\ -\frac{1}{f} & 1 \end{pmatrix} \tag{6.13}$$

Reflection from a Plane Mirror

The ray transfer matrix is $M_{ReflectionPlane}$.

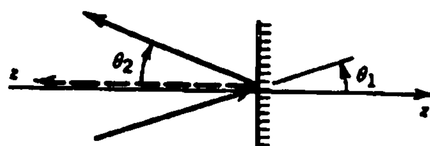


Figure 6.17: Reflection from a Plane Mirror

$$M_{ReflectionPlane} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{6.14}$$

Reflection from a Spherical Mirror

The ray transfer matrix is $M_{ReflSph}$.

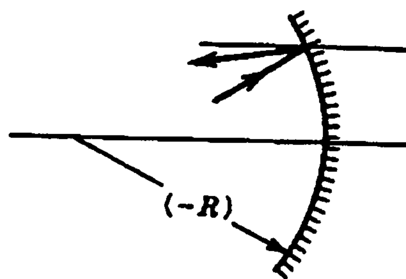


Figure 6.18: Reflection from a Spherical Mirror

$$M_{ReflSph} = \begin{pmatrix} 1 & 0 \\ \frac{2}{R} & 1 \end{pmatrix} \tag{6.15}$$

A **cascade** of optical components whose ray transfer matrices are M_1, M_2, M_3, ..., M_n is equivalent to a single optical component whose ray transfer matrix is M, the chain product of the individual transfer functions.

$$M = M_1 \times M_2 \times M_3 \times ...M_n \tag{6.16}$$

where $\times$ is matrix multiplication.

A paraxial ray travelling through a cascade of identical unit optical systems, each with a ray transfer matrix with elements (A, B, C, D) such that $AD - BC = 1$, follows a bounded trajectory if the condition $|(A+D)/2 \leq 1|$, called the stability condition is satisfied. The position at the mth stage is then $y_m = y_{max}\sin(m\varphi+\varphi_0)$, $m = 0, 1, 2, ...$, where $\varphi = cos^{-1}[(A + D)/2]$. The constants y_{max} and φ_0 are determined from the initial positions y_0 and $y_1 = Ay_0 + B\theta_0$, where θ_0 is the initial ray inclination. The ray angles are related to the positions by $\theta_m = \theta_{max}\sin(m\varphi + \varphi_1)$. For the paraxial approximation to be valid, $\theta_{max} << 1$. The ray trajectory is periodic with period s if $\varphi/2\pi$ is a rational number q/s.

Relationship to Other Theories

Ray optics is a special case of most other theories. It is the limit of wave optics (discussed next) when the wavelength is very short. The primary reason for studying ray optics is that it is simple (but limited) in its explanation capabilities.

6.2.2 Wave Optics

Light propagates in the form of waves. In free space, light travels with speed c_0. A homogenous, transparent medium such as glass is characterized by a single constant, its refractive index $n \geq 1$. In a medium of refractive index n, light travels with a reduced speed given by the following relationship:

$$c = \frac{c_0}{n} \tag{6.17}$$

An optical wave is described mathematically by a real function of position $\mathbf{r} = (x, y, z)$ and time t. denoted by $u(\mathbf{r}, t)$ and known as the **wavefunction.** It satisfies the following *wave equation*:

$$\nabla^2 u - \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = 0 \tag{6.18}$$

where ∇^2 is the Laplacian operator, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$. Any function satisfying the wave equation represents a possible optical wave.

Since the wave equation is linear, the *principle of superposition* applies, i.e., if $u_1(\mathbf{r}, t)$ and $u_2(\mathbf{r}, t)$ represent optical waves, then $u(\mathbf{r}, t) = u_1(\mathbf{r}, t) + u_2(\mathbf{r}, t)$ also represents a possible optical wave.

Postulates

Light travels as waves of expanding wave fronts. These wave fronts may be obtained by considering each point in a front as a potential source of light. Light is emitted from this source and travels a distance δd in time δt. Hence it can be represented as a circle of radius δd with center at the source. The union of all such circles of radius δd form the next wavefront.

Light is described as a scalar function of time and space, called the wave function that obeys/satisfies the Laplace's equation.

Principles and Laws

The **optical intensity**, $I(r,t)$, defined as the optical power per unit area (units of watts/cm^2), is proportional to the average of the following squared wave function.

$$I(r,t) = 2 < u^2(r,t) > \tag{6.19}$$

where the operation $< . >$ denotes averaging over a time interval that is much longer than the time of an optical cycle.

The **optical power**, $P(t)$, defined in watts flowing into an area A normal to the direction of propagation of light is the integrated intensity given by the following equation.

$$P(t) = \int_A I(r,t)dA \tag{6.20}$$

The **optical energy** measured in Joules, collected in a given time interval is the time integral of the optical power over the time interval.

A monochromatic wave is represented by a wavefunction with harmonic time dependence,

$$u(r,t) = a(r)cos[2\pi vt + \varphi(r)] \tag{6.21}$$

where, $a(r)$ is the amplitude, $\varphi(r)$ is the phase, v is the frequency (cycles/s or Hz.), $w = 2\pi v$ is the angular frequency (radians/s).

It is useful to represent the real wavefunction $u(r,t)$ in terms of the following complex function.

$$U(r,t) = a(r)e^{[j\varphi(r)]}e^{(j2\pi vt)} \tag{6.22}$$

so that,

$$u(r,t) = Re\{U(r,t)\} = \frac{1}{2}[U(r,t) + U^*(r,t)] \tag{6.23}$$

where $Re\{.\}$ represents the real part of any complex function.

The function $U(r,t)$ is called the **complex wavefunction**. It describes the wave completely. The wavefunction $u(r,t)$ is simply its real part.

Both the functions $U(r,t)$ and $u(r,t)$ satisfy the **wave equation** below.

$$\nabla^2 U - \frac{1}{c^2}\frac{\partial^2 U}{\partial t^2} = 0 \tag{6.24}$$

Substituting $U(r,t) = U(r)e^{(j2\pi vt)}$ into the wave equation, the following equation (called the **Helmholtz** equation) is obtained.

$$(\nabla^2 + k^2)U(r) = 0 \tag{6.25}$$

where, k is the **wavenumber** given by,

$$k = \frac{2\pi v}{c} = \frac{w}{c} \tag{6.26}$$

The **Optical Intensity** can be determined by,

$$2u^2(r,t) = 2a^2(r)cos^2[2\pi vt + \varphi(r)] \tag{6.27}$$

$$= |U(r)|^2\{1 + cos(2[2\pi vt + \varphi(r)])\} \tag{6.28}$$

is averaged over a time longer than the optical period, $1/v$, the second term of the above equation vanishes, giving the relation for the optical intensity ($I(r)$).

$$I(r) = |U(r)|^2 \tag{6.29}$$

The above equation shows that the optical intensity of a monochromatic wave *is the absolute square of its complex amplitude.*

Mathematical Model

A wave can be represented as a mathematical equation of the form,

$$U(r,t) = U(r)e^{(jwt)} \tag{6.30}$$

When two monochromatic waves of complex amplitudes $U_1(r)$ and $U_2(r)$ are superimposed, the result is a monochromatic wave of the same frequency and complex amplitude given by,

$$U(r) = U_1(r) + U_2(r) \tag{6.31}$$

Let the intensities of the waves be I_1 and I_2. Since $I_1 = |U_1|^2$ and $I_2 = |U_2|^2$, the total intensity I is given by

$$I = |U|^2 = |U_1 + U_2|^2 = |U_1|^2 + |U_2|^2 + U_1^* U_2 + U_1 U_2^* \tag{6.32}$$

Assuming that φ_1 and φ_2 are the phases of the two waves and $\varphi = \varphi_1 + \varphi_2$, and substituting $U_1 = I_1^{\frac{1}{2}} e^{(j\varphi_1)}$ and $U_2 = I_2^{\frac{1}{2}} e^{(j\varphi_2)}$, the following **interference equation** is obtained.

$$I = I_1 + I_2 + 2(I_1 I_2)^{\frac{1}{2}} cos\varphi \tag{6.33}$$

Components

In this section, the effect of some optical components on optical waves is examined.

Reflection from a Plane Mirror

A plane wave of wavevector k_1 is incident onto a planar mirror located in free space in the $z = 0$ plane. A reflected plane wave of wavevector k_2 is created. The angles of incidence and reflection are θ_1 and θ_2 (see Figure 6.19).

Assuming that some conditions are met for both waves at the mirror surface, it can be shown that $\theta_1 = \theta_2$, i.e., the ray-optics law of reflection holds.

Reflection/Refraction at a Boundary

Figure 6.20 shows a plane wave of wave-vector k_1 incident on a planar boundary formed by two homogenous media of refractive indices n_1 and n_2 respectively.

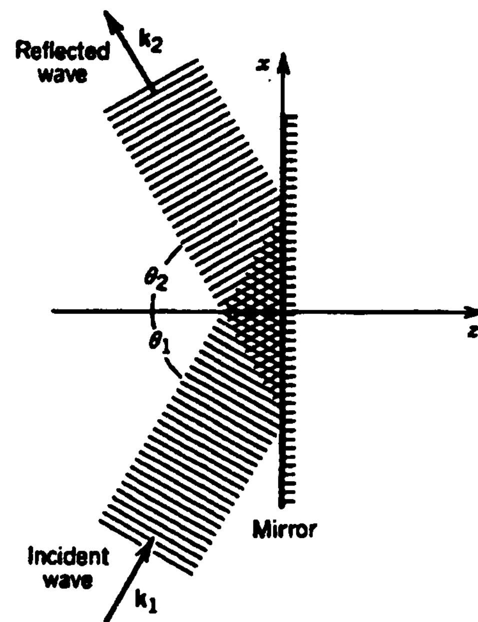


Figure 6.19: Reflection of a Plane Wave from a Planar Mirror

The boundary lies in the $z = 0$ plane. Refracted and reflected plane waves of wavevectors k_2 and k_3 respectively emerge.

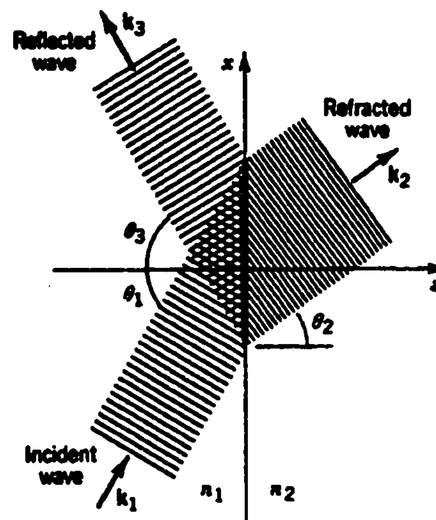


Figure 6.20: Reflection and Refraction of a Plane Wave at a Dielectric Planar Boundary

Since the waves lie on the same plane, assumptions such as equal phase lead to the results $\theta_1 = \theta_3$ and $n_1 \sin\theta_1 = n_2 \sin\theta_2$. These are the laws of reflection and refraction.

Thin Plate/Slab

The transmission of a plane wave through a thin plate (thickness $= d$, refractive index $= n$) surrounded by free space is considered here (see Figure 6.21). The surfaces of the plate are the planes $z = 0$ and $z = d$. The incident wave is travelling in the z direction. Let $U(x, y, z)$ be the complex amplitude of the plate. Ignoring internal and external reflections, $U(x, y, z)$ is assumed to be

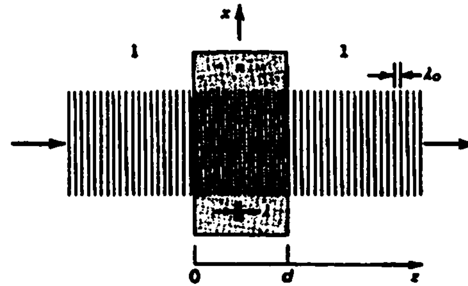


Figure 6.21: Transmission of a Plane Wave Through a Transparent Plate

continuous at the boundaries. The **complex amplitude transmission** of the plate is given by the ratio $t(x,y) = U(x,y,d)/U(x,y,0)$.

The incident plane wave continues to propagate inside the plate as a plane wave with wavenumber nk_0. Hence $U(x,y,z)$ is proportional to $e^{(-jnk_0z)}$ and $U(x,y,d)/U(x,y,0) = e^{(-jnk_0d)}$. The **Complex Amplitude Transmittance of a Transparent Plate** is given by the relation,

$$t(x,y) = e^{(-jnk_0d)} \tag{6.34}$$

which shows that the plate introduces a phase shift $nk_0d = 2\pi(d/\lambda)$.

If the incident plane wave makes an angle θ with the z axis (see Figure 6.22), then the complex amplitude transmittance changes to,

$$t(x,y) = e^{[-jnk_0(dcos\theta_1 + x\sin\theta_1)]} \tag{6.35}$$

Prism

Using the above relations, the complex transmittance of a thin prism with small

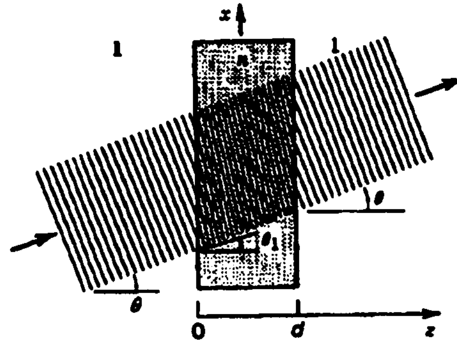


Figure 6.22: Transmission of a Plane Wave at an Angle θ Through a Transparent Plate

angle α and width d_0 is given by,

$$t(x, y) = h_0 e^{[-j(n-1)k_0 \alpha x]} \tag{6.36}$$

where $h_0 = e^{(-jk_0 d_0)}$.

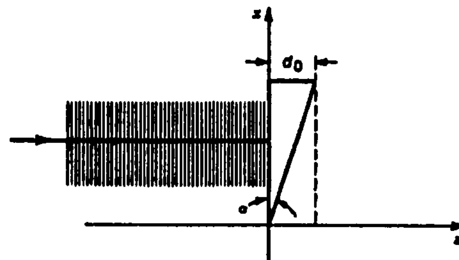


Figure 6.23: Transmission of a Plane Wave Through a Prism

Thin Lens

For a thin lens (shown in Figure 6.24), the transmittance is,

$$t(x, y) \approx h_0 e^{[jk_0 \frac{x^2+y^2}{2f}]} \tag{6.37}$$

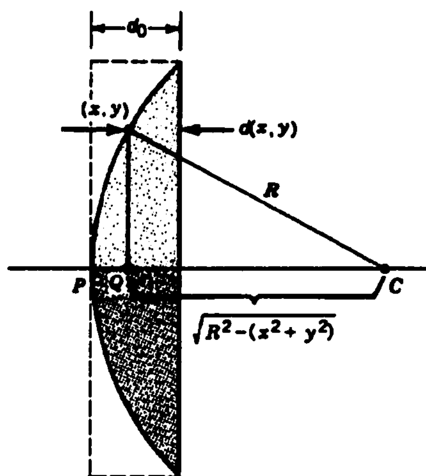


Figure 6.24: Transmission of a Plane Wave Through a Thin Lens

Double Convex Lens

Considering a double convex lens as a cascade of two plano-convex lenses (see Figure 6.25), the complex amplitude transmittance can be derived as,

$$\frac{1}{f} = (n-1)(\frac{1}{R_1} - \frac{1}{R_2}) \tag{6.38}$$

Diffraction Gratings

A **diffraction grating** is an optical component that serves to periodically modulate the phase or the amplitude of the incident wave. Consider a diffraction grating (see Figure 6.26) placed in the $z = 0$ plane whose thickness varies (spatially) periodically in the x direction with period Λ. The grating converts an

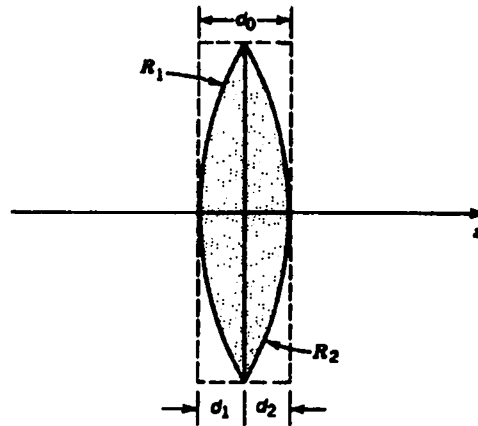


Figure 6.25: A Double Convex Lens

incident wave of wavelength $\lambda << \Lambda$, travelling at a small angle θ_i with respect to the z axis into several plane waves at small angles,

$$\theta_q = \theta_i + q\frac{\lambda}{\Lambda} \tag{6.39}$$

where $q = 0, \pm 1, \pm 2, \ldots$ with the z axis and is called the **diffraction order.** The diffraction waves are separated by an angle $\theta = \lambda/\Lambda$.

Graded-Index Components

The effect of a prism, lens, or diffraction grating lies in the phase shift it imparts to the incident wave, which serves to bend the wavefront in a prescribed manner. This phase shift is controlled by the variation of the thickness of the material with the transverse distance of the optical axis. The **complex amplitude** transmittance of a thin transparent planar plate of width d_0 and graded refractive index $n(x, y)$ is,

$$t(x, y) = e^{[-jn(x,y)k_0 d_0]} \tag{6.40}$$

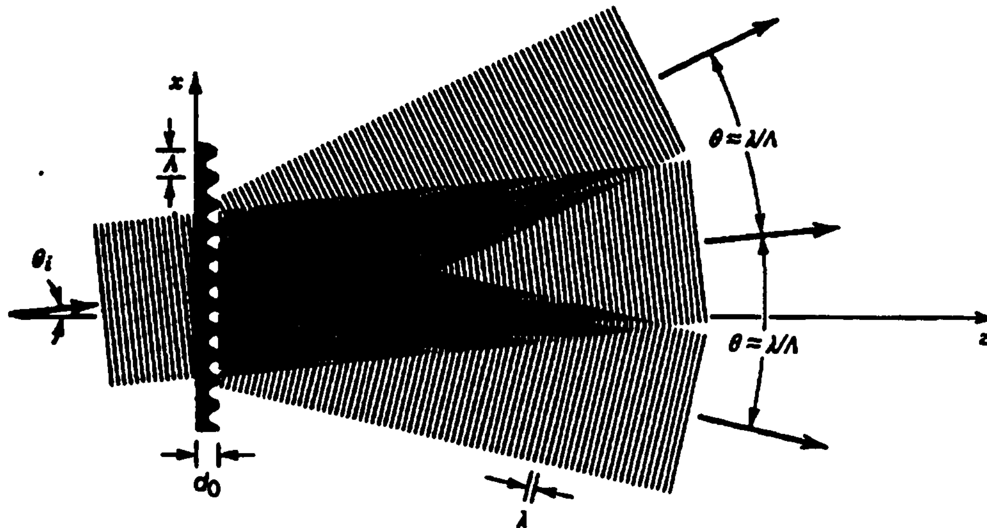


Figure 6.26: A Thin, Transparent Plate with Periodically Varying Thickness

By varying x and y in $n(x, y)$, the action of any constant-index thin optical component can be reproduced.

Interferometers

When two monochromatic waves of complex amplitudes $U_1(r)$ and $U_2(r)$ are superimposed, the result is a monochromatic wave of the same frequency and complex amplitude, i.e.,

$$U(r) = U_1(r) + U_2(r) \tag{6.41}$$

The intensities of the total wave is given by

$$I = |U|^2 = |U_1 + U_2|^2 = |U_1|^2 + |U_2|^2 + U_1^*U_2 + U_1U_2^* \tag{6.42}$$

Assuming that φ_1 and φ_2 are the phases of the two waves and $\varphi = \varphi_1 + \varphi_2$, and substituting $U_1 = I_1^{\frac{1}{2}}e^{(j\varphi_1)}$ and $U_2 = I_2^{\frac{1}{2}}e^{(j\varphi_2)}$, the following **interference**

equation is obtained.

$$I = I_1 + I_2 + 2(I_1 I_2)^{\frac{1}{2}} cos\varphi \qquad (6.43)$$

An **interferometer** is an optical instrument that:

1. splits a wave into two waves using a beamsplitter,
2. delays them by unequal distances,
3. redirects them using mirrors,
4. recombines them using another beamsplitter, and
5. detects the intensity of their superposition.

Some important interferometers are shown in Figure 6.27.

Analysis Techniques

Complex variable functions, and solutions to wave equations.

Relationship to other theories

As the wavelength (λ) approaches zero i.e., $\lambda \rightarrow 0$, wave optics becomes ray optics.

Wave optics has its limitations. It is not capable of providing a complete picture of the reflection and refraction of light at the boundaries between dielectric materials, nor of explaining those optical phenomena that require a vector formulation, such as polarization effects.

6.2.3 Beam Optics

A plane wave and a spherical wave represent the two opposite extremes of angular and spatial confinement. Waves with small angles and speed are called paraxial waves.

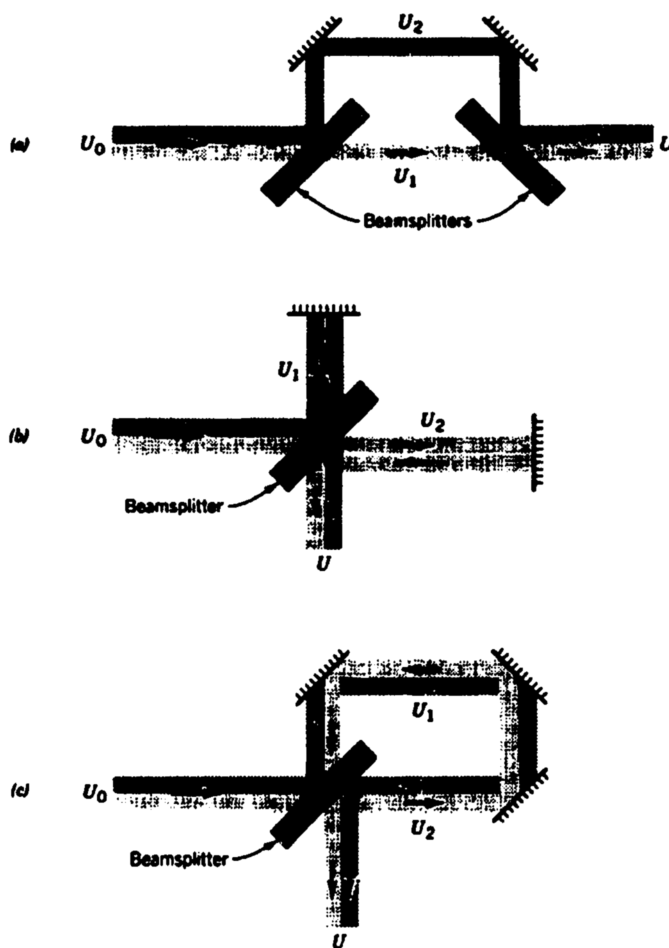


Figure 6.27: Interferometers: (a) Mach-Zehnder Interferometer (b) Michelson's Interferometer, and (c) Sagnac Interferometer

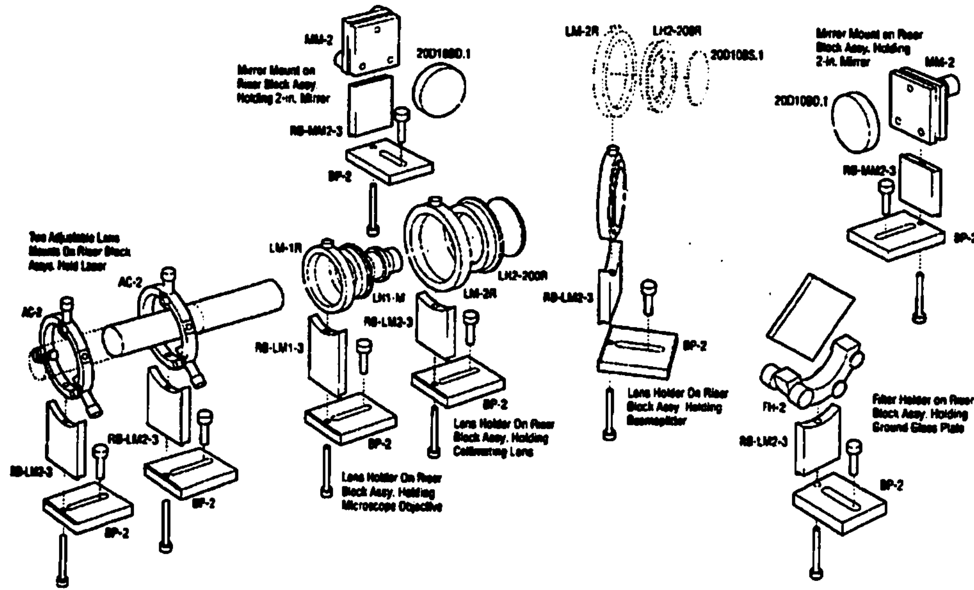


Figure 6.28: Setup of Michelson's Interferometer using Catalog Components

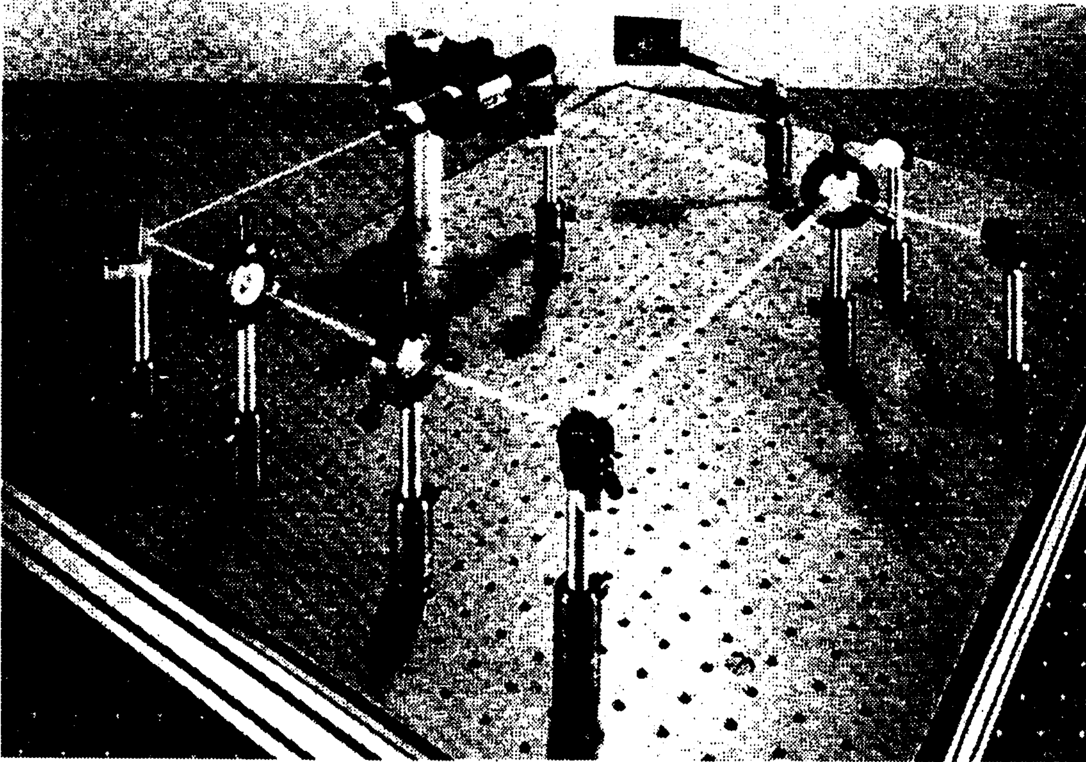


Figure 6.29: Snapshot of the Setup of Michelson's Interferometer

A Gaussian beam is a special wave in which much of the intensity is concentrated at the center.

Beam optics provides a good theoretical framework for the study of concentrated energy generated by laser and it is of vital importance in a number of systems.

Gaussian Beam and its Properties

A paraxial wave is a plane wave e^{-jkz} (with wavenumber $k = 2\pi/\lambda$ and wavelength λ) modulated by a complex envelope $A(r)$, i.e., a slowly varying function of position. The complex amplitude is

$$U(r) = A(r)e^{(-jkz)} \tag{6.44}$$

The envelope is assumed to be approximately constant within a neighborhood of size λ, so that the wave is locally like a plane wave with wavefront normals that are paraxial rays.

For the complex amplitude $U(r)$ to satisfy the Helmholtz equation, $\nabla^2 U + k^2 U = 0$, the complex envelope $A(r)$ must satisfy the paraxial Helmholtz equation

$$\nabla_T^2 A - j2k\frac{\partial A}{\partial z} = 0 \tag{6.45}$$

where $\nabla_T^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the transverse part of the Laplacian operator.

This gives the Gaussian beam,

$$A(r) = \frac{A_1}{q(z)}e^{[-jk\frac{\rho^2}{2q(z)}]} \tag{6.46}$$

where $q(z) = z - \xi$.

When ξ is purely imaginary, say $\xi = -jz_0$, where z_0 is real, gives rise to the complex envelope of the Gaussian beam,

$$A(r) = \frac{A_1}{q(z)}e^{[-jk\frac{\rho^2}{2q(z)}]} \tag{6.47}$$

where $q(z) = z + jz_0$ and z_0 is called the **Rayleigh range**.

To separate the amplitude and phase of this complex envelope, the complex function $\frac{1}{q(z)} = \frac{1}{z+jz_0}$ can be written in terms of its real and imaginary parts by defining two new real functions $R(z)$ (**wavefront radius**)and $W(z)$ (**beam width**) such that,

$$\frac{1}{q(z)} = \frac{1}{R(z)} - j\frac{\lambda}{\pi W^2(z)} \tag{6.48}$$

An expression for the complex amplitude $U(r)$ of the Gaussian beam is obtained as,

$$U(r) = A_0 \frac{W_0}{W(z)} e^{\left[-\frac{\rho^2}{W^2(z)}\right]} e^{\left[-jkz - jk\frac{\rho^2}{2R(z)} + j\zeta(z)\right]} \tag{6.49}$$

where $A_0 = \frac{A_1}{jz_0}$

The important results can be summarized as follows.

$$Beam\ Width\ W(z) = W_0 \left[1 + \left(\frac{z}{z_0}\right)^2\right]^{\frac{1}{2}} \tag{6.50}$$

$$Wavefront\ radius\ of\ curvature\ R(z) = z\left[1 + \left(\frac{z_0}{z}\right)^2\right] \tag{6.51}$$

$$\zeta(z) = tan^{-1}\frac{z}{z_0} \tag{6.52}$$

$$W_0 = \left(\frac{\lambda z_0}{\pi}\right)^{\frac{1}{2}} \tag{6.53}$$

Some of the properties of Gaussian beams are as follows.

Intensity

The optical intensity $I(r) = |U(r)|^2$ is a function of the axial and radial distances z and $\rho = (x^2 + y^2)^{\frac{1}{2}}$,

$$I(\rho, z) = I_0 \left[\frac{W_0}{W(z)}\right]^2 e^{\left[-\frac{2\rho^2}{W^2(z)}\right]} \tag{6.54}$$

where $I_0 = |A_0|^2$.

At each value of z the intensity is a Gaussian function of the radial distance ρ. The Gaussian function has its peak at $\rho = 0$(i.e., the axis) and drops monotonically with increasing ρ. The width $W(z)$ of the Gaussian distribution increases with the axial distance z.

On the beam axis ($\rho = 0$), the intensity

$$I(0,z) = I_0 \left[\frac{W_0}{W(z)}\right]^2 = \frac{I_0}{1+(z/z_0)^2} \tag{6.55}$$

has its maximum value at I_0 at $z = 0$ and drops gradually with increasing z reaching half its peak value at $z = \pm z_0$.

Power

The total optical power (P) carried by the beam is the integral of the optical intensity over a transverse plane,

$$P = \int_0^\infty I(\rho, z) 2\pi\rho d\rho \tag{6.56}$$

which gives

$$P = \frac{1}{2} I_0 (\pi W_0^2) \tag{6.57}$$

Beam Radius

Within any transverse plane, the beam intensity assumes its peak value on the beam axis, and drops by the factor $1/e^2 \approx 0.135$ at the radial axis $\rho = W(z)$. Since 86% of the power is carried within a circle of radius $W(z)$, the same quantity is called the beam radius.

The dependence of the beam radius on z is governed by

$$W(z) = W_0 \left[1 + \left(\frac{z}{z_0}\right)^2\right]^{\frac{1}{2}} \tag{6.58}$$

Beam Divergence

Far from the beam center, when $z >> z_0$, the beam radius increases approximately linearly with z, defining a cone with half angle θ_0. About 86% of the beam power is confined within this cone. The angular divergence of the beam is therefore defined by the angle

$$\theta_0 = \frac{2}{\pi}\frac{\lambda}{2W_0} \tag{6.59}$$

The beam divergence is directly proportional to the ratio between the wavelength λ and the beam-waist diameter $2W_0$.

Depth of Focus

Since the beam has its minimum width at $z = 0$, it achieves its best focus at the plane $z = 0$. In either direction, the beam gradually grows out of focus. The axial radius within which the beam radius lies within a factor of $\sqrt{2}$ of its minimum value is known as the **depth of focus**. The depth of focus is twice the Rayleigh range,

$$2z_0 = \frac{2\pi W_0^2}{\lambda} \tag{6.60}$$

Phase

The phase of the Gaussian beam is

$$\phi(\rho, z) = kz - \zeta(z) + \frac{k\rho^2}{2R(z)} \tag{6.61}$$

Wave Fronts

This represents the deviation of the phase at off-axis points in a given transverse plane from that at the axial point. The surfaces of constant phase satisfy

$$k\left[z + \frac{\rho^2}{2R(z)}\right] - \zeta(z) = 2\pi q \tag{6.62}$$

Since $\zeta(z)$ and $R(z)$ are relatively slowly varying, they are approximately constant at points within the beam radius on each wavefront. Therefore,

$$z + \frac{\rho^2}{2R} = q\lambda + \zeta\lambda/2\pi \tag{6.63}$$

where $R = R(z)$ and $\zeta = \zeta(z)$.

The above is the equation of a paraboloidal surface of radius of curvature R.

Following are some of the properties of Gaussian beams at special points.

At the plane $z = z_0$

At an axial distance z_0 from the beam waist, the beam has the following properties.

1. The beam radius is $\sqrt{2}$ times greater than the radius at the beam waist, and the area is larger by a factor of 2.
2. The intensity on the beam axis is half the peak intensity.
3. The phase on the beam axis is retarded by an angle $\pi/4$ relative to the phase of a plane wave.
4. The radius of curvature of the wavefront is the smallest, so that the wavefront has the greatest curvature ($R = 2z_0$).

Near the Beam Center

At points for which $z << z_0$ and $\rho << W_0$,

$$e^{\left[-\frac{\rho^2}{W^2(z)}\right]} \approx e^{\left[-\frac{\rho^2}{W_0^2}\right]} \approx 1 \tag{6.64}$$

so that the beam intensity is approximately constant. Also,

$$R(z) \approx \frac{z_0^2}{z} \tag{6.65}$$

and

$$\zeta(z) \approx 0 \tag{6.66}$$

so that the phase

$$k\left[z+\frac{\rho^2}{2R(z)}\right] = kz(1+\frac{\rho^2}{2z_0^2}) \approx kz. \tag{6.67}$$

As a result the wavefronts are approximately planar. The Gaussian beam may therefore be approximated near its center by a plane wave.

Far from the Beam Waist

At points within the beam-waist radius ($\rho < W_0$) but far from the beam waist ($z >> z_0$), the wave is approximately like a spherical wave. Since

$$W(z) \approx \frac{W_0 z}{z_0} >> W_0 \tag{6.68}$$

and

$$\rho < W_0 \tag{6.69}$$

$$e^{\left[-\frac{\rho^2}{W^2(z)}\right]} \approx 1 \tag{6.70}$$

so that the beam intensity is approximately uniform. Since

$$R(z) \approx z \tag{6.71}$$

the wavefronts are approximately spherical. Thus except for an excess phase

$$\zeta(z) \approx \frac{\pi}{2} \tag{6.72}$$

the complex amplitude of the Gaussian beam approaches that of the paraboloid wave, which in turn approaches that of the spherical wave.

The above can either be used to compute the characteristics of a gaussian beam with the given parameters or to synthesize gaussian beams of desired properties/shape by computing the parameters.

For example a gaussian beam could be synthesized for a given width and curvature.

Transmission of Gaussian Beams through Optical Components

This section discusses the effect of optical components on Gaussian beams.

Transmission Through a Thin Lens

The complex amplitude transmittance of a thin lens of focal length f is proportional to $e^{(jk\rho/2f)}$. When a Gaussian beam crosses the lens, its complex amplitude is multiplied by this phase factor. As a result, its wave-front is bent, but the beam radius is not altered.

A Gaussian beam centered at $z = 0$ with waist radius W_0 is transmitted through a thin lens located at a distance z (see Figure 6.30). The phase at the plane

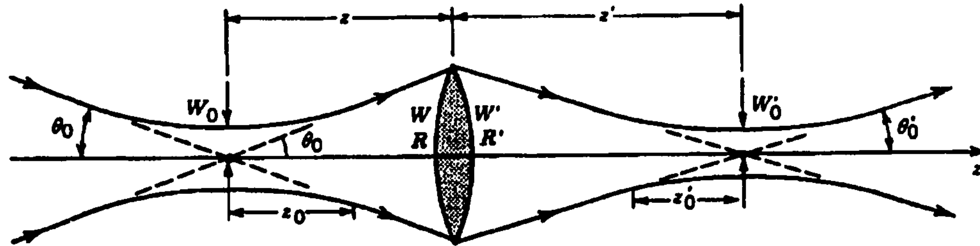


Figure 3.2-1 Transmission of a Gaussian beam through a thin lens.

Figure 6.30: Transmission of a Gaussian Beam through a Thin Lens

of the lens is $kz + k\rho^2/2R - \zeta$, where $R = R(z)$ and $\zeta = \zeta(z)$. The phase of the altered wave is,

$$kz + k\frac{\rho 2}{2R} - \zeta - k\frac{\rho^2}{2f} = kz + k\frac{\rho^2}{2R'} - \zeta \tag{6.73}$$

where

$$\frac{1}{R'} = \frac{1}{R} - \frac{1}{f} \tag{6.74}$$

The transmitted wave is itself a Gaussian beam with width $W' = W$, and radius of curvature R', where R' satisfies the imaging equation $1/R - 1/R' = 1/f$. R is positive since the wavefront of the incident beam is diverging and R' is negative since the wavefront of the transmitted beam is converging.

The waist radius of the new beam is

$$W_0' = \frac{W}{[1 + (\pi W^2/\lambda R')^2]^{1/2}} \tag{6.75}$$

and the center is located at a distance

$$-z' = \frac{R'}{1 + (\lambda R'/\pi W^2)^2} \tag{6.76}$$

from the lens.

In addition, the following expressions which relate the parameters of the two beams are obtained.

Characteristic	Equation
Waist radius	$W_0' = MW_0$
Waist location	$(z' - f) = M^2(z - f)$
Depth of focus	$2z_0' = M^2(2z_0)$
Divergence	$2\theta_0' = \frac{2\theta_0}{M}$
Magnification	$M = \frac{M_r}{(1+r^2)^{1/2}}$

where

$$r = \frac{z_0}{z - f} \tag{6.77}$$

$$M_r = \left|\frac{f}{z - f}\right| \tag{6.78}$$

Beam Shaping

Selected aspects of beams can be changed by letting the beams pass through specially designed/chosen components. The issues of beam focusing, relaying, collimation and expansion can be controlled.

Analysis Techniques

The transmission-matrix approach for ray optics of analyzing systems of optical components can also be adapted for the analysis of beam-optic systems. The transmission matrices will be generalized to deal with the q-parameter of the incident beam. Since the q-parameter identifies the width and radius of curvature the simple law i.e., the $ABCD$ law

$$q_2 = \frac{Aq_1 + B}{Cq_1 + D} \tag{6.79}$$

or transfer function of $(A\# + B)/(C\# + D)\&$ would render itself for efficient analysis. The values of A, B, C, D are component dependent, for example $(A, B, C, D) = (1, d, 0, 1)$ for free-space. Such simplified analysis will be of immense use for the design and analysis of a number of image processing architectures.

Relationship to other Theories

Beam optics is a special case of wave optic theory where there are smaller angles of divergence. It reduces to wave optics in the special case of zero beam diameter.

6.2.4 Fourier Optics

Fourier optic theory provides a description for the propagation of light based on *harmonic analysis* (the Fourier transform) and linear systems. Harmonic analysis is based

on the expansion of an arbitrary function of time $f(t)$ as a sum (or integral) of harmonic functions of time and different frequencies. The harmonic function $F(v)e^{j2\pi vt}$, which has a frequency v and complex amplitude $F(v)$, is the building block of the theory. Several of these functions, each with its own value of $F(v)$ are added to obtain a function $f(t)$. The complex amplitude $F(v)$, as a function of frequency is called the Fourier transform of $f(t)$.

A function $f(x, y)$, of two variables x and y, representing the spatial coordinates in a plane, may similarly be written as a superposition of harmonic functions of x and y of the form $F(v_x, v_y)e^{-2j\pi(v_x x+v_y y)}$ where $F(v_x, v_y)$ is the complex amplitude and v_x and v_y are the spatial frequencies in the x and y directions respectively. The harmonic function $F(v_x, v_y)e^{-2j\pi(v_x x+v_y y)}$ is the two dimensional building block of the theory. It can be used to generate an arbitrary function of two variables.

A plane wave is represented by $U(x, y, z) = Ae^{-j(k_x x+k_y y+k_z z)}$ where (k_x, k_y, k_z) are components of the wavevector $\mathbf{k}$ and A is a complex constant. Since an arbitrary function $f(x, y)$ can be analyzed as a superposition of harmonic waves, an arbitrary travelling wave $U(x, y, z)$ may be analyzed as a sum of plane waves.

The wave theory of light is responsible for many of the new applications of light. One of the most common applications of the wave theory is in the field of holography. Holography is based on recording optical waves onto appropriate materials employing the superposition principle of Fourier optics, and then reconstructing optical waves. The field of holography is of immense practical value because of their numerous applications in display systems, storage and interconnect potential.

Mathematical Model

Let x and y represent the coordinates of a point in 2-D space, then $f(x,y)$ represents a spatial pattern. The harmonic function $Fe^{[-j2\pi(v_x x+v_y y)]}$ is regarded as a building block from which other functions may be composed by superposition. The variables v_x and v_y represent spatial frequencies in the x and y directions respectively. Since x and y have units of length (mm), v_x and v_y have units of cycles/mm or lines/mm.

The Fourier theorem may be generalized to variables of two variables. A function $f(x,y)$ may be decomposed as a superposition integral of harmonic functions of x and y.

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(v_x,v_y)e^{[-j2\pi(v_x x+v_y y)]}dv_x dv_y \tag{6.80}$$

where the coefficients $F(v_x,v_y)$ are determined by use of the two-dimensional Fourier transform

$$F(v_x,v_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{[j2\pi(v_x x+v_y y)]}dxdy \tag{6.81}$$

Components

Spatial Filters

Consider the two-lens system shown in Figure 6.31. The shown system can be viewed as a cascade of two Fourier-transforming subsystems. The first subsystem (between the object plane and the Fourier plane) performs a Fourier transform, and the second subsystem (between the Fourier plane and the image plane) performs an inverse Fourier transform.

Let $f(x,y)$ be the complex amplitude of a transparency placed in the object plane and illuminated by a plane wave $e^{(-jkz)}$ travelling in the z-direction (see Figure 6.32), and let $g(x,y)$ be the complex amplitude in the image plane. The

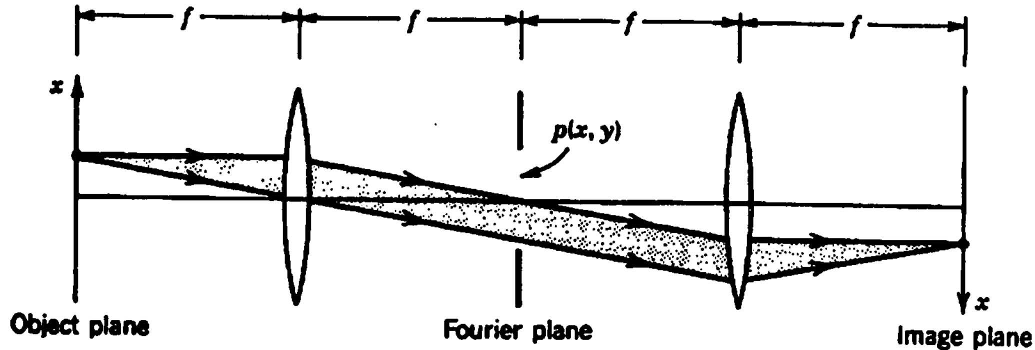


Figure 6.31: The $4 - f$ Imaging System. If an Inverted Coordinate System is used in the Image Plane, the Magnification is Unity

first lens system analyzes $f(x, y)$ into its spatial Fourier-transform and separates its Fourier components so that each point in the Fourier plane corresponds to a single spatial frequency. These components are then recombined by the second lens system and the object distribution is perfectly reconstructed.

The $4 - f$ imaging system can be used as a spatial filter in which the image $g(x, y)$ is a filtered version of the object $f(x, y)$. Since the Fourier components of $f(x, y)$ are available in the Fourier plane, a mask may be used to adjust them selectively, blocking some components and transmitting others (see Figure 6.33). The Fourier component of $f(x, y)$ at the spatial frequency (v_x, v_y) is located in the Fourier plane at the point $x = \lambda f v_x$, $y = \lambda f v_y$. To implement a filter of transfer function $H(v_x, v_y)$, the complex amplitude transmittance $p(x, y)$ of the mask must be proportional to $H(x/\lambda f, y/\lambda f)$. Thus the transfer function of the filter realized by a mask of transmittance $p(x, y)$ is

$$H(v_x, v_y) = p(\lambda f v_x, \lambda f v_y) \tag{6.82}$$

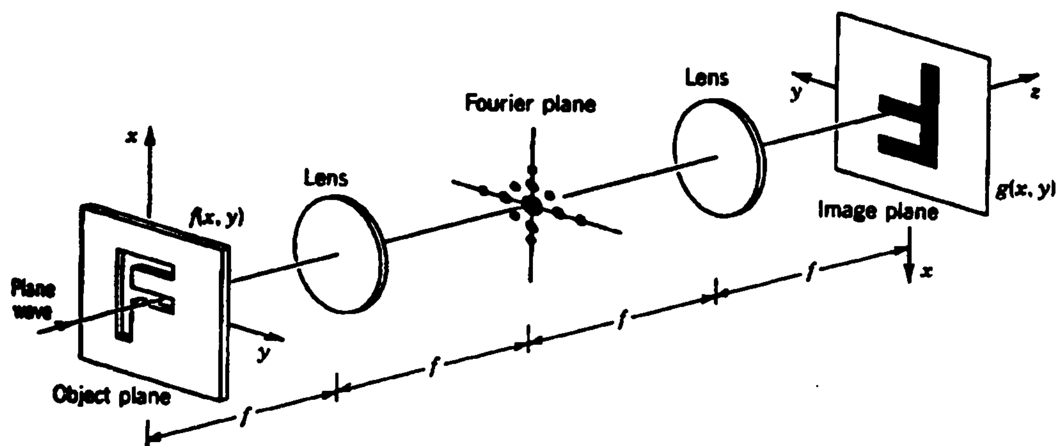


Figure 6.32: The $4-f$ System Performs a Fourier Transform Followed by an Inverse Fourier Transform.

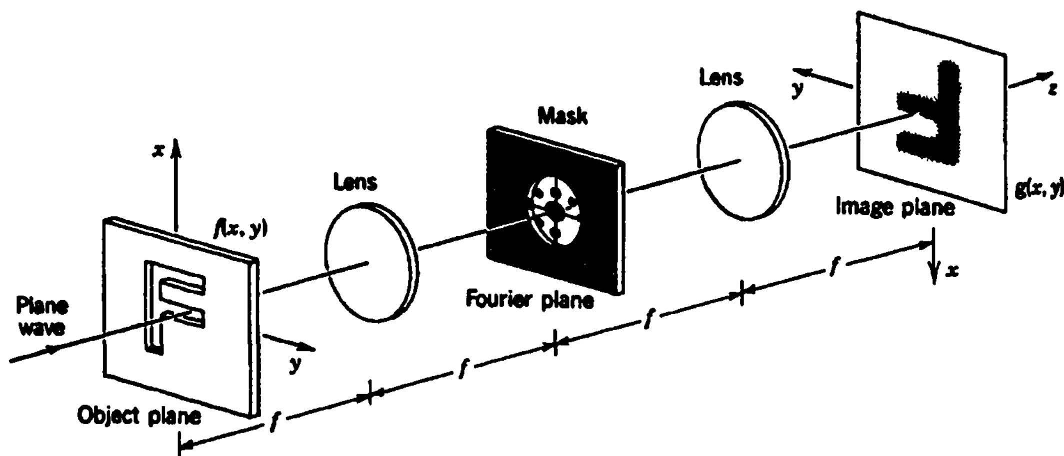


Figure 6.33: The $4-f$ System Performs a Fourier Transform Followed by an Inverse Fourier Transform.

An example of a spatial filter is shown in Figure 6.34. A *low-pass filter* passes spatial frequencies that are smaller than a cutoff frequency and blocks higher frequencies. A *high-pass filter* is the complement of low-pass filter. A *vertical-pass filter* blocks horizontal frequencies and transmits vertical frequencies.

Analysis Techniques

Transfer function of a linear system is its impulse response. For example, the transfer function of free-space is

$$H(v_x, v_y) = e^{\left[-j2\pi\left(\frac{1}{\lambda^2}-v_x^2-v_y^2\right)^{1/2}d\right]} \tag{6.83}$$

Product of the transfer functions of a cascaded system gives the transfer function of the cascaded system.

Output/response of a system is the product of input and its transfer function.

6.2.5 Electro-Magnetic Optics

Light is an electromagnetic wave phenomenon described by the same theoretical principles that govern all forms of electromagnetic radiation. Optical frequencies occupy a band of electromagnetic spectrum that extends from the infrared through the visible to the ultraviolet. As the wavelength of light is relatively short ($10nm - 1mm$), the techniques used for generating, transmitting, and detecting optical waves have traditionally differed from those used for electromagnetic waves of longer wavelength. However, the recent miniaturization of optical components has caused these differences to become less significant.

Electromagnetic radiation propagates in the form of two mutually coupled vector waves, an electric field wave and a magnetic field wave. The wave optic theory is an

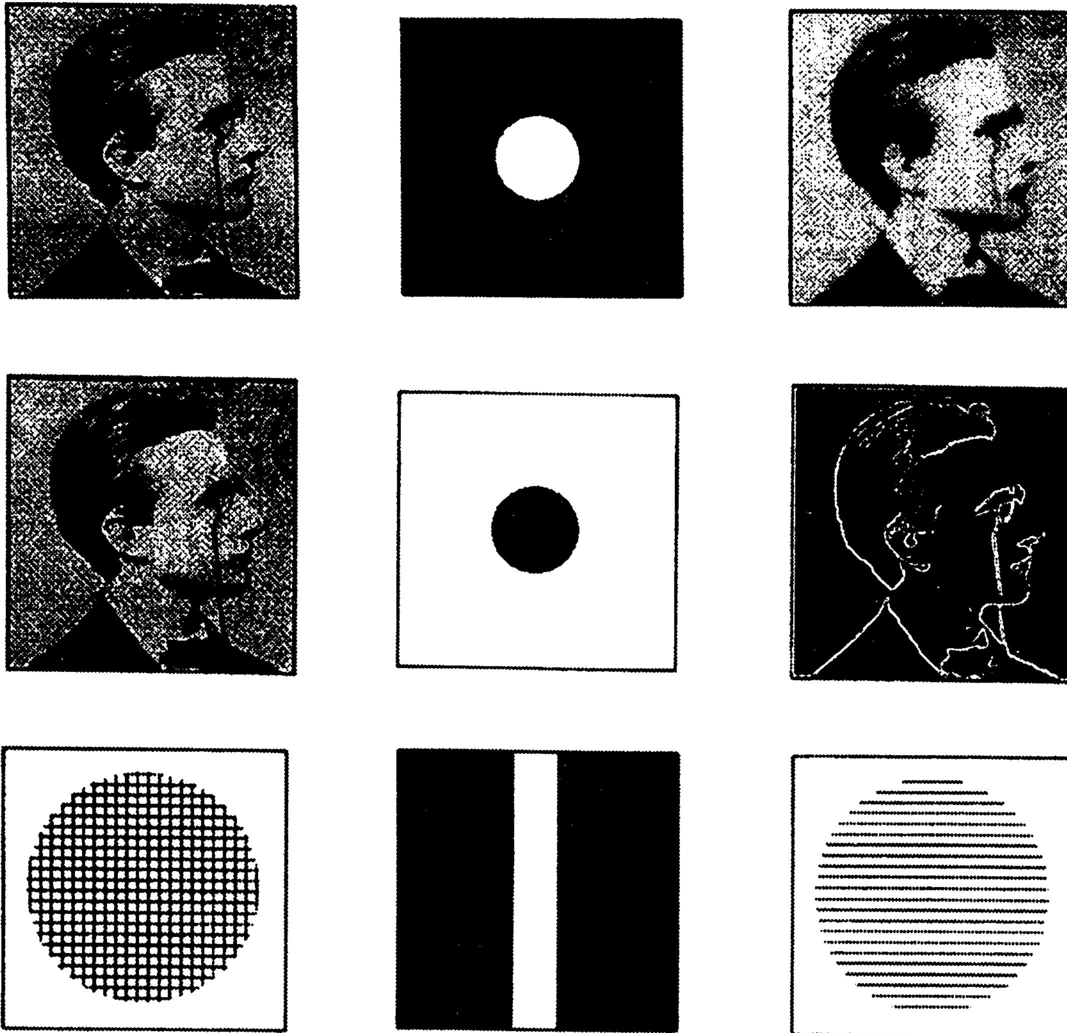


Figure 6.34: Examples of Object, Mask, and Filtered Image for Three Spatial Filters: (a) Low-pass Filter; (b) High-pass Filter; (c) Vertical-pass Filter. Black Shows No Transmittance While White Shows Full Transmittance

approximation of the electromagnetic theory, in which light is described by a single scalar function of position and time.

Theory

An electromagnetic field is described by two related vector fields: the electric field $\mathcal{E}(r,t)$ and the magnetic field $\mathcal{H}(r,t)$. Both are vector functions of position and time. In general six scalar functions of position and time are required to describe light in free-space. These are all related and satisfy the following **Maxwell's equations in free-space.**

$$\nabla \times \mathcal{H} = \epsilon_0 \frac{\partial \mathcal{E}}{\partial t} \quad (6.84)$$

$$\nabla \times \mathcal{E} = -\mu_0 \frac{\partial \mathcal{H}}{\partial t} \quad (6.85)$$

$$\nabla \cdot \mathcal{E} = 0 \quad (6.86)$$

$$\nabla \cdot \mathcal{H} = 0 \quad (6.87)$$

where the constants ϵ_0 and μ_0 (in MKS units) are the **electric permittivity** and the **electric permeability** respectively of free space; $\nabla\cdot$ and $\nabla\times$ are the divergence and curl operations.

Relationship to Other Theories

The wave optic theory is an approximation of the electromagnetic theory, in which light is described by a single scalar function of position and time. Hence electromagnetic optics encompasses wave optics which in turn encompasses ray optics.

6.3 Representation of Information

This Section presents various ways in which information can be coded as one of the many possible attributes of light.

6.3.1 Motivation

As the previous sections indicate, the phenomena exhibited by light are complex to understand, to model and to manipulate in a controlled manner. There are too many parameters/attributes that could be used to represent information. The numerous architectures proposed and analyzed in the literature tended to be fairly ad hoc in the way information was coded and represented on to the various attributes of light.

Hence there is a need for a systematic treatment of information representation and codification in the field of optical computing.

6.3.2 Attributes

The very first step in such an approach is the examination of various attributes of light and the degree to which they can be altered, and the precision with which they can be adjusted and detected and the levels of signal/noise ratio they can tolerate either due to external disturbances or due to component malfunctions such as abberations.

Some of the attributes onto which, continuous variables can be mapped are as follows:

Frequency

$$domain = visible\ spectrum\ with\ \lambda_{violet} = 380nm \rightarrow \lambda_{red} = 750nm$$

This is rarely employed because of the difficulty in changing frequencies.

Position of the Wave in 3D-Space

domain = x-range × y-range × z-range

The position attribute has been extensively exploited in 2-D by most architectures designed to exploit the spatial parallelism. However, full exploitation of 3-D offers further scope in selected areas like volume holography, 3-D interconnection networks.

Intensity/Amplitude of the Wave Function

At a given point in space and time, the intensity/amplitude varies within the domain where

domain = 0 to Maximum Intensity.

Polarization Angle at a Given Point in Space

domain = 0 to π

Phase of the Wavefront at a Point in Space

domain = 0 to 2π

6.3.3 Discussion

Depending upon the ability of the detector/manipulator devices as well as the signal/noise ratio and other factors like decay, fading etc., different analog ranges could be mapped into the ranges (domains) of the above attributes.

This does not however mean that all the above forms of analog information coding are usable in all contexts. They are presented for the sake of completeness. Whether

or not devices would be available to exploit all the above attributes to their full extent is not clear.

Different forms of information encoding are suitable for different kinds of operations. One of the open areas for the design of optical architectures is in the design of architectures that convert information coded on one attribute into information in another attribute. Such information translator units greatly facilitate the task of an architect.

There are limits on spatial resolution because of diffraction limits. Interpretation of continuous 2-D/3-D space digital pixels/voxels is usually restricted by the interference which in turn is dependent upon frequency. Each pixel in turn may carry analog information either in the third dimension or in time. Most image processing applications use 2-D information function.

In addition to the traditional analog to digital mapping schemes that could right away be incorporated on to the appropriate attribute, a number of ingenious and unique coding schemes like 2-D logic, shadow casting are also possible and are specific to the optics field.

A comprehensive compilation and classification of such techniques which is outside the scope of this Thesis work, is highly desirable.

6.4 Component Models

Following the modelling techniques of Section 6.1, theories of light presented in Section 6.2, information representation techniques presented in Section 6.3, a number of components were modelled. These components include different kinds of sources, such as laser, LED; detectors such as photodiodes; and manipulators such as lenses,

mirrors, prisms, and SLMs.

Each component model has two aspects:

1. *Structural Model.*

   This includes the geometry, graphical shape, and an iconic representation.

2. *Behavioral Model.*

   This includes a specification for what the component under study should do when excited or when presented an event.

Primarily due to the incomplete nature of ongoing work in producing sufficiently accurate and validated models for components, and secondarily due to space limitations, attention is confined to the model of one component – the laser. Even with the laser, the behavioral model is fairly crude in its current form.

6.4.1 Structural Model of a Laser

The `draw[]` function described in Chapter 5 is used to display icons of components. One example of a laser is described next.

The draw function of the *Laser* object goes through the following steps of construction.

1. *External parameters*

   height is the total height of laser.

   positionedHeight is the height of the actual laser bar on the vertical stand.

   theta is the resolution of the graphics produced. This defines the smoothness of the object's surface.

laserLength is the length of the laser bar.

```
height=2;
positionedHeight=1.5;
theta=0.25;
laserLength=1.5;
```

2. The base of the laser stand is conical with specified height and radius. The height of the base is 1/10 of the laser's total height. The radius at the bottom of the base is 1/5 of the laser's total height. The radius at the top of the base is 1/5 of the laser's total height. The bottom of the base should be closed so as to give a solid appearance.

   ```
   base = truncatedCone[height/5, height/10, height/9, theta,
                                       BottomCover -> True];
   ```

   Details of `truncatedCone[]` can be found in Part IV.

   Figure 6.35 shows a snapshot of the base from one camera position.

3. The vertical bar to fit on top of the base. This should start from the height of the base and extend upwards for 9/10 of the laser's total height. Hence combining the base and this bar will be the total height of the laser. This involves instantiating a cylinder object of height 9/10 of the total height. The top cover will give it the solid appearance. Translating this bar by 1/10 of the laser's height will bring it to the top of the base.

   ```
   verBar = transformMma3D[
              cylinder[height/10, 9/10 height,
   ```

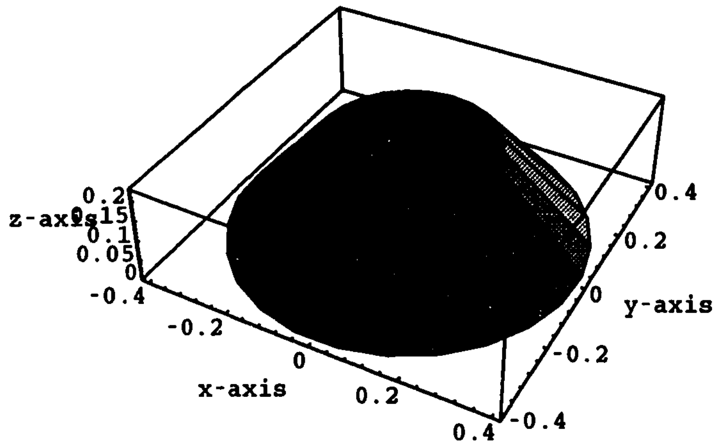


Figure 6.35: The Base of the Laser

```
                theta, TopCover -> True],
            {translateBy[{0, 0, height/10}]}
];
```

Figure 6.36 shows the vertical bar on its own.

The base and the vertical bar combined are shown in Figure 6.37.

4. A movable sleeve that slides over the vertical bar is needed at the height specified by positionedHeight. This is done by instantiating a cylinder of the appropriate dimensions and translating it to the desired height. Both top and bottom should be closed.

```
sleeve =
    transformMma3D[
        cylinder[height/7.5, 9/40 height, theta,
                  TopCover -> True, BottomCover -> True],
        {translateBy[{0, 0, positionedHeight-1/2 9/40 height}]}
```

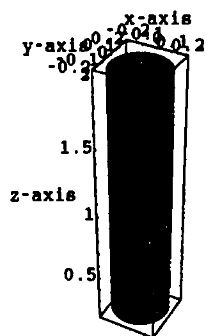


Figure 6.36: The Vertical Bar of the Laser

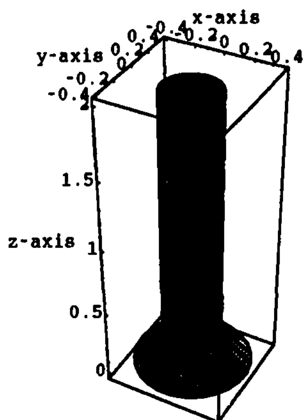


Figure 6.37: Combining the Vertical Bar and the Base

```
];
```

Figure 6.38 shows the cylinder that acts as the sleeve. Figure 6.39 shows the translated sleeve with reference to the base. Figure 6.40 shows the vertical bar with the sleeve sliding over it. Figure 6.41 finally shows the complete laser stand.

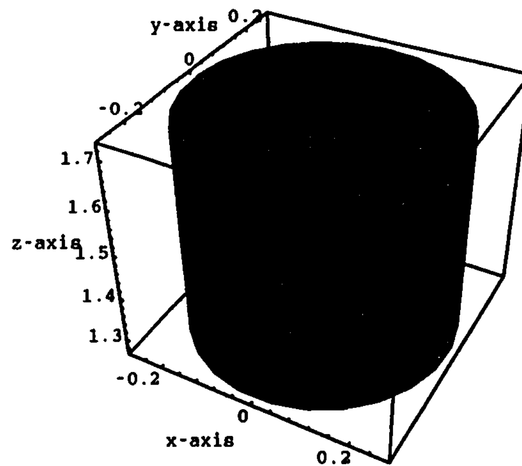


Figure 6.38: The Sliding Sleeve for the Vertical Bar

5. The actual laser is simply a horizontal bar of a certain length. This can be easily made by instantiating a cylinder object and rotating it by an angle of 90°. Then it can be translated to the height of the sleeve.

```
laserBar =
    transformMma3D[
         cylinder[laserLength/10, laserLength, theta,
                  TopCover -> True, BottomCover -> True],
         {
          rotateAroundYBy[angle[90 degree]],
          translateBy[{-laserLength/3, -height/7.5,
```

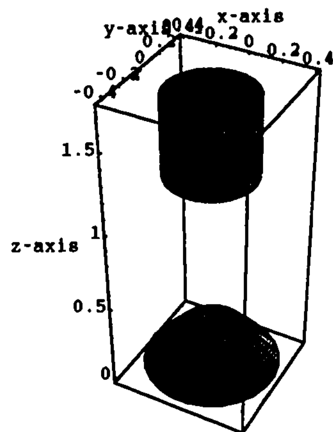


Figure 6.39: Positioning the Sleeve Over the Base

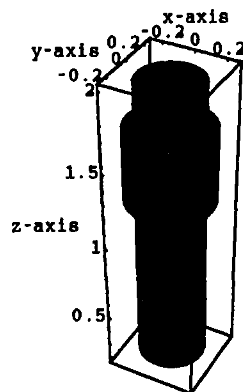


Figure 6.40: "Sliding" the Sleeve on the Vertical Bar

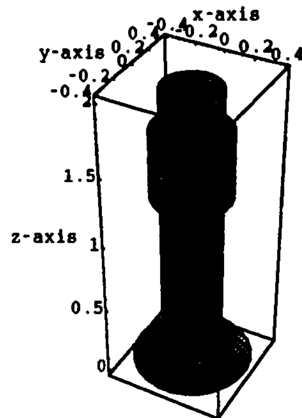


Figure 6.41: The Complete Stand for the Laser

```
                                positionHeight}]
        }
    ];
```

Figure 6.42 shows the cylinder that will be used for the laser bar. Figure 6.43 shows the laser bar with the sleeve. Figure 6.44 shows the laser bar with reference to the base to establish the correct positioning. Figure 6.45 shows the sleeve and laser bar with the base. The laser bar and the vertical bar are shown in Figure 6.46. Sliding the sleeve, laser bar combination over the vertical bar yields the setup shown in Figure 6.47. Figure 6.48 finally shows the complete laser setup.

6. The emitting beam can be shown as a cylinder inside the laser bar. Transformations similar to that of the laser bar will yield the cylinder that acts as the beam. The red color is appropriate for this cylinder.

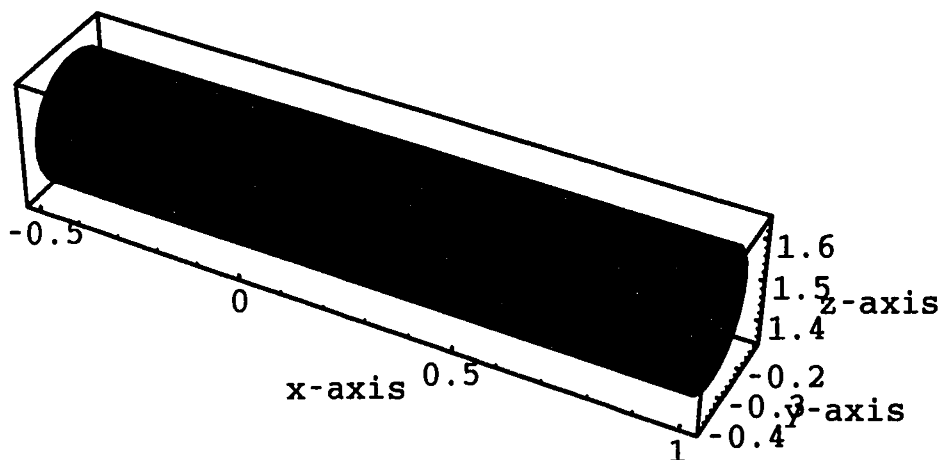


Figure 6.42: The Laser Itself – Horizontal Cylinderical Shape

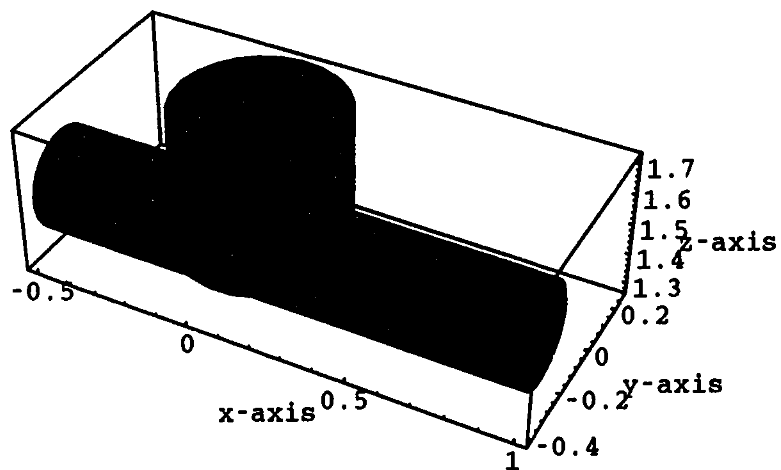


Figure 6.43: “Welding” the Sleeve to the Laser Bar

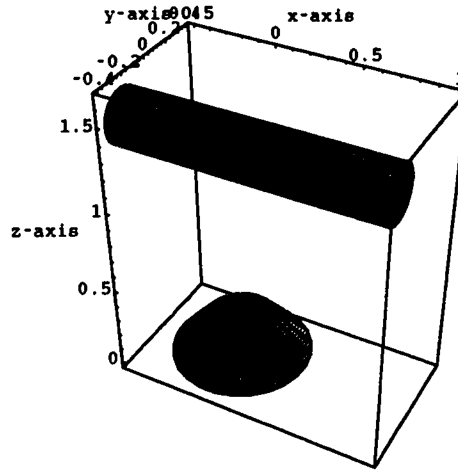


Figure 6.44: Positioning the Laser Bar over the Base

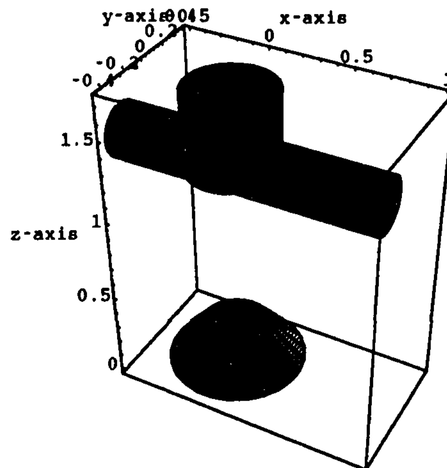


Figure 6.45: View of the Sleeve and Laser Combination with the Base

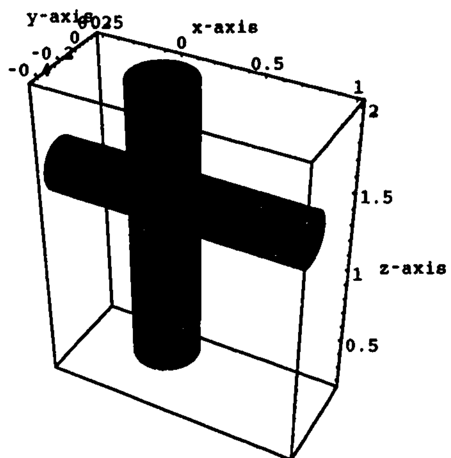


Figure 6.46: Snapshot of the Vertical Bar Perpendicular to the Laser

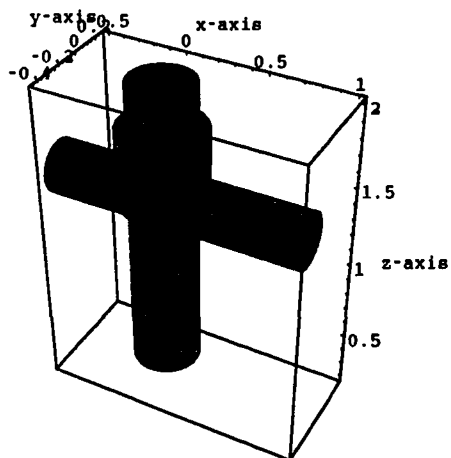


Figure 6.47: "Mounting" the Sleeve and Laser Combination over the Vertical Bar

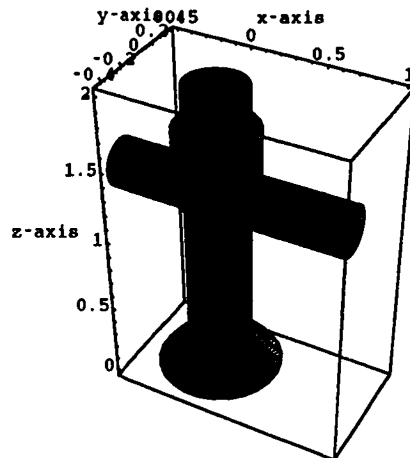


Figure 6.48: The Complete Laser Assembly

```
beam =
    {SurfaceColor[RGBColor[1, 0, 0]],
     transformMma3D[
         cylinder[laserLength/20, laserLength, theta,
                 TopCover -> True, BottomCover -> True],
          {
           rotateAroundYBy[angle[90 degree]],
           translateBy[{-laserLength/500, -height/7.5,
                positionHeight}
    ]
    }
]}
```

Figure 6.49 shows the cylinder that is used as a beam. Figure 6.50 show this beam with the laser bar. The complete laser along with the emitting beam is shown in Figure 6.51.

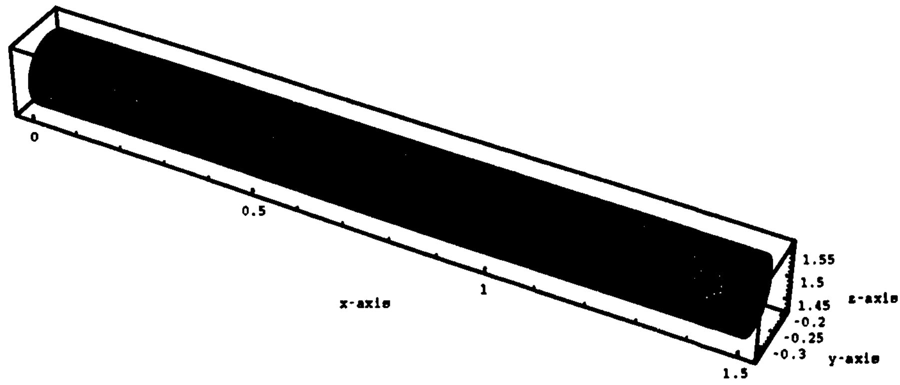


Figure 6.49: The Beam to be Emitted

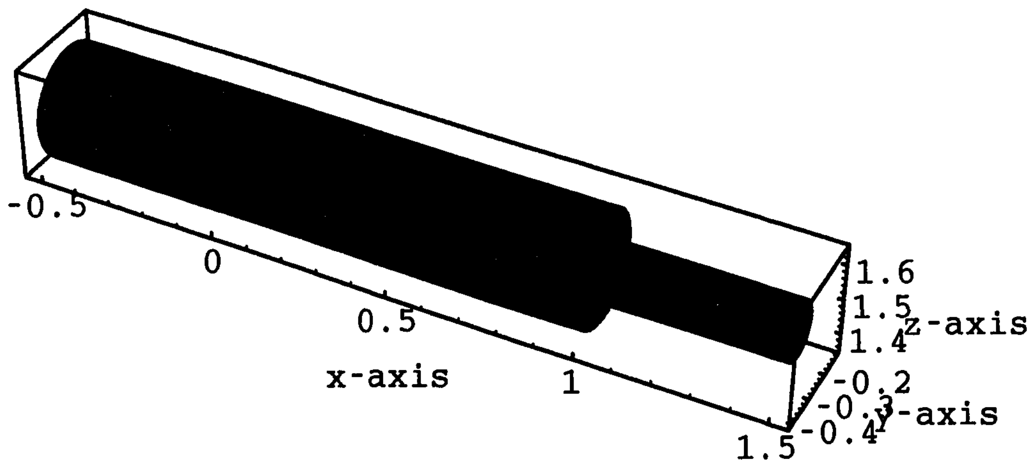


Figure 6.50: The Laser "Emitting" the Beam

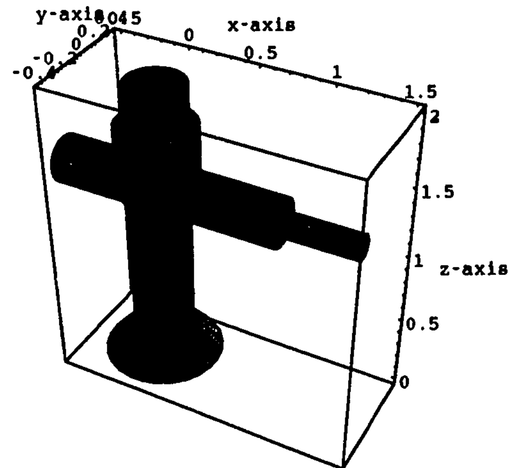


Figure 6.51: The Laser Assembly Complete with an Emitting Beam

6.4.2 Behavioral Model of a Laser

The behavioral model of every component is described in **simulateComponent[]** of the *OptiCAD* system. This essentially codes the finite state machine model of the component. The next state of the component and its output is a function of its current state and the type of input message it receives.

The state machine for the laser is summarized in Table 6.1.

A part of the *Mathematica* code for the laser model is given next.

```
state = State /. object;
messageType = MessageType /. message;

Switch[messageType,
    Start,
        Switch[state,
            Down,
              changeState[ObjectNumber /. object, Up];
              action = none;
```

Current State	Input Type	Next State	Output Produced
Down	Start	Up	Laser Beam
Down	Stop	Error	-NA-
Down	Normal	Error	-NA-
Up	Start	-NA-	Laser Beam
Up	Stop	Down	None
Up	Normal	Up	Laser Beam

Table 6.1: The Finite State Machine for Laser.

```
          DPrint["Initialized Laser"],
        Up,
          action = illegalAction,
        _,
          action = none;
          Print["Error::SimulateComponent[]:Laser already Up"]
    ],
Stop,
    Switch[state,
        Up,
          changeState[ObjectNumber /. object, Down];
          action = none;
          DPrint["Shutting off Laser"],
        Down,
          action = illegalAction,
        _,
          action = none;
          Print["Error::SimulateComponent[]:Laser already Down"]
    ],
Normal,
    Switch[state,
        Up,
          action = If[eqQ[First[orientationOf[object]], 0.0],
                      {absolute[{1, 0, 0}]},
                      {absolute[{-1, 0, 0}]}
                   ];
          DPrint["Laser ", action],
        Down,
```

```
                action = illegalAction,
              -,
                action = none;
                Print["Error::SimulateComponent[]:Laser Down"]
          ],

    -,
          Print["Warning::simulateComponent[Laser[]]:",
                "Unknown Message Type", message]
];

Switch[action,
    illegalAction,
          Print["simulateComponent[Laser[]]: Unexpected message ",
                  object, message]; {},
    none, {},
    -,
          performActions[object, PickFirst[Prepend[message,
                                                Action -> action]]]
];
```

6.5 Vendor Supplied Information & Catalog Data

The previous section presented components/devices in an idealized and abstract manner. While comprehensive work along those lines is desirable for the design and implementation of a realistic CAD system, often it is desirable to make use of only those components that are produced and supplied by some supplier.

Also different vendors use different manufacturing processes/techniques and materials to realize components. Thus the same generic component from different vendors could have different behavior.

Many components are parameterized and the different parameter combinations are tabulated in the catalogs supplied by different vendors. In addition, many vendors

accept specifications for components from users and custom design such components.

The catalogs that were consulted for this work were supplied by the following vendors.

1. Melles-Griot
2. Newport
3. Uniphase
4. Lambda Physic
5. CVI
6. Spectra Physics
7. laser Precision
8. EKSMA
9. Edmund Scientific
10. Hamamatsu

For a better understanding of the practices of optical components, the significant attributes of the supplied components were extracted. These details are maintained in the form of component databases. Using these databases of off-the-shelf components in the design of an architecture should make it easier for the architect to port his design to a working prototype. The diversity, parameters and data is so voluminous that they are omitted from this thesis for want of space and time.

6.6 Summary

This Chapter presented the modelling of optical components. Modelling involves the study of the behavior of components and encoding this as executable specifications. Basic concepts of modelling including the motives for model building, different types of models, and a modelling approach for optical components were presented. Ray optics, wave optics, beam optics and Fourier optics were discussed in detail. The detailed model of a laser was presented.

Chapter 7

Simulation

Chapter Abstract

Issues related to simulation in general and to optical architectures in particular are presented. Various kinds of simulation techniques are discussed. A methodology for the simulation of optical architectures is presented.

7.1 Introduction

Simulation is the imitation of operations of various kinds of real-world facilities or processes. Assumptions must be made about the working of each system under simulation study. The set of assumptions, usually in the form of mathematical or logical relationships, constitutes a *model*. A model is used to gain understanding of how the corresponding system behaves.

The next Section discusses the motivation for the analysis of optical architectures. Section 7.1.2 introduces simulation and the different ways of studying a system. Sec-

tion 7.1.3 outlines different simulation models and their classification. Section 7.1.4 discusses discrete event simulation and its different components.

7.1.1 Motivation for the Analysis of Optical Architectures and Systems

In order to design and test an optical architecture and establish its correctness, it is necessary to set up an actual experiment involving physical components. This is a time consuming task. Design of large practical optical systems requires availability and use of CAD systems. These CAD systems should allow the description of experimental setups involving optical architectures. Analysis of the architecture is a must to establish its correctness.

The very nature of light requires keeping track of analog quantities in the course of a simulation. These quantities are complex, often governed by complicated laws of physics, involving several differential equations. A true simulation would involve maintaining exact values of these quantities throughout the process. In addition to maintaining the attributes of components/light during simulation, other concerns such as mechanical stability, feedback loops and overall power dissipation must also be addressed.

Following are some of the important simulation results of interest to optical architectures.

Surfaces, Lenses, Image Planes

A count of how many times a signal passes through each of these is important in determining the final output's intensity, polarization, delay and other attributes.

Input Power

Calculating the losses encountered by a signal travelling through an architecture, the intensity of the input signal can be determined so as to achieve an acceptable output power.

Delay Analysis in Space

Related to calculating the delay encountered by a signal travelling through an architecture. Delay analysis could be:

input dependent which is specific to particular input signals.

input independent independent of the input signals. The results of this analysis depends only on the signals originating from sources within the architecture.

Heat Dissipation in Space and Time

An important concern in optical architectures is the heat dissipation problem. It is necessary to place the components in such a way that the heat is quickly taken out of the system. Many components have operating temperature ranges. They exhibit unpredictable behavior outside these ranges.

Maximum Operable Clock

The delay of signals from inputs to outputs will determine the maximum clock speed for synchronous architectures.

Number of Fourier Transforms

Each pass through a lens has the effect of a Fourier transform on the signal passing through it. The total number of lens passes is important in an optical architecture.

Number of Image Multiplications

Since most of the time information is encoded in the form of a 2-D matrix, most operations are realized as matrix multiplications.

Number of Operations per Seconds

Once the clock speed is computed for synchronous architectures, the number of operations can be easily determined.

Intensity

As the signal travels from component to component, its intensity changes. It is important to keep track of these changes and report to the designer in case the intensity falls below a certain minimum.

Polarization

Conditional decisions in an optical architecture are usually taken by changing the angle of polarization of the light signal. The architect might verify the functionality of the architecture by probing at various points and finding out about the polarization angle of the signal.

Crosstalk and Noise

It is important to keep track of any noise introduced due to any reason. Although there is very little interference between light signals, there may be some crosstalk/noise because of other factors such as magnetic fields, temperature.

7.1.2 Preliminaries for Simulation

A *system* is a collection of entities, e.g., people or machines that act and interact with each other towards the accomplishment of some logical end. The system contains only

those entities of interest to a particular aspect under study.

The *state* of a system is a collection of variables needed to describe a system at a particular time. These variables also depend on the objectives of the study.

Optical systems may be classified either as *discrete* or *continuous.* A discrete system is one in which the state variables change instantaneously at discrete points in time. For example, in an optical system, variables may be changed only when a beam of light strikes or leaves a component. A continuous system on the other hand is one in which the state variables change continuously with time. For example light travelling from one component to another may fit into that category and can be modelled by a set of partial differential equations. These will describe a continuous system.

Before attempting to simulate the behavior of a system, it is necessary to understand how the system works. The better the system is studied, the more accurate its model will be which will lead to more accurate simulation results.

Following are the different ways of studying a system (see Figure 7.1).

- *Experimentation with the actual system vs. experimentation with a model of the system:*

  In real life it is rarely possible to experiment with an actual system. This involves making changes to the existing system and studying its behavior. This, although would be the most accurate form of study, it would be too costly and threatens to disrupt the system. Moreover, the actual system may not exist at all. The experiment may involve varying some parameters of a proposed system in order to come up with a good alternative.

  In optical computing, experimenting with the actual system involves setting up an architecture containing all the components and then varying some of its

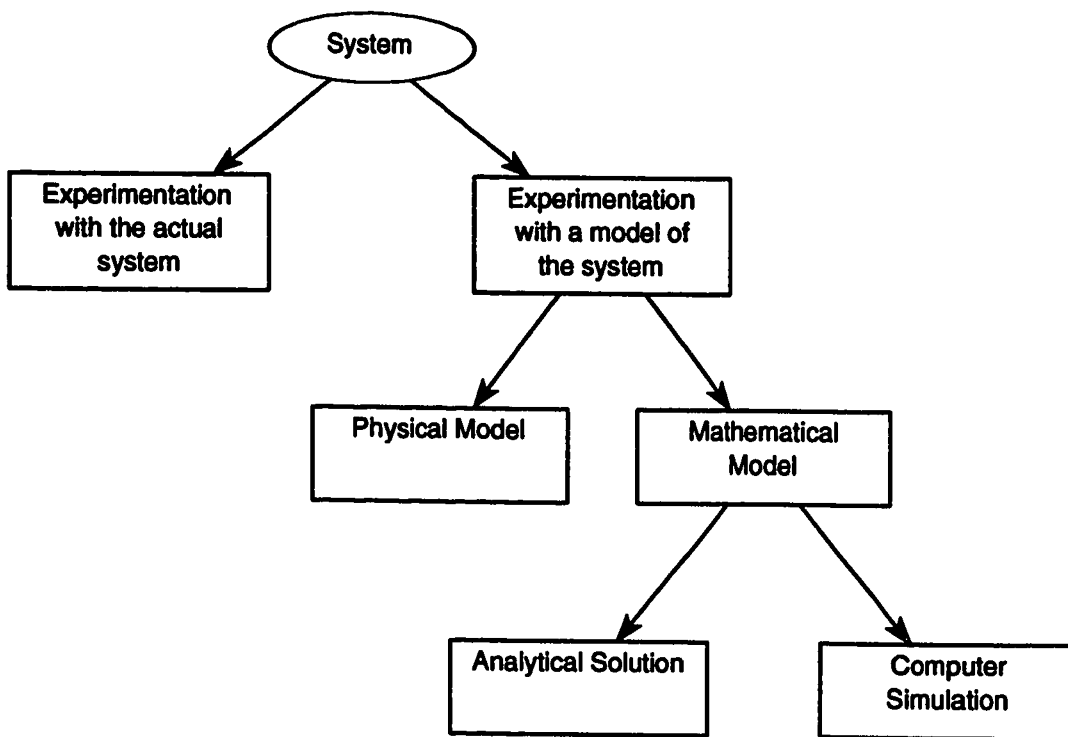


Figure 7.1: Ways to Study a System

attributes to study their effect on the output. This is time consuming but is the best (and only) alternative as models of optical components are not well understood.

Experimenting with a model of the system is a more cost effective solution. The only problem is the *validity* of the model, i.e., how accurately it represents the actual system.

- *Physical model vs. mathematical model:*

  Physical models are concrete models that are built so as to capture most of the physical characteristics of the actual system. For example to design a register of 8 memory elements, a single memory system can be set up as an experiment involving actual components. This could be treated as a simple physical model of the entire register.

  Mathematical models represent the system in terms of logical and quantitative relationships that are then manipulated and changed to see how the model reacts, and thus how the system would react (provided the mathematical model is a valid one).

- *Analytical solution vs. simulation:*

  Once a mathematical model is built, it must be examined to see whether it can answer the questions of interest about the system it is supposed to represent. If the model is simple, it may be possible to work analytically to get a solution. However, mostly the models are complex involving several complex relations. It is difficult to get a closed form solution using analytical techniques. In such a case, it is necessary to take the brute force approach in simulating the system.

This involves numerically exercising the model for the inputs in question to see how they effect the output in question.

7.1.3 Simulation Models and Their Classification

- *Static vs. Dynamic Simulation Models*

  A static simulation model is one that represents a system at a particular time, or one in which time does not play any role. A dynamic model is one that evolves with time.

- *Deterministic vs. Stochastic Simulation Models*

  A simulation model that does not contain any probabilistic components is called a deterministic model. For example a system of differential equations representing a system could be one such model. The output is determined once the set of input quantities is known.

  Systems that are modelled as having some random input components are called stochastic. Stochastic simulation models produce output that is itself random, and must therefore be treated as an estimate of the true characteristics of the model.

- *Continuous vs. Discrete Simulation Models*

  Discrete models of a system evolve over time by a representation in which the state variables change instantaneously at separate points in time. In contrast, continuous simulation models evolve over time in which state variables change continuously in time. A discrete (continuous) model is not always used to model a discrete (continuous) system.

7.1.4 Discrete Event Simulation

Discrete event simulation concerns the modelling of a system that can change only a countable number of points in time. These points in time are the ones at which an event occurs, where an event is defined as an instantaneous occurrence that may change the state of a system.

The *simulation clock* is the variable that gives the current value of simulated time. As the simulation progresses, the value of this clock increases. Two methods can be used to advance the simulation clock – the *next-event time advance* and *fixed-increment time advance.* The former is more popular and relies on the occurrence of an event to advance the simulation clock.

Discrete event simulation models share a number of common components. Some of them are discussed as follows.

System State:

The collection of state variables necessary to describe a system at a particular time.

Simulation Clock:

A variable giving the current value of simulated time.

Event List:

A list containing all outstanding scheduled events that are likely to occur, usually arranged in the ascending order of time.

Statistical Counters:

Variables used for storing statistical information about system performance.

Initialization Routines:

A subprogram to initialize the simulation model at time zero.

Timing Routine:

A subprogram that determines the next event from the event list and then advances the simulation clock to the time when the event is to occur.

Event Routine:

A subprogram that updates the system state when a particular type of event occurs.

Library Routines:

A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation method.

Report Generator:

A subprogram that computes estimates of the desired measures of performance and produces a report when the simulation ends.

Main Program:

A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.

7.2 A Discrete Event Simulator for Digital Optical Architecture

This section presents a discrete event simulator for digital optical architectures. The working are presented along with the motivation for the design decisions.

Figure 7.2 shows the top level view of the simulator. It consists of a *message handler*, an *event handler* and *objects/components*. Communication between these three is done through message exchanges. The next few subsections describe in detail the working of the simulator.

7.2.1 Global (System Level) vs. Local (Component Level) Simulation

There are two levels of abstraction when studying the *OptiCAD* simulator. One is the *global* view (inter-component) and the other is the *local* view (intra-component).

The global simulation deals with the interaction among objects. The details of each object/component are hidden/ignored at this level. Messages are sent to and from objects. They are converted into events to be scheduled at a later time.

The local simulation deals with the internals of each component. Each component can be viewed as a state machine responding to events. Upon receiving an event, the component transits to the next state. As a side-effect a set of messages is generated.

This distinction and separation of the simulation task into local vs. global simulation has numerous advantages. The most important one is the simplicity and flexibility of design. Design issues are quite different at both these levels. They are

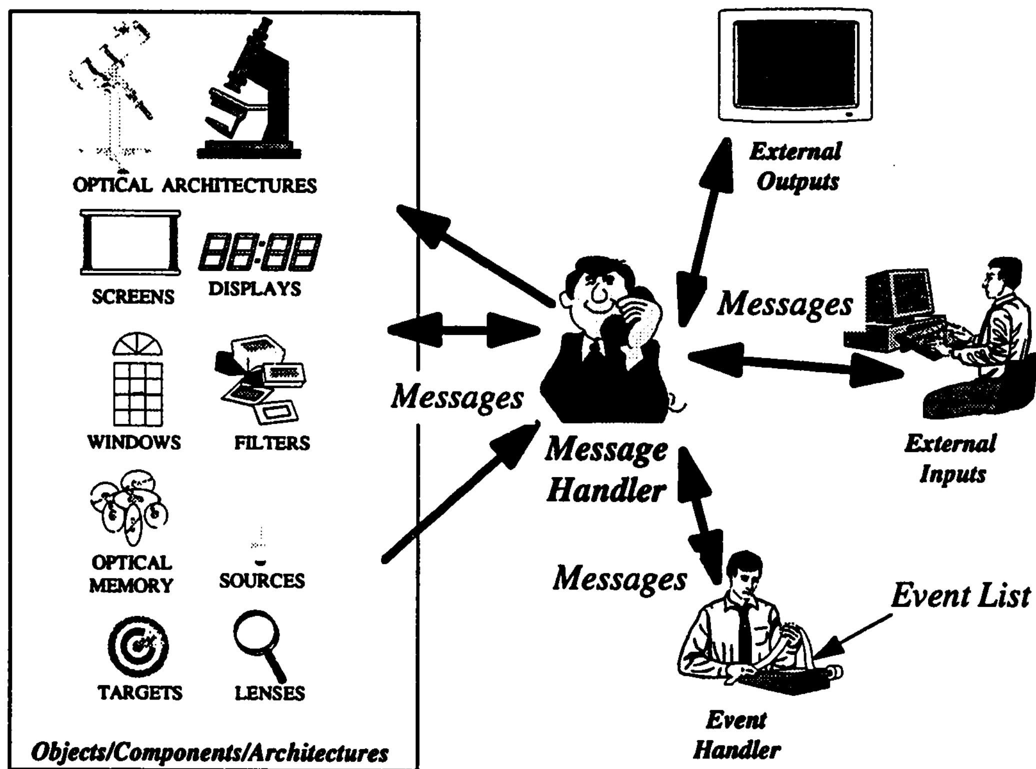


Figure 7.2: Event List/Handler, Components, Architectures, & Message Handler

better handled at two different levels. Moreover, separation of functionalities between global and component level means that new components can be added without affecting global simulator. The internal details of the component itself are hidden from other objects in the system. A completely different simulation methodology might be employed by each component's local simulator. Another advantage is that a component could be an architecture itself. Here, a recursive (hierarchical) simulation will be needed. This would be very difficult to realize in the absence of abstraction.

7.2.2 Events and Event Handling

Events are occurrences of importance to the optical system. These include a light beam striking an object or a beam leaving an object. Additional events include the violation of a constraint e.g. a component's temperature might exceed the specified operating temperature.

At the global level, one of the major objects participating in the simulation is the event handler. The event handler maintains an agenda of events with event times. These entries in the event list are the result of messages from components that specify the occurrence of events. Other entries can be given from external inputs.

The event handler performs the following tasks.

- *Receive messages from components*

  The components after processing a message send as their output other messages to the event handler. These messages may contain instructions to schedule events at different times. They may also instruct the message handler to remove certain scheduled events from the event agenda.

  For example consider a component – *beam-splitter.* An event is scheduled so

that a beam strikes a face of the beam-splitter. As a result of this event, the beam-splitter splits the beam and determines the destination of its two output beams and the time when they will hit other components. This information is sent to the event handler as a message. This message instructs the message handler to schedule two events corresponding to the two output beams.

- *Trigger events*

  The event handler is responsible for controlling the working of the entire simulator. The events and their order dictate the signals that will be travelling through the architecture. The event handler must pick the next event and take the appropriate action by initiating a message to the component responsible for processing that event.

- *Modify the event list*

  Some messages may cause the deletion of entries of the event list. Such messages can be external, sent by the global simulator, or internal, sent by components.

7.2.3 Messages and Their Handling

Messages are the means of communication between different objects participating in the simulation. They are sent from and received by components, event handlers and external inputs.

In any simulation system the entities need to communicate. This communication is essential for ensuring that they work together as a single system. One way of realizing this communication is to have communication protocols between different types of components. This solution is too rigid as the type of components is not known

beforehand. Message passing is the most general mechanism for this communication.

The message is a sequence of pairs of the form (*AttributeName*, *AttributeValue*). The values are given by the message sender and keep track of many of the attributes of the signals e.g. delay, intensity, polarization etc. The message also contains values for manipulating the event list. These values are interpreted by the event handler to make the necessary modifications to the event list.

7.2.4 Geometric Calculations for Next Component Determination

One of the most important calculations in the simulation of optical architectures is the determination of the next component i.e., which component does a beam strike after it leaves a certain component.

The problem is complex because the optical architecture is a collection of components in 3-D. Each component is some arbitrary shape placed and oriented in a specified manner. Once a beam leaves a component, it travels according to the laws of light propagation and either strikes the surface of a component or escapes from the system.

The problem is to determine exactly which surface the beam strikes and with what orientation. This is a problem of geometric calculations when considering ray optics where light travels in a straight line as a ray.

This problem is in general computationally expensive to solve when the surfaces of components involved are complex. However, the problem can be simplified to a manageable one after making certain simplifying assumptions. One assumption is that each component is of a regular shape – cuboid. There is no loss of generality

here since this assumption is in line with the notion of a bounding box of a component. Since all the six faces of the bounding box of every component are known, the problem reduces to finding the point of intersection of the ray (represented as a vector) with all the surfaces. The subsequent task of determining what happens inside the bounding box can be best left to the individual component level calculations.

The following subsection summarizes the idea as an algorithm.

7.2.5 The Algorithm for Determination of the Next Component During Simulation

The algorithm employed for the determination of the next component is presented as a pseudo code.

1. **[Formation of Attribute Vector]**:

   The ray leaves a component C_1. It is represented as a vector v with attributes.

2. **[Line Corresponding to Vector]**:

   Find the equation of the line representing the vector v.

3. **[Determination of Points of Intersection with the Faces of Bounding Boxes of Components]**:

   For all six faces of each of the components' bounding boxes in the architecture do:

   (a) **[Equation of the Plane Passing through the Face Under Consideration]**:

   Find the equation of the plane representing the face.

(b) **[Point of Intersection of Line with the Plane]**:

Find point of intersection of the vector v with all faces.

(c) **[Does the Line Intersect the Plane?]**:

If point is real then accept otherwise reject.

(d) **[Is the Point of Intersection Within the Face]**:

If the point of intersection lies within the boundaries of the face, then accept otherwise reject. At this point the face is a four sided polygon (not a rectangle). Finding whether the point of intersection lies in this polygon is a standard basic problem in computational geometry. The steps for determining whether the point of intersection lies on the face are as follows.

i. **[Triangulation]**:

Divide the polygon into *non-intersecting* triangles.

ii. **[Is the Point in a Triangle?]**:

Find out whether the point lies in any of these triangles. The following steps will determine whether a point lies in triangle.

A. Divide the triangle into three triangles so that the point under consideration is a vertex of all three.

B. If the sum of the areas of these is equal to the area of the full triangle, then the point lies in the triangle.

iii. If it does, then it lies on the polygon otherwise not.

4. **[Closest Point]**:

From all the found points, choose one that is nearest to the vector's origin in the direction of the vector.

5. **[Nearest Face in the way of Vector]**:

   The found plane is a side of the target component.

7.3 Ray Tracing

One of the simplest ways of modelling light is as a ray. This model is sufficient in determining the paths that the light will take through the architecture. As simulation progresses it is necessary to *trace* the rays through the architecture so that other analyses can be done later.

As inputs/sources are responsible for introducing rays in the architecture, ray tracing starts from the sources, progresses through manipulators and ends at detectors/output screens or escapes through the system's bounding box. Hence rays are traced from one component to the next. In terms of events, a ray is formed by an *exit of light* event from a component to an *entry of light* event to another component.

7.4 Additional Issues

In addition to the issues already discussed, the following constitute a major portion of the simulator.

7.4.1 Geometric Calculations

Simulation of light through an architecture involves many geometric calculations. It was this need that prompted the development of a library of geometry computation routines. Following are some of the important geometry routines.

squaredDistanceBetween:

Finds the square of the distance between two points.

distanceBetween:

Finds the shortest distance between two points.

pointOfIntersection:

Finds the point of intersection between a vector and a polygon.

pointInPolygonQ:

A predicate that returns true if the given point lies in the polygon.

areaOfTriangle:

Returns the area of a triangle.

pointInTriangleQ:

A predicate that returns True if the given point lies in the triangle.

pointInPolygonQ:

A predicate that returns true if the point lies in the polygon.

boundingBoxEquations:

A simplified function different from the one in Simulator, based on Faces info in the object. Returns a set of six equations each representing a plane passing through one face of the bounding box of a specific object.

colinearQ:

A predicate that checks whether the given set of points is colinear.

axisQ:

A predicate that checks whether the given line is the same as one of the principal axes.

xAxisQ:

A predicate that checks whether the given line is the same as the x-axis.

yAxisQ:

A predicate that checks whether the given line is the same as the y-axis.

zAxisQ:

A predicate that checks whether the given line is the same as the z-axis.

lineQ:

A predicate that checks whether a specific line is valid i.e., not a point.

pointOnThePlaneQ:

A predicate that checks whether the given point lies on the given plane.

equationOfPlane:

Given a set of points (at least three), it returns an equation of the plane on which those points lie.

normalize:

Given a vector, it returns its normalized vector.

pointInPlane:

Returns an arbitrary point on the given plane.

rayAtBoundary:

Given an input ray and a medium together with the surface between two media, returns a bundle of output rays that conform to the laws of reflection, reflection and the application of Snell's law, total internal reflection as needed.

vectorAtAnAngle:

A function that takes a ray, and an angle, computes and returns unit rays that make an angle given with the given vector.

7.4.2 Representation of Light

Light is modelled as a generic message. The message structure is similar to the one already discussed in Section 7.2.3. The ray/beam attributes are bundled into the message. This allows changing the model of light by simply introducing new attributes and/or modifying the existing ones. There is no major change at the global simulator level. As the system evolves and component models are refined, sophisticated models of light can be introduced by changing only a few parameters of the message. Obviously the interpretation and modification of such attributes requires the satisfaction of more complex laws and equations corresponding to the sophisticated model.

7.4.3 Cache - *Trading Space for Time*

The current simulator relies heavily on geometric computations. All issues of ray tracing, finding the next component, determining the exact position and orientation with which the beam strikes a component are handled geometrically. When a beam

escapes from a component, it is up to the global simulator to decide the next component that is going to receive the beam. These calculations are computationally expensive but are needed for almost every event processing in the simulation. The simulation becomes too slow when dealing with a large architecture.

Observation of the simulation led to the conclusion that most of the computations are repeated i.e., the output of a component is usually another fixed component. Instead of repeating the same computations over and again, the results can be stored in a temporary space. This space acts like a small cache memory. When a calculation is needed, the contents of the cache are consulted. If there is a *cache hit* then the results are taken from the cache. Otherwise there is a *cache miss* and geometric computations have to be carried out. The cache is updated by inserting the appropriate entry.

7.4.4 Hybrid Simulation

Due to the separation of global (system level) and local (component level) simulation, it is possible to mix several simulation methods into one. Each component has its own simulation model. The overall structure is responsible for coordinating the responses and behavior of all these models.

At the component level, it is possible to have analog simulation. Since every component can itself be an architecture, this allows integrating analog and discrete simulation techniques freely at different levels of abstraction for different parts of the system.

7.4.5 Hierarchical Simulation - *Trading Time for Space and Compactness*

An optical architecture usually involves many components. It is possible to divide an architecture into logical parts by associating functionality with groups of components. Each of these groups can be separately modelled, tested and added to the component library. Hence an architecture is a collection of smaller architectures and components.

At simulation time, it is too complex to maintain a flat view of an architecture. The simulation would not only be cumbersome, and complex for debugging but it would also be inefficient in terms of calculations such as next component determination. The simulation can be approached in a hierarchical manner.

At the highest level all the components and sub-architectures receive similar treatment. As soon as a signal strikes the bounding box of one of these, the simulation focus is shifted inside the component/sub-architecture. A simulation is carried out inside that component i.e., the local simulator is invoked recursively for that particular component, the incident signal being treated as an external input. This recursion is repeated until a primitive component is encountered whose local simulator is responsible for mimicking the functionality of the component. The bounding box of an assembly acts like a component (local simulator) to its peers and like a global simulator for the components within the assembly.

There is a considerable amount of extra overhead with this recursive approach. However, the cost is justified in terms of saving on debugging and coming up with a simple, hierarchical design.

The algorithm for simulating optical architectures can be summarized as follows.

```
simulate[object: instance of an architecture]
```

```
If object is a primitive component then
        - simulate object using the component-specific simulator
else
    for each instance architecture i in componentsOf[object] do
        - compute bounding box info for i;
        - compute orientation info for i
        - assemble an instance component ic;
        - simulate[ic]
```

The above algorithm is recursive returning only if a primitive **simulate[]** definition is encountered.

7.4.6 Ray Tracing as the Basis for the Computation of Performance Measures

There are several performance measures for optical architectures. Some important ones are delay and power analysis. These can be determined by tracing light through the architecture and recording the paths taken. Once the components through which the light passes are known, functions corresponding to the analysis can be applied at each component and the measures computed.

Ray tracing is both simple and accurate as far as path determination is concerned. Once this is known, numerous analyses can be done for the architecture.

7.4.7 Distributed/Parallel Simulation

The simulator structure is general i.e., the whole design is based on objects interacting through message passing. As the architecture becomes more complex (some architectures involve thousands of components), it is impractical to confine the simulation to one processor and expect simulation results for a large time period. Different components/sub-architectures can be distributed to different computers or processors. Message passing can still remain as the means of communication, although it may be implemented in different ways based on whether the simulation is done on a single processor machine or a parallel processor machine or on a distributed system and on the availability/non-availability of shared memory. The overall structure of the simulator, however, will remain much the same.

7.5 A Template for a Simulator

The generic format of the simulator remains similar for different architectures. It is possible to make a general purpose simulator for optical architectures. It will contain information about all possible components. Practically, this design is too inefficient and inflexible. It is not possible to know beforehand all types of components. Moreover, maintaining all components would render it too slow for practical architectures.

After developing and studying different simulators for different optical architectures, the common portions of these simulators are extracted and made into a general template. Architecture specific information i.e., the functionality of specific components can be plugged in, to generate a specific simulator for that architecture. Figure 7.3 shows the essential parts of the simulator template.

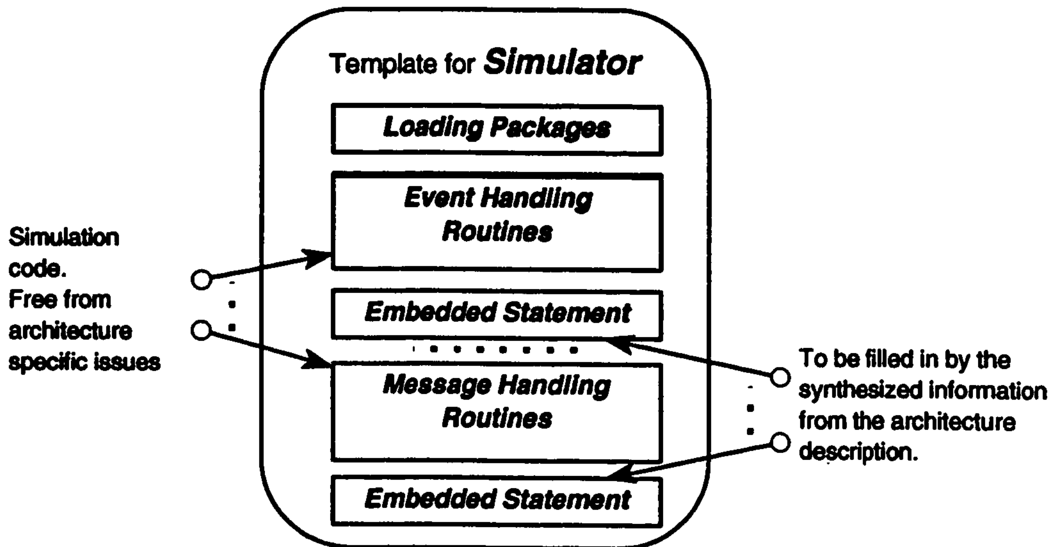


Figure 7.3: A Template for a Simulator

7.6 An Example Simulation of Trip-Flop

A generated simulator by the techniques illustrated so far was run for the Trip-Flop architecture (see details in Chapter 2). Figures 7.4 through 7.7 show some of the events in the simulation. Associated with each event is its state table (which shows the state of each component at that instant).

7.7 Summary

This Chapter presented a simulation methodology for optical architectures.

A discrete event simulator for digital optical architectures was presented. The notions of global (system level) vs. local (component level) simulation were presented. An algorithm for the simulation of optical architectures was discussed. A generic

#	Component Type	State
1	Pulsed Laser (High)	Emitting
2	Beam Splitter	InActive
3	Interference Filter	InActive
4	Beam Splitter	InActive
5	Beam Splitter	InActive
6	Beam Splitter	InActive
7	Pulsed laser (Low)	InActive
8	Mirror	InActive
9	Beam Splitter	InActive
10	Beam Splitter	InActive
11	Mirror	InActive
12	Pulsed Laser (s)	InActive

Simulation Clock = 1

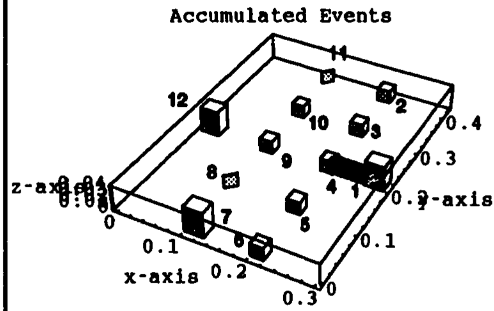


Figure 7.4: System Configuration when $Clock = 1$

#	Component Type	State
1	Pulsed Laser (High)	Emitting
2	Beam Splitter	InActive
3	Interference Filter	InActive
4	Beam Splitter	Splitting
5	Beam Splitter	Splitting
6	Beam Splitter	InActive
7	Pulsed laser (Low)	InActive
8	Mirror	InActive
9	Beam Splitter	Splitting
10	Beam Splitter	InActive
11	Mirror	InActive
12	Pulsed Laser (s)	Emitting

Simulation Clock = 5

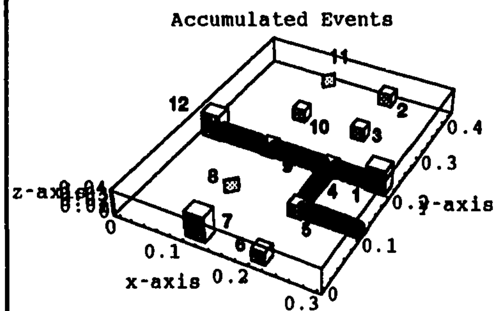


Figure 7.5: System Configuration when $Clock = 5$

#	Component Type	State
1	Pulsed Laser (High)	Emitting
2	Beam Splitter	InActive
3	Interference Filter	InActive
4	Beam Splitter	Splitting
5	Beam Splitter	Splitting
6	Beam Splitter	Splitting
7	Pulsed laser (Low)	InActive
8	Mirror	InActive
9	Beam Splitter	Splitting
10	Beam Splitter	Splitting
11	Mirror	Receiving
12	Pulsed Laser (s)	Emitting

Simulation Clock = 10

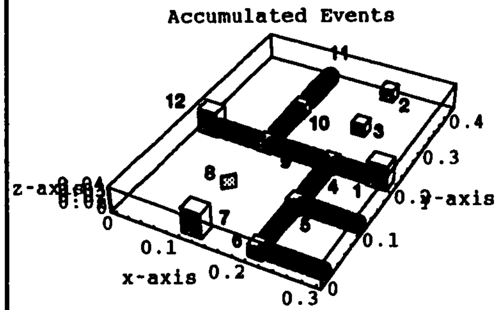


Figure 7.6: System Configuration when $Clock = 10$

#	Component Type	State
1	Pulsed Laser (High)	Emitting
2	Beam Splitter	InActive
3	Interference Filter	Reflecting
4	Beam Splitter	Splitting
5	Beam Splitter	Splitting
6	Beam Splitter	Splitting
7	Pulsed laser (Low)	InActive
8	Mirror	Receiving
9	Beam Splitter	Splitting
10	Beam Splitter	Splitting
11	Mirror	Reflecting
12	Pulsed Laser (s)	Emitting

Simulation Clock = 15

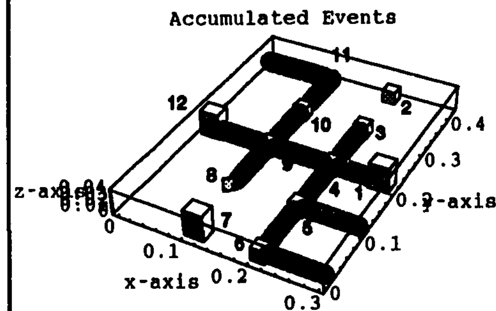


Figure 7.7: System Configuration when $Clock = 15$

template for a simulator was presented.

Chapter 8

Power and Delay Analyses

Chapter Abstract

The role of analysis in designing optical architectures is presented. A technique for obtaining the power and delay analysis by using path enumeration is outlined as an algorithm.

8.1 Introduction

Power and delay (timing of signals) analyses constitute important aspects of studying an optical architecture. This Chapter starts by giving the motivation for power and delay analyses in Section 8.1.1. It then presents a method of carrying out these analyses by using signal flow graph construction in Section 8.2. Section 8.3 describes an algorithm for using the constructed signal flow graph to enumerate the paths followed by the signals within the architecture. Section 8.1.2 and 8.1.3 introduce power and delay analysis respectively. Generic templates for power and delay analyzers are pre-

sented in Sections 8.5 and 8.7 respectively. Limitations of the current implementation and possible improvements are discussed in Sections 8.8 and 8.10. Sections 8.9 and 8.11 present other kinds of analysis and alternative approaches for analysis.

8.1.1 Need for Power and Delay Analyses

Once an architecture has been designed and placed in 3D, the next step is to find out whether the architecture fulfills its intended functionality. There is a need to simulate the architecture for different input patterns and analyze its behavior. Chapter 7 explained the mechanisms employed in simulation. The results of the simulation need to be analyzed and the correctness of the architecture established.

Analyses requires an understanding of the different kinds of measures that will be of interest to an evaluator. Some common ones follow:

- Surfaces, Lens Passes, Image Planes
- Input Power
- Power Distribution in Space and Time

  input dependent
  input independent
- Delay Analysis in Space

  input dependent
  input independent
- Heat Dissipation in Space and Time
- Scatter in Space and Time
- Volume and Surface Analyses
- Throughput
- Maximum Operable Clock for Synchronous Architectures

- Hardware Cost
- Mechanical Stability
- Number of Fourier Transforms
- Number of Image Multiplications
- Number of Operations per Second
- Degree of Parallelism
- Maximum Permitted Signal to Noise Ratio
- Resolution (Number of Lines/Pixels per Unit Length)
- Power
- Delay
- Intensity
- Polarization
- Distribution
- Crosstalk and Noise
- Coherence
- Correlation
- Volume-Time^3 Measure

Some of the above measures are *static* and others are *dynamic*. The static measures can be determined from architecture description without simulation. These are usually done by using calculations from architecture specification. The dynamic measures require simulation to help determine temporal behavioral characteristics.

In this chapter, the primary focus is to analyze the spatio-temporal aspects of power distribution and signal propagation delay. The power distribution includes checking the signal power at various points in the architecture. Delay analysis includes

finding out the time difference between the time of supplying inputs and the time of obtaining outputs.

The two measures (power and delay) can be obtained by solving the Maxwell's equations for the electric and magnetic fields. However, in practice this is too slow because of its enormous computation demands. Faster techniques are needed especially for large architectures involving several dozens to several hundreds of components.

Simulation (discussed in Chapter 6), is needed to determine the paths through which light travels. Once these are known, analyses can be carried out on these paths by applying appropriate functions at the components.

8.1.2 Power Analysis

Given an architecture, a particular excitation, and a rudimentary output out of simulation in the form of important pairs of points through which light travels, the power analysis provides power/intensity values at different points in the architecture.

For example consider the Trip-Flop architecture shown in Figure 8.1.

Following are some of the questions of interest when designing the Trip-Flop architecture.

- *What will be the intensity of the signal at points P_1, P_2, and P_3 as functions of time ?*
- *Will the input signal at point P_I be of enough intensity so that the signal incident to the interference filter at point P_4 has enough power to change the state of the interference filter ?*
- *What is the minimum power needed at point P_5, so that the beam at P_3 is detectable ?*
- *Is the assertion*

ASSERT[Input[IF] =!= s]

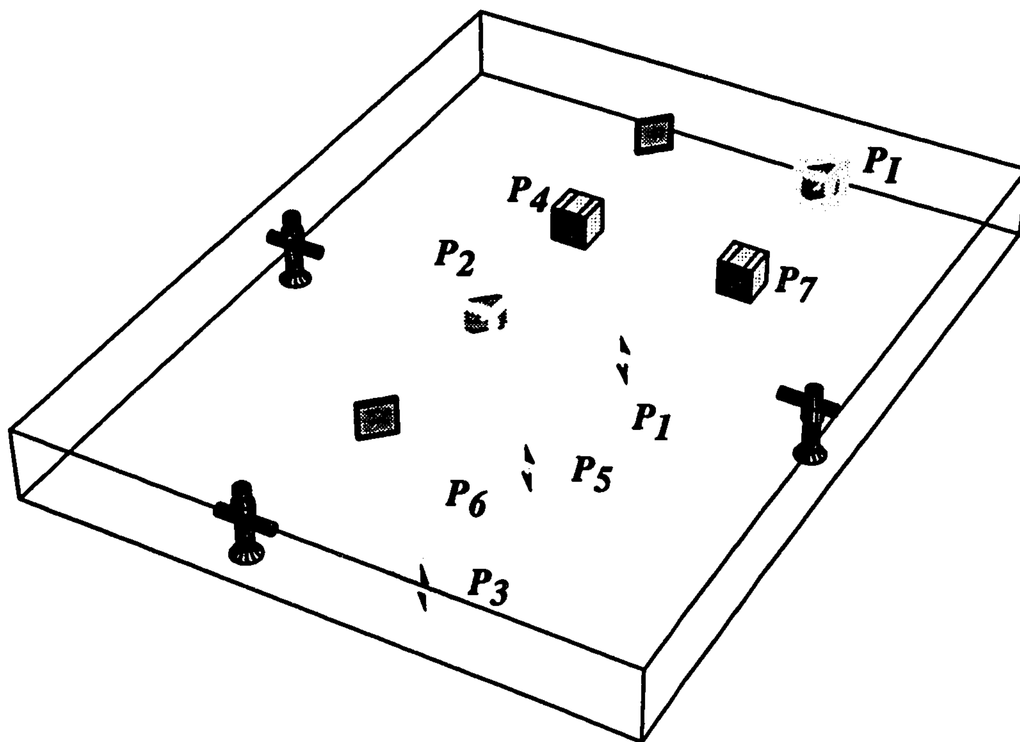


Figure 8.1: The Trip-Flop Architecture with a Few Important Points Marked

met at the point P_7 (prior to the low power interference filter) ?

- *How does the intensity decay as light travels from sources to detectors ?*
- *Where and how many light restorers (similar to repeaters) should be kept to raise the dwindling intensity levels for satisfactory performance ?*
- *Can a laser be replaced by another lower power laser without affecting the functionality of the architecture ?*

These are some of the questions that the designer/architect will be able to ask and get answers from a power analyzer.

8.1.3 Delay Analysis

In a similar fashion, given an architecture, a particular excitation, and a rudimentary output out of simulation in the form of important pairs of points through which light travels, the delay analysis provides time values relative to the input signal time at different points in the architecture. The primary interest is in finding out the time lags from reference times. The reference time is the leading edge of clock in synchronous architectures and the input time in asynchronous architectures.

Following are some of the questions of interest related to timing.

- *Do two signals arrive in phase as the inputs to a beam-merger ?*
- *What is the minimum clock period of the architecture ?*
- *How compact can an architecture be made while retaining the same functionality ?*
- *Is the assertion*

  $$\boxed{\text{Assert[Time[C1] - Time[C2]} \leq \delta]}$$

  meaning 'do signals C1 *and* C2 *reach the point of assertion with a time lag of δ' met ?*

In an idealized scenario the designer needs the above kinds of information independent of a particular input. However, deduction of proper answer for an arbitrary input could be intractable. Hence the requirements of a feasible analysis and that too a quick one with respect to measures like power, delay make the need for pragmatic analysis all the more important.

Hence the decision was taken to restrict to a path based analysis for a specific input on which simulation was carried out to assess the signal/light paths.

8.2 Signal Flow Graph Construction

This section describes in a step-by-step manner the construction of a signal flow graph.

8.2.1 Simulation Produces Reference Points

The simulator when executed produces a sequence of events. These events correspond to the light signals travelling from one component to the next. The event contains among other attributes, the source and destination components of the event. By using these it is possible to trace a signal travelling through the architecture. Table 8.3 shows some events together with their important attributes from an output of a simulation. The sender and receiver are indicated by the component number as shown in Figure 8.2 and the types are shown in Table 8.1. Table 8.2 shows the positions of each of the components in 3D.

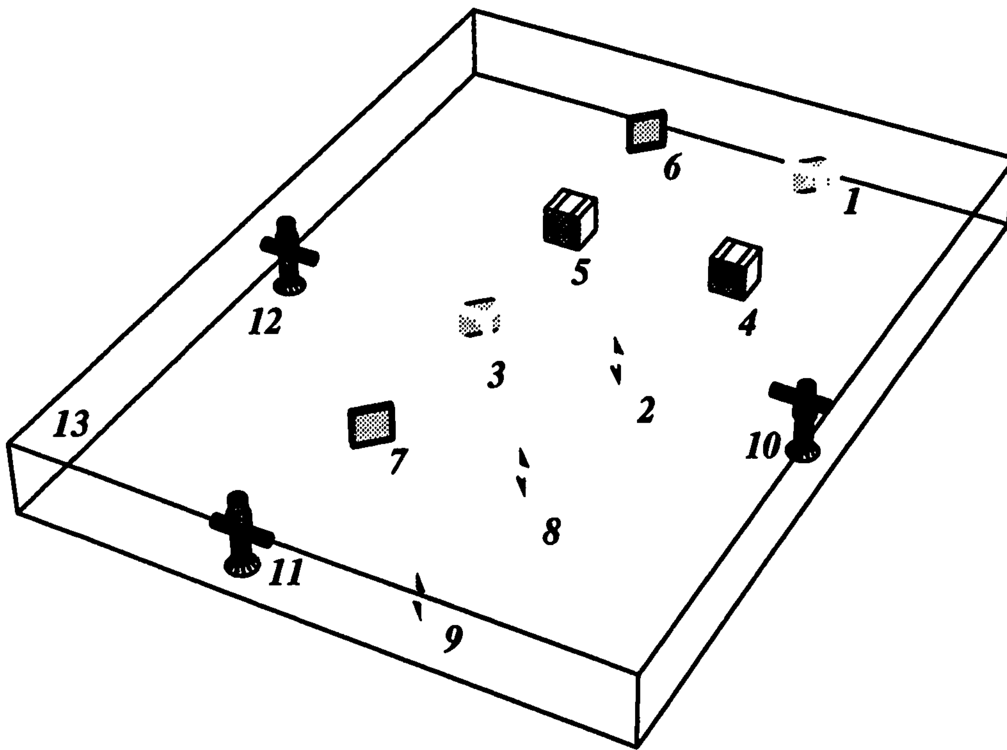


Figure 8.2: The Trip-Flop Architecture with Numbered Components.

Component #	Component Type
1	PolarizingBeamSplitter
2	PolarizingBeamSplitter
3	PolarizingBeamSplitter
4	InterferenceFilter
5	InterferenceFilter
6	Mirror
7	Mirror
8	BeamSplitter
9	BeamSplitter
10	PulsedLaser
11	PulsedLaser
12	PulsedLaser
13	BoundingBox

Table 8.1: Component Numbers and Their Types.

Component #	Component Position
1	(0.2,0.4,0)
2	(0.22,0.2,0)
3	(0.1,0.2,0)
4	(0.2,0.3,0)
5	(0.1,0.3,0)
6	(0.1,0.4,0)
7	(0.1,0.1,0)
8	(0.22,0.1,0)
9	(0.22,0,0)
10	(0.3,0.225,0)
11	(0.1,0,0)
12	(0,0.2,0)

Table 8.2: Component Numbers and Their Positions.

Msg Number	Msg Type	Sender Object	Receiver Object	Time of Impact ns	Point of Impact
1	*Start*	*Simulator*	1	*-NA-*	*-NA-*
15	Normal	10	2	0.27	(0.22, 0.21, 0.01)
18	Normal	2	3	0.57	(0.12, 0.21, 0.01)
19	Normal	2	8	0.57	(0.21, 0.12, 0.01)
22	Normal	3	12	0.83	(0.03, 0.21, 0.01)
23	Normal	3	5	0.87	(0.11, 0.3, 0.01)
26	Normal	8	9	0.87	(0.21, 0.02, 0.01)
29	Normal	12	3	1.1	(0.1, 0.21, 0.01)
35	Normal	9	13	1.2	(0.3, 0.01, 0.01)
38	Normal	3	2	1.4	(0.2, 0.21, 0.01)
39	Normal	3	7	1.4	(0.11, 0.113, 0.01)
44	Normal	2	10	1.6	(0.27, 0.21, 0.01)
45	Normal	2	4	1.7	(0.21, 0.3, 0.01)
47	Normal	7	8	1.7	(0.2, 0.113, 0.01)
55	Normal	8	2	2.	(0.21, 0.2, 0.01)
97	Normal	7	8	3.4	(0.2, 0.113, 0.01)

Table 8.3: Some Important Attributes of Events/Messages from a Simulation Log File.

8.2.2 Construction of Signal-Flow Graph

The information from the simulation can be converted into a graph that models spatial light paths in the architecture. The vertices of the graph can be the significant points on the surface of components in the architecture through which light passes. An edge is introduced from a vertex x to another y if there is a signal going from x to y. An example is shown in Figure 8.3. A beam emitting from component L_1 strikes component PBS_1 and goes further to component M_1. The corresponding graph is shown with vertices marked with the same numbers and edges showing the signal flow.

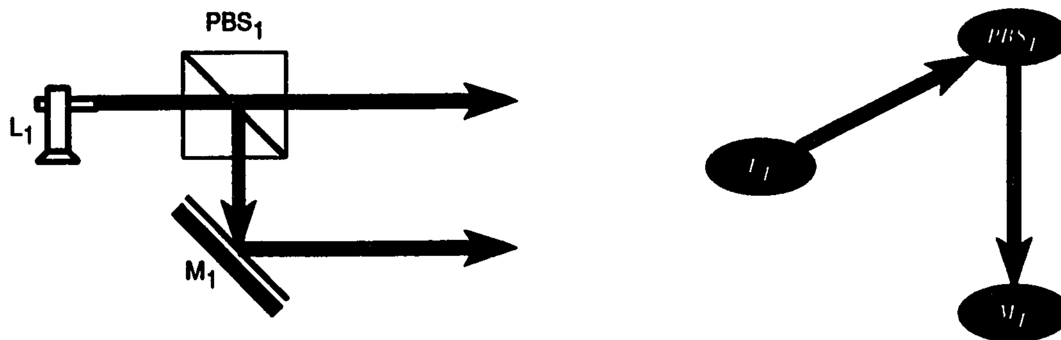


Figure 8.3: Three Components and their Corresponding Flow Graph.

This graph can be further enhanced into a weighted directed graph. The weight of an edge is calculated based on the objective of analysis. For power it considers the medium decay characteristics and distance. For delay it considers the optical distance and the refractive index of the material.

The signal flow graph for the Trip-Flop architecture is shown in Figures 8.4 through 8.6. The numbers on the vertices indicate the component numbers (as given in Figure 8.2).

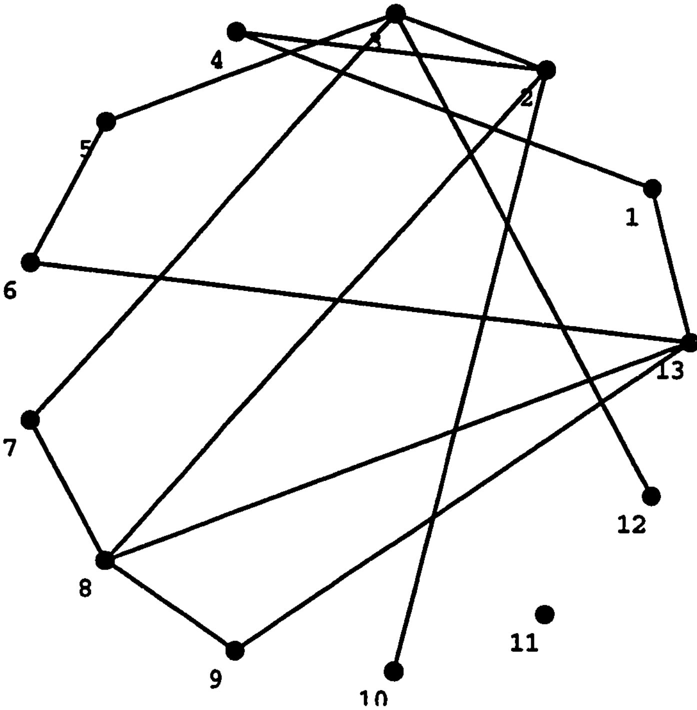


Figure 8.4: The Signal Flow Graph with Vertex Labels

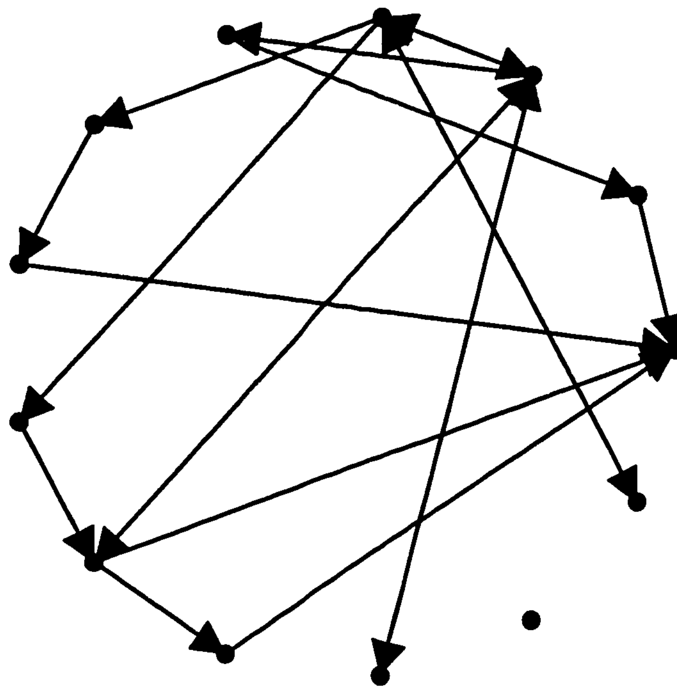


Figure 8.5: The Directed Signal Flow Graph without Vertex Labels

Figure 8.6: The Directed Signal Flow Graph with Vertex Labels

8.3 Path Enumeration

Having constructed a signal flow graph there is a need to enumerate various simple paths from all sources to all detectors. Such an enumeration is necessary for assessing the signal decay characteristics and the delay involved.

The architecture contains the information about components that could be used to label the vertices of signal-flow graphs such as: sources (lasers, LEDs), detectors (screens, bounding-box), neither (manipulators or components) and both (regenerators or repeaters).

Paths of interest are those that start from a source/regenerator, pass through zero or more manipulators and end at a detector/regenerator. This can be done directly from the graph using the following backtrack algorithm [32].

---

algorithm $BACKTRACK(n)$

1. This pseudo code describes the backtracking process. All solutions are generated in $X(1:n)$ and printed as soon as they are generated and determined. $T(X(1),...,X(k-1))$ gives all possible values of $X(k)$ given that $X(1),...,X(k-1)$ have already been chosen. The predicates $B_k(X(1),...,X(k))$ determine those elements $X(k)$ which satisfy the implicit constraints.
2. **integer** k, n; **local** $X(1:n)$
3. $k \leftarrow 1$
4. **while** $k > 0$ **do**
5.   **if** there remains an untried $X(k)$ such that
6. $X(k) \in T(X(1),...,X(k-1))$ **and** $B_k(X(1),...,X(k)) =$ **True**
7.   **then if** $X(1),...,X(k)$ is a path to an answer node
8.   **then print** $(X(1),...,X(k))$ **endif**
9. $k \leftarrow k+1$

10. **else** $k \leftarrow k - 1$
11. **endif**
12. **repeat**
13. **end** *BACKTRACK*

Enumeration of all simple paths can be handled by the above backtrack algorithm starting from all sources and ending on all detectors. All paths of $3 <= length <= 6$, that are produced by the backtrack algorithm for path enumeration for the signal flow graph of Trip-Flop architecture are shown in Table 8.4.

8.4 Power Analysis

The approach followed here is derived from one of tracing a beam through a simple path. In a simple path the light beam starts from a source, travels alternatively through free-space and components and ends at a detector. The intensity is maximum at the source and decays as it passes through the various components.

Each component has a transfer function which captures the losses in that component's medium due to various affects. This transfer function of a component can be obtained in a variety of ways. Some of the common techniques are *Paraxial Approximation*, *Fourier Analysis*, and *Diffusion Analysis.* All of these are discussed in some detail in Section 8.11.

The transfer function of a simple path can be obtained from the transfer functions of the individual components. For power analysis the transfer function can be obtained as a composition (chain product) of all the transfer functions. Iteration through the

Path Length	Source	Detector	Path
3	2	13	{ 2,8,13}
3	4	13	{ 4,1,13}
3	5	13	{ 5,6,13}
3	7	13	{ 7,8,13}
3	8	13	{ 8,9,13}
4	2	13	{ 2,4,1,13}
4	2	13	{ 2,8,9,13}
4	3	13	{ 3,2,8,13}
4	3	13	{ 3,5,6,13}
4	3	13	{ 3,7,8,13}
4	7	13	{ 7,8,9,13}
4	10	13	{ 10,2,8,13}
5	2	13	{ 2,3,5,6,13}
5	2	13	{ 2,3,7,8,13}
5	3	13	{ 3,2,4,1,13}
5	3	13	{ 3,2,8,9,13}
5	3	13	{ 3,7,8,9,13}
5	8	13	{ 8,2,4,1,13}
5	10	13	{ 10,2,4,1,13}
5	10	13	{ 10,2,8,9,13}
5	12	13	{ 12,3,2,8,13}
5	12	13	{ 12,3,5,6,13}
5	12	13	{ 12,3,7,8,13}
6	2	13	{ 2,3,7,8,9,13}
6	7	13	{ 7,8,2,4,1,13}
6	8	13	{ 8,2,3,5,6,13}
6	10	13	{ 10,2,3,5,6,13}
6	10	13	{ 10,2,3,7,8,13}
6	12	13	{ 12,3,2,4,1,13}
6	12	13	{ 12,3,2,8,9,13}
6	12	13	{ 12,3,7,8,9,13}

Table 8.4: All Paths of 3 <= *length* <= 6 for the Signal Flow Graph of Trip-Flop Architecture.

set of paths allows one to examine the spatio-temporal aspects of power distribution in a system.

Figures 8.7 through 8.9 show some examples of power analysis results with the signal flow graph highlighting (red in color and thick in black-and-white) the path under consideration. The bar-chart also indicates a simplified transfer function for each component on the path.

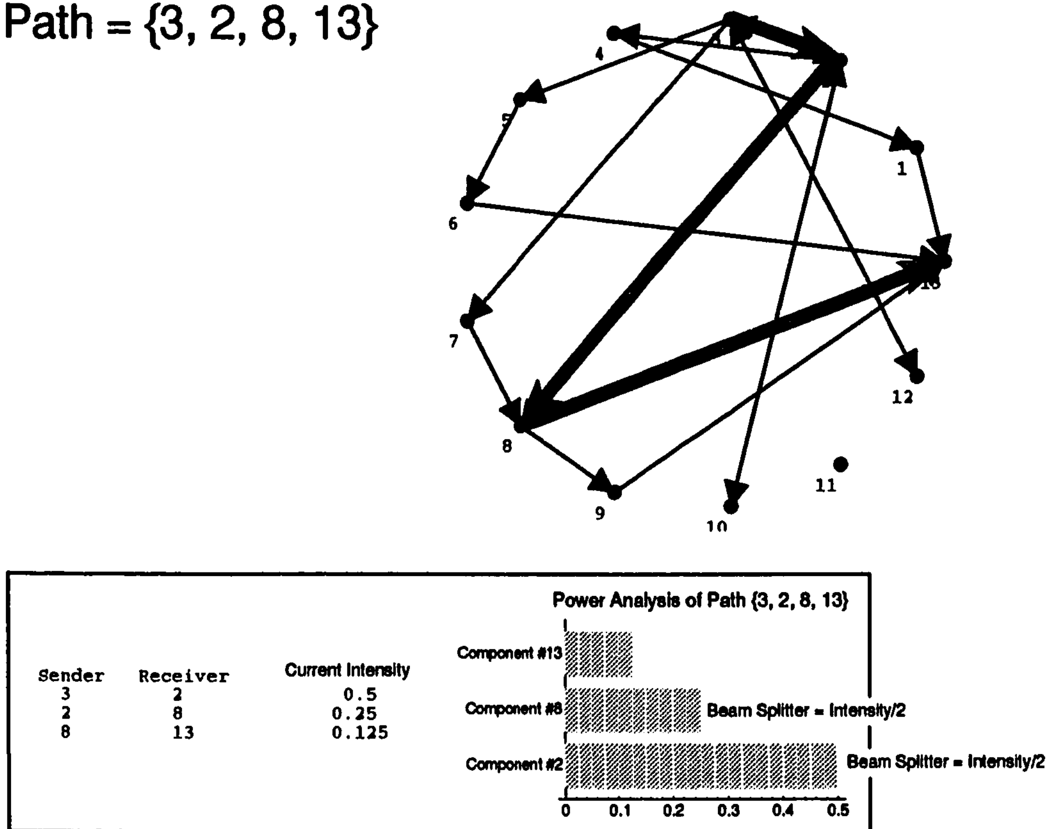
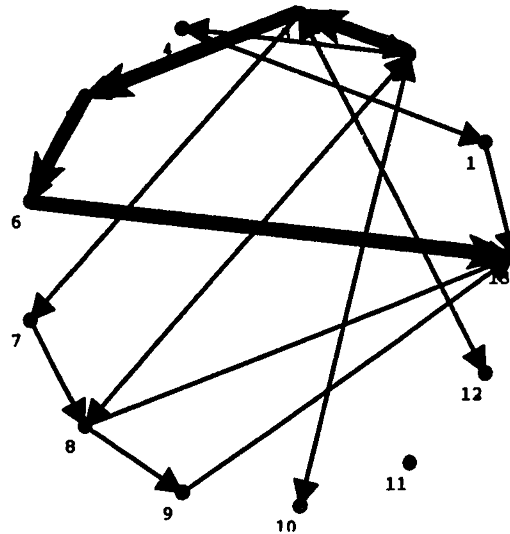
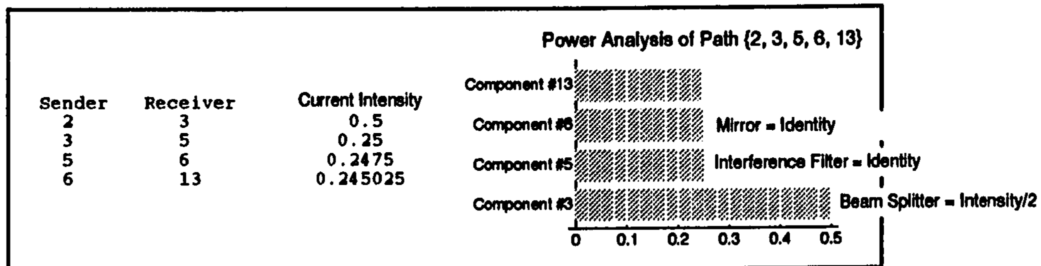


Figure 8.7: Signal Flow Graph Highlighting One Simple Path $\{3,2,8,13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function

Path = {2, 3, 5, 6, 13}

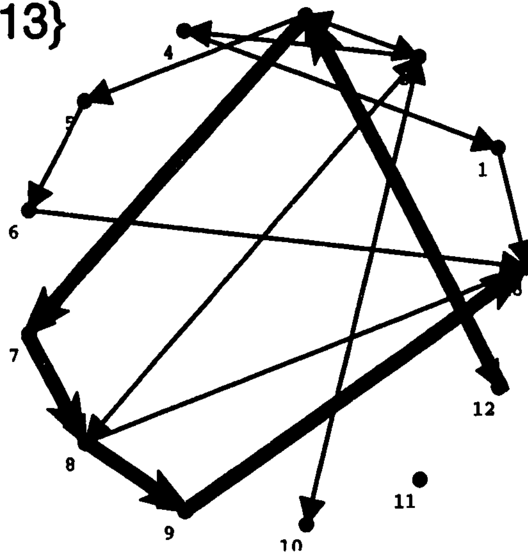


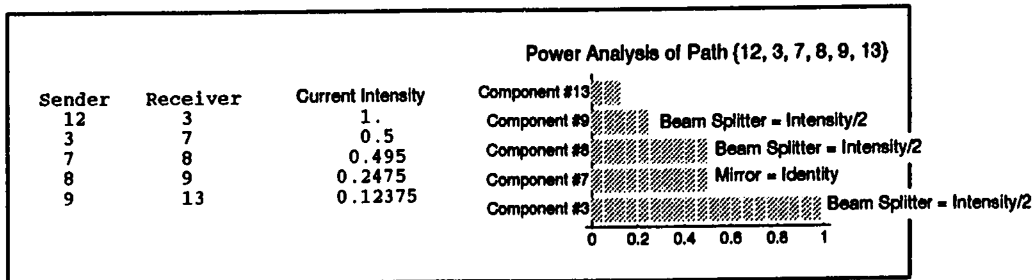


Sender	Receiver	Current Intensity
2	3	0.5
3	5	0.25
5	6	0.2475
6	13	0.245025

Figure 8.8: Signal Flow Graph Highlighting One Simple Path $\{2,3,5,6,13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function

Path = {12, 3, 7, 8, 9, 13}





Sender	Receiver	Current Intensity
12	3	1.
3	7	0.5
7	8	0.495
8	9	0.2475
9	13	0.12375

Figure 8.9: Signal Flow Graph Highlighting One Simple Path $\{12, 3, 7, 8, 9, 13\}$ and the Corresponding Power Analysis Bar Graph with Simplified Transfer Function

8.5 A Template for a Power Analyzer

Power analysis is a common evaluation criterion for optical architecture. It is necessary to generate a separate power analyzer for every architecture. A power analyzer template is needed for reasons similar to those for introducing a simulator template (discussed in previous chapter).

The power analyzer template (see Figure 8.10) contains generic code that is needed for analyzing power in an architecture. This generic code contains routines for loading generic library routines, slots for architecture specific information, slots for component power models, and code for power analysis. Figure 8.11 summarizes the process of generating a power analyzer for a specific architecture using the generic power analyzer template and different libraries.

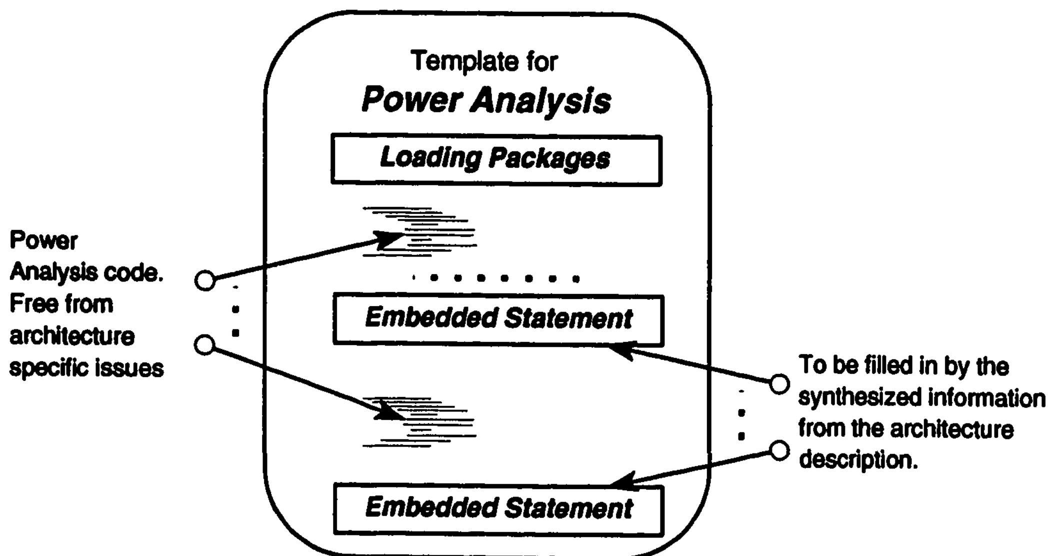


Figure 8.10: The Generic Power Analyzer Template.

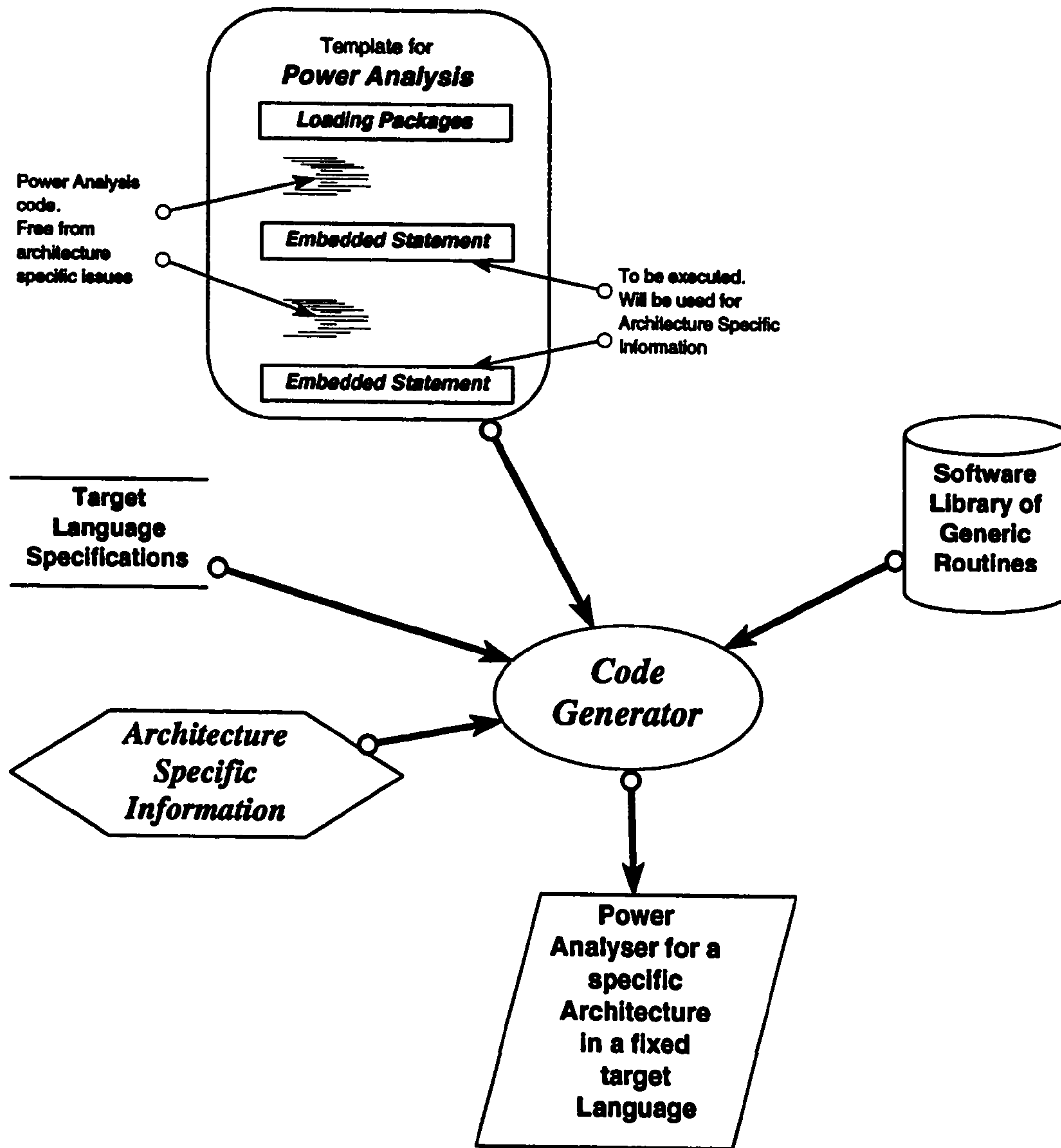


Figure 8.11: Generating a Power Analyzer for an Architecture by Using the Generic Power Analyzer Template.

8.6 Delay Analysis

As in the case of power analysis, the approach followed here is derived from one of tracing a beam through a simple path. In a simple path the light beam starts from a source, travels alternatively through free-space and components and ends at a detector. The time is considered zero when the beam starts from a source, increasing as it passes through manipulators and ends at a detector.

Each component has a transfer function which computes the time needed to travel through that component. This transfer function for delay of a component can be obtained in a variety of ways (Section 8.11).

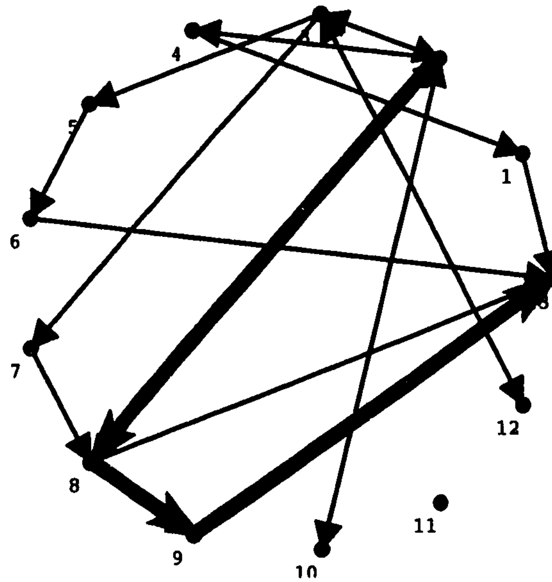
The transfer function of a simple path can be obtained from the transfer functions of the individual components. For delay analysis the transfer function can be obtained as an accumulation (chain addition) of all the transfer functions. Iteration through the set of paths allows one to determine the worst case time path (or critical path) that will determine the clock period of the architecture.

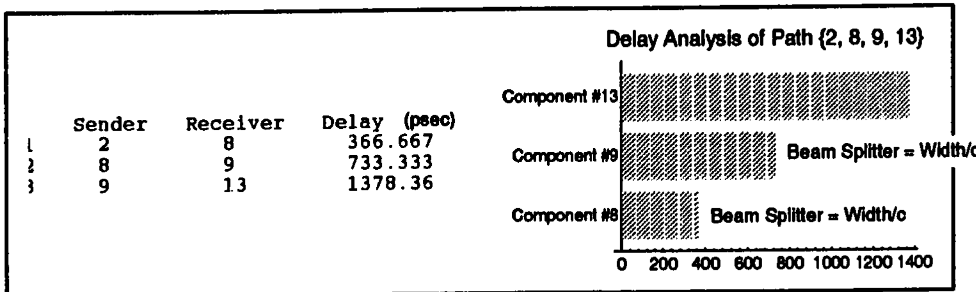
Figures 8.12 through 8.14 show some examples of results obtained by delay analysis with the signal flow graph highlighting the path under consideration. The bar-chart indicates a simplified delay function for each component on the path.

8.7 A Template for a Delay Analyzer

Delay analysis is an important evaluation criterion for optical architecture. It is necessary to generate a separate delay analyzer for every architecture. A delay analyzer template is needed for reasons similar to those for introducing a power analyzer template.

Path = {2, 8, 9, 13}

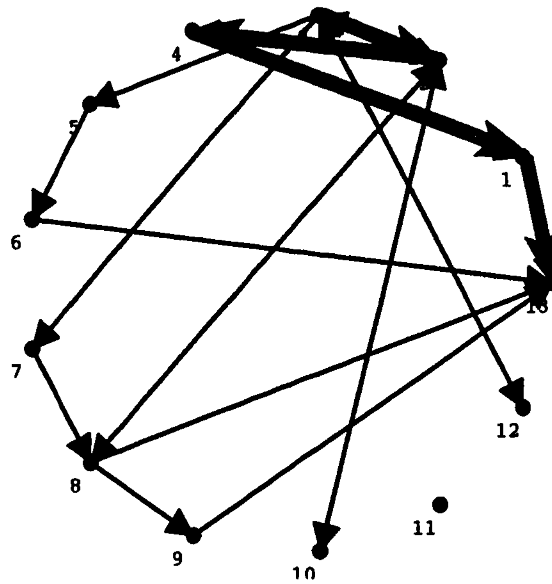


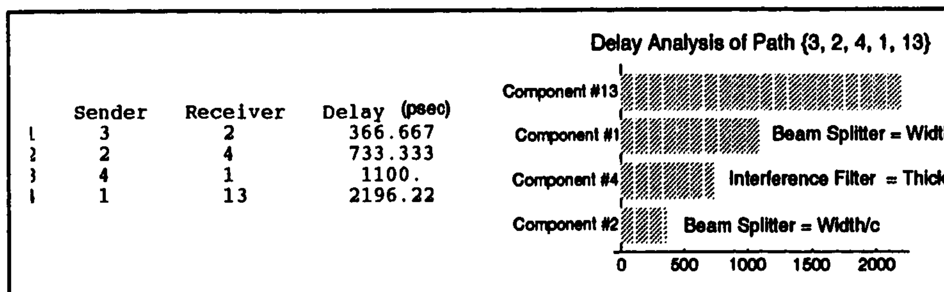


	Sender	Receiver	Delay (psec)
1	2	8	366.667
2	8	9	733.333
3	9	13	1378.36

Figure 8.12: Signal Flow Graph Highlighting One Simple Path {2,8,9,13} and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function

Path = {3, 2, 4, 1, 13}

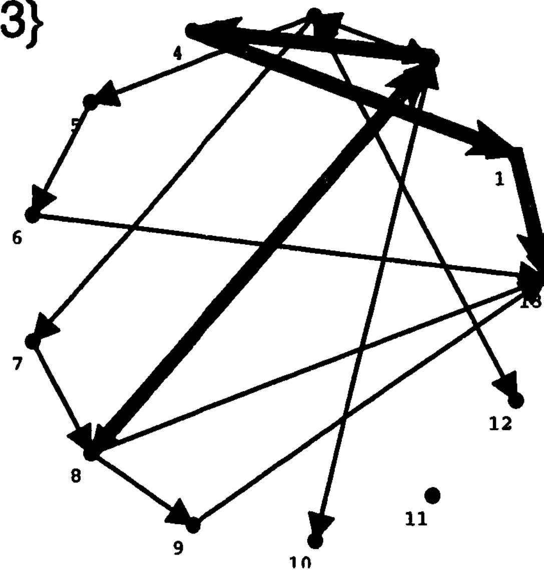


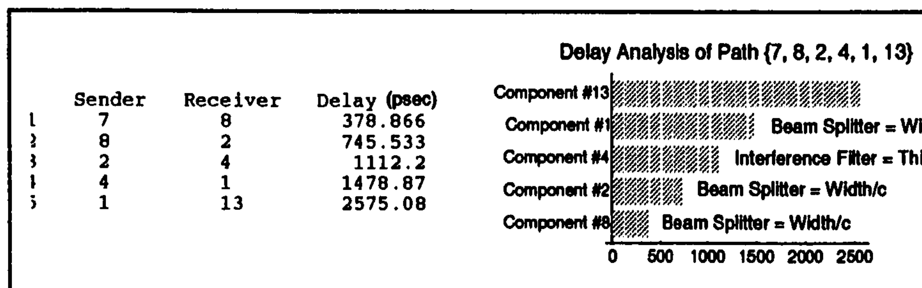


	Sender	Receiver	Delay (psec)
1	3	2	366.667
2	2	4	733.333
3	4	1	1100.
4	1	13	2196.22

Figure 8.13: Signal Flow Graph Highlighting One Simple Path $\{3, 2, 4, 1, 3\}$ and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function

Path = {7, 8, 2, 4, 1, 13}





	Sender	Receiver	Delay (psec)
1	7	8	378.866
2	8	2	745.533
3	2	4	1112.2
4	4	1	1478.87
5	1	13	2575.08

Figure 8.14: Signal Flow Graph Highlighting One Simple Path $\{7, 8, 2, 4, 1, 3\}$ and the Corresponding Delay Analysis Bar Graph with Simplified Transfer Function

The delay analyzer template (see Figure 8.15) contains generic code that is needed for analyzing delay in an architecture. This generic code contains routines for loading

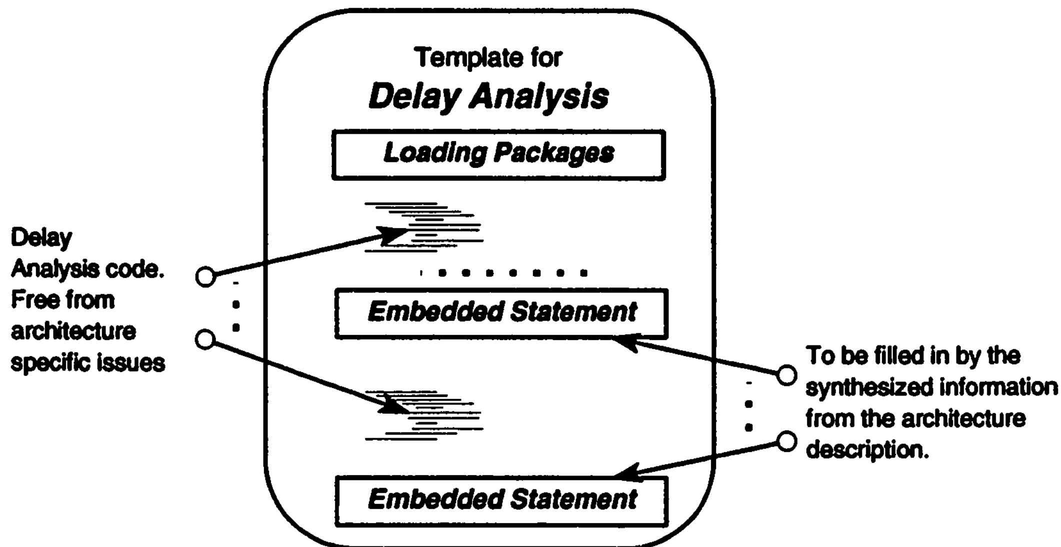


Figure 8.15: The Generic Delay Analyzer Template.

generic library routines, slots for architecture specific information, slots for component power models, and code for delay analysis. Figure 8.16 summarizes the process of generating a delay analyzer for a specific architecture using the generic delay analyzer template and different libraries.

8.8 Limitations of the Current Analyses

The approach taken in the current implementation for power and delay analysis is crude and at best an initial solution. In practice, for large systems, this kind of analysis becomes infeasible due to the combinatorial explosion of paths. The limitations of

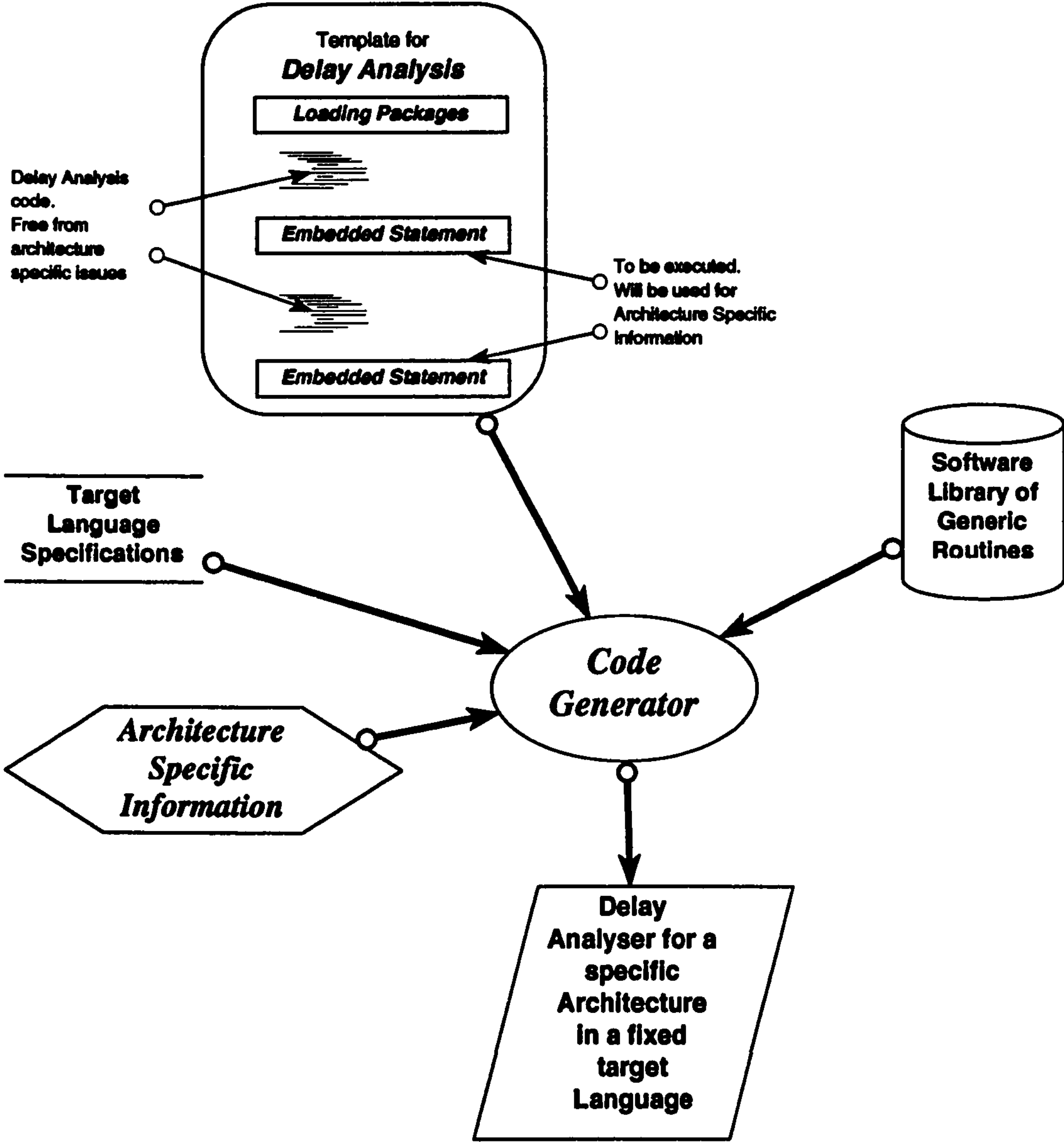


Figure 8.16: Generating a Delay Analyzer for an Architecture by Using the Generic Delay Analyzer Template.

the current system and possible improvements are discussed in this section.

Following is a brief mention of the limitations in the analyses currently employed in the system.

- The power decay and delay in a component doesn't consider all possible states. Such a consideration would amount to conducting thorough simulation for the purpose of analysis.
- At components such as beam-mergers/splitters or in other switched components or synchronized signals the analysis becomes inaccurate. Since blocking effect is not considered, delays tend to be smaller than true values. Likewise intensities tend to be underestimates as accumulation is not considered at beam-mergers.
- Interactions among paths are not considered.
- In view of the above, the analysis tends to be optimistic/pessimistic depending on the kind of analysis.
- Exponential increase in the cost of the analysis with the size of the architecture. As the size of the architecture increases, the signal flow graph becomes bigger and the cardinality of enumerated paths increases drastically.

8.9 Other Analyses

In addition to power and delay, other kinds of measures should be kept track of. These include:

- *Number of passes through surfaces, lenses, and image planes*

In order to obtain a more accurate picture of the power decay and delays involved, statistics are usually obtained for these parameters. They also provide some indication of the scatter and beam divergence.

- *Input power*

  The input power needs constant monitoring as the level of the outputs has to be maintained enough for them to be detectable. As the input beam travels through the architecture it undergoes losses due to several factors. The output obtained has to be strong enough so that it can serve as an input to some other architecture.

- *Heat dissipation in space and time*

  Compactness of an architecture is largely determined by its ability to distribute heat and remove it quickly from the system to the external environment. It is necessary to get the heat patterns that emerge in the architecture for different inputs to determine its smallest possible volume.

- *Scatter in space and time*

  As a beam travels through an architecture it diverges. It is necessary to keep track of the divergence patterns so that condensers can be placed at appropriate points. Also when a beam strikes a surface, part of it is reflected in various directions depending on the nature of the surface. The behavior of these stray rays must be traced so the shutters/absorbers are placed at different points in the architecture to absorb scattered/stray light.

- *Volume and surface analyses*

Optical architectures tend to be assembled in 3D. They occupy some volume. Once an architecture has been assembled, it may be possible to adjust the components so that the whole architecture occupies a smaller volume.

- *Throughput*

  Once the clock speed has been determined, it is straightforward to calculate the throughput.

- *Maximum operable clock for synchronous architectures*

  The architect may have some idea of the clock speed of an architecture. This is usually determined by using the statistics obtained from the delay analysis. It is useful to have an analysis which can determine the path that contributes most to the delay (i.e., the critical path). Once the critical path has been marked, efforts can be made to minimize it. The maximum operable clock can be determined. The same analysis can be used to reduce the cost/power of an architecture by replacing components with less expensive/low power ones on non-critical paths.

- *Hardware cost*

  One measure of comparing two systems that have the same functionality is by using their cost. If the cost of each component is known, the total hardware cost can relatively easily be calculated. However, it involves additional cost of realization of a particular setup. This could be measured as setup time cost.

- *Mechanical stability*

  Various architectures in the literature are quoted to have better mechanical stability than others. This is a measure of how the components are placed and how sensitive the setup is to mechanical disturbances. Although such a measure may

not be readily quantifiable using the current knowledge, a subjective decision can usually be made.

- *Number of Fourier Transforms*

  Some manipulation in the Fourier domain can result in displacement in the normal domain. Usually architectures exploit this property of light to subject it to various operations that tend to be easier/cheaper in the Fourier domain. Many interconnection networks realize some permutations using this phenomenon. It is useful to have an estimate of the number of Fourier transforms in an architecture.

- *Number of Image Multiplications*

  Information is encoded in light as a 2D image matrix. Image multiplications will achieve operations on the information in a much faster way than other techniques.

- *Degree of Parallelism*

  Optical architectures exploit spatial parallelism. Usually an architecture that is designed for some resolution of data can be used for a much higher resolution without any major changes to the architecture. Improving the resolution of the data will improve the degree of parallelism.

- *Maximum Permitted Signal to Noise Ratio*

  Although optics does not suffer to the same degree from the conventional noise in electronics, some noise may be introduced by temperature variations and reflection from surfaces. It is important to know the maximum signal rate that can be achieved without being affected by noise.

- *Polarization*

  Decisions are usually taken in optical architectures by using polarization encoding. Analysis of the different polarizations at various points in the architecture is useful in the debugging process.

- *Volume-Time^3 Measure*

  In many solutions to problems, one finds a spectrum of solutions employing hardware and software to different degrees. It is hard to assess the quality of one solution with another. A measure Volume-Time^3 is proposed that can be the basis for comparison of alternative solutions based on 3D-architecture types. It is widely understood that a design that takes more space could be faster than another that takes less space due to the extra/parallel hardware. However, there is a limit to the space that can be given to an architecture. Volume-Time^3 measure will indicate the tradeoff between time and space for 3D-architecture solutions.

8.10 Possible Improvements to the Current Analysis

The following improvements can be made to the current analysis to make the system more practical.

- *Path pruning using dominating/dominated path heuristics.*

  The current system considers all possible paths of a given length for analysis. Analyzing some of these does not yield any useful result. This is because the

results of some other path dominates over the results of these paths. It is possible to devise heuristics to drastically cut down the state space and examine only the dominating paths.

- *Bottleneck analysis*

  In cases of analyses like delay, it is sufficient to find the path that takes the longest time i.e., the critical path. Algorithms need to be modified so that they examine and enumerate only the worst delay path.

- *Enumeration of k-worst paths*

  Techniques exist using which it is possible to enumerate the k-worst paths starting from the worst. If such techniques are used, they will cut down enormously on the overhead of producing and then examining useless paths.

8.10.1 Hierarchical Analysis

Large scale architectures will employ thousands of optical components. It is infeasible to carry out a flat analysis of such architectures. Closer examination of their development will reveal that these architectures are subdivided into smaller more manageable parts, each having its own functionality. Hence the analysis is done using different levels of abstraction.

Carrying out analysis at different levels of abstraction requires techniques in which there is constant communication and interaction between a level and its higher/lower level. Analyzing at a higher level usually gives some idea of the results of analysis for better planning at lower levels. If this information could be passed to the lower levels when they are being analyzed, it narrows the search space considerably resulting in

faster convergence. The lower level, after carrying out analysis at its level will pass the results to the higher level. The higher one is now in a better position to narrow the bounds even further for other sub-architectures. These techniques are widely used in several iterative computations of systems of mathematical equations. However, the discussion of their details and issues like convergence are beyond the scope of this work.

8.11 Alternative Approaches

Numerous techniques are available in the literature for analyzing particular types of optical components, phenomena, or systems. Some of these are presented in subsequent sections.

8.11.1 Paraxial Approximation and Matrix Analysis

Matrix optics is a technique for tracing *paraxial rays*[1] [67]. The rays are assumed to travel only within a single plane, so that the formalism is applicable to systems with planar geometry.

A ray is described by its position and its angle with the optical axis. These variables are altered as the ray travels through the system. In the paraxial approximation, the position and angle of the input and output planes of an optical system are related by two linear algebraic equations. As a result the optical system is described by a 2×2 matrix called the *ray transfer matrix*.

[1] A ray whose angular deviation from the cylinderical (z) axis is small enough that the sine and the tangent can be approximated by the angle itself.

The advantage of using matrix methods lies in the fact that the ray-transfer matrix of a cascade of optical components is a product of the ray-transfer matrices of the individual components. Matrix optics, therefore provides a formal mechanism for describing complex optical systems in the paraxial approximation.

8.11.2 Jones Calculus

Many optical systems e.g. electro-optic modulators involve the passage of light through a train of polarizers and retardation plates. The effect of each individual element, either polarizer or retardation plate, on the polarization state of the transmitted light can be described by simple means. However, the calculations become more complicated when the optical system consists of many elements oriented at different azimuthal angle.

A systematic approach is provided in the Jones method [85]. It is a powerful matrix method in which the state of polarization is represented by a two component vector. Each optical element is represented by a 2×2 matrix. The overall transfer matrix for the whole system is obtained by multiplying all the individual element matrices, and the polarization state of the transmitted light is computed by multiplying the vector representing the input beam by the overall matrix.

8.11.3 Additional Analysis Techniques

Discussion about some of the other popular techniques 2D Fourier Transform Analysis, Scatter/Transient Analysis, and Ray-tracing/Beam-tracing for Gaussian Beams can be found in the literature [7, 67, 85].

8.12 Summary

This Chapter presented a techniques for the analysis of optical architectures based on signal-flow graph construction. The power and delay analysis of the trip-Flop architectures was carried out. Several new analysis techniques were suggested.

Chapter 9

Code Generation

Chapter Abstract

Code generation plays an important and vital part in OptiCAD for architecture design and analysis. Instead of simulating an architecture in a general framework, code is generated to produce a customized and independent program that carries out a particular analysis of the specific architecture. This generated program can be separately executed to obtain the desired analysis results.

9.1 Introduction

Chapters 6–8 explained how optical components can be modeled and analyzed using the language described in Chapter 4. This chapter deals with the issue of taking the architecture description and producing a specific purpose analyzer.

One of the fundamental drawbacks of most contemporary design automation systems or simulators is the price to be paid for generality. They are designed to accommodate a wide variety of instances for simulation in their particular domain. This

leads to the disadvantage of carrying all the extra overhead that comes with the generality.

Usually, most of the computations during simulation take enormous CPU-time/ memory. Even a small percentage of overhead caused by any generality beyond the needs of the example under study will amount to substantial overall CPU-time and memory overhead. Even with lens systems, the well understood task of ray tracing takes hours to days on supercomputers for moderate sized systems containing no more than a few dozen components. This is the main reason that specific purpose simulators are developed for carrying out large simulations.

The situation is similar in a number of other domains. Examples include:

1. Specific simulation tools designed for specific domains.

   - General purpose simulators for computer communication systems modeling and analysis like COMNET, Simlan, LNet, SimFac etc.
   - Timing and delay analysis in VLSI design automation systems.
   - Petrinet (or some such graphical model) based systems whether they are for simulation or reachability analysis, or Markovian analysis.
   - Numerical and analytical solvers of queuing systems.
   - Finite element analysis tools.

2. At the other end of the spectrum one can resort to writing customized simulators. This can be done in either of two ways.

   - Developing the simulator in a special purpose language like SimScript, GPSS or Simula. These high-level languages are meant for writing simulators. They provide the user with various facilities in the form of general

purpose data-structures usually used in simulation. The disadvantage of these languages/systems is that their use introduces enormous overhead.

- Writing the simulation software from scratch using a general purpose programming language like C/Pascal. This is not only too tedious to work with but also demands expertise in computational algorithms since all these must be developed from scratch.

In either of the above two cases, the need to manually develop a custom-built simulator for each architecture is a daunting alternative.

Optical architectures are complex themselves and the principles that govern optics requires enormous computation for simulation. Moreover there is a wide variety of optical components in use and the number is constantly growing. It is impractical to have a system that contains information on all these components. The computation involved along with the volume of information to be maintained renders development of one simulation system infeasible.

The practical alternative is to write a specific simulator for every architecture in a general purpose programming language. However, this (as discussed before) is not a very attractive alternative. In view of the all the factors considered, it was decided to go with generating architecture-specific code automatically from the specifications of the architecture. This code would be specific to the architecture. Hence it would not carry the extra overhead of a general simulation system. Since the code is to be automatically generated, it can be done for each architecture without much effort on the architect's part.

The only problem in automatic program writing is that there is very little literature on producing high-level language code in a general framework. Issues of semantics,

program correctness are tricky and vary from one target language to another.

9.1.1 Objectives

The objectives for generating custom-made analyzers are

1. *Automatic generation of correct code from specifications*

   The primary objective of automatically producing code is that the code should be correct i.e., it should function as it is intended to. Another concern is that the code should comply with the syntax and semantics of the target language. Automatic generation of high-level code is very different from generating low-level machine code from a high-level program. In conventional compilers the source and target languages are fixed. There are well defined one-to-one translation rules that govern the code generation process.

   In this setting the requirements are quite different. High-level language code has to be generated from very-high level specifications. The target languages are not fixed. The translation rules are given as a part of the specification.

2. *Efficiency of the generated code*

   The generated code should be efficient i.e., its execution time efficiency should be comparable to that of a hand crafted code for the architecture under consideration. If it proves to be much slower than conventional simulators then it defeats the primary purpose of generating code. Efficiency is one of the advantages of code generation over the interpretation of high-level specifications. Issues that can be resolved and checks that can be performed are done so at compilation/generation time rather than waiting and repeatedly carrying them

out at run time.

Another advantage of producing an architecture specific code is that complete information about the architecture is available at code generation time. This reduces the enormous overhead of input/output from the system. The complete information about the architecture can be built into the generated code as crafted data structures, thus eliminating the need for input altogether.

3. *Portability*

   The target language chosen should be widely available so that the generated code can be executed on different platforms.

9.2 Previous Work

Substantial body of literature exists in the areas translators, translator design, translator generators and syntax directed translation schemes. Basic issues are covered in [2, 69, 6]. Two generators are discussed in [48].

9.3 A Template Driven Code Generator

As a practical alternative to have a working system without getting too much into the issue of language semantics and their preservation, the following approach is advocated.

The code generation process is divided into different roles.

1. A person well-versed with a particular kind of analysis will code in an appropriate language, leaving out the details of a specific architecture or example.

2. The models of components are likewise coded to be stand alone functions/ procedures that can be linked to any example specific analyzer.

3. Specifications for each of the target languages is provided by an expert in that language (see Chapter 3).

4. In addition to these there could be a general purpose software library implementing common functions.

Of the above, three of these (general purpose library, component models and analysis template) are expected to be realized in a general purpose language. The code generator consults these and synthesizes the code in the desired target language by consulting the specifications provided by the language expert. The process is summarized in Figure 9.1.

Code synthesis is essentially carrying out the *embedded actions* specified in the templates. The template files are written in the meta-language with embedded actions. The code generator reads the template files and builds a 2-level tree. Some of the leaves are in the representational language, others correspond to embedded actions. The code generator repeatedly rewrites the tree by picking up any leaf corresponding to an embedded action, rewriting and replacing that node by the tree produced from the evaluation of the embedded action. This evaluation continues until there are no more embedded actions. Once the evaluation completes the resulting representation is subjected to dependency analysis and all the missing code and component simulators are respectively picked up from the libraries and component simulators respectively. Then an analyzer is transcribed in the target language. The steps of these operations are shown as an example in Figures 9.2 through 9.5.

Consider a template file shown as a tree in Figure 9.2. It contains the embedded

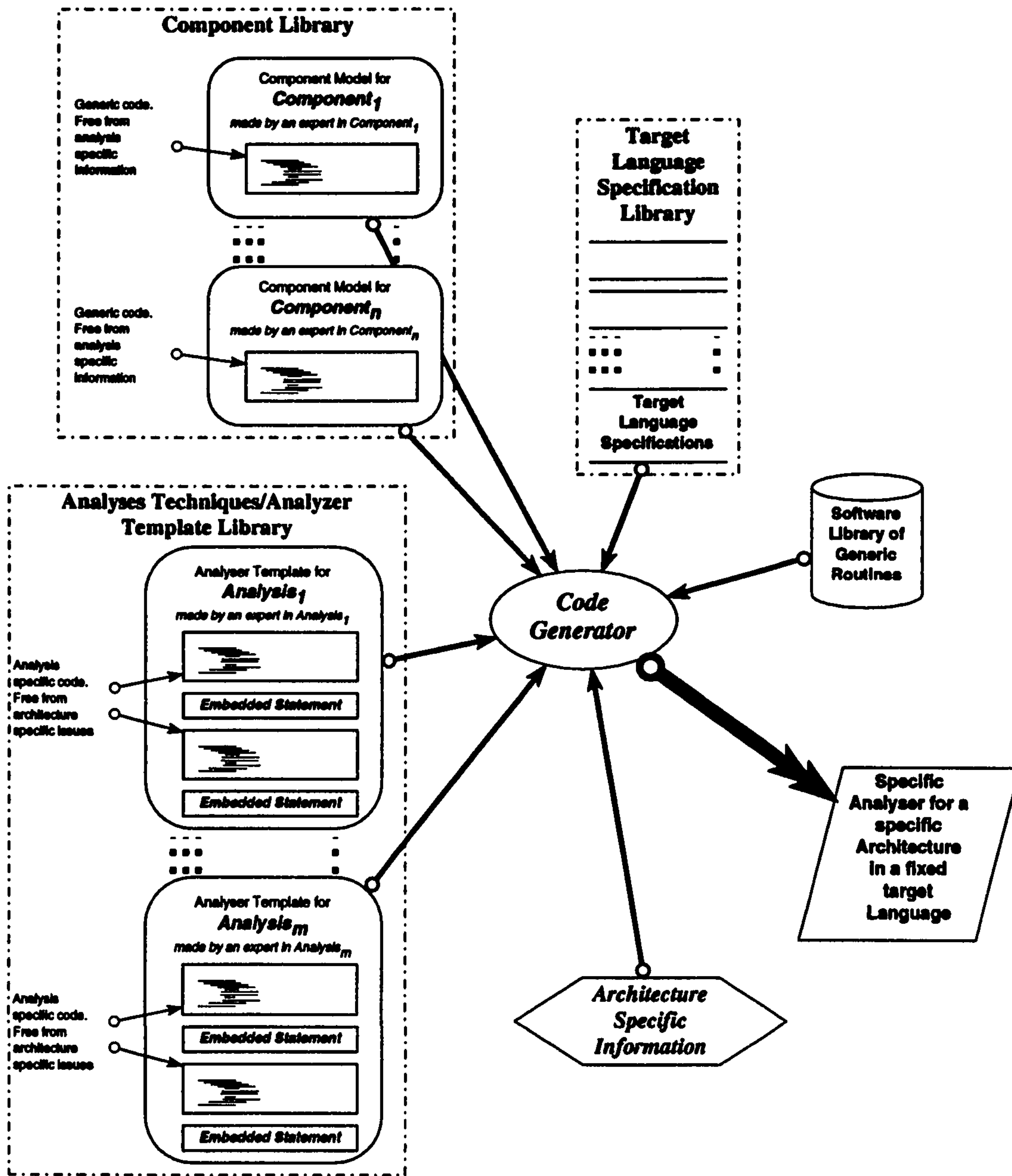


Figure 9.1: The General Framework for Code Synthesis

action marked by EA_1.

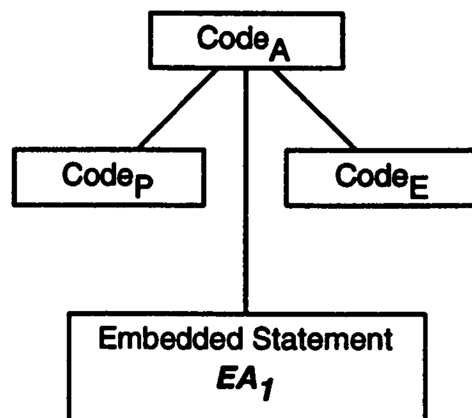


Figure 9.2: A Template with an Embedded Action EA_1

Execution of this embedded action leads to a new tree structure shown in Figure 9.3. This new template has further actions marked by EA_2 and EA_3.

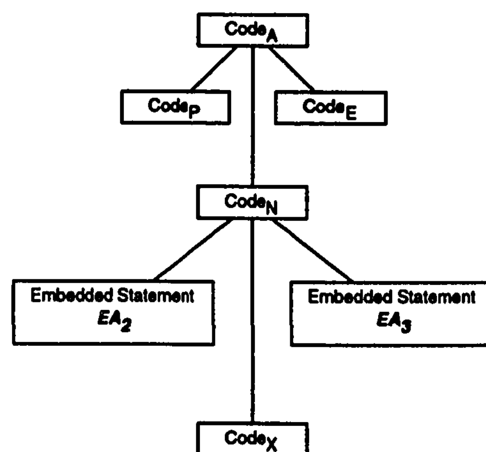


Figure 9.3: Executing the Embedded Action EA_1 to Produce a New Template Containing EA_2 and EA_3

Execution of these leads to the intermediate result shown in Figure 9.4.

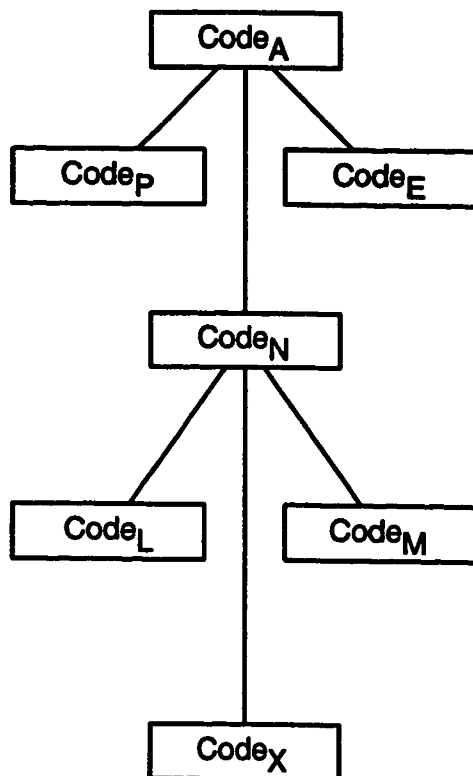


Figure 9.4: The Intermediate Result Produced After Evaluation of All the Embedded Actions

Dependency analysis on this and the evaluation of the syntax functions of the target language produces the final code in the target language shown in Figure 9.5.

Not all the above steps are incorporated in the current version of the system. Additional work needs to be done with respect to the realization of libraries and component models in different languages. For the time being, the target language is *Mathematica* and the source language is $\mathcal{OHDL}$.

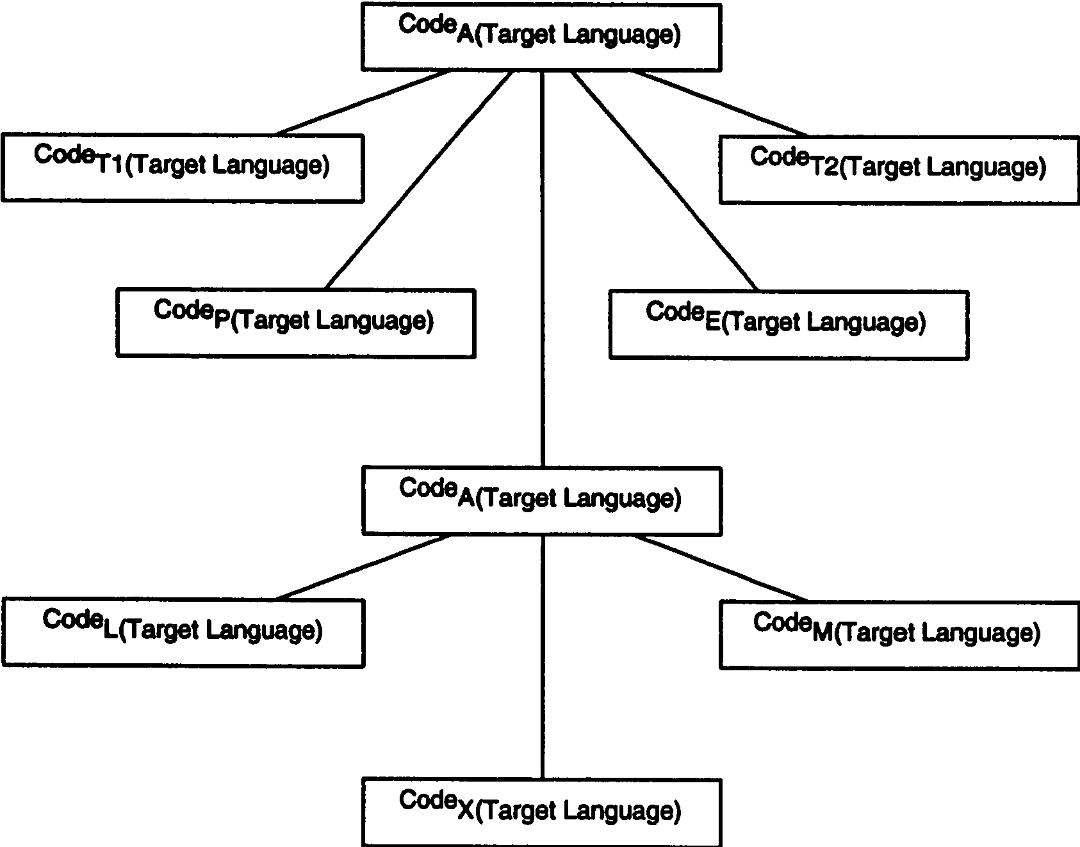


Figure 9.5: The Final Code Produced for the Specific Architecture.

9.4 Splicing and its Restriction to Semantic Preserving Substitutions

Definition: *Spliced Evaluation*

Let E be an execution environment or a context. Let L_I be an intermediate language. Let L_T be the target language. For a splicing evaluation and replacement rules, the splicing of a template T written in T with embedded statements of I is the limiting value of evaluating the embedded statements of I in the template T. The evaluations are carried out in the context E.

9.4.1 Examples/Applications

Figure 9.6 shows an example of a template with an embedded statement. This when evaluated produces a similar embedded statement with reduced parameters (see Figure 9.7). This is exactly like recursion used in numerous applications.

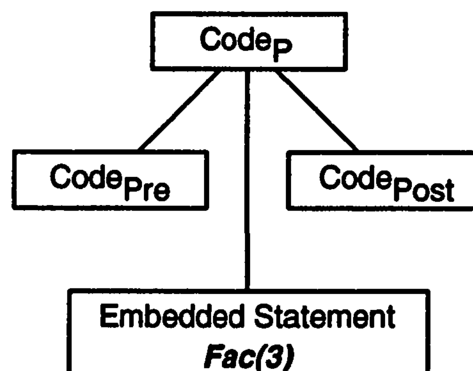


Figure 9.6: An Example of a Template with an Embedded Statement.

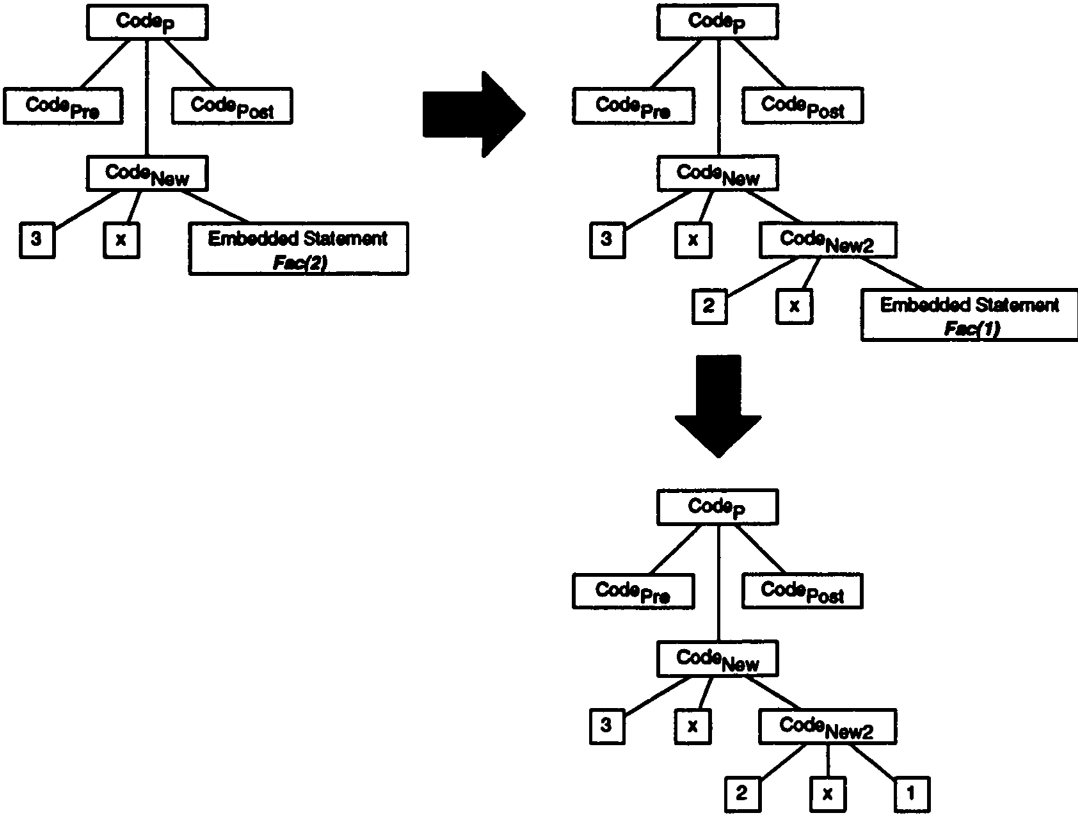


Figure 9.7: Recursive evaluation of the embedded statements.

The problem with blind splicing is that evaluation of the embedded statement might lead to infinite recursion (see Figure 9.8). Careful checks must be placed to avoid this.

9.5 Summary

This Chapter presented a method for generating code for a simulator of optical architectures. This code is executed to produce different analyzers. Template based code generation was discussed with splicing and its restriction to semantic preserving substitutions. Some examples/applications were presented.

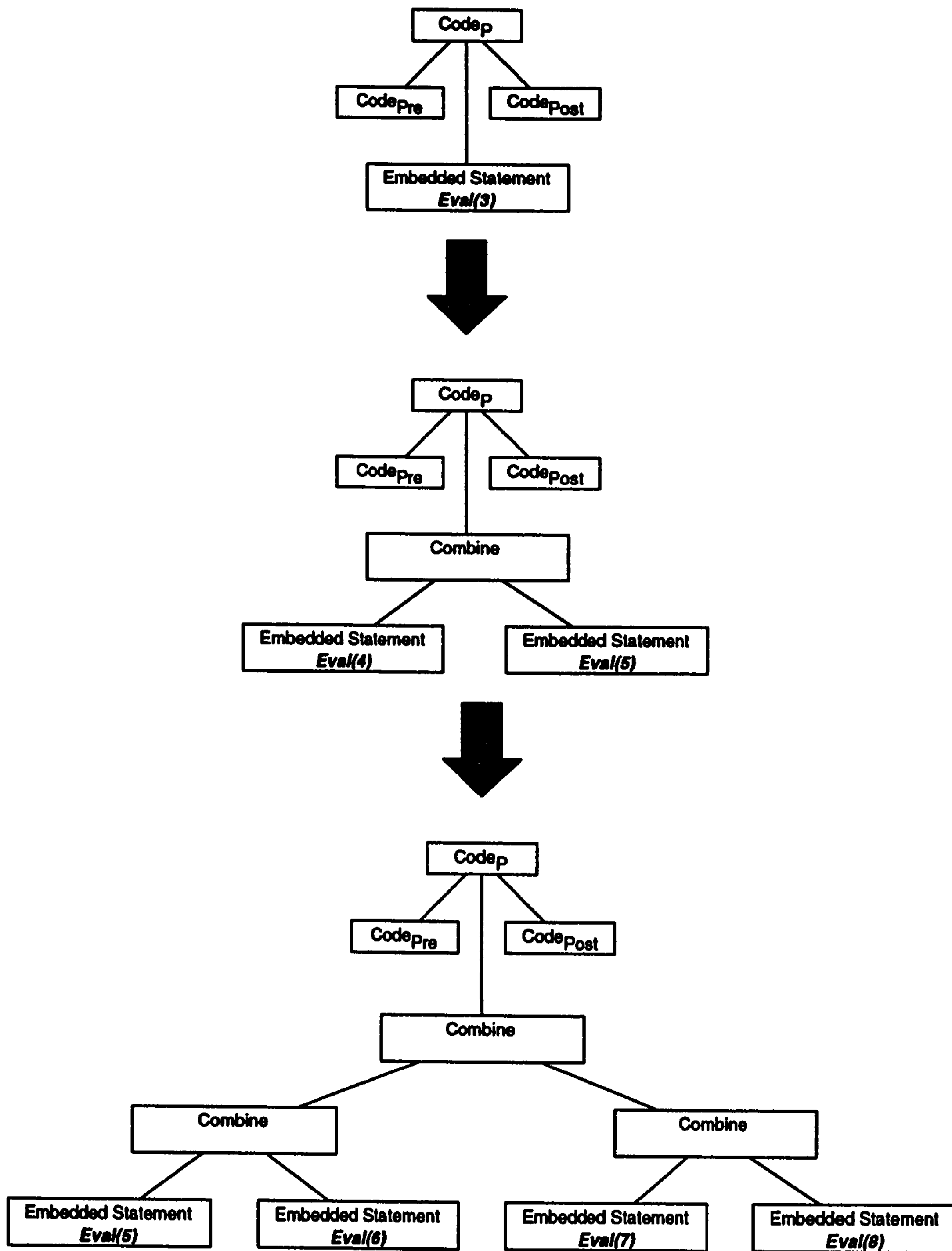


Figure 9.8: Infinite Recursion with Splicing.

Chapter 10

Conclusion

Chapter Abstract

This chapter summarizes the work presented in this thesis. It highlights the specific contributions to the field of CAD software for optical systems, the limitations of OptiCAD in its current form, and outlines possible extensions and future work.

10.1 Contribution of the Thesis

Optical computing is the field that exploits various optical phenomena and components towards the realization of different special/general purpose computer systems. With the demonstration of numerous prototypes, the widespread deployment of fiber optics, increasing use of optical storage, it is becoming an increasingly viable proposition to talk about optical systems as alternatives to the traditional electronic systems/architectures. It appears that the next generation high-performance computer systems will employ optical technologies at least in some subsystems – such as inter-

connection networks/storage.

So far most approaches towards the realization of optical architectures appear to be ad hoc. What is needed is a systematic approach towards the design of these architectures. This requires the development of reasonably broad methodologies, standards for components and devices, and CAD tools/systems.

This thesis attempts to provide a methodology for the development of optical architectures together with the design and implementation of a CAD system for facilitating the design activity.

More specifically the following have been achieved:

---

1. Identification of end user requirements by describing various optical architectures.
2. Design of a special purpose language for computer architects employing optical devices.
3. Modelling of optical components.
4. Development of a design library.
5. Database design and support for vendor-specific databases.
6. A 3D-layout system derived as a byproduct of constraints specification and satisfaction.
7. Design of a hierarchical simulator and its implementation at two levels.
8. Development of power and delay analyzers.
9. Development of several tools for translator development and code generation in the implementation language *Mathematica*.
10. Design of two new architectures using the proposed methodology and *OptiCAD*.

10.2 Synopsis of the Thesis

The thesis is divided into four logical parts.

Part I describes the principles involved in the design of the complete system omitting all the implementation details. It describes issues such as algorithms employed and the design decisions.

Part II contains numerous architectures described and simulated using *OptiCAD*. These have been chosen to illustrate several features/facets of *OptiCAD*.

Part III contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. These include issues such as component modeling, component libraries, simulation and different kinds of analyses.

Part IV presents a large collection of tools as *Mathematica* notebooks. These facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to contribute to other domains.

Part V is a reference to *OptiCAD* in its current form.

The following paragraphs summarize all the Chapters in Part I highlighting the main points.

Chapter 1 outlines the basic concepts of optical computing and the previous work in the area. In particular it defines optical computing and presents some of its applications. It discusses the ongoing work in the area, highlighting the general problems with setting up experiments involving optical components. It also discusses the motivation of the thesis and the objectives of the work. The Chapter discusses the desired trend for computer evolution and the role of optics in this regard along with the recent discoveries and their impact on today's computers. Various Sections pro-

vide a discussion on hardware description languages, functional, object-oriented and rule-based programming language paradigms, constraint specification and satisfaction techniques, discrete event and continuous simulation, code generation and generators, geometry and 3D-graphics.

Chapter 2 presents a user's view of the CAD system. It describes the steps of describing an architecture and analyzing it. A problem of designing a tri-state memory unit (Trip-Flop) is taken up giving complete specifications of the Trip-Flop architecture. These include specifications for instantiation of components, placement and orientation, viewing the architecture from different camera positions, and generating a simulator for delay and power analysis. The process of describing the architecture is presented as a general methodology. The Chapter concludes with a generic user's template for architecture specification.

Chapter 3 presents the top-level design of *OptiCAD*. It presents the four different views of *OptiCAD* – user's/architect's view, domain expert's view, language expert's view and the *OptiCAD* designer's view. It gives the different interfaces of *OptiCAD* with user/architect, domain expert, language expert, outside world tools, the implementation platform and databases. The Chapter gives a module-level description of *OptiCAD*.

Chapter 4 gives details of the hardware description language *OHDL* for *OptiCAD*. It presents a flow-chart outlining the steps to follow when describing an architecture. It gives the *OHDL* definitions of operators, data types, constants, parameters, variables, units, expressions, assertions and constructs. The Chapter also discusses the different errors reported by *OptiCAD*.

Chapter 5 discusses some of the placement/orientation problems and their solutions. It introduces constraints, their role in geometric modelling, and constraint

specification and satisfaction techniques. It introduces the notion of bounding box of a component/assembly. It discusses in detail the formulation of equations from position and orientation specifications and obtaining their solutions.

Chapter 6 presents modelling of optical components and architectures. It introduces the basic concepts of modelling and the motives for model building. It presents an approach for the modelling of optical components It discusses in detail the theories of ray optics, wave optics, beam optics and Fourier optics. For each of these theories, it discusses the postulates, principles and laws, mathematical models, components and their behavior, analysis techniques and relationship to other theories.

Chapter 7 presents a simulation methodology for optical architectures. It presents simulation models and their classification. It describes a discrete event simulator for digital optical architectures. It discusses the difference between global (system level) vs. local (component level) simulation. The Chapter describes in detail a simulation algorithm for optical architectures, and presents a template for a simulator.

Chapter 8 presents analysis as one of the best ways of establishing the correctness of an architecture. It describes signal-flow graph construction using the events produced by simulation. It later uses this graph for power and delay analysis.

Chapter 9 describes a method for generating code for a simulator. This code is executed to produce different analyzers. Template based code generation is discussed with splicing and its restriction to semantic preserving substitutions. Some examples/applications are presented.

10.3 Limitations of the *OptiCAD* System in its Current Form

OptiCAD in its current form suffers from a number of limitations. Several of them are conceptual in nature and these are the most difficult ones to overcome. Several others are related to the design of *OptiCAD*. These can be overcome by redesigning the system with the hindsight. Others are mostly related to the implementation and the availability of appropriate tools and resources. Following are some of the known limitations:

1. support for better theories of light - beam, wave, fourier optics.
2. lack of quality models for the components.
3. methodologies and corresponding algorithms for various types of analyses.
4. validation of the whole system against actual prototypes.
5. development and standardization of interfaces for domain experts, and language experts.
6. database user friendliness.
7. insufficient interactive facilities.

10.4 Possible Extensions and Future Work

The list of limitations indicates that there is ample opportunity to enhance the system in several ways. In addition to those straightforward but time consuming extensions

to the system, the following ideas could be pursued based on the reported work.

- *Formalization of interfaces presented in Chapter 3 facilitating independent evolution of subsystems.*

  This is due to the evolution history of *OptiCAD*. The different roles of user, language expert, domain expert and overall system designer became clearer as the system evolved to its current state. All the subsystems corresponding to each of these can be enhanced in parallel. This requires fixing the interfaces between the subsystems.

- *Describe more example architectures to identify, fix bugs and validate the system.*

  Each of the example architectures described in Part II of this thesis helped in identifying problems with the system. As more and more architectures were described, these problems gradually reduced in number. However, now and then some new architecture magnifies particular limitations that were not apparent before.

- *Devising better architectural solutions to problems.*

  Describing the architectures using the methodology proposed in this thesis, various weaknesses in the architectures' design become apparent. Different approaches using some other architecture would yield better solutions to the same problem. Using *OptiCAD*, different alternatives can be explored and analyzed.

- *Support for formal verification of assemblies.*

  Architectures employ components and assemblies of components as the basic building blocks. Component models are assumed to be correct as they are

described by the domain expert. Since the assemblies are designed by the user/architect, they are prone to errors. To put both components and assemblies on equal footing, it is necessary to have a mechanism wherein an assembly design can be validated. If this is provided then assemblies can receive the same treatment as components with a well defined functionality.

- *Improved support for constraint management.*

  The current implementation provides features for definition and enforcement of constraints. However, internally *Mathematica*'s equation solving capabilities are being exploited to find feasible solutions that satisfy all the constraints. Better techniques need to be developed that provide direct support for constraint management and enforcement.

- *Enhancement of utilities to incorporate better graphics and language translation support.*

  Better support for graphics will greatly improve the 3D layout output. Routines for projecting the layout onto any arbitrary plane will facilitate the task of visual verification of placement and orientation. Additional routines for language translation can make the code generation task much simpler and more powerful. Translation from one form to another by using specifications will make it possible to generate code in several target languages.

- *Synthesis of optical architectures.*

  Given a problem description/specifications, is it possible to come up with an architecture that will solve the problem ? It is not possible to answer this question in a general setting. However, if the problem solution is precise and the

application domain is limited e.g. interconnection networks, then it is possible to vary a few parameters to come up with an architecture to realize this solution in hardware.

- *Applications to other domains.*

  1. *Formal modelling, analysis and realization in hardware/software of protocols.*

     A very high-level detailed specification of a protocol along with a few general templates of protocol implementations will make it possible to automatically generate code for a wide variety of protocols. This description need not consider low level issues such as implementation language. Once a complete specification has been provided, it is possible to generate analyzers for different scenarios for testing and validation purposes.

  2. *Computer network simulators.*

     Computer network simulation can be relatively easily done using the general framework of *OptiCAD*. An expert in the domain can describe different network protocols as components. For example different network protocols such as Novell, TCP/IP can be modelled as components. Each component is responsible for its own functionality. At a higher level a system could be assembled using these components. The connections between them could be given in the form of connection constraints. Once a setup has been described, a comprehensive simulation can be done by using the general message passing mechanism of *OptiCAD*.

  3. *Distributed program synthesis.*

Distributed program writing involves identifying portions of programs that can be executed simultaneously and allocating these tasks to different processors in a network environment. The objective is to maximize parallelism using all the available resources. This is not a trivial task especially if the topology is an evolving one and requires frequent changes to task allocation. It is desirable to simulate the behavior of such a distributed computation before actually carrying it out. Each of the tasks can be modelled as a separate object. The whole computation can be viewed as a collection of objects residing on separate processors interacting by message passing. The objects are similar to the components used in *OptiCAD*. Several instances of these can be created. Constraints can include the processor limitations, communication cost and processing time. After a simulation has been carried out the design can be modified to get maximum throughput. In the end code for a distributed application can be generated. This code will contain essential calls for task allocation, migration at appropriate places. Embedded in the code will be the code to be executed on different processors. Regeneration can be done when the processor topology changes.

4. *Visual languages.*

   Visual programming involves development of programs using 2D icons corresponding to constructs and defining some constraints between them. These are in the form of spatial constraints. *OptiCAD* already has an extensive graphics library for designing different component icons. It is also possible to define relationships between these objects in the form of spatial constraints. The whole mechanism of *OptiCAD* can relatively easily be modified to incorporate definition of visual languages.

10.5 Summary

Contribution of this work were highlighted. Known limitations of the *OptiCAD* system, and of the design were presented. Possible extensions and applications to other domains were sketched.

Bibliography

[1] Ghafoor A., Guizani M., and Sheikh S. "All-Optical Circuit-switched Multistage Interconnection Networks". *IEEE, Journal on Selected Areas of Communication*, 9(8):1218–1226, October 1991.

[2] Alferd V. Aho, Ravi Sethi, and Jeffery D. Ullman. *"Compilers: Principles, Techniques, and Tools"*. Addison-Wesley, 1986.

[3] Nabeel Al-Mosli. "Design & Implementation of a 3D-Graphics System: An Interactive Hierarchical Modelling Approach". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, June 1995.

[4] John Backus. "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs". *Communications of the ACM*, 21(8):613–641, August 1978.

[5] Roger Bailey. *"Functional Programming with HOPE"*. Ellis Horwood Series in Computers and Their Applications. Ellis Horwood ltd., 1990.

[6] Henri E. Bal and Dick Grune. *"Programming Language Essentials"*. Addison-Wesley, 1994.

[7] Partha P. Banerjee and Ting-Chung Poon. *"Principles of Applied Optics"*. Asken Associates Incorporated Publishers, 1991.

[8] D. Barnhard. "Lens Lab 3D". Technical report, 1993.

[9] Alain Bergeron, Henri H. Arsenault, and Denis Gingras. "Single-Rail Translation-Invariant Optical Associative Memory". *Applied Optics*, 34(2):352–357, January 1995.

[10] S. Bian, K. Xu, and J. Hong. "Near Neighbouring Neurons Interconnected Neural Network". *Optics Communications*, 76(3,4):199–202, May 1990.

[11] K. Brenner and A. Huang. "Optical Implementation of the Perfect Shuffle Interconnection". *Applied Optics*, 27(1):135–137, January 1988.

[12] L. Z. Cai and T. H. Chao. "Optical Image Subtraction using a LCTV SLM and White Light Imaging Grating Interferometer". *Journal of Modern Optics*, 37(6):1127–1138, 1990.

[13] H. J. Caulfield. "Improved Relaxation Processor for Parallel Solution of Linear Algebric Equations". *Applied Optics*, 29(20):2978–2981, July 1990.

[14] H. J. Caulfield, J. Shamir, J. E. Ludman, and P. Greguss. "Reversibility and Energetics in Optical Computing". *Optics Letters*, 15(16):912–914, August 1990.

[15] T. J. Cloonan et al. "Architectural Issues Related to the Optical Implementation of an EGS Network based on Embedded Control". *Optical Quantum Electronics*, 24:S415–S442, 1992.

[16] T. J. Cloonan and M. J. Herron. "Optical Implementation and Performance of One-Dimensional and Two-Dimensional Trimmed Augmented Data-Manipulator Networks for Multi-Processor Computer Systems". *Optical Engineering*, 28(4):305–314, 1989.

[17] McAulay Alastair D. *"Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers"*. John Wiley and Sons Inc., 1991.

[18] S. Dasgupta. "The Structure of Design Processes". *Advances in Computers*, 28:1–67, 1989.

[19] A. K. Datta, A. Basuray, and S. Mukhopadhyay. "Arithmetic Operations in Optical Computations Using a Modified Trinary Number System". *Optics Letters*, 14(9):426–428, May 1989.

[20] E. R. Davies. *"Machine Vision: Theory, Algorithms, Practicalities"*. Academic Press, 1990.

[21] L. Dron. "Multiscale Veto Model: A Two-Stage Analog Network for Edge Detection and Image Reconstruction". *International Journal of Computer Vision*, 11(1):45–61, August 1993.

[22] P. Egbert and W. Kubitz. "Application Graphics Modeling Support Through Object Orientation". *COMPUTER*, pages 84-90, Oct 1992.

[23] M. T. Fatehi. "Optical Flip-Flops and Sequential Logic Circuits Using LCLV". *Applied Optics*, 23(13):2163–2171, 1 July 1984.

[24] C. Ferrera and C. Vazquez. "Anamorphic Multiple Matched Filter for Character Recognition, Performance with Signals of Equal Size". *Journal of Modern Optics*, 37(8):1343–1354, 1990.

[25] M. A. Flavin and J. L. Horner. "Average Amplitude Matched Filter". *Optical Engineering*, 29(1):31–37, January 1990.

[26] D. Foley and A. van Dam. *"Fundamentals of Interactive Computer Graphics"*. Addison-Wesley, 1990.

[27] A. Ghosh and P. Paparao. "Matrix Preconditioning: A Robust Operation For Optical Linear algebra Operations". *Applied Optics*, 26(14):2734–2737, July 1987.

[28] G. R. Gindi, A. F. Gmitro, and K. Parthasarathy. "Hopfield Model Associative Memory with Nonzero Diagonal Terms in Memory Matrix". *Applied Optics*, 27(1):129–134, January 1988.

[29] K. Hara, K. Kojima, K. Mitsunga, and K. Kyuma. "Optical Flip-Flop Based on Parallel Connected Algaas/gaas PNPN Structure". *Optics Letters*, 15(13):749–751, July 1990.

[30] P. Henderson. "Functional Programming, Formal Specification, and Rapid Prototyping". *IEEE Transactions on Software Engineering*, SE-10(5):241–250, 1986.

[31] H. S. Hinton et al. "Free-Space Digital Optical Systems". *Proceedings of the IEEE*, 82(11):1632–1649, November 1994.

[32] Ellis Horowitz and Sartaj Sahni. *"Fundamentals of Computer Algorithms"*. Computer Science Press, 1978.

[33] H. Huang, L. Lilu, and Z. Wang. "Parallel Multiple Matrix Multiplication Using an Orthogonal Shadow Casting and Imaging System". *Optics Letters*, 15(19):1085–1087, October 1990.

[34] M. N. Islam. "All Optical Cascadable NOR Gate With Gain". *Optics Letters*, 15(8):417–419, April 1990.

[35] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[36] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[37] J. Jahns and B. A. Brumback. "Integrated Optical Shift and Split Model Based on Planar Optics". *Optics Communications*, 76(5,6):318–320, May 1990.

[38] J. Jahns and M. Murdocca. "Crossover Networks and their Optical Implementations". *Applied Optics*, 27(15):3155–3160, August 1988.

[39] J. Jahns and S. J. Walker. "Imaging with Planar Optical Systems". *Optics Communications*, 76(5,6):313–317, May 1990.

[40] H. F. Jordan. *"Digital Optical Computers at Boulder, Center for Optoelectronic Computing Systems"*. University of Colorado, Boulder, 1991.

[41] A. Kemper and M. Wallrath. "An Analysis of Geometric Modeling in Database Systems". *ACM Computing Surveys*, 19(1):47–91, March 1987.

[42] E. Kerbis, T. J. Cloonan, and F. B. McCormick. "An All-Optical Realization of a 2×1 Free-Space Switching Node". *IEEE Photon Technology Letters*, 2(8):600–602, August 1990.

[43] B. Kliewer. "HOOPS: Powerful Portable 3D-Graphics". *BYTE*, pages 193–194, July 1989.

[44] Kubota Pacific Computer Inc. *"Dore Reference Manual"*, 1991.

[45] A.O. Lafi. "The Design and Implementation of a Structured Programming Language for the Description of Optical Architectures". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, January 1992.

[46] P. Lalanee, J. Taboury, and P. Chavel. "A Proposed Generalization of Hopfields Algorithm". *Optics Communications*, 63(1):21–25, July 1987.

[47] A. L. Lentine et al. "Symmetric Self-Electrooptic Effect Device: Optical Set-Reset Latch, Differential Logic Gate and Differential Modulator/Detector". *IEEE Journal of Quantum Electonics*, 25(8):1928–1936, August 1989.

[48] John R. Levine, Tony Mason, and Doug Brown. *"Lex & Yacc"*. OReilly & Associates, Inc., 1992.

[49] S. Lin, L. Liu, and Z. Wang. "Optical Implementation of the 2D-Hopfield Model for a 2D-Associative Memory". *Optics Communications*, 70(2):87–91, Feburary 1989.

[50] A. W. Lohmann, W. Stroke, and G. Stucke. "Optical Perfect Shuffle". *Applied Optics*, 25:1530, 1986.

[51] Guizani M. "Picosecond Multistage Interconnection Network Architectures for Optical Computing". *Applied Optics*, 33(8):1587–1599, March 1994.

[52] M. A. Memon, S. Ghanta, and S. Guizani. "An Optical Architecture for Edge Detection". In *Seventh IASTED International Conference on Parallel and Distributed Computing and Systems.* Georgetown University, Washington D.C., 1995.

[53] F. L. Miller, J. Maeda, and H. Kubo. "Template Based Method of Edge Linking using a Weighted Decision". In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1808–1815, 1993.

[54] M. Mirsalehi and T. K. Gaylord. "Analytic Expressions for the Sizes of Logically Minimized Truth Tables for Binary Addition and Subtraction". *Applied Optics*, 29(23):3339–3344, August 1990.

[55] R. L. Morrison. "Symmetries that Simplify the Design of Spot Array Phase Gratings". *Journal of Optical Society of America A*, 1992.

[56] S. Mukhopadhyay. "An Optical Conversion System: From Binary to Decimal and decimal to Binary". *Optics Communications*, 76(5,6):309–312, May 1990.

[57] M. Murdocca. *"A Digital Design Methodology for Optical Computing"*. The MIT Press, 1990.

[58] V. K. Murty. "Exact Parallel Matrix Inversion Using para-Hensel Codes with Systolic Processors". *Applied Optics*, 27(10):2022–2024, May 1988.

[59] J. A. Neff. "Major Initiatives for Optical Computing". *Optical Engineering*, 26(1):002–009, January 1987.

[60] M. Oita, J. Ohta, S. Tai, and K. Kyuma. "Optical Implementation of Large Scale Neural Networks Using a Time Division Multiplexing Technique". *Optics Letters*, 15(4):227–229, Feburary 1990.

[61] D. V. Pentelic. "Optical Computation of Determinants". *Optics Communications*, 64(5):421–424, 1987.

[62] D. Peri. "Optical Implementation of a Phase Retrieval Algorithm". *Applied Optics*, 26(9):1782–1785, May 1987.

[63] J.P. Pratt. *"HATCH Users Manual"*. Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[64] J.P. Pratt and V.P. Heuring. "A Methodology for the Design of Continuous-Dataflow Synchronous System". Technical report, Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[65] M. E. Prise et al. "Design of an Optical Digital Computer". In W. Firth, N. Peyhambarian, and A. Tallet, editors, *Optical Bistability IV*, pages C2–15–C2–18. Les Editions de Physique, 1988.

[66] A. Requicha. "Representation for Rigid Solids: Theory, Methods and Systems". *ACM Computing Surveys*, 12(4):437–463, Dec 1980.

[67] Bahaa E. A. Saleh and Malvin Carl Teich. *"Fundamentals of Photonics"*. John Wiley & Sons Inc., 1991.

[68] J. M. Senior and S. D. Cusworth. "Wavelength Division Multiplexing in Optical Fiber Sensor Systems and Networks: A Review". *Optics and Laser Technology*, 22(2):113–126, 1990.

[69] Ravi Sethi. *"Programming Languages: Concepts and Constructs"*. Addison-Wesley, 1989.

[70] Ben A. Sijtsma. "Requirements for a Functional Programming Enviornment". In Rogardt Heldal, Carsten Kehler, and Philip Wadler, editors, *Functional Programming, Glasgow 1991*, pages 339–346. Springer-Verlag, 1992.

[71] H. A. Simon. *"Science of the Artificial"*. The MIT Press, second edition, 1981.

[72] WRI Staff. "*MathLink* External Communication in *Mathematica*". Technical report, WRI, 1991.

[73] WRI Staff. "The 3-Script File Format". Technical report, WRI, 1991.

[74] WRI Staff. "The *Mathematica* Compiler". Technical report, WRI, 1991.

[75] WRI Staff. "Guide to Standard *Mathematica* Packages". Technical report, WRI, 1993.

[76] N. Streibl. "Beam Shaping with Optical Array Generators". *Journal of Modern Optics*, 36(12):1559–1573, 1989.

[77] Boleslaw K. Szymanski, editor. *"Parallel Functional Languages and Compilers"*. ACM Press Frontier Series. Addison-Wesley, 1991.

[78] David A. Turner, editor. *"Research Topics in Functional Programming"*. The UT Year of Programming series. Addison-Wesley, 1990.

[79] U.S. Air Force. *"VHDL Users Manual"*, 1985.

[80] P. Wegner. "Dimensions of Objected-Oriented Modeling". *COMPUTER*, pages 12–21, Oct 1992.

[81] B. S. Wherret, S. Desmond Smith, F. A. P. Tooley, and A. C. Walker. "Optical Components for Digital Optical Circuits". *Future Generation Computer Systems*, 3:253–259, 1987.

[82] D. R. J. White, K. Atkinson, and J. D. M. Osburn. "Taming EMI in Microprocessor Systems". *IEEE Spectrum*, 22(12):30–37, December 1990.

[83] P. Wisskirchen. *"Object-Oriented Graphics"*. Springer-Verlag, 1990.

[84] S. Wolfram. *"Mathematica: A System for Doing Mathematics by Computer"*. Addison-Wesley, second edition, 1991.

[85] Amnon Yariv. *"Optical Electronics"*. Saunders College Publishing: HBJ College Publishers, 1991.

[86] E. Yee and J. Ho. "Neural Network Recognition and Classification of Aerosol Particle Distributions Measured with a Two-Spot Laser Velocimeter". *Optics Communications*, 74(5):295–300, January 1990.

[87] L. Zhang and L. Liu. "Incoherent Optical Implementation of 2D-Complex Discrete Fourier Transform and Equivalent 4-F System". *Optics Communication*, 74(5):295–300, January 1990.

A System for Prototyping Optical Architectures

Volume III: Applications

BY

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

December 1995

Preface

OptiCAD is a Computer Aided Design (CAD) system for designing optical architectures. It was designed and implemented to facilitate the design and verification of optical architectures.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system omitting the implementation details. It is however a complete document describing critical issues, design decisions and algorithms employed.

The second part contains several optical architectures designed and described using *OptiCAD*. These have been carefully chosen so as to illustrate most of the features of *OptiCAD*. This part essentially contains the user's view of the system.

The third part contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. Each notebook is presented as a self contained chapter. These include issues such as component modeling, component libraries, simulator and different kinds of analyses.

A rich collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. These utilities facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to be useful in other domains.

Reading through Parts III and IV is by itself expected to be an illustrative and interesting exercise in appreciating the power of functional programming in general and *Mathematica* in particular. The code is not necessarily efficient. Whenever there was a choice between efficiency and readability/clarity, the latter was preferred.

Part V is a reference manual for *OptiCAD* and the *OHDL* language.

Part II

Example Optical Architectures

Chapter 11

Michelson Setup for the Banyan Network

Chapter Abstract

The architecture presented in this Chapter describes one of the ways of achieving a Banyan interconnection using all-optical components [35].

11.1 Introduction

The Banyan network is a multistage interconnection network that connects N input ports of a computer or switching system with the same number of output ports using a minimum number of connections. The Banyan network consists of $\log_2(N)$ stages. Each node of a specific stage has 2 input connections and 2 output connections. The individual function of the nodes varies with the specific application for which the

network is used. For a sorting network, for example, the nodes may be compare-and-exchange units.

This Chapter describes one approach for realizing the Banyan interconnect. An architecture is presented that achieves one stage of the interconnection network. The *OptiCAD* system is used to describe the architecture in *OHDL*. Results are obtained as a 3D-layout.

11.2 Optical Implementation

11.2.1 Approach

Input:

A vector $V_i = (i_1, i_2, ..., i_n)$.

Output:

A vector $Vo = (o_1, o_2, ..., o_n)$

Function:

V_o is a permutation of V_i. $V_o = P.V_i$ where P is the permutation matrix of Banyan network.

One optical implementation of the Banyan network is based on the use of polarization optics. Space-variant arrays of individually addressable half-wave plates can be used to encode pixels of light. Polarization sensitive optical components like beam splitters can then be used to deflect the light rays and to obtain the space-variant interconnection pattern. The use of polarization optics allows the network to be implemented with minimum light losses between input and output. The Michelson setup is one

way of realizing the Banyan network optically. It uses the principle of reflection for implementing one stage of the Banyan network. The setup consists of input and output planes that are arrays of optical logic gates. The data from the input array are interconnected with those of the output array by one stage of the Banyan network. There are four convex lenses (L_1 to L_4), two polarizing beam-splitters (PBS_1 and PBS_2), one quarter-waveplate (QWP) and one half-waveplate (HWP) in the setup. In addition, three mirrors (M_1, M_2 and M_3) are also used in the implementation. The encoding and decoding of the pixels is done in one plane using only one component (P_1 for encoding and P_2 for decoding). The relative shift of the rows of pixels is achieved by the two tilted mirrors.

11.2.2 Operation

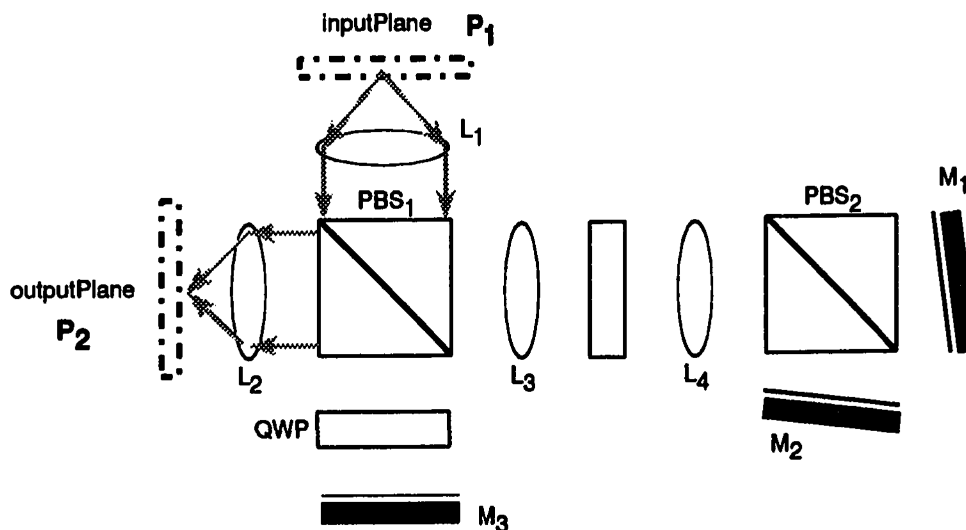


Figure 11.1: Michelson's Setup for the Banyan Network.

Figure 11.1 shows the schematic of the Banyan network. M_3, is placed parallel to

the beam-splitter PBS_1. The input light is split into two by the beam-splitter PBS_1. One component hits the convex lens L_3 and the other strikes the parallel mirror, M_3. The former component travels from lens L_3 and is split by PBS_2 into two components that are reflected by tilted mirrors M_1 and M_2. This tilting is responsible for the crossed interconnection of the Banyan network. After reflecting from the mirrors, the two components (now tilted) merge at beam-merger PBS_2, travel further to PBS_1 where they are merged with the component coming from QWP. QWP and HWP are responsible for the proper direction of the light as it travels through the architecture.

11.3 OHDL Description of the Architecture

11.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

11.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Michelson setup for the Banyan Network";
Architect = "";
```

11.3.3 Components instantiation

Components instantiation

In[3]:=

```
{pbs1, pbs2} = GetObjects[2, "PolarizingBeamSplitter", CatalogNumber ->
"PBS1"];

{m1, m2, m3} = GetObjects[3, "Mirror", CatalogNumber -> "M1"];

inLaser = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

outPlane = GetObject["SLM", CatalogNumber -> "SLM1"];

{lens1, lens2, lens3, lens4} = GetObjects[4, "ConvexLens", CatalogNumber ->
"CL1"];

hwp = GetObject["HalfWavePlate", CatalogNumber -> "HWP1"];

qwp = GetObject["QuarterWavePlate", CatalogNumber -> "QWP1"];
```

11.3.4 Placement Constraints

Parameters

In[4]:=

```
f1 = 0.1; f2 = 0.1; f3 = 0.1; f4 = 0.1; a = 0.1; b = 0.1; c = 0.1; d = 0.1; e
= 0.1;
g = 0.1; h = 0.1; j = 0.1;
```

Constraints

In[6]:=

```
AddPlacementConstraint[default[pbs1]];

AddPlacementConstraints[{
 relativeTo[pbs1, lens1, {0, d, 0}],
 relativeTo[lens1, inLaser, {0, f1, 0}],
 relativeTo[pbs1, lens2, {-a, 0, 0}],
 relativeTo[lens2, outPlane, {-f2, 0, 0}],
 relativeTo[pbs1, qwp, {0, -e, 0}],
 relativeTo[qwp, m3, {0, -g, 0}],
 relativeTo[pbs1, lens3, {b, 0, 0}],
 relativeTo[lens3, hwp, {f3, 0, 0}],
 relativeTo[hwp, lens4, {f4, 0, 0}],
 relativeTo[lens4, pbs2, {c, 0, 0}],
 relativeTo[pbs2, m2, {0, -h, 0}],
 relativeTo[pbs2, m1, {j, 0, 0}]}
];
```

In[7]:=

```
ComputePositions[];
```

11.3.5 Orientation Constraints

Constraints

In[10]:=

```
AddOrientationConstraints[{
 orientationOf[pbs1, {angle[0], angle[0], angle[0]}],
 orientationOf[lens2, {angle[0], angle[90 degree], angle[0]}],
 orientationOf[lens1, {angle[90 degree], angle[90 degree], angle[0]}],
 orientationOf[outPlane, {angle[0], angle[90 degree], angle[0]}],
 orientationOf[inLaser, {angle[-90 degree], angle[0], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[pbs2, pbs1],
 parallelTo[lens3, lens2],
 parallelTo[hwp, lens3],
 parallelTo[lens4, hwp],
 parallelTo[qwp, lens1],
 parallelTo[m3, qwp]
 }
 ];

 AddOrientationConstraints[{
 relativeAngles[m1, pbs2, {angle[20 degree], angle[90 degree], angle[0]}],
 relativeAngles[m2, pbs2, {angle[-110 degree], angle[90 degree], angle[0]}]}
 ];
```

In[11]:=

```
ComputeOrientations[];
```

11.3.6 Layout

In[12]:=

```
ShowArchitecture[];
```

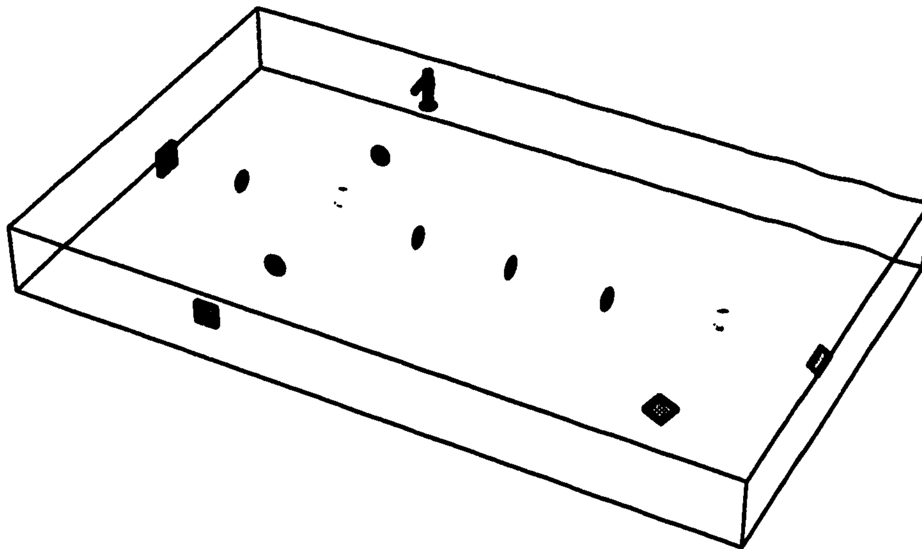


Figure 11.2: 3D-Layout of Michelson Setup for the Banyan Network

11.4 Conclusion

In this Chapter, an optical architecture for realizing one stage of the Banyan interconnect was described using $\mathcal{OHDL}$. All the placement and orientation constraints were supplied to produce the final result as a 3D-layout of the architecture.

Chapter 12

Optical Comparator Unit

Chapter Abstract

This Chapter presents an architecture capable of comparing the input data. It has applications in building switches for interconnection networks.

12.1 Introduction

In a 2 × 2 switch, it is usually important to compare the data received at the switch input channels and generate a control signal that can be used for routing the data to the appropriate outputs. In addition, the switch should be capable of detecting contentions. The architecture presented in this Chapter shows the optical implementation of such an architecture, the *optical comparator.*

Section 12.2.1 describes an approach for realizing the comparator unit using only optical components. A brief sketch of the working of the comparator is given in Section 12.2.2. The architecture is then described using $\mathcal{OHDL}$ of *Opti*$\mathcal{CAD}$. The

results are obtained as a 3D-layout of the architecture.

12.2 Optical Implementation

12.2.1 Approach

Input:

Two inputs A and B.

Output:

Two outputs with respective functionality AB' and $A'B$.

Function:

Compares two bits for equivalence.

An Exclusive-OR (XOR) latching logic can provide this function which can be implemented using *interference filters* (IFs). Based on the incident signals on both sides, these IFs can be in one of three different states – *transmission*, *reflection* and *absorption*. In order to detect contention between two inputs A and B, the generation of signals AB' and $A'B$ is essential. The generation of these two signals can be achieved optically using a single bi-directional, reflection mode, Fabry-Perot type IF [51]. Based on this type of etalon, the optical realization of the desired logic circuit is proposed in Figure 12.1. This system consists of an optical comparator unit and a contention detection unit.

For comparison, first the optical signals AB' and $A'B$ are generated. This is achieved by using the interference filter IF_1 and two quarter-wave plates (QWP_3 and

QWP_4). The input data beams at channels A and B are assumed to be circular-polarized (c-polarized). The idea of having the beams polarized in a certain direction is to be able to control their propagation. Details of the functionality of both units are found in [1].

12.2.2 Operation

The schematic of the comparator unit is given in Figure 12.1. The input laser source L_A is transmitted through a quarter wave plate (QWP_1) that changes its polarization accordingly. Then it is reflected by mirror (M_1) to be split by the polarizing beam-splitter (PBS_1). PBS_1 allows perpendicular-polarized light to pass straight and reflects the parallel-polarized light. The parallel component passes through QWP_3 which makes its polarization circular. Then it is received by the interference filter IF_1. Depending on the state of IF_1, it will either transmit or reflect the light beam. When reflected, it passes again through QWP_3 which changes the polarization to parallel hence causing reflection by PBS_1. In a similar fashion, L_B is reflected by IF_1 which is eventually merged with L_A at PBS_3. Both beams hit IF_2 on its right side. Depending on the state of IF_2, (transmission or reflection), the output is either L_S or *none*. An L_S output indicates that $L_A = L_B$.

12.3 OHDL Description of the Architecture

12.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

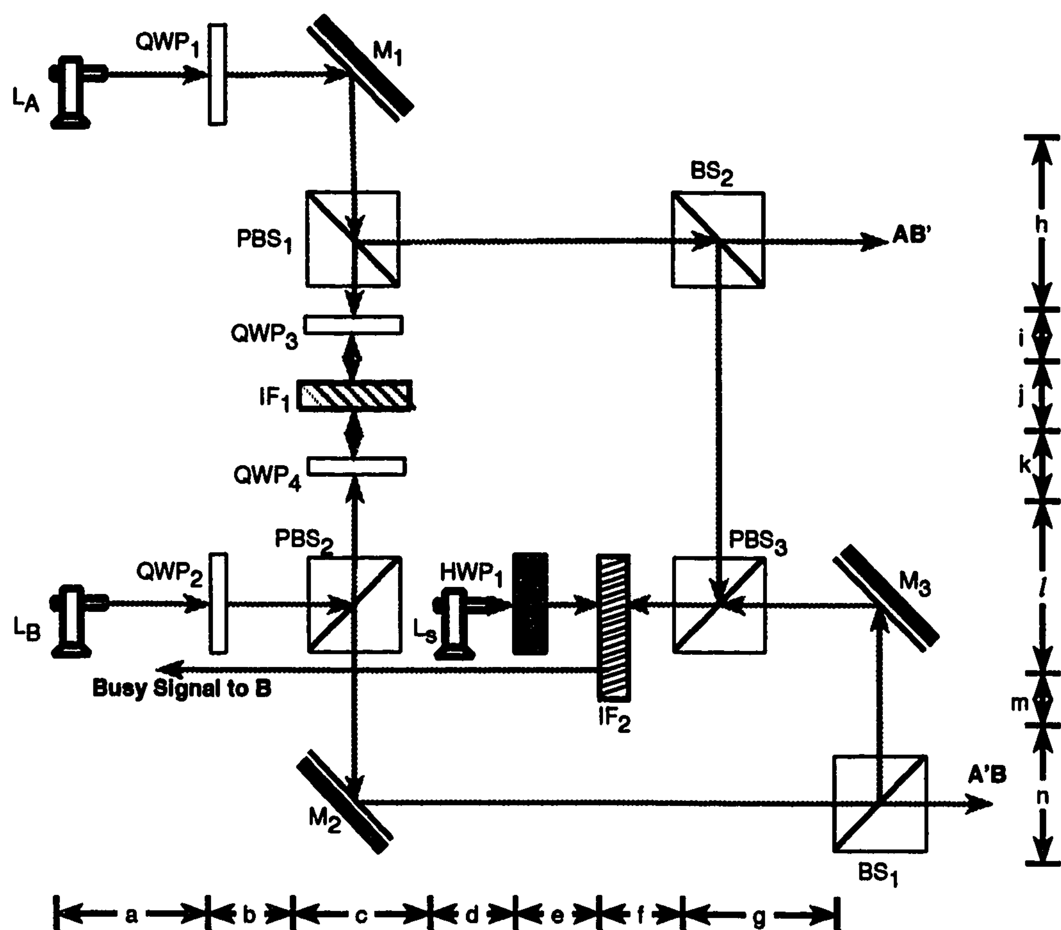


Figure 12.1: The Schematic of the Optical Comparator.

12.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Comparator";
Architect = "";
```

12.3.3 Components instantiation

Parameters

In[4]:=

```
medium = 0.5; high = 1;
```

Components instantiation

In[5]:=

```
{la, lb} = GetObjects[2, "PulsedLaser", CatalogNumber -> "PL1",
 Intensity -> medium];
ls = GetObject["PulsedLaser", CatalogNumber -> "PL2", Intensity -> high];
{qwp1, qwp2, qwp3, qwp4} = GetObjects[4, "QuarterWavePlate",
 CatalogNumber -> "QWP1"];
hwp1 = GetObject["HalfWavePlate", CatalogNumber -> "HWP1"];
{pbs1, pbs2, pbs3} = GetObjects[3, "PolarizingBeamSplitter",
 CatalogNumber -> "PBS1"];
{bs1, bs2} = GetObjects[2, "BeamSplitter", CatalogNumber -> "BS1"];
{m1, m2, m3} = GetObjects[3, "Mirror", CatalogNumber -> "M1"];
{if1, if2} = GetObjects[2, "InterferenceFilter", CatalogNumber -> "IF1"];
```

12.3.4 Placement Constraints

Parameters

In[9]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1; i =
0.1; j = 0.1; k = 0.1; l = 0.1; m = 0.1; n = 0.1;
```

Constraints

In[10]:=

```
AddPlacementConstraint[default[la]];
 AddPlacementConstraints[{
 relativeTo[la, qwp1, {a, 0, 0}],
 relativeTo[qwp1, m1, {b, 0, 0}],
 relativeTo[m1, pbs1, {0, -h, 0}],
 relativeTo[pbs1, qwp3, {0, -i, 0}],
 relativeTo[qwp3, if1, {0, -j, 0}],
 relativeTo[if1, qwp4, {0, -k, 0}],
 relativeTo[qwp4, pbs2, {0, -l, 0}],
 relativeTo[pbs2, qwp2, {-b, 0, 0}],
 relativeTo[qwp2, lb, {-a, 0, 0}],
 relativeTo[pbs2, m2, {0, -(n+m), 0}],
 relativeTo[m2, bs1, {(c+d+e+f+g), 0, 0}],
 relativeTo[pbs2, ls, {c, 0, 0}],
 relativeTo[ls, hwp1, {d, 0, 0}],
 relativeTo[hwp1, if2, {e, -m, 0}],
 relativeTo[if2, pbs3, {f, 0, 0}],
 relativeTo[pbs3, m3, {g, 0, 0}],
 relativeTo[pbs1, bs2, {(c+d+e+f), 0, 0}]}
];
```

In[13]:=

```
ComputePositions[];
```

12.3.5 Orientation Constraints

Constraints

In[15]:=

```
AddOrientationConstraints[{
 orientationOf[m1, {angle[135 Degree], angle[90 Degree], angle[0]}] ,
 orientationOf[pbs1, {angle[90 Degree], angle[0], angle[0]}]
 }];

 AddOrientationConstraints[{
 parallelTo[bs2, pbs1],
 parallelTo[m1, m2],
 parallelTo[m1, m3]
 }];
```

In[16]:=
```
ComputeOrientations[];
```

12.3.6 Layout

In[20]:=
```
ShowArchitecture[];
```

12.4 Conclusion

This Chapter presented a design for the realization of a comparator unit. It is constructed using all-optical components. The design was described in $\mathcal{OHDL}$ of $\mathcal{O}pti\mathcal{CAD}$ which takes care of the alignment, placement and orientation of optical components required for the architecture. The results were obtained as a 3D-layout of the architecture.

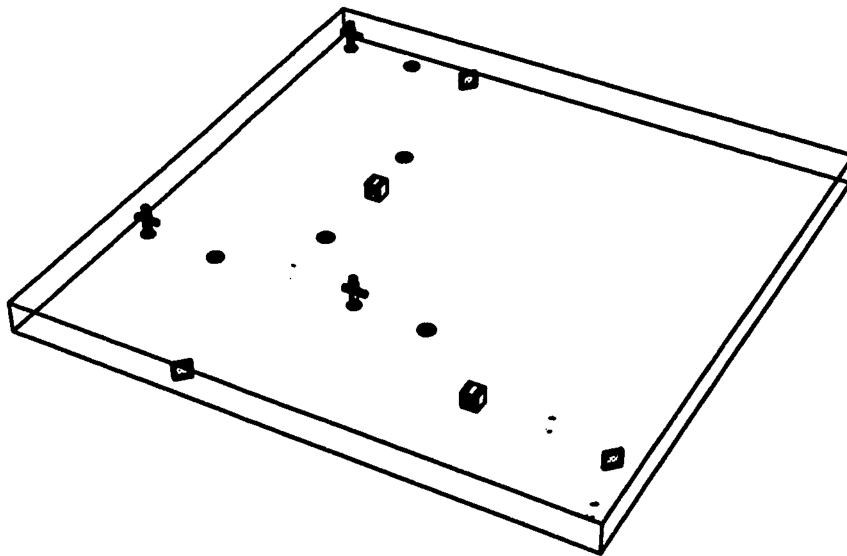


Figure 12.2: 3D-Layout of Optical Comparator Unit

Chapter 13

Mechanically Adjustable Shifter Module

Chapter Abstract

The architecture presented in this Chapter achieves the spatial shift of beams [52]. This is needed in applications where a beam must be directed to a certain output. Free-space interconnection is exploited. The amount of shift may be controlled by a mechanically adjustable parameter.

13.1 Introduction

Numerous optical architectures exploit the free-space connectivity of optics. This usually requires guiding the light from one component to another. A shifter is useful for such manipulations. The shifter also finds applications in architectures operating on 2D-image matrices of data [52].

Section 13.2.1 describes one approach for realizing a shifter module by employing classical optical components. Section 13.2.3 discusses the functionality of the module. The architecture is described using $\mathcal{OHDL}$ of *Opti$\mathcal{CAD}$*. Results are presented as a 3D-layout.

13.2 Optical Implementation

13.2.1 Approach

Input:

A matrix of Beams.

Output:

The input matrix shifted by an amount d.

Function:

Shifts the input matrix by an amount d.

The approach followed here is simple. The shift is realized by reflection of mirrors only. The mirrors are positioned so that the signal gets shifted in the specific dimension by a specified quantity.

13.2.2 Internal parameters and Constraints

The functionality of this architecture is to shift the input beam. Te amount of shift is denoted by the parameter d. The other parameter used here is s. This represents the length of the mirror. For simplicity, we assume that all the mirrors used in this architecture are of length s.

13.2.3 Operation

The schematic of the shifter is shown in Figure 13.1.

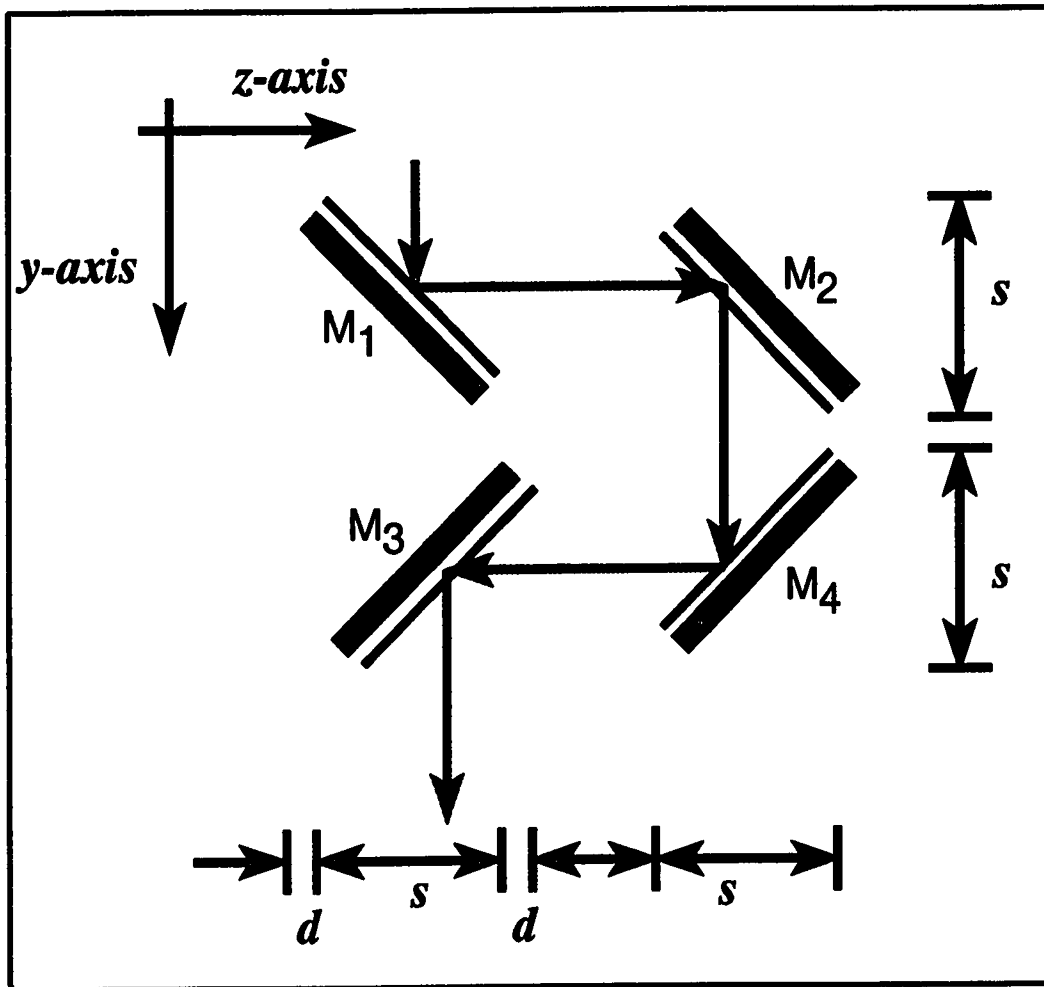


Figure 13.1: The Schematic of the Shifter Module.

The input signal strikes the mirror M_1 which is oriented at an angle of 45° relative to the input beam. This orientation guarantees a 90° rotation of the signal towards mirror M_2. It then travels to mirrors M_3 and M_4. The output signal is parallel to

the input signal but is shifted in the z-direction by an amount d. The parameter d is controllable and can be changed by moving the mirror M_3 horizontally.

13.3 OHDL Description of the Architecture

13.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

13.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Edge Detector: Shifter Module";
Architect = "";
```

13.3.3 Components instantiation

Components instantiation

In[4]:=

```
{m1, m2, m3, m4} = GetObjects[4, "Mirror",
 CatalogNumber -> "M1"];
```

13.3.4 Placement Constraints

Parameters

In[5]:=

```
d = 0.1 cm; s = 1 cm;
```

Constraints

In[6]:=

```
AddPlacementConstraint[default[m1]];

 AddPlacementConstraints[{
 relativeTo[m1, m2, {0, 0, (2d + 2s)}],
 relativeTo[m1, m3, {0, (s + d), d}],
 relativeTo[m3, m4, {0, 0, (2s + d)}]}
 ];
```

In[7]:=

```
ComputePositions[];
```

13.3.5 Orientation Constraints

Constraints

In[8]:=

```
AddOrientationConstraints[{
 orientationOf[m1, {angle[135 Degree], angle[90 Degree], angle[0]}],
 orientationOf[m3, {angle[45 Degree], angle[90 Degree], angle[0]}]}
 ];
 AddOrientationConstraints[{
 parallelTo[m1, m2],
 parallelTo[m3, m4]}
 ];
```

In[10]:=

```
ComputeOrientations[];
```

13.3.6 Layout

In[12]:=

```
ShowArchitecture[];
```

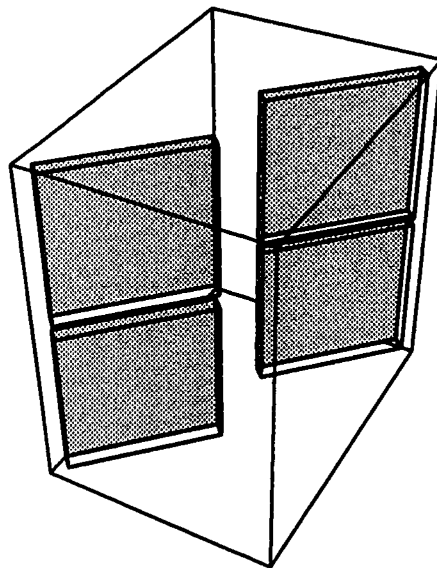


Figure 13.2: 3D-Layout of Mechanically Adjustable Shifter Module

13.4 Conclusion

In this Chapter, the design of a mechanically adjustable shifter unit was proposed. The working and operation of the shifter was discussed. The shifter was described using $\mathcal{OHDL}$ of *Opti$\mathcal{CAD}$*. The results of the description was presented as a 3D-layout.

Chapter 14

Optical Edge Detector

Chapter Abstract

Edge detection is the identification of the boundaries of an image [20]. Several computer vision and robotics applications require realtime edge detection. This may require processing tens of thousands of images per second. Numerous computer vision applications require decision-making at various levels involving the examination of the image at different resolutions. Noise distorts some features of the image. This could mean introduction of new edges or breaking an existing edge or the elimination of an edge. To a limited extent these could be compensated for if edge detection can be carried out at different resolutions. These require a solution for edge detection in an image at different resolutions. Software solutions of edge detection may not be able to provide realtime response. Although hardware solutions may provide realtime response, they tend to be application and resolution dependent and are difficult to generalize for multiple resolutions. Analog hardware solutions suffer from the deteriorated quality at high resolutions. Digital electronic hardware architectures do not scale well with variable resolutions. Optics provides free-space connectivity, spatial parallelism, non-interference among intersecting beams [17]. A parallel architecture that exploits the advantages of optics to solve the realtime, variable-resolution edge detection problem with applications to image processing, computer vision and robotics is presented.

14.1 Introduction

Computer vision has numerous applications in the areas of robotics, automated manufacturing. Eyes/cameras capture the visual information in the form of 2-Dimensional images. The process of constructing object/scene descriptions from images or image sequences is called *image analysis.* This process usually consists of extracting features such as edges from these image bit maps followed by grouping processes which isolate objects from one another and their surroundings.

Edges arise from various types of discontinuities in the 3-D scene such as boundaries of objects, sudden changes of surface orientation, etc. On an image matrix, edges can be distinguished as the common border of two homogenous regions of different intensity values. Edge extraction plays a key role in the inference of 3-D surface structure from an image. They are important in discriminating regions of different parts of the image without being affected by the smooth change in the intensity values.

The classical approach to edge detection is to linearly convolve the image with a set of masks representing ideal step edges in various directions. If we want to apply edge detection to an image $x(M, N)$ by convolving an operator w over the image, then at each point (m, n), the convolution is computed as:

$$y(m,n) = \sum_i \sum_j w(i,j) \cdot x(m-i, n-j)$$

For an $M \times N$ matrix x_{MN}, the complexity is $O(MN)$.

Typically, x can vary from 256×256 to 8192×8192. A sequential software solution to this would be too slow for real time applications since images need to be processed 30 to 60 times a second.

Section 14.2.1 describes one approach for realizing the edge-detector using classical

optical components. A brief discussion of the working of the edge-detector is presented in Section 14.2.3. The architecture is described using *OHDL* of *OptiCAD*. The results are obtained as a 3D-layout.

14.2 Optical Implementation

14.2.1 Approach

Input:

An image encoded in a matrix.

Output:

The input image with enhanced edges.

Function:

Extracts the edges from an input image.

The approach taken in this architecture is one of achieving splitting the input image matrix into two components. One of these components is slightly shifted. They are later merged after undergoing proper encoding. The merging process produces the edges.

Figure 14.1 shows the block diagram for the architecture. Its working can be summarized in the following steps:

1. Split the incoming image to obtain two copies.

2. From one image, obtain a shifted copy.

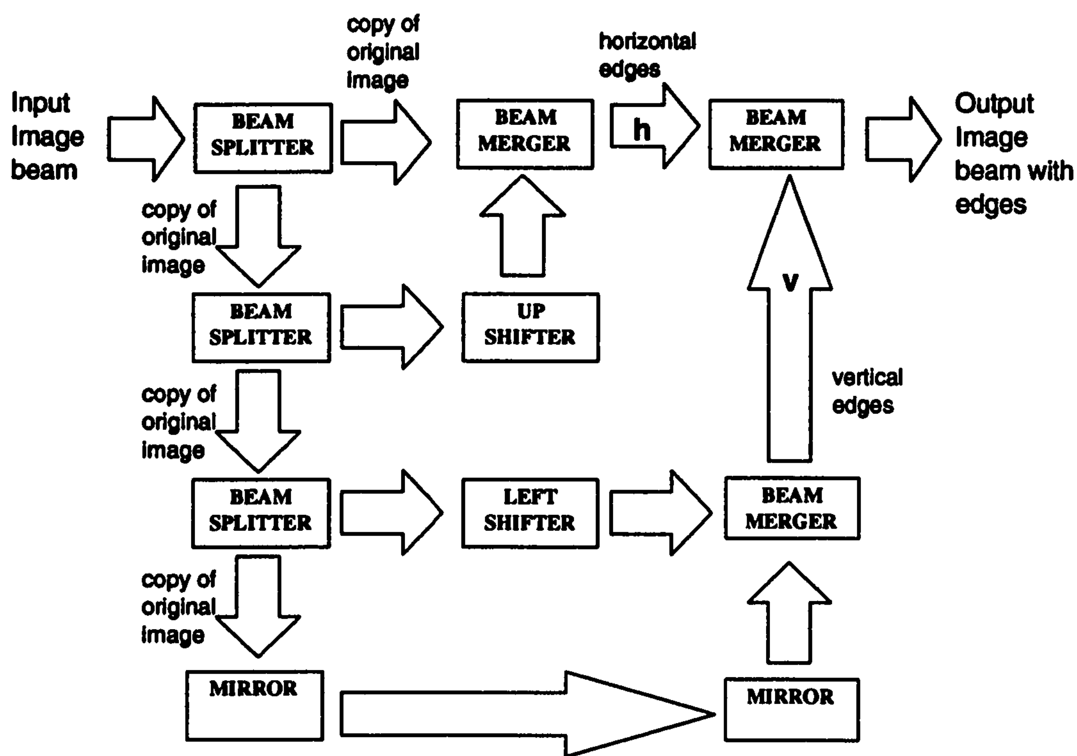


Figure 14.1: Block Diagram of the Architecture.

3. Encode both the image copies so as to realize the appropriate functionality at the output.
4. Merge the two encoded copies.
5. The Merged/Output image will have extracted edges.

14.2.2 Internal parameters and Constraints

The resolution of the input image is defined by a parameter which is dictated by the shifter module. This is given machanically.

14.2.3 Operation

All the operations are carried out in parallel on the 2-D image matrix using spatial logic of optical components.

The realization of the block diagram of Figure 14.1 is shown in Figure 14.2. It makes use of a shifter module, six beam splitters (BS_1 to BS_3 and PBS_4 to PBS_6), two polarization plates (HWP_1 and QWP_1) and two mirrors (M_5 and M_6), a laser input (L_1), a spatial light modulator (SLM_1) and a detector.

Figures 14.3 and 14.4 show a random input figure and its corresponding output. Figures 14.5 and 14.6 refer to the snapshots of the image at points (**v**) and (**h**) in the block diagram.

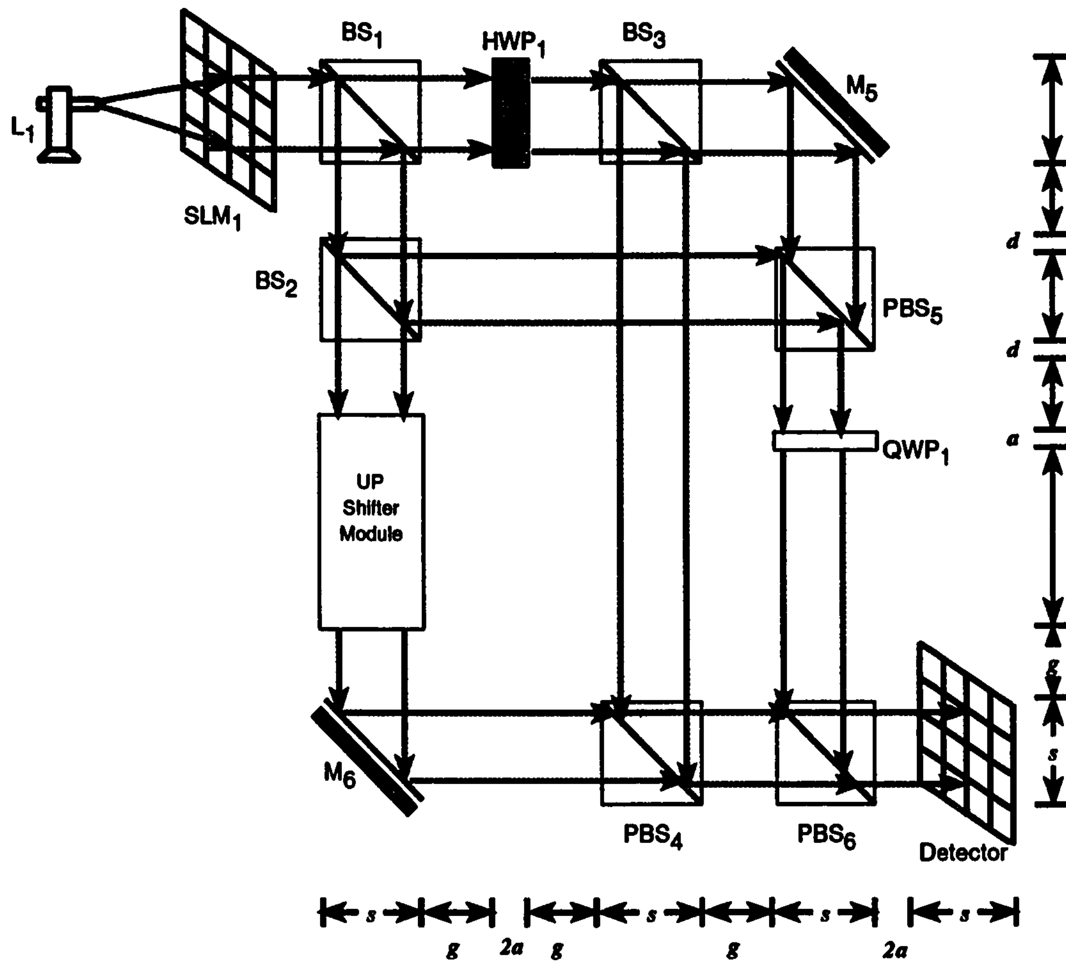


Figure 14.2: Schematic of the Edge Detector.

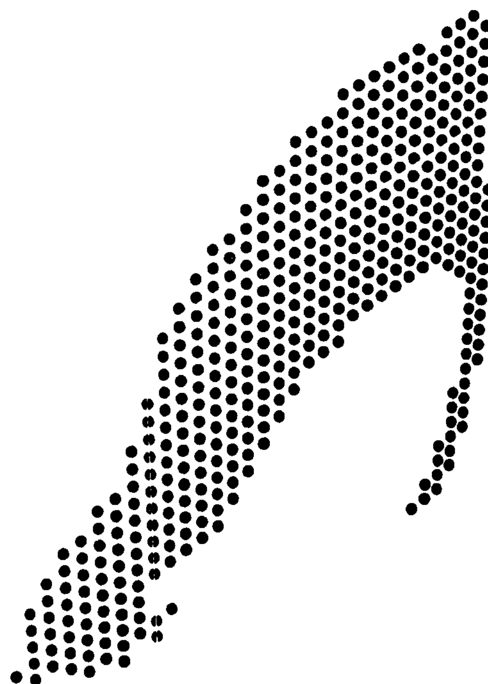


Figure 14.3: The Original Input Image.

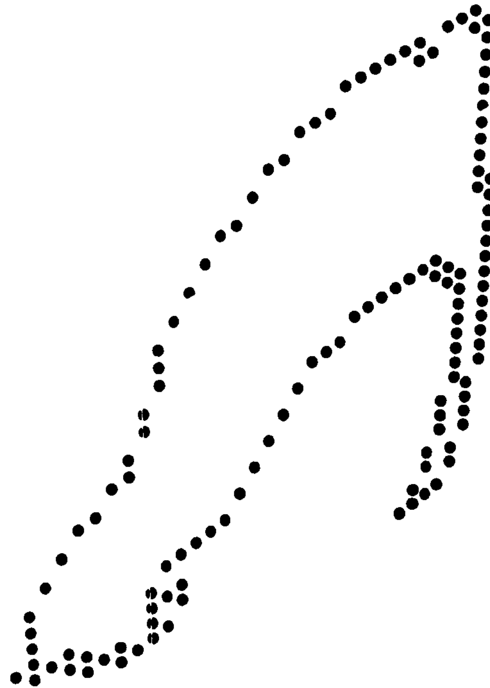


Figure 14.4: The Output Showing the Edges.

Figure 14.5: The Vertical Edges.

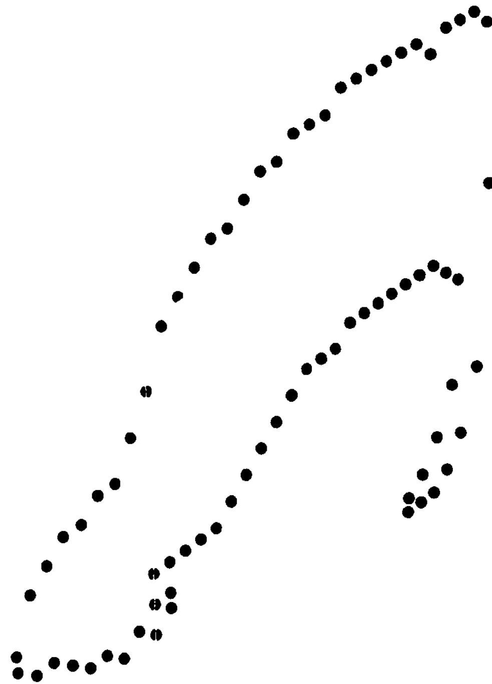


Figure 14.6: The Horizontal Edges.

14.3 OHDL Description of the Architecture

14.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

14.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Main Edge Detector Module";
Architect = "";
```

14.3.3 Components instantiation

Parameters

In[4]:=

```
medium = 1; s = 2 cm;
```

Components instantiation

In[5]:=

```
l1 = GetObject["PulsedLaser", CatalogNumber -> "PL1", Intensity -> medium];

{slm1, slm2} = GetObjects[2, "SLM",
CatalogNumber -> "SLM1"];

{bs1, bs2, bs3, bs4, bs5, bs6} = GetObjects[6, "BeamSplitter",
CatalogNumber -> "BS1"];

{m5, m6} = GetObjects[2, "Mirror",
CatalogNumber -> "M1"];

hwp1 = GetObject["HalfWavePlate", CatalogNumber -> "HWP1"];

qwp1 = GetObject["QuarterWavePlate", CatalogNumber -> "QWP1"];

shifter = GetObject["Shifter", CatalogNumber -> "SH1",
Size -> {s, 2s, 2s}];
```

14.3.4 Placement Constraints

Parameters

In[9]:=

```
g = 3 cm; d = 0.1 cm; a = 1 cm;
```

Constraints

In[11]:=

```
AddPlacementConstraint[default[bs1]];

AddPlacementConstraints[{
relativeTo[bs1, bs2, {0, -(s+g), 0}],
relativeTo[bs2, shifter, {0, -(2s + g), 0}],
relativeTo[shifter, m6, {0, -(g + s), 0}],
relativeTo[bs1, hwp1, {(s + g), 0, 0}],
relativeTo[hwp1, bs3, {(2a + g), 0, 0}],
relativeTo[bs3, m5, {(s+g), 0, 0}],
relativeTo[m5, bs5, {0, -(g + s + d), 0}],
relativeTo[bs5, qwp1, {0, -(g + a), 0}],
relativeTo[qwp1, bs6, {0, -(2s + g), 0}],
relativeTo[m6, bs4, {(s + 2g + 2a), 0, 0}],
relativeTo[bs1, slm1, {-(s + 2a), 0, 0}],
relativeTo[slm1, l1, {-(g), 0, 0}],
relativeTo[bs6, slm2, {(s + 2a), 0, 0}]}
];
```

In[16]:=

```
ComputePositions[];
```

14.3.5 Orientation Constraints

Constraints

In[17]:=

```
AddOrientationConstraints[{
 orientationOf[hwp1, {angle[0 Degree], angle[90 Degree], angle[0]}],
 orientationOf[qwp1, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[m5, {angle[45 Degree], angle[90 Degree], angle[0]}]
} ];

AddOrientationConstraints[{
parallelTo[m6, m5],
parallelTo[hwp1, slm1],
parallelTo[slm1, slm2]
}
];
```

In[18]:=

```
ComputeOrientations[];
```

14.3.6 Layout

In[21]:=
```
ShowArchitecture[];
```

14.4 Conclusion

Edge detection is a fundamental operation in image processing. Real-time edge detection is necessary for computer vision and robotics. The ability to carry out edge detection with different resolutions is needed in several applications where different decisions are taken at different levels of detail. Another application could be detecting edges in noisy images [53, 21]. At a coarse resolution, an edge could be identified and that information could be used at a finer resolution to compensate for the loss of information due to noise. An optical architecture for variable resolution edge detection is presented. Its simulated results on one sample image were shown. The current architecture employs only conventional optical components. Alignment of these components could be tricky and can limit the use of the architecture. The use of *OptiCAD* showed how to ease such a complication. In the proposed architecture different resolutions can only be achieved through mechanical movement. Ongoing and future work includes

1. alternative representations for the encoding of information which may simplify the architecture.
2. delay and power analysis of the architecture and associated changes.
3. eliminating the mechanical alignment and replacing it with either an electronic or optical control.

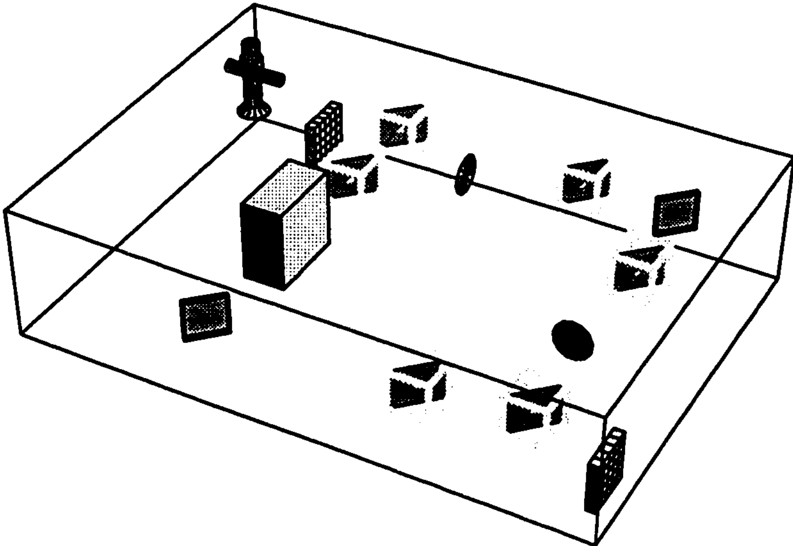


Figure 14.7: 3D-Layout of Optical Edge Detector

Chapter 15

Optical Serial Adder

Chapter Abstract

The architecture of a serial adder using all optical components is presented. It employs carry look-ahead to speed up the addition operation.

15.1 Introduction

This Chapter describes the working of an adder constructed from optical components. It is a serial adder that employs the carry look-ahead algorithm for speeding up the addition process. In this algorithm, all the carries are generated simultaneously. Hence the results can be obtained faster as there is no need to wait for carry propagation.

Section 15.2.1 describes the approach followed for realizing the addition. Section 15.2.2 outlines the operation of the serial adder. The complete architecture is described using $\mathcal{OHDL}$ of the $Opti\mathcal{CAD}$ system. Results are obtained as a 3D-layout.

15.2 Optical Implementation

15.2.1 Approach

Input:

Vectors $A = (a_4, a_3, a_2, a_1)$ and $B = (b_4, b_3, b_2, b_1)$.

Output:

Sum of the Two vectors $S = A + B$.

Function:

The intermediate carries c_i are obtained by:

$$c_{i+1} = g_i + p_i.c_i \tag{15.1}$$

where

$$g_i = a_i.b_i \tag{15.2}$$

$$p_i = a_i \oplus b_i. \tag{15.3}$$

Hence

$$c_2 = g_1 + p_1.c_1 \tag{15.4}$$

$$c_3 = g_2 + p_2.c_2 \tag{15.5}$$

$$= g_2 + p_2.g_1 + p_2.p_1.c_1 \tag{15.6}$$

$$c_4 = g_3 + p_3.c_3 \tag{15.7}$$

$$= g_3 + p_3.g_2 + p_3.p_2.g_1 + p_3.p_2.p_1.c_1 \tag{15.8}$$

are all the carries obtained simultaneously.

All the bits s_i of the sum are obtained in parallel by:

$$s_i = a_i \oplus b_i \oplus c_i = p_i \oplus c_i. \qquad (15.9)$$

Figure 15.1 shows the skeleton of block-diagram of a serial adder. Look-Ahead carry adders are often used to speed up the addition operation, which otherwise depends on the number of bits because of the time for the carry to propagate. In the look-ahead carry adder, the carries for all bits are computed in parallel by additional logic gates. This sort of application is well suited when using optics where speed is obtained by low level parallelism. The optical implementation of a look-ahead carry adder can be obtained using S-SEED devices. The computation time is very fast, essentially the delay of optical propagation through the system, plus a small multiple of the switching time for S-SEEDs. One side of each flip-flop in the S-SEED array is considered the control side and the other side provides the read output.

15.2.2 Operation

The top part of the Figure 15.1 is used to compute the look-ahead carries for all bits simultaneously. The threshold for SS_1 and SS_2 are set by optical or electrical means. SS_1 is set to that the signal at the control must be present at both inputs (AND) in order to exceed the threshold. SS_2 is set so that only one signal (OR) is needed at the control to switch the device on. Sources S_1 and S_2 now read the status of the devices SS_1 and SS_2 respectively by illuminating both sides of the flip-flop to preserve the flip-flop settings. The output signal is obtained by reading one side of the flip-flop.

A fixed holographic interconnection network H_1 fans the signals out to the appropriate bit positions. The resulting signals fall on the control sensors of the 2-D

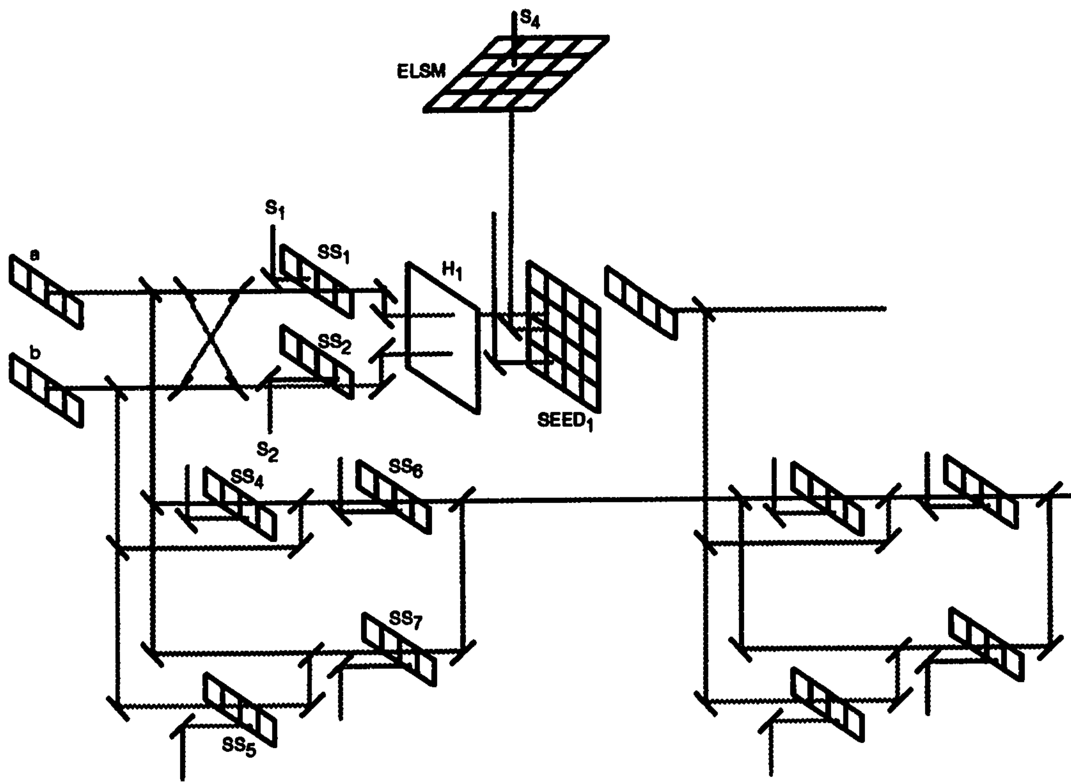


Figure 15.1: A Serial Adder using Optical Look-Ahead.

S-SEED array, $SEED_1$. Threshold are set electronically via $ESLM$ so that a pixel of the S-SEEDs will switch on only if there is sufficient intensity for an AND for the required incident signals. The result for each of the AND gates in a column is ORed together by adding with a cylindrical lens C_1.

The lower part of the figure computes the sum for each bit. 1-D S-SEED devices SS_4 and SS_5 invert each of the inputs, and SS_6 and SS_7 perform the operations $a.b'$ and $a'.b$ respectively. These are ORed together at the output of SS_6 to form $d = a$ XOR b. A second similar system is used to perform the exclusive OR between result d and the carrier c. Such a look-ahead carry adder is fast and feasible in optics.

The functionality of the adder can be summarized in the following points:

1. The top part is used to compute the look-ahead carries for all bits simultaneously.

2. The threshold for SS_1 and SS_2 are set by optical or electrical means.

3. SS_1 is set so that the signal at the control must be present at both inputs (AND) in order to exceed the threshold.

4. SS_2 is set so that only one signal (OR) is needed at the control to switch the device on.

5. A fixed holographic interconnection network H_1 fans the signals out to the appropriate bit positions.

6. The resulting signals fall on the control sensors of the 2-D S-SEED array seed.

15.3 OHDL Description of the Architecture

15.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

15.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Lookahead Carry Adder using SEEDs";
Architect = "";
```

15.3.3 Components instantiation

Components instantiation

In[4]:=

```
{m1, m2, m3, m4, m5, m6, m7, m8, m9,
 m10, m11, m12, m13, m14, m15, m16, m17,
 m18, m19, m20, m21, m22, m23, m24, m25, m26, m27} =
 GetObjects[27, "Mirror", CatalogNumber -> "M1"];

{bs1, bs2, bs3, bs4, bs5, bs6, bs7,
 bs8, bs9, bs10, bs11, bs12, bs13,
 bs14, bs15, bs16, bs17} =
 GetObjects[17, "BeamSplitter", CatalogNumber -> "BS1"];

{ss1, ss2, ss3, ss4, ss5, ss6, ss7,
 ss8, ss9, ss10, ss11} =
 GetObjects[11, "SSEED", CatalogNumber -> "SSEED1"];

{operandA, operandB} = GetObjects[2, "SLM", CatalogNumber -> "SLM1"];

eslm = GetObject["SLM", CatalogNumber -> "SLM1", SLMType -> Electronic];

hol  = GetObject["Hologram", CatalogNumber -> "H1"];

seed = GetObject["SEED", CatalogNumber -> "SEED1"];

lens = GetObject["PlanoConvexCylindricalLens", CatalogNumber -> "PCCL1"];
```

15.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.2; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1;
i = 0.1; j = 0.2; k = 0.1; l = 0.1; m = 0.1;
n = 0.1; o = 0.2; p = 0.1; q = 0.1; r = 0.1; s = 0.1; t = g; u = 0.3;
v = 0.1; w = 0.1; x1 = 0.2; y1 = 0.1; z1 = 0.1;
aa = 0.4; ab = 0.4 ; ac = 0.2; ad = 0.2; ae = 0.2; af = 0.1; ag = 0.2;
ah = 0.1; ai = 0.2; aj = 0.4;
ak = 0.1; al = 0.1; am = 0.2; an = 0.1; ao = 0.1; ap = 0.1;
aq = 0.3;
ar = 0.1; as = 0.1; at = 0.1;
au = 0.3; av = 0.2; aw = 0.1; ax = 0.1;
h1 = 0.4; h2 = 0.1; h3 = 0.1; h4 = 0.3; h5 = 0.1; h6 = 0.5; h7 = 0.4;
h8 = 0.2; h9 = 0.2; h10 = 0.1;
h11 = 0.1; h12 = 0.2; h13 = 0.2; h14 = 0.5; h15 = 0.2; h16 = 0.2; h17 = 0.3;
h18 = 0.1; h19 = 0.1;
h20 = 0.2; h21 = 0.1;
```

Constraints

In[10]:=

```
AddPlacementConstraint[default[operandA]];

AddPlacementConstraints[{
 relativeTo[ operandA, bs1, {a, 0, 0}],
 relativeTo[ bs1 , bs2, {b, 0, 0}],
 relativeTo[ bs2 , bs3, {c, 0, 0}],
 relativeTo[ bs3 , m1 , {d, 0, 0}],
 relativeTo[ m1 , ss1, {e, 0, 0}],
 relativeTo[ ss1 , m2 , {f, 0, 0}],
 relativeTo[ m2, m3, {0, -h3, 0}],
 relativeTo[ m3, hol, {g, 0, 0}],
 relativeTo[ hol, m7 , {j, 0, 0}],
 relativeTo[ hol, m8 , {i, -h2, 0}],
 relativeTo[ hol, eslm , {j, h1, 0}],
 relativeTo[ hol, seed, {u, 0, 0}],
 relativeTo[ seed, lens, {k, 0, 0}],
 relativeTo[ lens, ss3, {l, 0, 0}],
 relativeTo[ ss3, bs16, {m, 0, 0}],
 relativeTo[ lens, m9, {l, 0, 0}],
 relativeTo[ lens, ss3, {l, 0, 0}],
 relativeTo[ ss3, bs16 , {m, 0, 0}],
 relativeTo[ operandA, operandB, {0, -h4, 0}],
 relativeTo[ operandB, bs4, {n, 0, 0}],
 relativeTo[ bs4 , bs5, {o, 0, 0}],
 relativeTo[ bs5 , bs6, {p, 0, 0}],
 relativeTo[ bs6, m6, {q, 0, 0}],
 relativeTo[ m6, ss2, {r, 0, 0}],
 relativeTo[ ss2, m5, {s, 0, 0}]}];
```

In[11]:=

```
AddPlacementConstraints[{
 relativeTo[ m5, m4 , {0, h5, 0}],
 relativeTo[ bs1, bs7, {0, -h6, 0}],
 relativeTo[ bs7, m10, {v, 0, 0}],
 relativeTo[ m10, ss4, {w, 0, 0}],
 relativeTo[ ss4, bs8, {x1, 0, 0}],
 relativeTo[ bs8, m11, {y1, 0, 0}],
 relativeTo[ m11, ss6, {z1, 0, 0}],
 relativeTo[ ss6, bs9, {ad, 0, 0}],
 relativeTo[ bs4, bs12, {0, -h7, 0}],
 relativeTo[ bs12, m14, {aa, 0, 0}],
 relativeTo[ bs7, m15, {0, -h8, 0}],
 relativeTo[ m15, bs14, {ab, 0, 0}],
 relativeTo[ bs14, ss7, {ac, 0, 0}],
 relativeTo[ bs14, m18, {ac-af, 0, 0}],
 relativeTo[ ss7, m19, {ae, 0, 0}],
 relativeTo[ bs12, m16, {0, -h9, 0}],
 relativeTo[ m16 , m17, {ag, 0, 0}],
 relativeTo[ m17, ss5, {ah, 0, 0}],
 relativeTo[ ss5, m20, {ai, 0, 0}],
 relativeTo[ bs9, bs10, {aj, 0, 0}],
 relativeTo[ bs10, m12, {ak, 0, 0}],
 relativeTo[ m12, ss8, {al, 0, 0}],
 relativeTo[ ss8, bs11, {am, 0, 0}],
 relativeTo[ bs11, m13, {an, 0, 0}],
 relativeTo[ m13, ss10, {ao, 0, 0}],
 relativeTo[ ss10, bs17, {ap, 0, 0}]}];
```

In[11]:=

```
AddPlacementConstraints[{
 relativeTo[ bs16, bs13, {0, -h14, 0}],
 relativeTo[ bs13, m21, {ag, 0, 0}],
 relativeTo[ bs10, m22, {0, -h15, 0}],
 relativeTo[ m22, bs15, {au, 0, 0}],
 relativeTo[ bs15 , m26, {av-aw, 0, 0}],
 relativeTo[ bs15 , ss11, {av, 0, 0}],
 relativeTo[ bs13, m23, {0, -h16, 0}],
 relativeTo[ m23, ss9, {ar, 0, 0}],
 relativeTo[ m23, m24, {ar-as, 0, 0}],
 relativeTo[ m24, ss9, {as, 0, 0}],
 relativeTo[ ss9, m25, {at, 0, 0}],
 relativeTo[ ss11, m27, {ax, 0, 0}]}
];
```

In[12]:=

```
ComputePositions[];
```

15.3.5 Orientation Constraints

Constraints

In[13]:=

```
AddOrientationConstraint[
 orientationOf[bs1, {angle[90 degree], angle[0], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[bs4, bs1],
 parallelTo[bs12, bs4],
 parallelTo[bs2, bs1],
 parallelTo[bs7, bs2],
 parallelTo[bs6, bs2],
 parallelTo[bs16, bs2],
 parallelTo[bs10, bs2],
 parallelTo[bs13, bs2]}
 ];
```

In[14]:=

```
AddOrientationConstraint[
 orientationOf[m1, {angle[135 degree], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[m3, m1],
 parallelTo[m7, m3],
 parallelTo[m8, m7],
 parallelTo[m9, m8],
 parallelTo[m10, m9],
 parallelTo[m11, m10],
 parallelTo[m12, m11],
 parallelTo[m13, m12],
 parallelTo[m15, m13],
 parallelTo[m16, m15],
 parallelTo[m22, m1],
 parallelTo[m23, m1]}
 ];
```

In[15]:=

```
AddOrientationConstraint[
 orientationOf[m2, {angle[-45 degree], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraint[
 orientationOf[m4, {angle[45 degree], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[m6, m4],
 parallelTo[m17, m6],
 parallelTo[m18, m17],
 parallelTo[m24, m17],
 parallelTo[m26, m17]}
 ];

 AddOrientationConstraint[
 orientationOf[m5, {angle[225 degree], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[m14, m5],
 parallelTo[m19, m5],
 parallelTo[m20, m5],
 parallelTo[m21, m5],
 parallelTo[m25, m5],
 parallelTo[m27, m5]}
 ];
```

In[16]:=

```
AddOrientationConstraint[
 orientationOf[ss1, {angle[0], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[ss2, ss1],
 parallelTo[ss3, ss1],
 parallelTo[ss4, ss1],
 parallelTo[ss5, ss1],
 parallelTo[ss6, ss1],
 parallelTo[ss7, ss1],
 parallelTo[ss8, ss1],
 parallelTo[ss9, ss1],
 parallelTo[ss10, ss1],
 parallelTo[ss11, ss1]}
 ];

 AddOrientationConstraint[
 orientationOf[seed, {angle[0], angle[90 degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[operandA, seed],
 parallelTo[operandB, operandA]}
 ];

 AddOrientationConstraint[
 orientationOf[eslm, {angle[0], angle[90 degree], angle[90 degree]}]
 ];
```

In[17]:=

```
ComputeOrientations[];
```

15.3.6 Layout

In[18]:=

```
ShowArchitecture[];
```

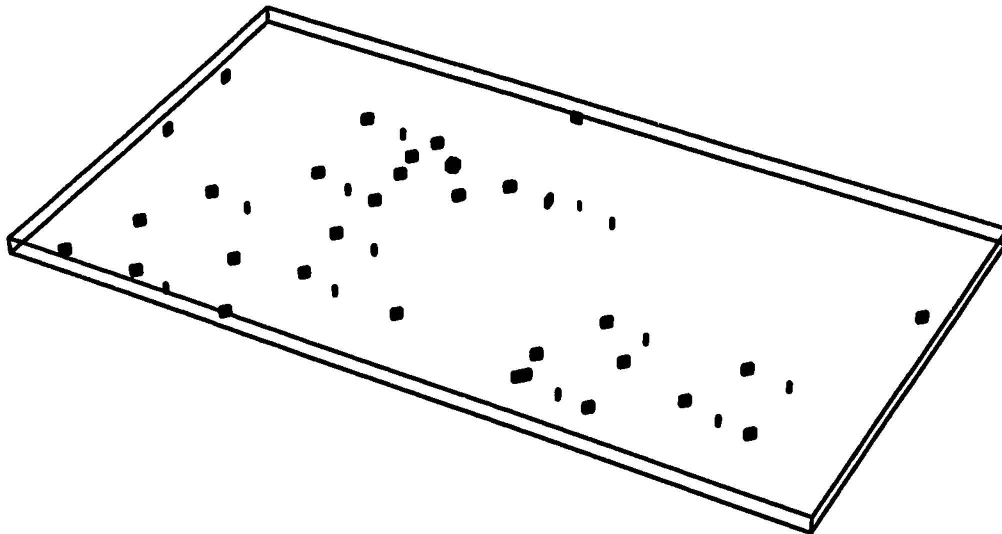


Figure 15.2: 3D-Layout of Optical Serial Adder

15.4 Conclusion

In this Chapter, an architecture for carrying out addition of two operands was presented. The architecture uses the carry look-ahead technique for speeding up the addition process. The design was described in $\mathcal{OHDL}$ – the specification language of $\mathcal{O}pti\mathcal{CAD}$. Results of the description were obtained as a 3D-layout of the architecture.

Chapter 16

Perfect Shuffle Interconnection

Chapter Abstract

An optical implementation of the perfect shuffle interconnection is presented [50]. The architecture is constructed from classical optical components.

16.1 Introduction

Originally, the term *perfect shuffle* (PS) referred to a strategy for mixing a deck of cards. The upper half of the deck is removed and then interlaced with the lower half. The PS shifts the binary address columns one step to the right, cyclically. Thus, the last column reappears on the left. This binary cyclic shift behavior of the PS suggests its usefulness as a communication network. If in addition to the PS network one has also the capability of exchanging nearest neighbors using "exchange boxes". These exchange boxes add more flexibility into the PS network. It is possible to permute

any address configuration into any other configuration with only the order of $(logN)^2$ steps. Such permutations are useful for several transforms, for sorting, and for certain algorithms.

Section 16.2.1 describes an approach for realizing the perfect shuffle interconnect. The functionality of the setup is described in Section 16.2.2. The architecture is described using $\mathcal{OHDL}$. Results are obtained as a 3D-layout.

16.2 Optical Implementation

16.2.1 Approach

Input:

A vector $(i_1, i_2, i_3, ..., i_n)$.

Output:

A vector $(o_1, o_2, o_3, ..., o_n)$ where $o_j \in \{i_1, i_2, i_3, ..., i_n\}$ for $1 \leq j \leq n$.

Function:

Realizes a permutation of the input vector.

The perfect shuffle can be implemented in several ways. These permutations can be performed very fast using classical optics. Optical shuffle with free space propagation can be implemented using either holograms or classical optics. The approach followed here employs classical optical components (lenses and prisms). The principle of operation is one of transforming the input signals to Fourier domain and delaying each one by a different time. Delay in the Fourier domain corresponds to displacement in the ordinary domain. Hence an inverse Fourier Transform realizes the desired

interconnection.

16.2.2 Operation

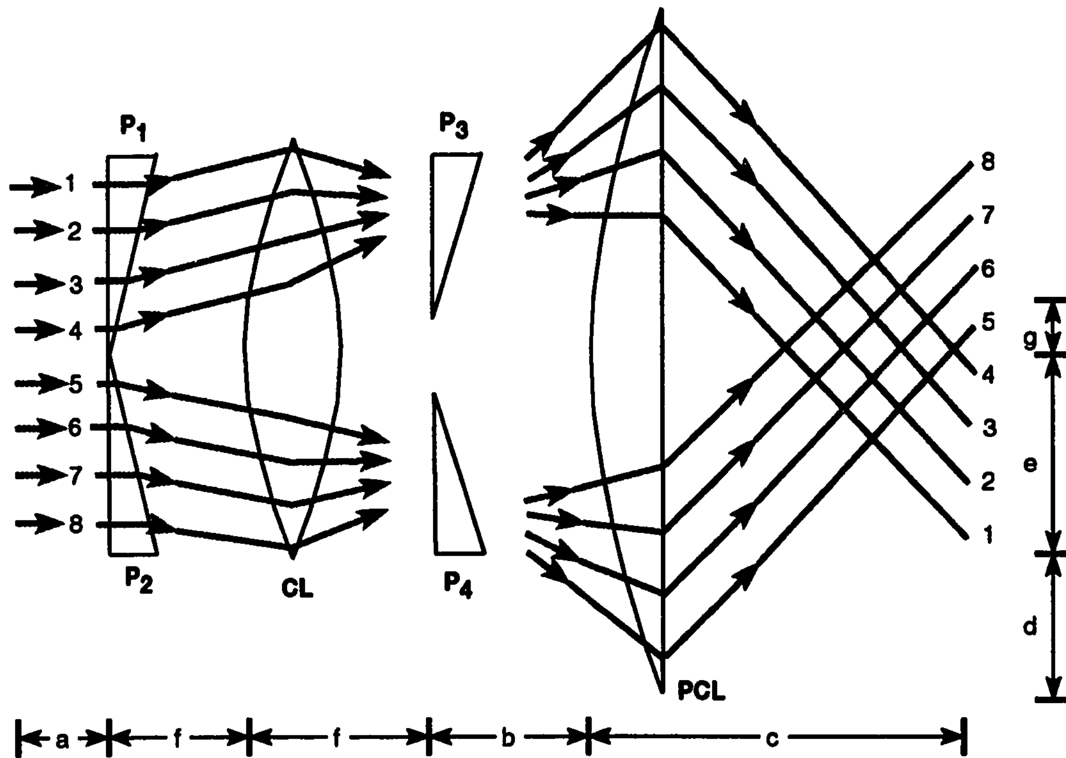


Figure 16.1: The Schematic for the Perfect Shuffle Architecture.

Figure 16.1 shows a possible setup of the optical PS implementation. The input object, indicated by the numbers 1-8, may be a spatial light modulator. The first pair of prisms separates the two halves, so that different shifts can be applied to the upper and lower halves of the input. These shifts are caused by a second pair of prisms at the focal length of the first lens (the Fourier plane). In the output plane the shuffled version of the input object appears with an overall reversed sequence. This

reversal can be compensated by optical means using lenses, mirrors, prisms, and/or other standard components.

16.3 OHDL Description of the Architecture

16.3.1 Additional Definitions for the Component Library

In[1]:=

```
Clear[rightAngledTriangle];
rightAngledTriangle[size_]:=
Module[{len, width, dummy},
 {len, width, dummy} = size;
 {{0, 0, 0}, {len, 0, 0}, {0, width, 0}}
]
```

In[2]:=

```
Clear[prism];
prism[size_, options___]:=
Module[{base, height, top, lines, sides},
 base = BasePolygon /. {options, BasePolygon -> rightAngledTriangle[size]};
 height = size[[3]];
 top = Map[({#[[1]], #[[2]], #[[3]]+height})& , base];
 lines = Transpose[{base, top}];
 sides = Map[ Flatten[#, 1]&, Partition[ Append[lines, lines[[1]] ], 2, 1]];
 sides = Map[{#[[1]], #[[2]], #[[4]], #[[3]]}&, sides]; (* ordering the
vertices *)
 Map[Polygon, Join[{base, top}, sides]]
]
```

16.3.2 Initialization of the Simulator

In[3]:=

```
initialize[];
```

16.3.3 Architecture Information

In[4]:=

```
ArchitectureName = "Optical Implementation of Perfect Shuffle";
Architect = "";
```

16.3.4 Components instantiation

Components instantiation

In[5]:=

```
{p1, p2, p3, p4} = GetObjects[4, "Prism", CatalogNumber -> "P1",
 Size -> {0.5 cm, 1.0 cm, 2 cm}];
cl = GetObject["ConvexLens", CatalogNumber -> "CL1",
 Size -> {3 cm, 3 cm, 0.3 cm}];
pcl = GetObject["PlanoConvexCylindricalLens", CatalogNumber -> "PCCL1",
 Size -> {3 cm, 0.3 cm, 3 cm}];
```

16.3.5 Placement Constraints

Parameters

In[6]:=

```
b = 3 cm; c = 3 cm; d = 0.8 cm; e = 1.0 cm; f = 3 cm; g = 0.1 cm;
```

Constraints

In[7]:=

```
AddPlacementConstraint[default[p2]];

 AddPlacementConstraints[{
 relativeTo[p4, p2, {-2 f, 0, 0}],
 relativeTo[p2, cl, {f, 2e, 0}],
 relativeTo[p1, p2, {0, -2e, 0}],
 relativeTo[p3, p4, {0, -2e - 2g, 0}],
 relativeTo[p4, pcl, {b, -d, 0}]}
 ];
```

In[8]:=

```
ComputePositions[];
```

16.3.6 Orientation Constraints

Constraints

In[9]:=

```
AddOrientationConstraints[{
 orientationOf[pcl, {angle[-60 Degree], angle[0], angle[0]}],
 orientationOf[cl, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[p1, {angle[180 Degree], angle[0], angle[0]}]
 }
 ];

 AddOrientationConstraints[{
 parallelTo[p3, p1]}];
```

In[10]:=

```
ComputeOrientations[];
```

16.3.7 Layout

In[11]:=

```
ShowArchitecture[];
```

16.4 Conclusion

This Chapter presented an optical architecture for realizing the perfect shuffle interconnect. The architecture exploits the Fourier domain to achieve the interconnection. The *OptiCAD* system was used to describe the architecture and obtain results. The description is given in $\mathcal{OHDL}$ and the results are obtained as a 3D-layout.

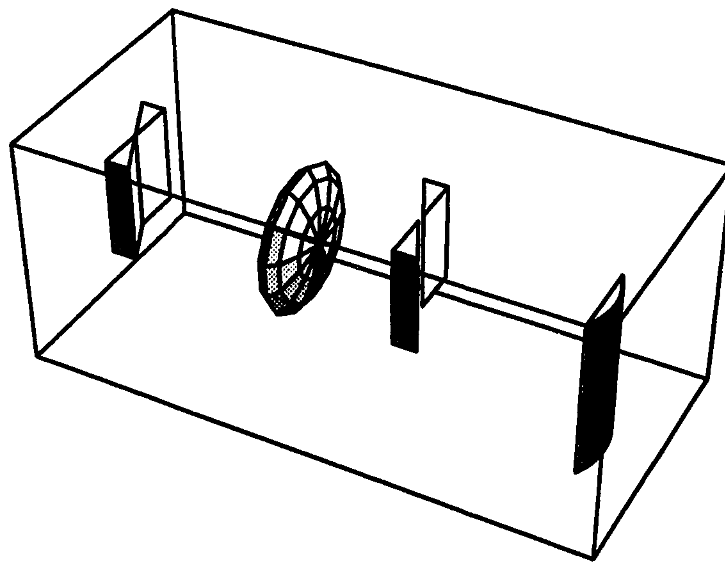


Figure 16.2: 3D-Layout of Perfect Shuffle Interconnection

Chapter 17

All Optical Trip-Flop

Chapter Abstract

The optical Trip-flop is an all optical single unit memory. It can be used as a building block in higher order memory units like registers and latches. It has applications in designing optical buses.

17.1 Introduction

The optical Trip-Flop is an all optical single unit memory. It has the capability to store any of three state values *Low, High* or *S*, i.e., it functions as a tri-state flip-flop.

The Trip-Flop can be used as a building block in higher order memory units like registers and latches. Its three state behavior enables use in buses where three levels are needed.

17.2 Optical Implementation

17.2.1 Approach

Input:

$i \in \{none, low, high, S\}$.

Output:

$o \in \{low, high, S\}$.

Function:

The Trip-Flop has a current configuration c where $c \in \{low, high, S\}$.

The configuration, c, changes depending on the input. If $(i \neq none)$ then $c \leftarrow i$

Bi-stable devices are used to route the light through the architecture. The different logic levels are encoded as different intensities and polarizations. Depending on the attributes of the input signal, the configuration of the Trip-Flop changes so as to continue giving the same output.

17.2.2 Operation

A schematic of the all-optical Trip-Flop is shown in Figure 17.1.

The configuration c of the Trip-Flop is defined using the following settings of the individual components:

$$\mathrm{c} = Low \triangleq \quad \text{config(if1)} = \text{transmitting } \&\&$$
$$\text{config(if2)} = \text{transmitting.}$$

$$c = High \triangleq \text{config(if1)} = \text{reflecting \&\&}$$
$$\text{config(if2)} = \text{transmitting.}$$
$$c = S \triangleq \text{config(if1)} = \text{don't care \&\&}$$
$$\text{config(if2)} = \text{reflecting.}$$

Input: when it is *Low/High* it is p-polarized (circular).

when it is S it is S-polarized.

PBS_1 splits circularly polarized input and reflects S-polarized input.

17.3 OHDL Description of the Architecture

17.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

17.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Trip Flop";
Architect = "";
```

17.3.3 Components instantiation

Parameters

In[3]:=

```
none = 0; low = 0.1; high = 1; s = 2;
```

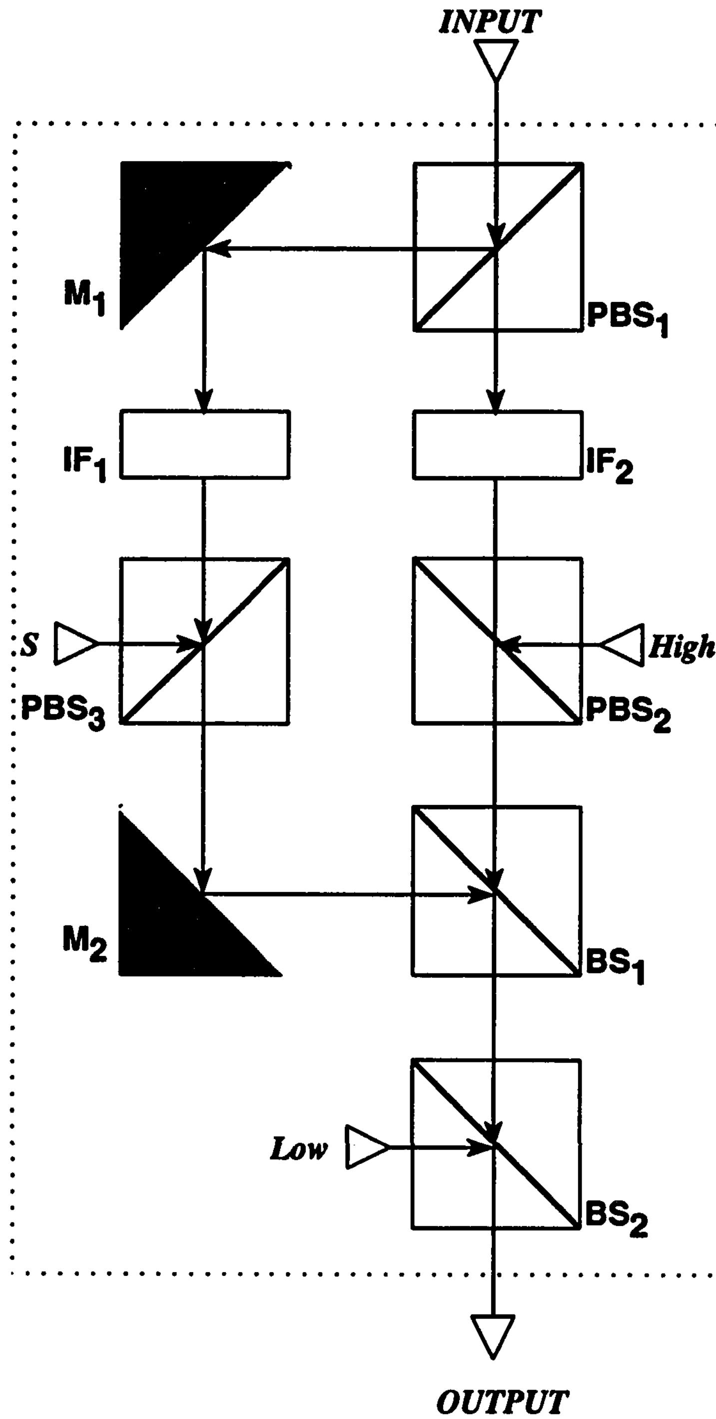


Figure 17.1: The Trip-Flop Architecture.

Components instantiation

In[4]:=

```
{pbs1, pbs2, pbs3} = GetObjects[3, "PolarizingBeamSplitter", CatalogNumber ->
"PBS1"];

{if1, if2} = GetObjects[2, "InterferenceFilter", CatalogNumber -> "IF1"];

{m1, m2} = GetObjects[2, "Mirror", CatalogNumber -> "M1"];

{bs1, bs2} = GetObjects[2, "BeamSplitter", CatalogNumber -> "BS1"];

plh = GetObject["PulsedLaser", CatalogNumber -> "PL1", Intensity -> high];

pll = GetObject["PulsedLaser", CatalogNumber -> "PL1", Intensity -> low];

pls = GetObject["PulsedLaser", CatalogNumber -> "PL2", Intensity -> s];
```

17.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

Constraints

In[6]:=

```
AddPlacementConstraint[default[pbs1]];
AddPlacementConstraints[{
 relativeTo[pbs1, m1, {-a, 0, 0}],
 relativeTo[m1, if2, {0, -b, 0}],
 relativeTo[if2, pbs3, {0, -c, 0}],
 relativeTo[pbs3, pls, {-f, 0, 0}],
 relativeTo[pbs3, m2, {0, -d, 0}],
 relativeTo[pbs1, if1, {0, -b, 0}],
 relativeTo[if1, pbs2, {0, -c, 0}],
 relativeTo[pbs2, plh, {g, 0, 0}],
 relativeTo[pbs2, bs1, {0, -d, 0}],
 relativeTo[bs1, m2, {-a, 0, 0}],
 relativeTo[bs1, bs2, {0, -c, 0}],
 relativeTo[bs2, pll, {-h, 0, 0}]}
];
```

In[6]:=

```
ComputePositions[];
```

17.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
 orientationOf[pl1, {angle[0], angle[0], angle[0]}],
 orientationOf[bs1, {angle[Pi/2], angle[0], angle[0]}]}
];

AddOrientationConstraints[{
 parallelTo[pl1, pls],
 parallelTo[pbs1, pbs3],
 parallelTo[if1, if2],
 parallelTo[pbs2, bs1],
 parallelTo[bs1, bs2],
 parallelTo[pbs3, pls],
 parallelTo[pbs1, if1]}
];

AddOrientationConstraint[perpendicularTo[bs1, pbs1]];

AddOrientationConstraints[{
 atAnAngle[m1, angle[45 degree], if2],
 atAnAngle[pls, angle[180 degree], plh]}
];

AddOrientationConstraints[{
 relativeAngles[m1, pbs1, {angle[45 degree], angle[90 degree], angle[0]}],
 relativeAngles[pbs3, m2, {angle[45 degree], angle[90 degree], angle[0]}]}
];
```

In[7]:=

```
ComputeOrientations[];
```

17.3.6 Layout

In[8]:=

```
ShowArchitecture[];
```

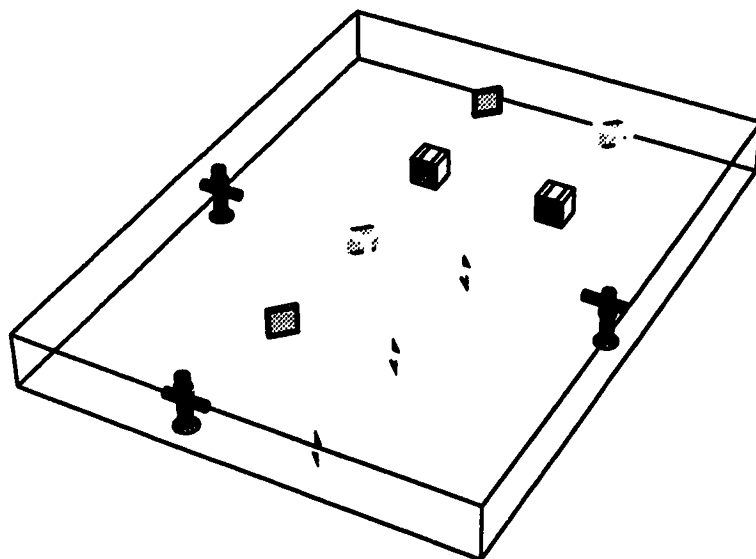


Figure 17.2: 3D-Layout of All Optical Trip-Flop

17.4 Conclusion

In this Chapter the design of a single unit tri-state memory was presented. This unit called the Trip-Flop was described in $\mathcal{OHDL}$ and the output was obtained as a 3D-layout. The architecture was also analyzed for power and delay and some of the results were discussed in Chapter 8-Part I of this thesis.

Chapter 18

The Correlator Associator

Chapter Abstract

The correlator associator extracts a stored image matrix identified by the input image. It has applications in neural networks, storage retrieval, holography. It will be later used as a component in an optical memory architecture [9].

18.1 Introduction

Associative memories played an important role in the evolution of optical neural network systems. They permit a good visual representation of the powerful properties of optics. Usually associative memories are based on a double correlator architecture, need for optical thresholding units and translation invariance.

The correlator associator is a part of the optical implementation of an associative memory. It represents the second layer of a two layer associative memory neural network system.

Section 18.2.1 outlines the approach taken for the realization of the architecture. Section 18.2.2 describes the operation of the associator. The architecture is described in $\mathcal{OHDL}$ of the *OptiCAD* system. Results are produced as a 3D-layout.

18.2 Optical Implementation

18.2.1 Approach

Input:

Input matrix (256x256).

Output:

Output image (128x128)

Function:

Extract the"identified" (by input) stored image.

The patterns are stored in a memory filter. A Fourier Transform of the input matrix is made to pass through this filter to extract the appropriate image matrix.

18.2.2 Operation

Figure 18.1 shows the correlator associator (CA). This represents the second layer of the optical memory architecture. The input matrix from SLM_1 goes through a convex lens CL_1. It then passes through a wave plate (polarizer) WP_1 which in turn transmits it through a memory filter MF_1 placed at the focal length of the Fourier lens CL_2. The collimated output of CL_2 is captured on the output screen $Screen_1$.

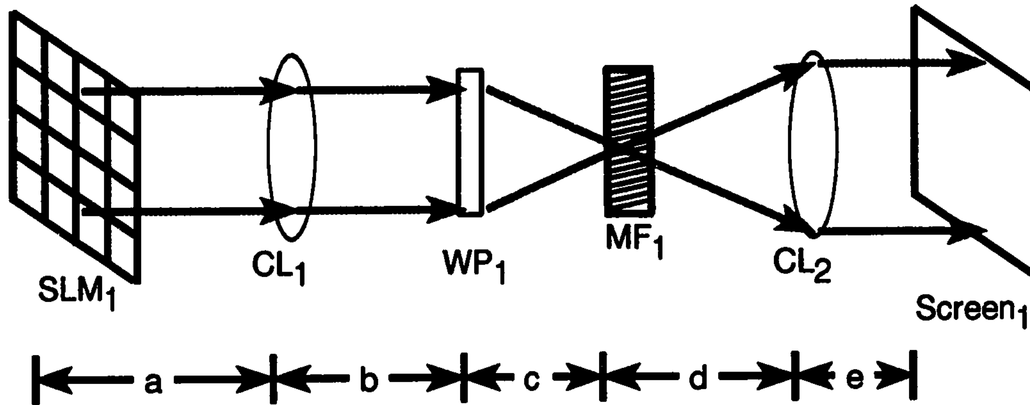


Figure 18.1: The Schematic of the Correlator Associator.

18.3 OHDL Description of the Architecture

18.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

18.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Memory Architecture: Correlator Associator";
Architect = "";
```

18.3.3 Components instantiation

Components instantiation

In[3]:=

```
input = GetObject["SLM", CatalogNumber -> "SLM1"];
{l1, l2} = GetObjects[2, "ConvexLens", CatalogNumber -> "CL1"];
wp1 = GetObject["WavePlate", CatalogNumber -> "WP1"];
mf1 = GetObject["MemoryFilter", CatalogNumber -> "MF1"];
output = GetObject["Screen", CatalogNumber -> "S1"];
```

18.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[input]];

 AddPlacementConstraints[{
 relativeTo[input, l1, {a, 0, 0}],
 relativeTo[l1, wp1, {b, 0, 0}],
 relativeTo[wp1, mf1, {c, 0, 0}],
 relativeTo[mf1, l2, {d, 0, 0}],
 relativeTo[l2, output, {e, 0, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

18.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[input, l2],
 parallelTo[input, wp1],
 parallelTo[input, mf1],
 parallelTo[input, output]}

 ];
```

In[8]:=

```
ComputeOrientations[];
```

18.3.6 Layout

In[9]:=

```
ShowArchitecture[];
```

18.4 Conclusion

The correlator associator module was described. Because of its image retrieval features, it will later be used in a neural network system. It will represent the second layer of the neural network.

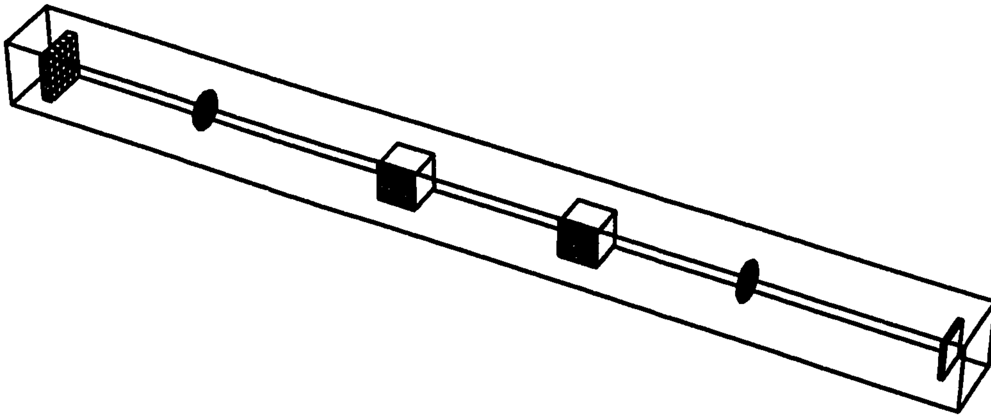


Figure 18.2: 3D-Layout of The Correlator Associator

Chapter 19

The Correlator Detector Module

Chapter Abstract

The correlation detector (CD) represents the first layer of an associative memory two layer neural network. This subsystem is used to correlate the input image with the stored patterns for an optical associative memory. The output of this subsystem is transferred to the opto-electronic thresholder of the associative memory neural-network [9].

19.1 Introduction

Associative memories played an important role in the evolution of optical neural network systems. They permit a good visual representation of the powerful properties of optics. Usually associative memories are based on a double correlator architecture, need for optical thresholding units and translation invariance.

The correlation detector is an integral part in the optical implementation of an associative memory. It represents the first layer of a two layer associative memory

neural network system.

The approach followed to realize this architecture is discussed in Section 19.2.1. Section 19.2.2 discusses the operation of the correlator detector. The $\mathcal{O}pti\mathcal{CAD}$ system is used to describe the architecture and obtain results. The description is given in $\mathcal{OHDL}$ and the results are obtained as a 3D-layout.

19.2 Optical Implementation

19.2.1 Approach

Input:

Input matrix.

Output:

Output image.

Function:

Correlate the input image with the stored patterns.

The input is encoded as a 2D-image matrix. The approach is to perform the Fourier Transform of the matrix and pass it through a phase filter. The phase filter determines the image to be transmitted to the next layer.

19.2.2 Operation

Figure 19.1 describes the correlation detector (CD) which corresponds to the first layer of the neural network. In addition to the input SLM_1 and output $Screen_1$, it is composed of two lenses (L_1 and L_2) and a phase filter (PF_1).

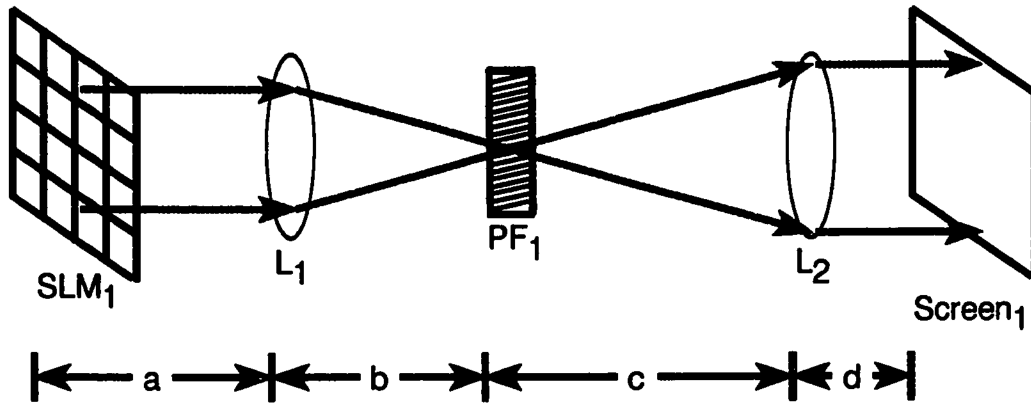


Figure 19.1: The Schematic of the Correlator Detector.

The input matrix passes through the convex lens (L_1) which takes its Fourier Transform and projects it on PF_1 placed at L_1's focal length. PF_1 determines the signal that will be transmitted to the next layer of the neural network. The output of PF_1 is collimated by L_2 and captured on $Screen_1$.

19.3 OHDL Description of the Architecture

19.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

19.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Memory Architecture: Correlator Detector";
Architect = "";
```

19.3.3 Components instantiation

Components instantiation

In[4]:=

```
input = GetObject["SLM", CatalogNumber -> "SLM1"];
{l1, l2} = GetObjects[2, "ConvexLens", CatalogNumber -> "CL1"];
pf1 = GetObject["PhaseFilter", CatalogNumber -> "PF1"];
output = GetObject["Screen", CatalogNumber -> "S1"];
```

19.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1;
```

Constraints

In[8]:=

```
AddPlacementConstraint[default[input]];

 AddPlacementConstraints[{
 relativeTo[input, l1, {a, 0, 0}],
 relativeTo[l1, pf1, {b, 0, 0}],
 relativeTo[pf1, l2, {c, 0, 0}],
 relativeTo[l2, output, {e, 0, 0}]}];
```

In[10]:=

```
ComputePositions[];
```

19.3.5 Orientation Constraints

Constraints

In[12]:=

```
AddOrientationConstraints[{
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[input, l2],
 parallelTo[input, pf1],
 parallelTo[input, output]}

 ];
```

In[15]:=

```
ComputeOrientations[];
```

19.3.6 Layout

In[16]:=

```
ShowArchitecture[];
```

19.4 Conclusion

This Chapter presented the correlator detector module. It will be used in a neural network architecture (Chapter 21) as the first layer of the network. The architecture was described using *OHDL* of *OptiCAD*. Results were presented as a 3D-layout.

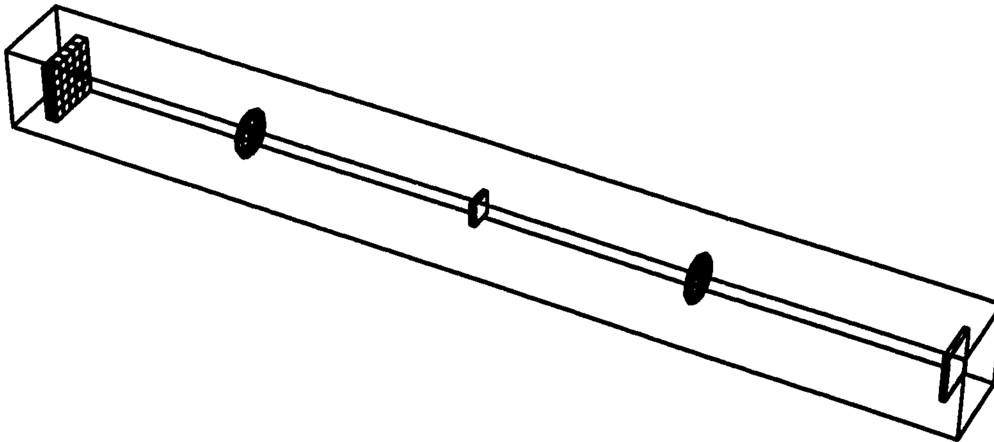


Figure 19.2: 3D-Layout of The Correlator Detector Module

Chapter 20

Optical Thresholder Unit

Chapter Abstract

The thresholder attenuates the input image employing an iterative scheme (attenuation). It serves as the connection between the first layer (correlator detector) and the second layer (correlator associator) in an optical memory architecture [9].

20.1 Introduction

This subsystem performs the thresholding operation of the optical associator memory. It taps the output of the first layer and modulates the output of an electrically addressed spatial light modulator. This is done in order to attenuate small signal intensities while keeping higher signal values unchanged. This type of thresholder permits the light path to be uninterrupted so that the photons arriving at the system output come directly from the input laser without going through the electronic relay.

20.2 Optical Implementation

20.2.1 Approach

Input:

Input matrix (input image) (256x256).

Output:

Output image (256x256).

Function:

Thresholds the input image employing an iterative scheme.

The output of the first layer (the image) is attenuated in a loop. At each iteration a copy of the image is sent to the second layer.

20.2.2 Operation

Figure 20.1 shows the opto-electronic thresholder which serves as the connection between the first and second layer of the optical memory architecture.

The output of the first layer passes through the liquid crystal TV ($LCTV_1$), which has an image from the CCD camera. The image is passed through the beam-splitter which splits the image into two parts. One component goes to the second layer (correlation associator) and the other passes through a polarizer, attenuator and then an imaging lens which eventually is fed back to the $LCTV_1$.

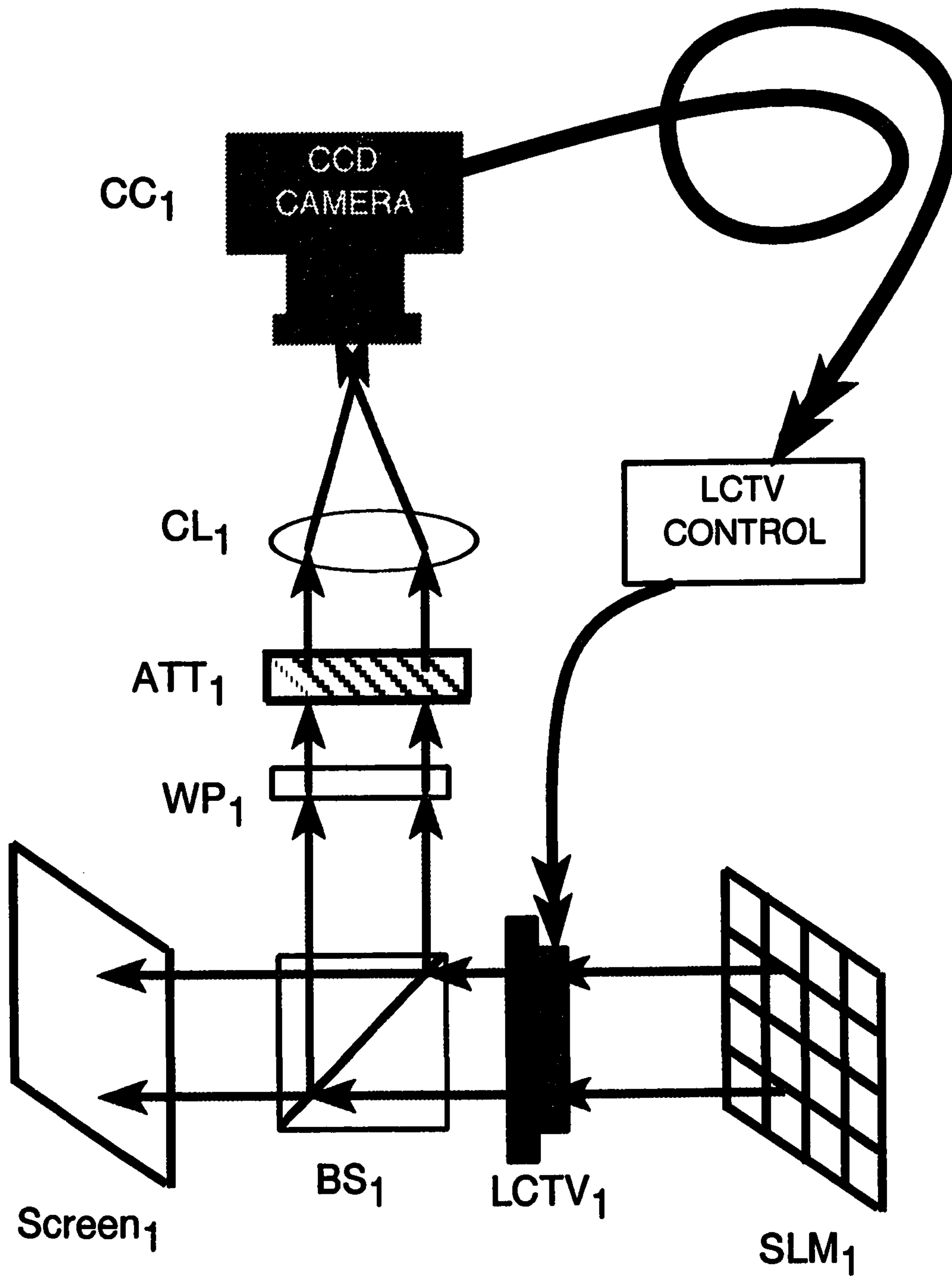


Figure 20.1: The Schematic of the Thresholder.

20.3 OHDL Description of the Architecture

20.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

20.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Memory Architecture: Thresholder";
Architect = "";
```

20.3.3 Components instantiation

Components instantiation

In[4]:=

```
input = GetObject["SLM", CatalogNumber -> "SLM1"];
lctv1 = GetObject["LiquidCrystalTV", CatalogNumber -> "LCTV1"];
cc1 = GetObject["CameraControl", CatalogNumber -> "CC1"];
bs1 = GetObject["BeamSplitter", CatalogNumber -> "BS1"];
wp1 = GetObject["WavePlate", CatalogNumber -> "WP1"];
att1 = GetObject["Attenuator", CatalogNumber -> "ATT1"];
cl1 = GetObject["ConvexLens", CatalogNumber -> "CL1"];
m1 = GetObject["Mirror", CatalogNumber -> "M1"];
output = GetObject["Screen", CatalogNumber -> "S1"];
```

20.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.4; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

Constraints

In[11]:=

```
AddPlacementConstraint[default[bs1]];

 AddPlacementConstraints[{
 relativeTo[bs1, lctv1, {-b, 0, 0}],
 relativeTo[lctv1, input, {-a, 0, 0}],
 relativeTo[lctv1, cc1, {0, d, 0}],
 relativeTo[bs1, output, {c, 0, 0}],
 relativeTo[bs1, wp1, {0, e, 0}],
 relativeTo[wp1, att1, {0, f, 0}],
 relativeTo[att1, cl1, {0, g, 0}],
 relativeTo[cl1, m1, {0, h, 0}]}];
```

In[13]:=

```
ComputePositions[];
```

20.3.5 Orientation Constraints

Constraints

In[15]:=

```
AddOrientationConstraints[{
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[cl1, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[input, output],
 parallelTo[cl1, att1],
 parallelTo[wp1, cl1]}
 ];
```

In[19]:=

```
ComputeOrientations[];
```

20.3.6 Layout

In[21]:=

```
ShowArchitecture[];
```

20.4 Conclusion

In this Chapter, an optical thresholder was presented. It was described using $\mathcal{OHDL}$. The results were obtained as a 3D-layout.

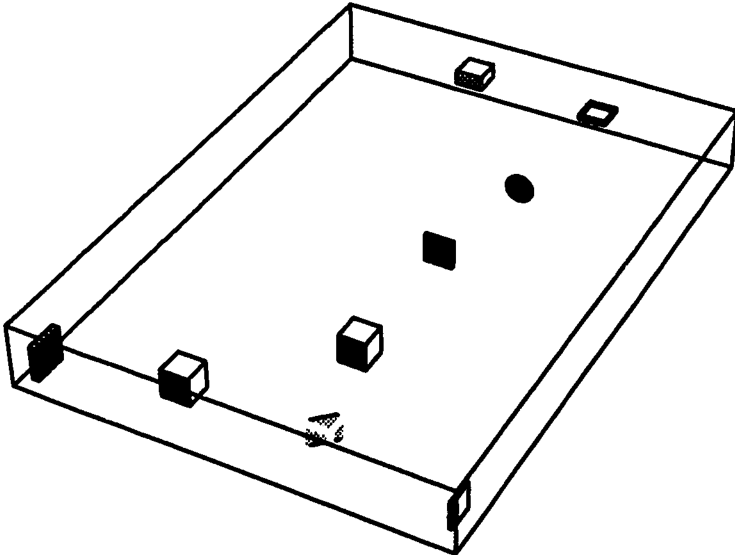


Figure 20.2: 3D-Layout of Optical Thresholder Unit

Chapter 21

Optical Memory Architecture

Chapter Abstract

A feed-forward associative-memory neural network architecture is presented [9]. It makes use of three of the architectures already presented, i.e., the correlator associator, correlator detector and the thresholder.

21.1 Introduction

The architecture presented in this Chapter realizes a two layer neural network. All the pixels of the input plane are analyzed so as to permit translational invariance. This is done by using the thresholder module presented in Chapter 20.

Section 21.2.1 describes the approach taken for designing this architecture. Section 21.2.2 explains its functionality.

21.2 Optical Implementation

21.2.1 Approach

Input:

Input matrix (input image) – a continuous function $f(x, y)$ (256×256 pixels) representing an image.

Output:

A screen to display a two variable continuous function.

Function:

Associative memory.

The approach taken is to create a two layer neural-network using the correlator detector and the correlator associator. The thresholder connects these two modules.

21.2.2 Operation

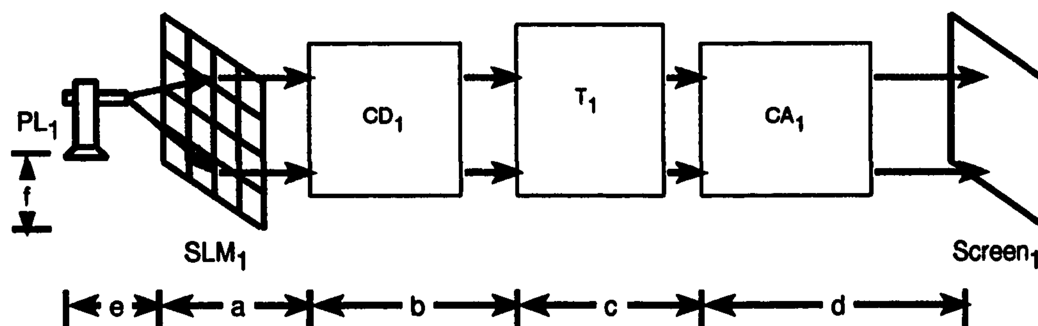


Figure 21.1: The Complete Optical Memory Architecture.

Figure 21.1 shows the schematic of the complete optical memory architecture. The input is supplied by a spatial light modulator (SLM_1) which passes to the correlator detector, CD_1. This is the first layer of the neural network. The output of CD_1 goes to the thresholder module T_1 which in turn sends it to the correlator associator CA_1. The output of CA_1 is projected on the screen. Details of the working of this architecture are found in [9].

21.3 OHDL Description of the Architecture

21.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

21.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Optical Memory Architecture";
Architect = "";
```

21.3.3 Components instantiation

Components instantiation

In[4]:=

```
pl1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];
slm1 = GetObject["SLM", CatalogNumber -> "SLM1"];
screen1 = GetObject["Screen", CatalogNumber -> "S1"];
{cd1, t1, ca1} = GetObjects[3, "Assembly", CatalogNumber -> "A1",
 Size -> {0.05, 0.05, 0.05}];
```

21.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.01;
```

Constraints

In[7]:=

```
AddPlacementConstraint[default[slm1]];

 AddPlacementConstraints[{
 relativeTo[slm1, pl1, {-e, f, 0}],
 relativeTo[slm1, cd1, {a, 0, 0}],
 relativeTo[cd1, t1, {b, 0, 0}],
 relativeTo[t1, ca1, {c, 0, 0}],
 relativeTo[ca1, screen1, {d, 0, 0}]}];
```

In[8]:=

```
ComputePositions[];
```

21.3.5 Orientation Constraints

Constraints

In[9]:=

```
AddOrientationConstraints[{
 orientationOf[slm1, {angle[0], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[slm1, screen1],
 parallelTo[slm1, cd1],
 parallelTo[slm1, t1],
 parallelTo[slm1, ca1]
 }
 ];
```

In[11]:=

```
ComputeOrientations[];
```

21.3.6 Layout

In[14]:=

```
ShowArchitecture[];
```

21.4 Conclusion

In this Chapter the complete optical memory architecture was presented. It used three previously described architectures (presented in Chapters 18 through Chapter 20) as building blocks. These had already added to the component library. The $\mathcal{OHDL}$ description of the optical memory architecture was presented. Results were shown as a 3D-layout.

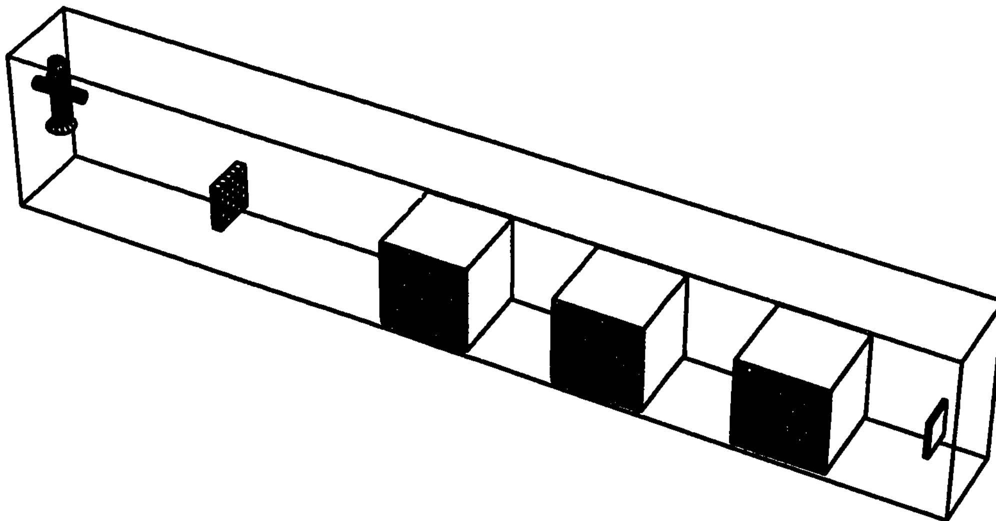


Figure 21.2: 3D-Layout of Optical Memory Architecture

Chapter 22

Optical Vector Operation

Chapter Abstract

The optical vector operation architecture implements S-SEED based optical logic gates. These gates are capable of performing optical logic functions such as NOR, OR, NAND and AND [47]. They can be used to implement complex switching building blocks such as 2×1 and 2×2 switching nodes. The inputs to these gates are differential to avoid any critical biasing of the device. The complete architecture was assembled using only off-the-shelf catalog components.

22.1 Introduction

This subsystem is a vector operator. It is generic optical architecture that can achieve different logic operations such as AND and OR. It is a free-space architecture that makes use of all-optical components. This architecture has many applications involving logic operations on vectors.

Section 22.2.1 discusses an approach to realize the vector operator. Details of the

working are presented in Section 22.2.3

22.2 Optical Implementation

22.2.1 Approach

Input:

Signal vector, control vector and clock.

Output:

Output vector.

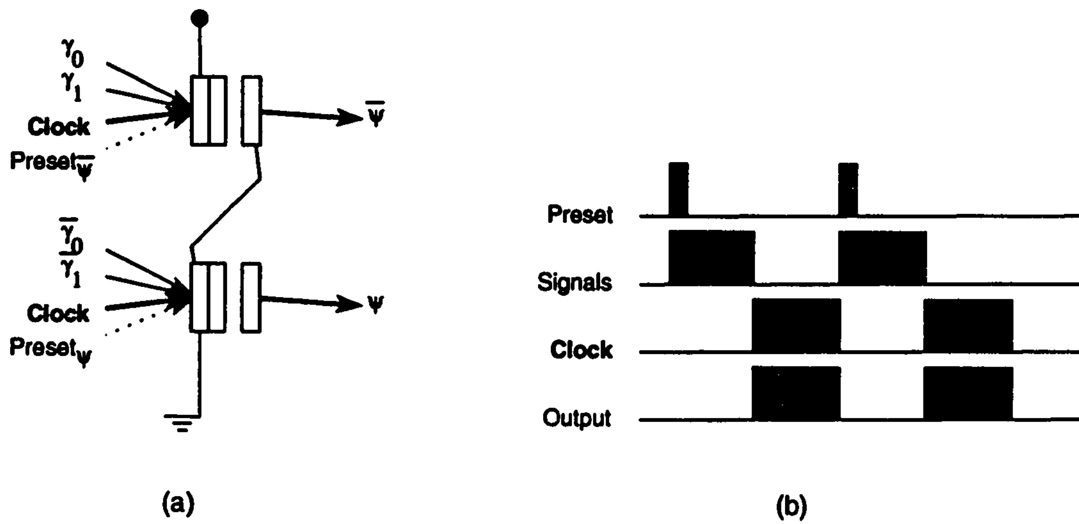
Function:

Realizes an operation (e.g. AND, OR) on a vector.

An approach for achieving logic gate operation is shown in Figure 22.1.

When the incident power to the upper (lower) S-SEED in Figure 22.1 is greater than that of the lower (upper) S-SEED, a logic "1" ("0") will be present on the input. For the non-inverting gates (OR and AND), the output logic level can be represented by the power of the signal coming from the output relative to the power of the signal coming from the inverted output. Similarly, when the power of the signal leaving the output is greater than the power of the signal leaving the inverted output, a logic "1" is represented on the output.

To achieve AND operation, the device is initially set to its "off" state (logic "0"). If both input signals have logic levels of "1" (SET = 1 and RESET = 0), then the S-SEED AND gate is set to its "on" state. For any other input combination, there is no change of state, resulting in AND operation.



γ_0	γ_1	$\overline{\gamma}_0$	$\overline{\gamma}_1$	Preset_{ψ}	$\text{Preset}_{\overline{\psi}}$	ψ		$\overline{\psi}$	
0	0	1	1	0	1	0	AND	1	NAND
0	1	1	0	0	1	0		1	
1	0	0	1	0	1	0		1	
1	1	0	0	0	1	1		0	
0	0	1	1	1	0	0	OR	1	NOR
0	1	1	0	1	0	1		0	
1	0	0	1	1	0	1		0	
1	1	0	0	1	0	1		0	

(c)

Figure 22.1: Logic Using S-SEED Devices.

For NAND operation, the logic level is represented by the power of the inverted output signal relative to the power of the output signal, i.e., when the power of the signal leaving the inverted output is greater than the power of the signal leaving the output, a logic "1" is present on the output.

The operation of the OR and NOR gates is identical to the AND and NAND gates, except that *preset* pulse is used instead of the inverted preset pulse (see Figure 22.1). Thus a single array of devices can perform any or all of the four logic functions and memory functions with the proper optical interconnections and preset pulse routing.

22.2.2 Internal parameters and Constraints

Number of elements in the vector n and the operation to be performed, e.g. AND and OR.

22.2.3 Operation

All the components are picked up from the available catalogs (off-the-shelf). Figure 22.2 shows the layout of the components required for this subsystem. The preset signals, $Enable_0$ and $Enable_1$ are triggered to set the S-SEED into its proper state.

To achieve AND operation, the device is initially set to its "off" state (logic "0"). If both input signals have logic levels of "1" (SET = 1 and RESET = 0), then the S-SEED AND gate is set to its "on" state. For any other input combination, there is no change of state, resulting in AND operation.

Figure 22.2 shows a schematic of the vector operation. Prior to receiving any input signals the preset signals for the S-SEED array ($Enable_0$, $Enable_1$) are triggered to set the S-SEEDs into their proper state prior to the reception of any valid input

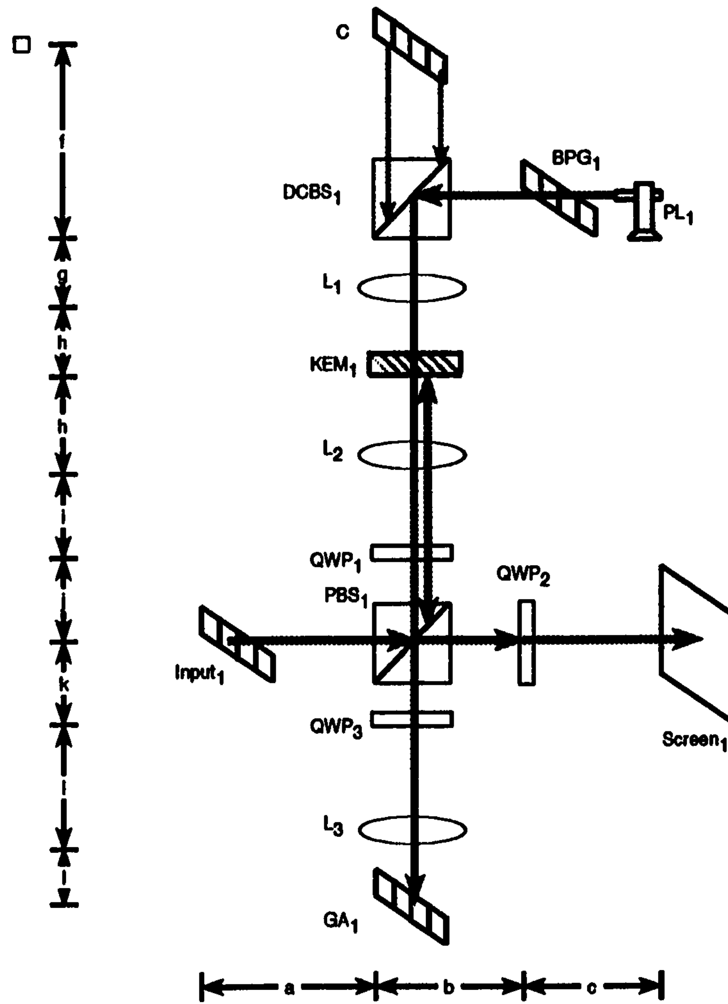


Figure 22.2: The Vector Operation.

signals. These preset signals are encoded on a 780-nm light source, passed through the dichronic beam-splitter ($DCBS_1$) through the knife-edge mirror (KE_1), through the polarization beam-splitter (PBS_1), and then imaged (infinite conjugate imaging) onto the appropriate optical windows of the S-SEED arrays. The quarter-wave plates in this system are aligned with the fast axis at 45° with respect to the PBS plane of incidence.

The input signals ($Input_1$, $Input_2$) then enter the system through PBS_1 where they are reflected up, passed through a quarter-wave plate and imaged onto KE_1 made of evaporated palladium on 2-mm glass substrates. These signals are then reflected, pass a second time through the quarter wave plate to complete the rotation of the polarization of the input signals to allow them to pass through PBS_1 and be imaged on the first S-SEED array.

The reflected signals leaving the S-SEED arrays (modulated clock signals) are directed towards the output plane via PBS_1.

22.3 OHDL Description of the Architecture

22.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

22.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "Vector Op of Systeml of AT&T Switching Fabrics";
Architect = "";
```

22.3.3 External Parameters

In[4]:=

```
n = 4; op = And; lambda = 850;
```

22.3.4 Components instantiation

Components instantiation

In[5]:=

```
dcbs1 = GetObject["DichroicBeamSplitter", CatalogNumber -> "DCBS1"];

bpg1 = GetObject["BinaryPhaseGrating", CatalogNumber -> "BPG1"];

{l1, l2, l3} = GetObjects[3, "ConvexLens", CatalogNumber -> "CL1"];

kem1 = GetObject["KnifeEdgeMirror", CatalogNumber -> "KEM1"];

{qwp1, qwp2, qwp3} = GetObjects[3, "QuarterWavePlate",  CatalogNumber ->
"QWP1"];

ga1 = GetObject["GateArray",  CatalogNumber -> "GA1", Operation -> op];

pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

pl1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

control = GetObject["SLM", CatalogNumber -> "SLM1"];

output = GetObject["SLM", CatalogNumber -> "SLM1"];

input = GetObject["SLM", CatalogNumber -> "SLM1"];
```

22.3.5 Placement Constraints

Parameters

In[13]:=

```
a = 0.1; b = 0.1; c = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
k = 0.1; l = 0.1; j = 0.1; i = 0.1;
```

Constraints

In[14]:=

```
AddPlacementConstraint[default[dcbs1]];

AddPlacementConstraints[{
 relativeTo[dcbs1, l1, {0, -g, 0}],
 relativeTo[l1, kem1, {0, -h, 0}],
 relativeTo[kem1, l2, {0, -h, 0}],
 relativeTo[l2, qwp1, {0, -i, 0}],
 relativeTo[qwp1, pbs1, {0, -j, 0}],
 relativeTo[pbs1, qwp3, {0, -k, 0}],
 relativeTo[qwp3, l3, {0, -l, 0}],
 relativeTo[l3, ga1, {0, -l, 0}],
 relativeTo[pbs1, input, {-a, 0, 0}],
 relativeTo[pbs1, qwp2, {b, 0, 0}],
 relativeTo[qwp2, output, {c, 0, 0}],
 relativeTo[dcbs1, bpg1, {b, 0, 0}],
 relativeTo[bpg1, pl1, {e, 0, 0}],
 relativeTo[dcbs1, control, {0, f, 0}]
}];
```

In[14]:=

```
ComputePositions[];
```

22.3.6 Orientation Constraints

Constraints

In[15]:=

```
AddOrientationConstraints[{
 orientationOf[pl1, {angle[180 Degree], angle[0 Degree], angle[0]}],
 orientationOf[ga1, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}]}];
```

In[15]:=

```
AddOrientationConstraints[{
 parallelTo[input, qwp2],
 parallelTo[input, output],
 parallelTo[input, bpg1],
 parallelTo[ga1, l3],
 parallelTo[ga1, qwp3],
 parallelTo[ga1, qwp1],
 parallelTo[ga1, l2],
 parallelTo[ga1, kem1],
 parallelTo[ga1, l1],
 parallelTo[ga1, control]}
 ];
```

In[16]:=

```
ComputeOrientations[];
```

22.3.7 Layout

In[17]:=

```
ShowArchitecture[];
```

22.4 Conclusion

This Chapter described a generic architecture that can be adapted to perform logic operations on vectors. This will be later used in higher level architectures as a sub-system. The architecture was described in *OHDL* of the *OptiCAD* system. Results

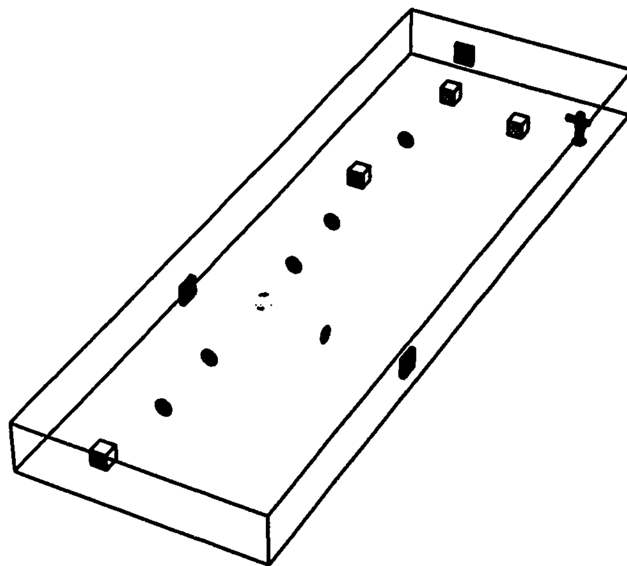


Figure 22.3: 3D-Layout of Optical Vector Operation

were summarized as a 3D-layout.

Chapter 23

The Shifter Module

Chapter Abstract

The shifter is a general purpose device with many applications. One application is the implementation of interconnections – straight and shifted. The input to the shifter is a pair of light beam signals. The output is pairs of straight, connected light beams and crossed light beams. This is the basic building block in many interconnection networks. It is constructed using off-the-shelf optical components.

23.1 Introduction

The shifter is an all-optical free-space architecture. It takes as input a vector and shifts its elements so as to achieve the straight and crossed connections. These are useful in switches for interconnection networks.

Section 23.2.1 discusses one approach for the realization of the shifter using only optical components. Section 23.2.2 describes the working of the shifter. The architecture is described in $\mathcal{OHDL}$ to verify its design.

23.2 Optical Implementation

23.2.1 Approach

Input:

Vector i.

Output:

Vector o which is a shifted copy of i.

Function:

Shifts the elements of the input vector.

One approach for realizing the shifting operation is by splitting the input beams into two components. This can be done using a polarizing beam-splitter if the inputs are appropriately polarized. One component is allowed to go straight (realizing the straight interconnection) and the other is tilted by a carefully oriented mirror which achieves the shifted interconnection.

23.2.2 Operation

Figure 23.1 shows the schematic of the shifter. One of the polarization components of the input is directed to M_2 to provide the straight connection to the output, while the other polarization component is directed to the tilted mirror M_3 which provides the shifted interconnect.

The amount of shift depends on the tilt of the mirror, M_3, and the distance between M_3 and beam-splitter PBS_1.

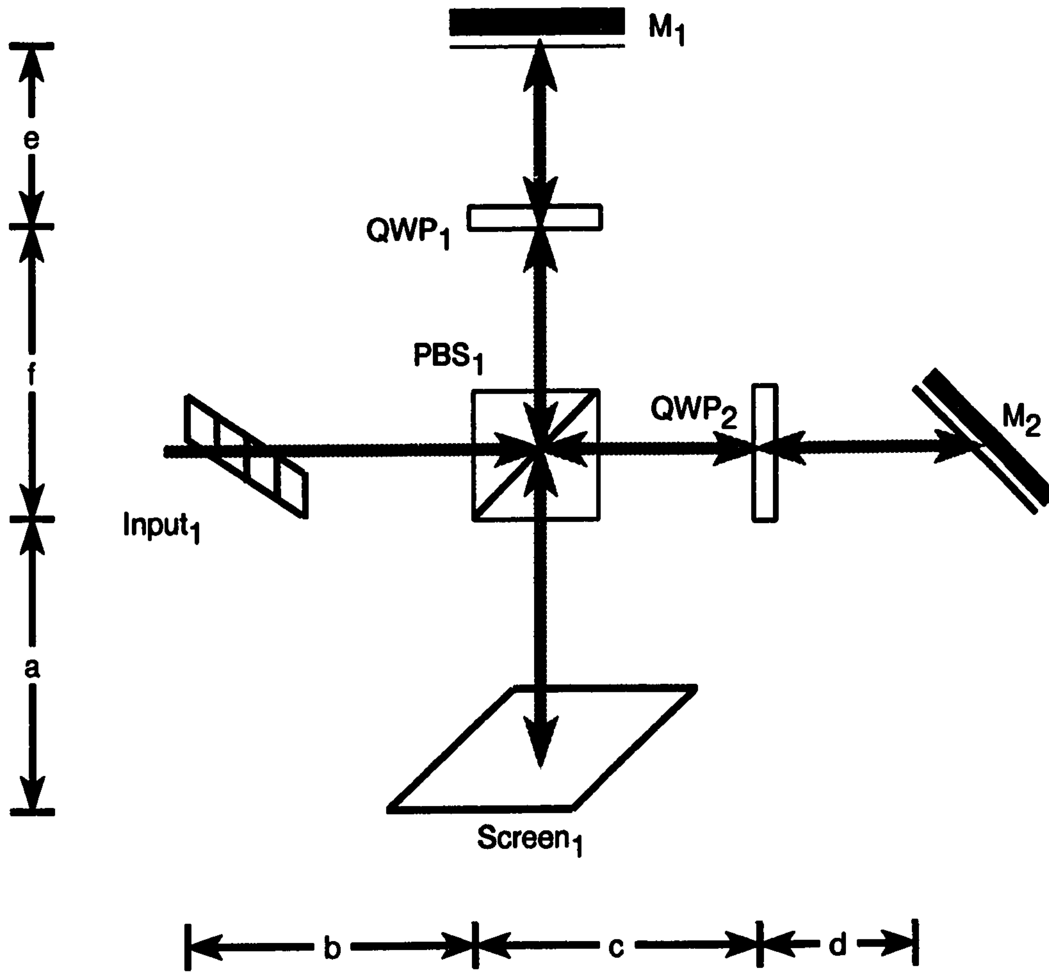


Figure 23.1: The Shifter Module.

23.3 OHDL Description of the Architecture

23.3.1 Initialization of the Simulator

In[1]:=
```
initialize[];
```

23.3.2 Architecture Information

In[2]:=
```
ArchitectureName = "Shifter of System1 of AT&T Switching Fabrics";
Architect = "";
```

23.3.3 Components instantiation

Components instantiation

In[4]:=
```
pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

{m1, m2} = GetObjects[2, "Mirror", CatalogNumber -> "M1"];

{qwp1, qwp2} = GetObjects[2, "QuarterWavePlate", CatalogNumber -> "QWP1"];

input = GetObject["SLM", CatalogNumber -> "SLM1"];

output = GetObject["SLM", CatalogNumber -> "SLM1"];
```

23.3.4 Placement Constraints

Parameters

In[5]:=
```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1;
```

In[10]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, qwp1, {-e, 0, 0}],
 relativeTo[pbs1, output, {a, 0, 0}],
 relativeTo[pbs1, input, {0, -b, 0}],
 relativeTo[pbs1, qwp2, {0, c, 0}],
 relativeTo[pbs1, m2, {0, c + d, 0}],
 relativeTo[pbs1, m1, {- e - f, 0, 0}]
 }];
```

Constraints

In[12]:=

```
ComputePositions[];
```

23.3.5 Orientation Constraints

In[14]:=

```
AddOrientationConstraints[{
 orientationOf[qwp2, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[m2, {angle[120 Degree], angle[90 Degree], angle[0]}],
 orientationOf[m1, {angle[0], angle[90 Degree], angle[0]}]}
 ];
```

In[15]:=

```
AddOrientationConstraints[{
 parallelTo[input, qwp2],
 parallelTo[m1, qwp1],
 parallelTo[m1, output]}
 ];
```

Constraints

In[16]:=

```
ComputeOrientations[];
```

23.3.6 Layout

In[17]:=
```
ShowArchitecture[];
```

23.4 Conclusion

In this Chapter, a shifter module was described using *OHDL* in *OptiCAD*. This module will later be used as a component in a higher level architecture. The results of the description were obtained as a 3D-layout.

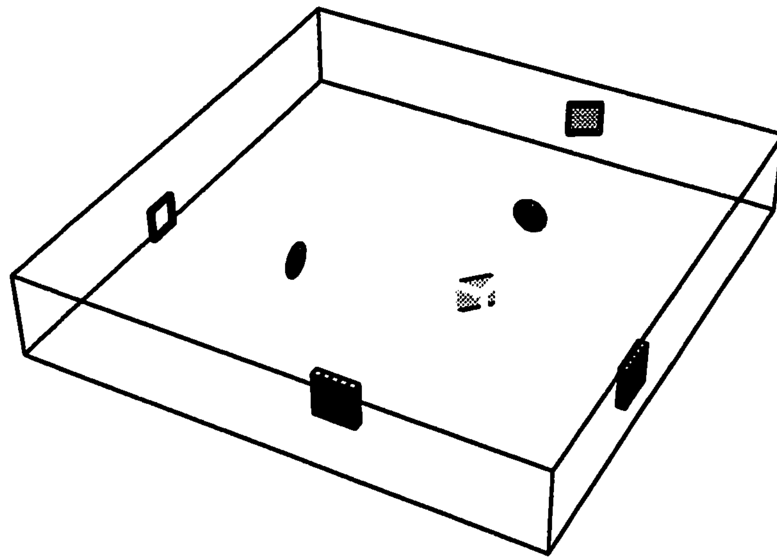


Figure 23.2: 3D-Layout of The Shifter Module

Chapter 24

$System_1$ of AT&T Switching Fabrics

Chapter Abstract

$System_1$ is the first of a series of five switching fabric demonstrators constructed at AT&T. These systems demonstrate two-dimensional arrays of optical and optoelectronic devices which employ free-space interconnections. These demonstrators use S-SEED technology as the device platform. $System_1$ is the S-SEED based switching demonstrator.

24.1 Introduction

$System_1$ is assembled from two instances of the vector operation subsystem (presented in Chapter 21), one for the ANDing and the other one for ORing. It connects these two using the shifter module.

24.2 Optical Implementation

24.2.1 Approach

Input:

Two inputs – $Input_0$ and $Input_1$.

Output:

One output – *Output*.

Function:

$System_1$ is a (2, 1, 1) node.

The purpose of this system was to create a (2, 1, 1) node. The approach was to interconnect two S-SEED arrays that would be used as optical logic gates. The S-SEEDs in the first array need to provide the functionality of digital AND gates while the second S-SEED array must function as OR gates.

24.2.2 Operation

The S-SEED based switching demonstrator using the logic gates described earlier, was to interconnect two S-SEED arrays and create a switching node prototype [42] as shown in Figure 24.1.

To realize this node, the S-SEED arrays were used as optical logic gates. In general, the S-SEEDs in the first array provide the digital AND gate operation while the second S-SEED array must function as an OR gate as shown in Figure 24.1.

$System_1$ is composed of three subsystems. The first subsystem is the vector-operator performing the basic AND operation. The second subsystem is the optical-

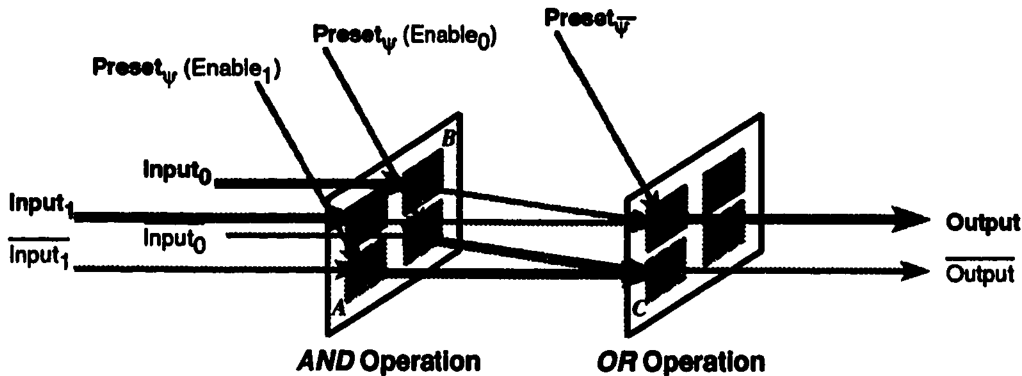


Figure 24.1: Schematic of S-SEED Implementation.

shifter which realizes the crossover interconnects. The third subsystem is the vector-operator performing the OR operation. These subsystems are connected as shown in Figure 24.2.

24.3 OHDL Description of the Architecture

24.3.1 Initialization of the Simulator

In[1]:=
```
initialize[];
```

24.3.2 Architecture Information

In[2]:=
```
ArchitectureName = "System1 of AT&T Switching Fabrics";
Architect = "";
```

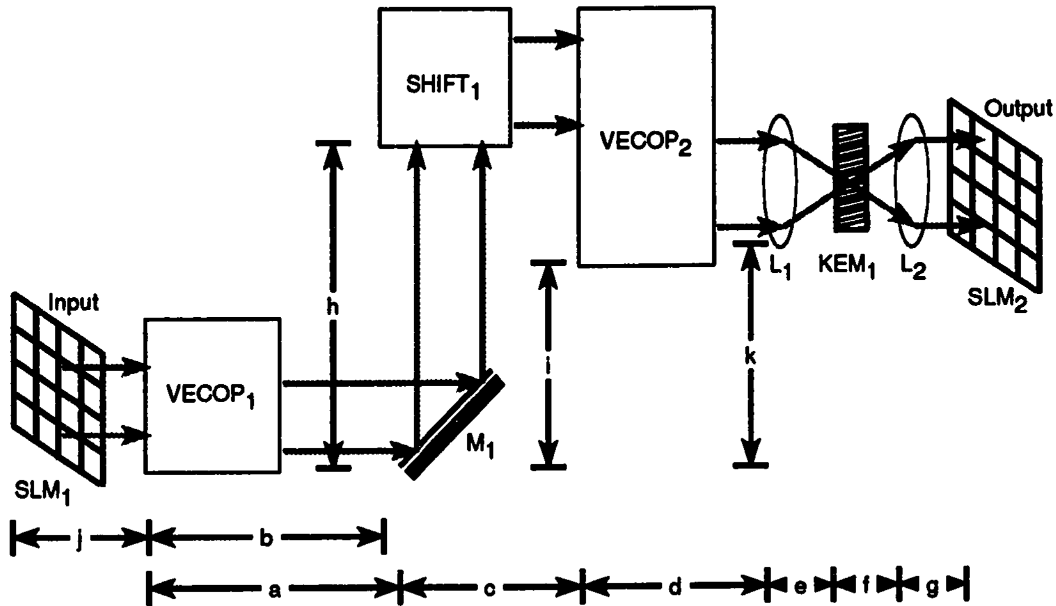


Figure 24.2: The Complete Layout of $System_1$.

24.3.3 Components instantiation

Components instantiation

In[4]:=

```
{vecop1, vecop2} = GetObjects[2, "Assembly", CatalogNumber -> "A1"];

shift1 = GetObject["Assembly", CatalogNumber -> "A1"];

{input, output} = GetObjects[2, "SLM", CatalogNumber -> "SLM1"];

{l1, l2} = GetObjects[2, "ConvexLens", CatalogNumber -> "CL1"];

m1 = GetObject["Mirror", catalogNumber -> "M1"];

kem1 = GetObject["KnifeEdgeMirror", CatalogNumber -> "KEM1"];
```

24.3.4 Placement Constraints

Parameters

In[7]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1;
 f = 0.1; g = 0.1; h = 0.1; i = 0.1; j = 0.1; k = 0.1;
```

Constraints

In[11]:=

```
AddPlacementConstraint[default[input]];

 AddPlacementConstraints[{
 relativeTo[input, vecop1, {j, 0, 0}],
 relativeTo[vecop1, m1, {a, 0, 0}],
 relativeTo[vecop1, shift1, {b, h, 0}],
 relativeTo[m1, vecop2, {c, i, 0}],
 relativeTo[m1, l1, {c + d, k, 0}],
 relativeTo[l1, kem1, {e, 0, 0}],
 relativeTo[kem1, l2, {f, 0, 0}],
 relativeTo[l2, output, {g, 0, 0}]
 }];
```

In[13]:=

```
ComputePositions[];
```

24.3.5 Orientation Constraints

In[16]:=

```
AddOrientationConstraints[{
 orientationOf[m1, {angle[45 Degree], angle[90 Degree], angle[0]}],
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}]}
 ];
```

In[17]:=

```
AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[input, l2],
 parallelTo[input, kem1],
 parallelTo[input, output]}
 ];
```

Constraints

In[18]:=
```
ComputeOrientations[];
```

24.3.6 Layout

In[19]:=
```
ShowArchitecture[];
```

24.4 Conclusion

This Chapter presented the complete $System_1$ of the AT&T switching fabrics. It realizes a (2,1,1) node by using two instances of the vector operator and one instance of the shifter. The system was described in $\mathcal{OHDL}$ and results were obtained in a 3D-layout.

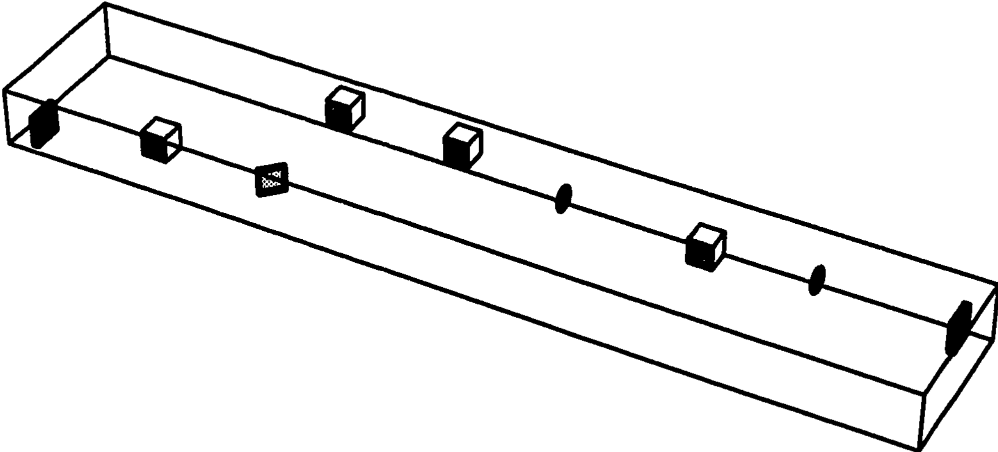


Figure 24.3: 3D-Layout of $System_1$ of AT&T Switching Fabrics

Chapter 25

Beam Combiner Unit

Chapter Abstract

In this chapter, we discuss the beam-combiner. Its basic functionality is to produce the combined image of the two super-imposed input images. It regenerates the desired signals out of clock to restore intensity and sharpness.

25.1 Introduction

One of the major problems in employing free-space optics is the beam-combination. If a simple beam-splitter/merger is used, then the beam undergoes big intensity losses. Moreover, the combination is not uniform because of the nature of the merging. This Chapter proposes a beam-merger [31] that achieves a beam-combination of several different images.

Section 25.2.1 describes the approach employed and Section 25.2.2 outlines the operation. The architecture is described in $\mathcal{OHDL}$ of the $Opti\mathcal{CAD}$ system. The

results are summarized as a 3D-layout.

25.2 Optical Implementation

25.2.1 Approach

Input:

Input Matrix (Image encoding: Each pixel is a combination of two linearly polarized signals, i.e., superimposed input images).

Output:

Output Matrix (linearly polarized, regenerated output image which is a "combined" image of inputs as per the functionality of SEED array).

Function:

Produces a combined image of the two superimposed input images. Preferably regenerates the desired signal out of clock to restore intensity and sharpness.

Separate the two input beams and feed them as inputs to SEED elements so that the combined signal can modulate the clock to produce the output.

25.2.2 Operation

For this operation, the free-space photonic switching fabrics, based on S-SEED arrays require that each device must be able to receive two input signals plus a clock [38, 65]. In addition, the reflected output signal must be directed from the device to the next stage of the switching fabric. The major constraints of this problem are that the

spots must be small ($< 5\mu m$). This often requires the entering signals to use the full aperture of an imaging lens, and that the signals must not interfere at the device's optical window.

An example of a beam combination system using image division (space multiplexing) is illustrated in Figure 25.1.

This beam-combiner is composed of a polarizing beam-splitter (PBS_1) that is surrounded on three sides by quarter-wave plates (QWP_1, QWP_2 ad QWP_3). The four lenses (L_1-L_4) are used in the infinite conjugate mode. At the bottom of the beam-combiner is the S-SEED array. The input image is composed of an array with each of the two input signals being associated with one of the two linear polarizations. As an input image enters the beam-combination system on the left, the perpendicular component is directed upward while the parallel component passes through the beam-splitter. Both components travel through quarter-wave plates where the linear polarization is converted to circular. The light travelling upwards is imaged onto patterned mirrors of mask M_1. This mask consists of an array of small mirrors that are located in the image plane of L_2 such that light reflected from them is imaged onto the upper half of the rectangular window of the S-SEED array. Therefore, the image travelling upwards reflects off M_1, passes again through QWP_1 which changes the polarization from circular to parallel allowing the light to be passed through PBS_1 onto the upper half of the S-SEED array.

The parallel component of the input image passes through the polarization beam-splitter, is imaged onto the small patterned mirrors of mask M_2, reflected by PBS_1, and then imaged onto the upper half of the S-SEED array. Since these two inputs have orthogonal circular polarizations, they do not interfere.

The optical clock enters the beam-combination unit by being imaged onto the

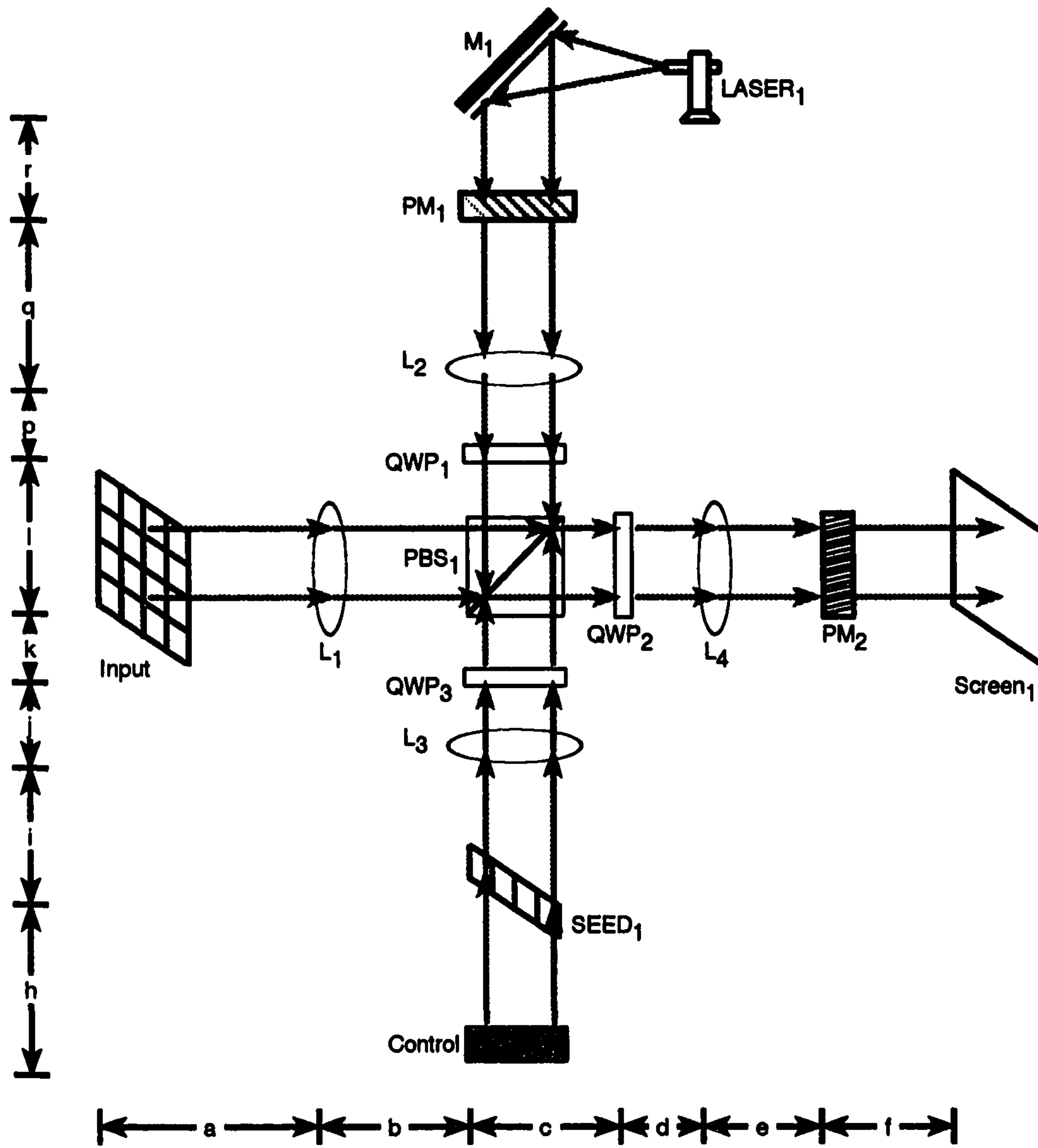


Figure 25.1: Image Division Beam-Combination Unit.

transparent region below the small mirrors of mask M_1. This array of high-power spots is imaged onto the bottom half of the S-SEED array. The reflected clock signal, which then becomes the output signal of the S-SEED array, will have its polarization rotated allowing it to be reflected by the polarization beam-splitter and then imaged onto the transparent region below the small mirrors of mask M_2. This output image can then be collected and used as an optical interconnect or another beam-combination unit.

25.3 OHDL Description of the Architecture

25.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

25.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System2: Beam Combiner";
Architect = "";
```

25.3.3 Components instantiation

Components instantiation

In[4]:=

```
{l1, l2, l3, l4} = GetObjects[4, "PlanoConvexLens", CatalogNumber -> "PCL1"];

{qwp1, qwp2, qwp3} = GetObjects[3, "QuarterWavePlate", CatalogNumber ->
"QWP1"];

{pm1, pm2} = GetObjects[2, "PatterenedMirror", CatalogNumber -> "PM1"];

pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

laser1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

{input, control} = GetObjects[2, "SLM", CatalogNumber -> "SLM1"];

s1 = GetObject["Screen", CatalogNumber -> "S1"];

seed1 = GetObject["SEED", CatalogNumber -> "SEED1"];
```

25.3.4 Placement Constraints

Parameters

In[7]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1;
 g = 0.1; h = 0.1; i = 0.1; j = 0.1; k = 0.1; l = 0.1;
 p = 0.1; q = 0.1; r = 0.1; t = 0.01;
```

Constraints

In[10]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, l1, {-b, 0, 0}],
 relativeTo[l1, input, {-a, 0, 0}],
 relativeTo[pbs1, qwp2, {c, 0, 0}],
 relativeTo[qwp2, l4, {d, 0, 0}],
 relativeTo[l4, pm2, {e, 0, 0}],
 relativeTo[pm2, s1, {f, 0, 0}],
 relativeTo[pbs1, qwp1, {0, 1, 0}],
 relativeTo[qwp1, l2, {0, p, 0}],
 relativeTo[l2, pm1, {0, q, 0}],
 relativeTo[pm1, laser1, {t, r, 0}],
 relativeTo[pbs1, qwp3, {0, -k, 0}],
 relativeTo[qwp3, l3, {0, -j, 0}],
 relativeTo[l3, seed1, {0, -i, 0}],
 relativeTo[seed1, control, {0, -h, 0}]}
 ];
```

In[16]:=

```
ComputePositions[];
```

25.3.5 Orientation Constraints

Constraints

In[18]:=
```
AddOrientationConstraints[{
 orientationOf[laser1, {angle[-90 Degree], angle[0], angle[0]}],
 orientationOf[l1, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[l2, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];
 AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[qwp2, l1],
 parallelTo[l4, l1],
 parallelTo[pm2, l1],
 parallelTo[s1, l1],
 parallelTo[pm1, l2],
 parallelTo[qwp1, l2],
 parallelTo[qwp3, l2],
 parallelTo[l3, l2],
 parallelTo[seed1, l2],
 parallelTo[control, l2]}
 ];
```

In[20]:=
```
ComputeOrientations[];
```

25.3.6 Layout

In[21]:=
```
ShowArchitecture[];
```

25.4 Conclusion

In this Chapter a beam combination architecture was discussed and described in $\mathcal{OHDL}$. Results were summarized as a 3D-layout. This architecture will later be used in building a higher level architecture.

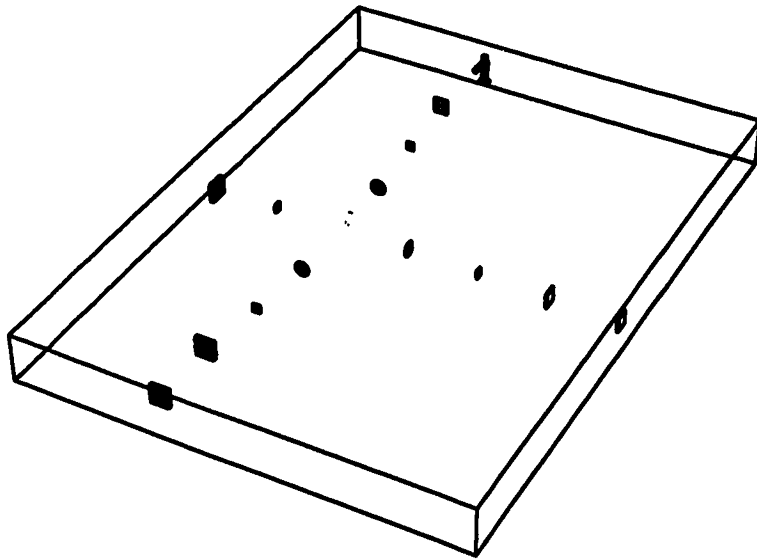


Figure 25.2: 3D-Layout of Beam Combiner Unit

Chapter 26

Crossover Interconnection Network

Chapter Abstract

This subsystem is to design a general optical multistage interconnection network (MIN). To create such a MIN requires a crossover interconnect between consecutive stages [38].

26.1 Introduction

Numerous architectures employ interconnection networks. The crossover interconnect provides two types of connections – straight and exchange. This Chapter discusses one optical implementation of this interconnect.

Section 26.2.1 outlines the approach employed in realizing the architecture. Section 26.2.2 explains its operation. A description of the architecture is provided in

$\mathcal{OHDL}$.

26.2 Optical Implementation

26.2.1 Approach

Input:

Input Vector

Output:

Output Vector

Function:

Splits the output into two components (a straight through and/or crossed) and connects them. The approach is to employ a polarizing beam-splitter (PBS) to split the beam based on its type of polarization, a reflector for straight through connection (in one path), and a reflector grating of desired period for crossover connection in the other path.

An optical retro-reflector grating is used to achieve the optical crossover interconnects. The retro-reflector grating is a ruled aluminum prismatic mirror array. The input image is split into two components. The use of the retro-reflector allows the delay and the shift of one component of the input before reflecting it. The other component is directly reflected allowing both components the straight-through and crossed interconnects to be achieved.

26.2.2 Operation

Figure 26.1 illustrates the operation of the crossover interconnect.

An input image enters the system from the left. The light associated with each pixel or spot is circularly polarized. The perpendicular component of the input light will be imaged onto the mirror (M_1). The reflected light from M_1 will have its polarization rotated by QWP_1, allowing it to pass through the polarizing beam-splitter (PBS_1) where it can be imaged onto the output image plane.

The parallel component of the input light passes through PBS_1 and is imaged onto the retro-reflector grating . The light incident on one mirror of each retro-reflector grating is reflected to the opposing mirror where it will be redirected back to PBS_1. This retro-reflection process shifts the position of each spot imaged onto the grating. This shifting implements the crossover patterns of the crossover interconnection topology. The light leaving the retro-reflector grating will have its polarization rotated by QWP_2 allowing it to be redirected and focused onto the output image plane. Thus, both the straight-through and crossover interconnects have been achieved. The amount of shift in space is related to the period of the retro-reflector grating.

26.3 OHDL Description of the Architecture

26.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

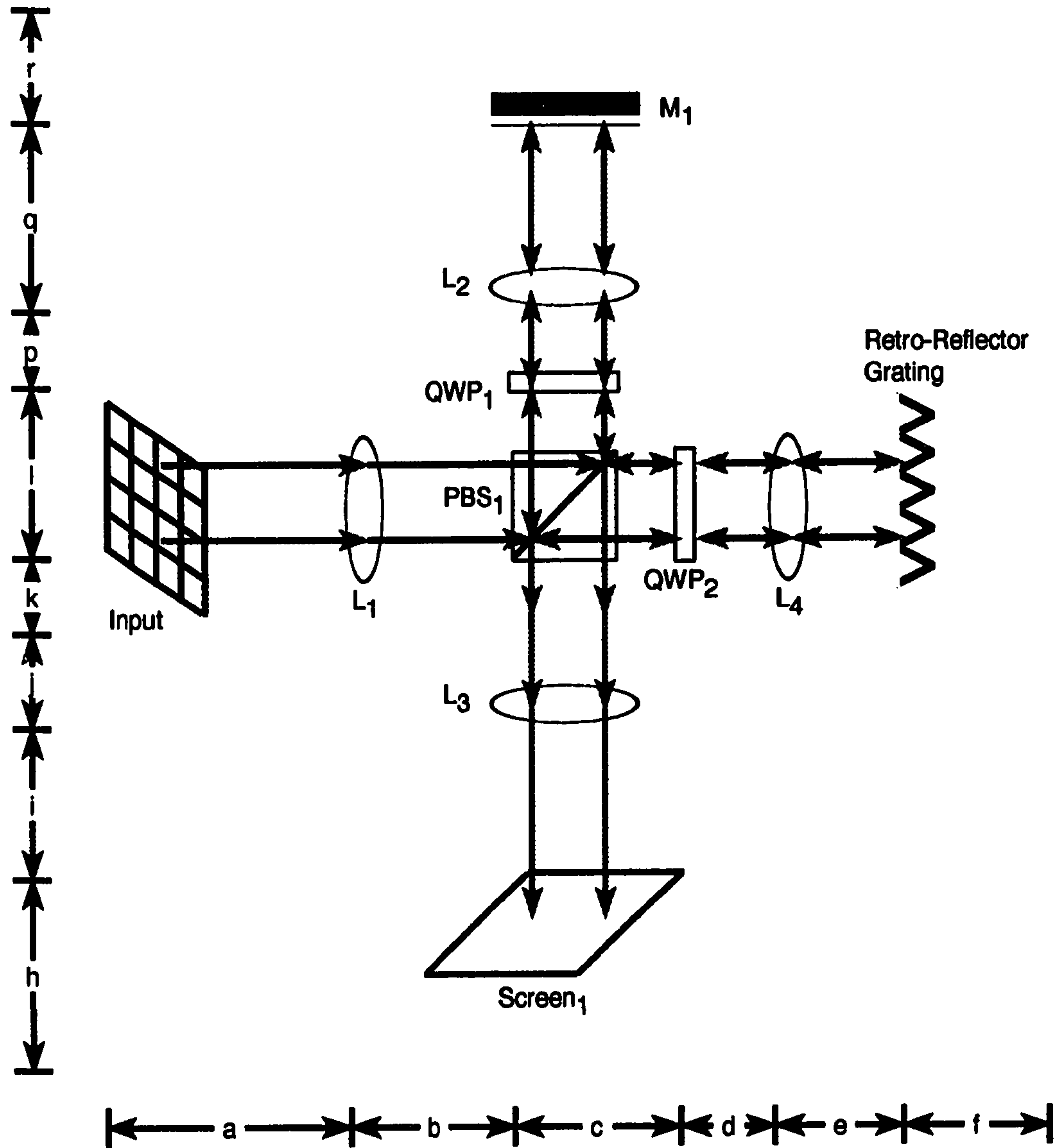


Figure 26.1: The Optical Hardware of the Crossover Interconnect.

26.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System2: CrossOver Interconnection Network";
Architect = "";
```

26.3.3 Components instantiation

Components instantiation

In[3]:=

```
{l1, l2, l3, l4} = GetObjects[4, "PlanoConvexLens", CatalogNumber -> "PCL1"];

{qwp1, qwp2} = GetObjects[2, "QuarterWavePlate", CatalogNumber -> "QWP1"];

m1 = GetObject["Mirror", CatalogNumber -> "M1"];

pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

rg1 = GetObject["ReflectorGrating", CatalogNumber -> "RG1"];

input = GetObject["SLM", CatalogNumber -> "SLM1"];

output = GetObject["Screen", CatalogNumber -> "S1"];
```

26.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, l1, {-b, 0, 0}],
 relativeTo[l1, input, {-a, 0, 0}],
 relativeTo[pbs1, qwp2, {c, 0, 0}],
 relativeTo[qwp2, l4, {d, 0, 0}],
 relativeTo[l4, rg1, {e, 0, 0}],
 relativeTo[pbs1, qwp1, {0, c, 0}],
 relativeTo[qwp1, l2, {0, d, 0}],
 relativeTo[l2, m1, {0, h, 0}],
 relativeTo[pbs1, l3, {0, -g, 0}],
 relativeTo[l3, output, {0, -f, 0}]}
 ];
```

In[5]:=

```
ComputePositions[];
```

26.3.5 Orientation Constraints

Constraints

In[6]:=

```
AddOrientationConstraints[{
 orientationOf[l1, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[l2, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];
 AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[qwp2, l1],
 parallelTo[l4, l1],
 parallelTo[rg1, l1],
 parallelTo[qwp1, l2],
 parallelTo[l3, l2],
 parallelTo[output, l2]}
 ];
```

In[6]:=

```
ComputeOrientations[];
```

26.3.6 Layout

In[7]:=

```
ShowArchitecture[];
```

26.4 Conclusion

An optical implementation of the crossover interconnect was discussed. It was described using $\mathcal{OHDL}$. Results were presented as a 3D-layout.

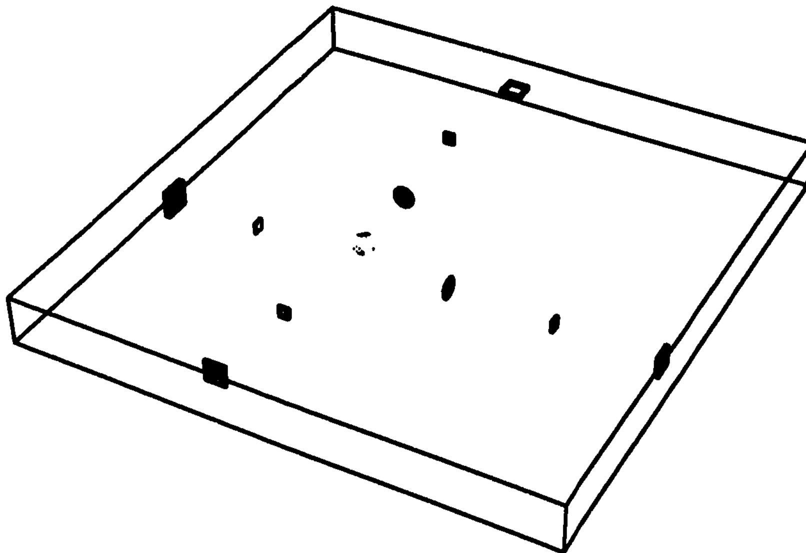


Figure 26.2: 3D-Layout of Crossover Interconnection Network

Chapter 27

Passive Beam Combiner

Chapter Abstract

The passive beam-combiner is actually a beam-combiner with the input laser source replaced by an SLM to provide collimated laser source.

27.1 Introduction

The passive beam combiner is exactly similar to the beam combiner presented in Chapter 24. The input source has been replaced by an SLM to provide a collimated laser source.

Sections 27.2.1 and 27.2.2 sketch the approach and the operation of the passive beam combiner. A description is provided using $\mathcal{OHDL}$.

27.2 Optical Implementation

27.2.1 Approach

Input:

Input image matrix. Each pixel of the image is a combination of linearly polarized signals, i.e., superimposed images.

Output:

Output matrix.

Function:

Produces a combined image of the two super-imposed input images.

The approach followed is similar to that of the beam-combiner.

27.2.2 Operation

The operation of the passive-beam combiner is similar to that of the beam-combiner of Chapter 24. The only difference is that the input is a collimated source after passing through an SLM (see Figure 27.1).

27.3 OHDL Description of the Architecture

27.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

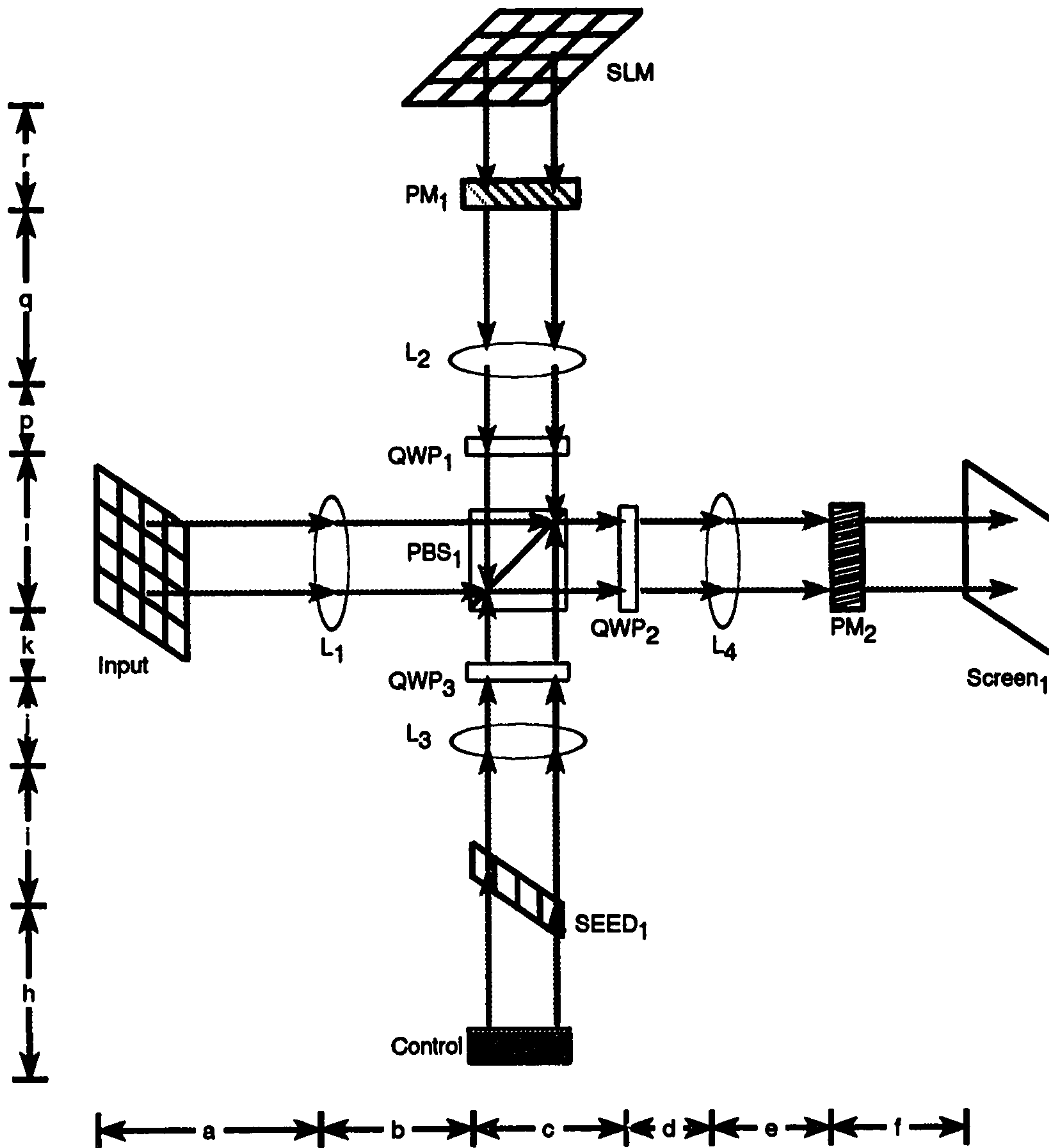


Figure 27.1: The Passive Beam-Combination Unit.

27.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System2: Passive Beam Combiner";
Architect = "";
```

27.3.3 Components instantiation

Components instantiation

In[3]:=

```
{l1, l2, l3, l4} = GetObjects[4, "PlanoConvexLens", CatalogNumber -> "PCL1"];

{qwp1, qwp2, qwp3} = GetObjects[3, "QuarterWavePlate", CatalogNumber ->
"QWP1"];

{pm1, pm2} = GetObjects[2, "PatterenedMirror", CatalogNumber -> "PM1"];

pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

{input, control, clock} = GetObjects[3, "SLM", CatalogNumber -> "SLM1"];

s1 = GetObject["Screen", CatalogNumber -> "S1"];

seed1 = GetObject["SEED", CatalogNumber -> "SEED1"];
```

27.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1; i =
0.1; j = 0.1; k = 0.1; l = 0.1; p = 0.1; q = 0.1; r = 0.1; t = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, l1, {-b, 0, 0}],
 relativeTo[l1, input, {-a, 0, 0}],
 relativeTo[pbs1, qwp2, {c, 0, 0}],
 relativeTo[qwp2, l4, {d, 0, 0}],
 relativeTo[l4, pm2, {e, 0, 0}],
 relativeTo[pm2, s1, {f, 0, 0}],
 relativeTo[pbs1, qwp1, {0, l, 0}],
 relativeTo[qwp1, l2, {0, p, 0}],
 relativeTo[l2, pm1, {0, q, 0}],
 relativeTo[pm1, clock, {0, r, 0}],
 relativeTo[pbs1, qwp3, {0, -k, 0}],
 relativeTo[qwp3, l3, {0, -j, 0}],
 relativeTo[l3, seed1, {0, -i, 0}],
 relativeTo[seed1, control, {0, -h, 0}]}
 ];
```

In[5]:=

```
ComputePositions[];
```

27.3.5 Orientation Constraints

Constraints

In[6]:=

```
AddOrientationConstraints[{
 orientationOf[l1, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[l2, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];
 AddOrientationConstraints[{
 parallelTo[input, l1],
 parallelTo[qwp2, l1],
 parallelTo[l4, l1],
 parallelTo[pm2, l1],
 parallelTo[s1, l1],
 parallelTo[pm1, l2],
 parallelTo[qwp1, l2],
 parallelTo[qwp3, l2],
 parallelTo[clock, l2],
 parallelTo[l3, l2],
 parallelTo[seed1, l2],
 parallelTo[control, l2]}
 ];
```

In[6]:=

```
ComputeOrientations[];
```

27.3.6 Layout

In[7]:=

```
ShowArchitecture[];
```

27.4 Conclusion

In this Chapter the architecture of the passive beam combiner was presented. The $\mathcal{OHDL}$ description was provided and the results were summarized as a 3D-layout.

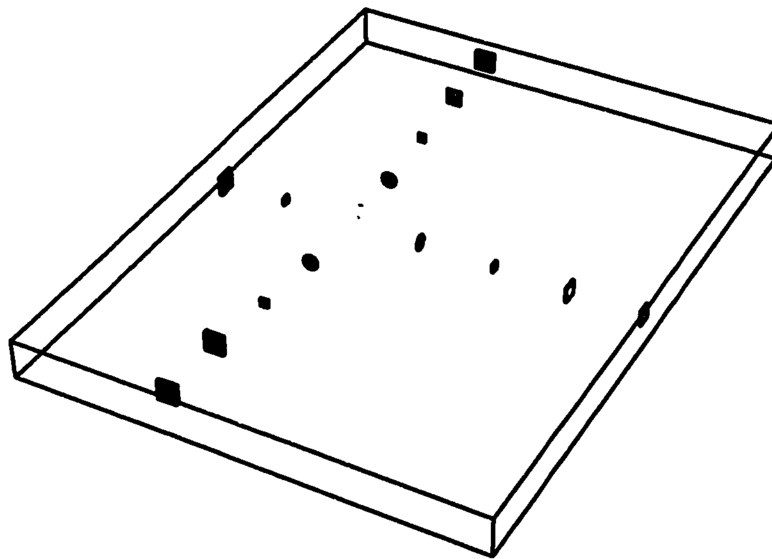


Figure 27.2: 3D-Layout of Passive Beam Combiner

Chapter 28

Spot Array Generator

Chapter Abstract

The two dimensional arrays of SEED technology based switching nodes require an optical power supply to clock the devices [76]. The generation of 2D-arrays of uniform intensity spots requires two basic components. The first is a high power, single frequency, diffraction limited laser that can provide the appropriate power per pixel needed to meet the system speed requirements. The second component is a spot-array generator that requires some mechanism to equally and uniformly divide the power from the laser and distribute it to the optical windows of the S-SEEDs.

28.1 Introduction

Architectures employing SEEDs have the problem of supplying equal power to all the SEED's windows. The spot array generator is one device that can deliver up to 2048 × 2048 equi-power spots of light from a single laser source. The architecture discussed in this Chapter is not that sophisticated. It is at best an initial design.

Section 28.2.1 outlines the approach followed when designing this architecture. Section 28.2.3 describes the operation of the architecture.

28.2 Optical Implementation

28.2.1 Approach

Input:

Collimated Gaussian beam.

Output:

Spot array.

Function:

Dividing the input (high-power, single frequency, diffraction limited) equally and uniformly to distribute it to a matrix of windows in the output plane (S-SEED's). The approach is to use a Fourier plane spot array generation using Binary Phase Gratings (BPG) to uniformly distribute the optical power to the S-SEED's.

There have been several approaches to provide required optical power to clock the S-SEED devices. The approach that has been pursued by AT&T system demonstrators is Fourier plane spot array generation using binary phase grating (BPG). This will uniformly distribute the optical power to the S-SEEDs [55].

28.2.2 Internal parameters and Constraints

The internal parameters of this architecture are (1) frequency of the laser light, (2) spot size, (3) number of spots, (4) characterization of grating, and (5) number of levels.

28.2.3 Operation

The phase gratings used to uniformly distribute the optical power to the S-SEEDs are made by etching glass with a repetitive multi-level pattern. For the case of a binary phase grating, there are two thickness of glass. This grating is illuminated by a plane wave from a laser source as illustrated in Figure 28.1.

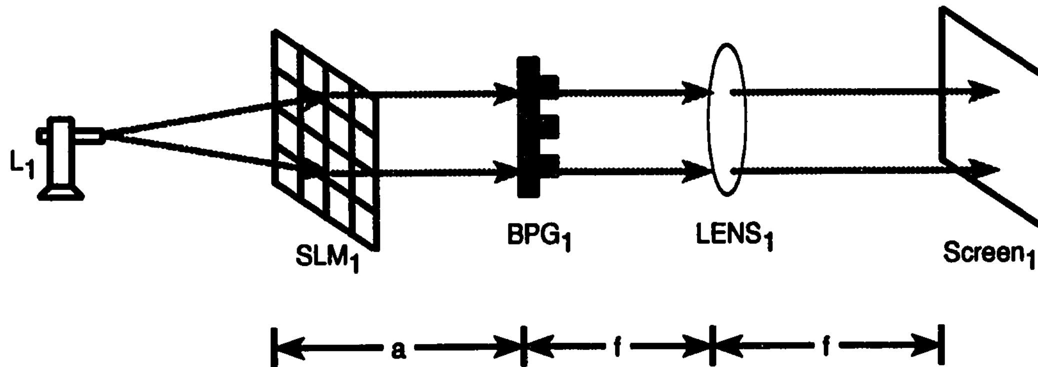


Figure 28.1: Fourier-Plane Spot Array Generation.

The light transmitted through the grating is Fourier transformed at the back focal plane of the lens ($LENS_1$). This focal plane represents the output plane of the spot-array generator and the location of the smart pixel array.

28.3 OHDL Description of the Architecture

28.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

28.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System2: Spot Array Generation";
Architect = "";
```

28.3.3 Components instantiation

Components instantiation

In[3]:=

```
bpg1 = GetObject["BinaryPhaseGrating",
 CatalogNumber -> "BPG1", Size -> {0.02,0.02,0.005}];

l1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

lens1 = GetObject["ConvexLens", CatalogNumber -> "CL1"];

s1 = GetObject["Screen", CatalogNumber -> "S1"];

slm1 = GetObject["SLM", CatalogNumber -> "SLM1"];
```

28.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.001; f = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[bpg1]];

 AddPlacementConstraints[{
 relativeTo[bpg1, slm1, {-a, b, 0}],
 relativeTo[slm1, l1, {-a, b, 0}],
 relativeTo[bpg1, lens1, {f, 0, 0}],
 relativeTo[lens1, s1, {f, 0, 0}]}
 ];
```

In[5]:=

```
ComputePositions[];
```

28.3.5 Orientation Constraints

Constraints

In[6]:=

```
AddOrientationConstraint[
 orientationOf[lens1, {angle[0], angle[90 Degree], angle[0]}]
 ];

 AddOrientationConstraints[{
 parallelTo[bpg1, lens1],
 parallelTo[slm1, lens1],
 parallelTo[s1, lens1]}
 ];
```

In[6]:=

```
ComputeOrientations[];
```

28.3.6 Layout

In[7]:=

```
ShowArchitecture[];
```

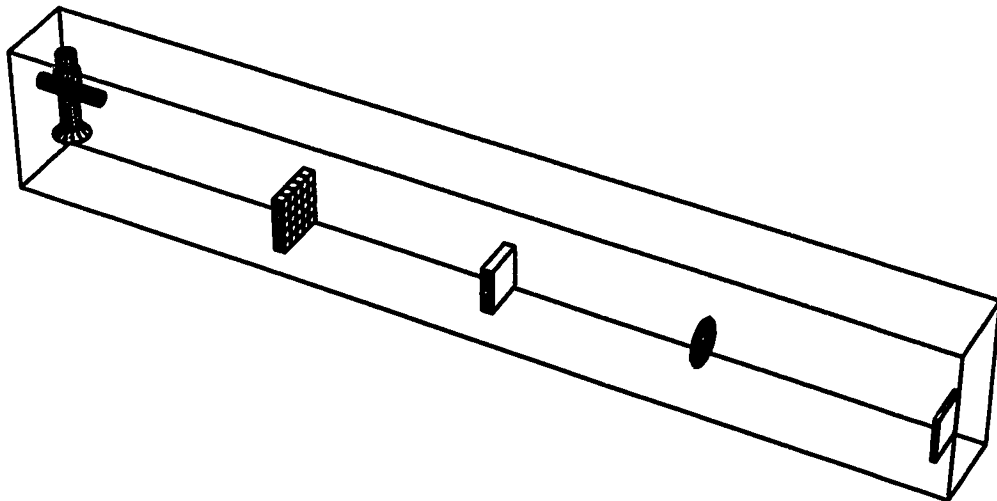


Figure 28.2: 3D-Layout of Spot Array Generator

28.4 Conclusion

This Chapter presented a preliminary version of the spot array generator. It provides a simple solution that is useful for small architectures. The $\mathcal{OHDL}$ description was provided that produced a 3D-layout of the architecture.

Chapter 29

$System_2$ of AT&T Switching Fabrics

Chapter Abstract

This Chapter describes $System_2$ of the AT&T switching fabrics [31]. It realizes a four-stage interconnection network.

29.1 Introduction

$System_2$ realizes a four-stage interconnection network. It is divided into submodules each of which has been described in previous Chapters (Chapters 25-28).

Section 29.2.1 outlines the approach taken for the realization of this architecture. Section 29.2.2 discusses the operation of the complete $System_2$.

29.2 Optical Implementation

29.2.1 Approach

Input:

Input matrix - an image.

Output:

Screen.

Function:

One stage of a cross-over MIN.

The approach is to use existing components that were previously designed. These include the crossover interconnection network (Chapter 26), passive beam-combiner (Chapter 27), and the spot-array generator (Chapter 28). Some additional components such as mirrors and lenses are used to guide the light through the various subsystems.

29.2.2 Operation

Figure 29.1 shows the schematic of $System_2$ of the AT&T switching fabrics. It is constructed from smaller sub-architectures. The clock is generated by the spot-array generator which is given directly as one of the inputs to the passive beam-combiner. This also receives the input that has passed through the crossover interconnect to achieve the proper connectivity. The output of the passive beam-combiner is collected at the screen ($SCREEN_1$).

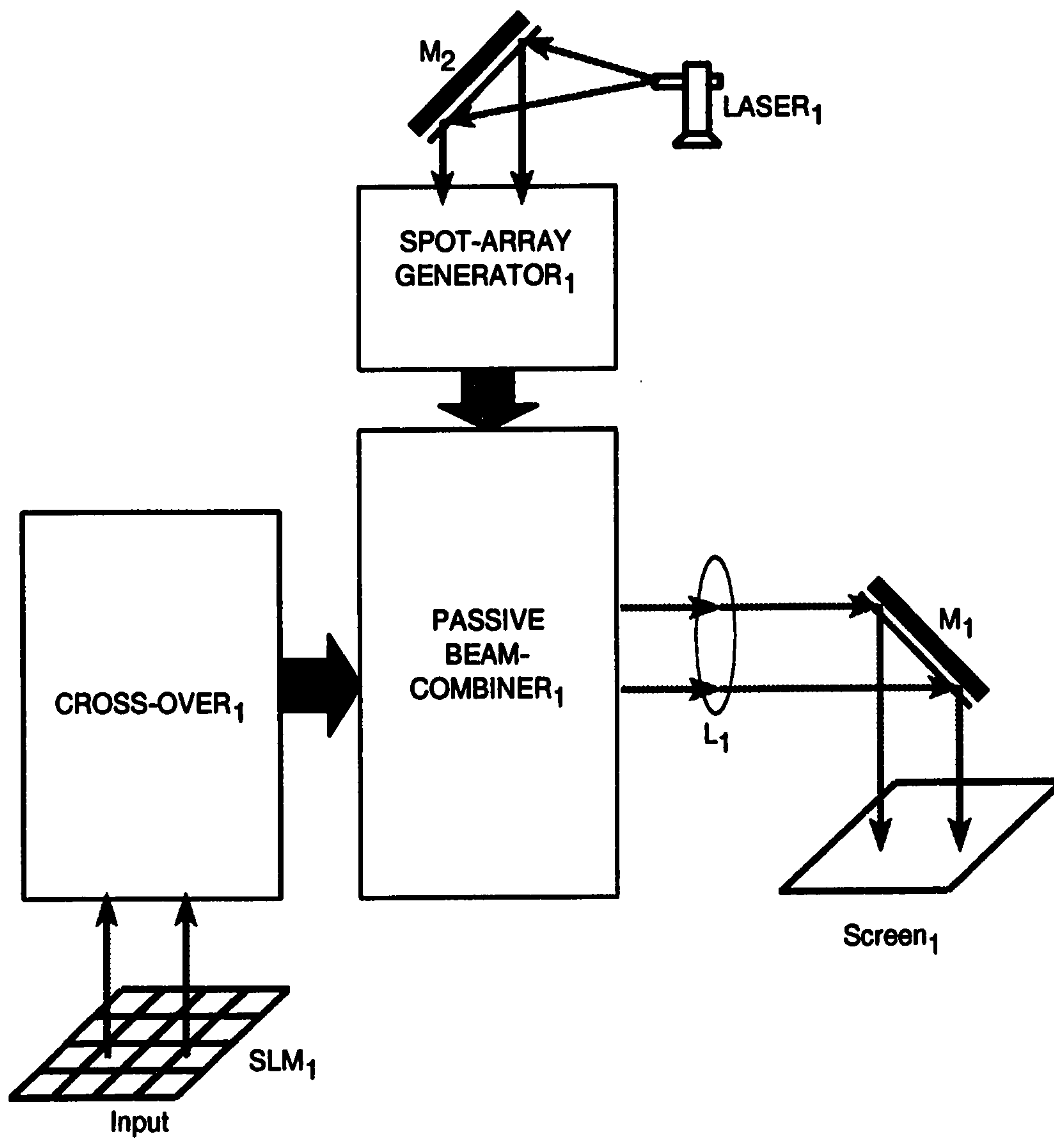


Figure 29.1: The Schematic of *System$_2$* of the AT&T Switching Fabrics.

29.3 OHDL Description of the Architecture

29.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

29.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System2: OHM Crossover";
Architect = "";
```

29.3.3 Components instantiation

Components instantiation

In[4]:=

```
l1 = GetObject["ConvexLens", CatalogNumber -> "CL1"];

m1 = GetObject["Mirror", CatalogNumber -> "M1"];

input = GetObject["SLM", CatalogNumber -> "SLM1"];

output = GetObject["Screen", CatalogNumber -> "S1"];

laser1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

crossover = GetObject["Assembly", CatalogNumber -> "A1"];

spotarraygenerator = GetObject["Assembly", CatalogNumber -> "A1"];

passivebeamcombiner = GetObject["Assembly", CatalogNumber -> "A1"];
```

29.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.01; g = 0.1; h = 0.001; i =
0.1; j = 0.1;
```

Constraints

In[8]:=

```
AddPlacementConstraint[default[laser1]];

 AddPlacementConstraints[{
 relativeTo[laser1, spotarraygenerator, {h, -c, 0}],
 relativeTo[spotarraygenerator, passivebeamcombiner, {0, -b, 0}],
 relativeTo[passivebeamcombiner, crossover, {-g, a, 0}],
 relativeTo[crossover, input, {f, -a, 0}],
 relativeTo[passivebeamcombiner, l1, {i, d, 0}],
 relativeTo[l1, m1, {j, 0, 0}],
 relativeTo[m1, output,{0, -e, 0}]
 }];
```

In[11]:=

```
ComputePositions[];
```

29.3.5 Orientation Constraints

Constraints

In[13]:=

```
AddOrientationConstraints[{
 orientationOf[l1, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[input, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[laser1, {angle[-90 Degree], angle[0 Degree], angle[0]}],
 orientationOf[m1, {angle[-45 Degree], angle[90 Degree], angle[0]}]
 }];
 AddOrientationConstraints[{
 parallelTo[input, output]
 }];
```

In[14]:=

```
ComputeOrientations[];
```

29.3.6 Layout

In[16]:=

```
ShowArchitecture[];
```

29.4 Conclusion

In this Chapter three of the previously designed architectures were used to construct a four-stage interconnection network. The $\mathcal{OHDL}$ description was provided and the resulting 3D-layout was presented.

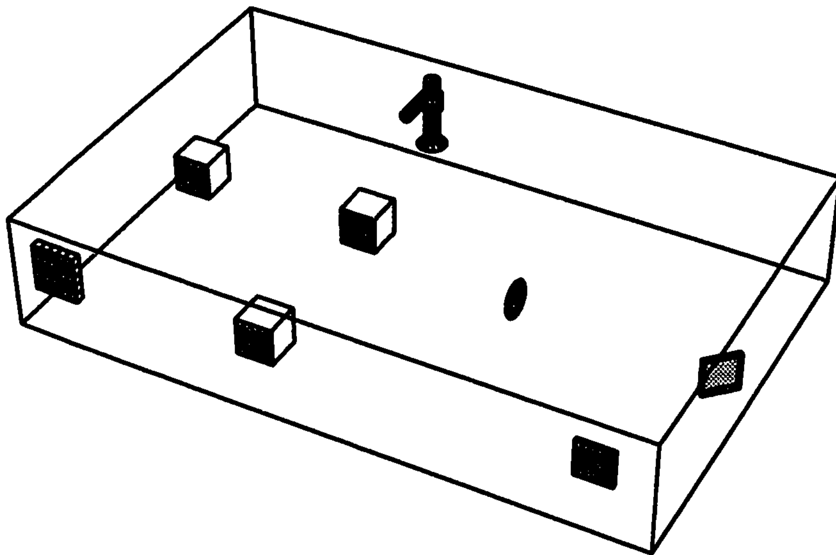


Figure 29.2: 3D-Layout of $System_2$ of AT&T Switching Fabrics

Chapter 30

Integrated Spot Array Generator

Chapter Abstract

The spot array generator provides the optical power supply to clock devices [76]. These are generated as 2D-arrays of uniform intensity spots. This is done by using two basic components. The first is a high power, single frequency, diffraction limited laser that can provide the appropriate power per pixel needed to meet the system speed requirements. The second component in a spot-array generator provides a mechanism to equally and uniformly divide the power from the laser and distribute it to the optical windows of the S-SEEDs.

30.1 Introduction

This Chapter describes another way of generating a spot-array that has numerous applications in free-space optics. One such application is generating the clock signal for a SEED array.

Section 30.2.1 discusses the approach used to generate the spots of equal power. Section 30.2.3 outlines the working of the spot array generator.

30.2 Optical Implementation

30.2.1 Approach

Input:

None.

Output:

Clock, preset and illumination for alignemnt.

Function:

Generates the spot array.

The approach followed here is exactly the same as the one followed for the spot array generator described in Chapter 28, i.e., Fourier plane spot array generation using binary phase grating (BPG).

30.2.2 Internal parameters and Constraints

The internal parameter is the spot array spacing.

30.2.3 Operation

Figure 30.1 shows the components and their placements to realize the integrated spot-array generator.

The clock signal is generated by the pulsed laser ($LASER_2$) and then distributed to the beam-combiner by M_2, BPG_2 (1×2 grating), L_9, SF (spatial filter), L_8, PBS_3, BS_1, and PBG_1.

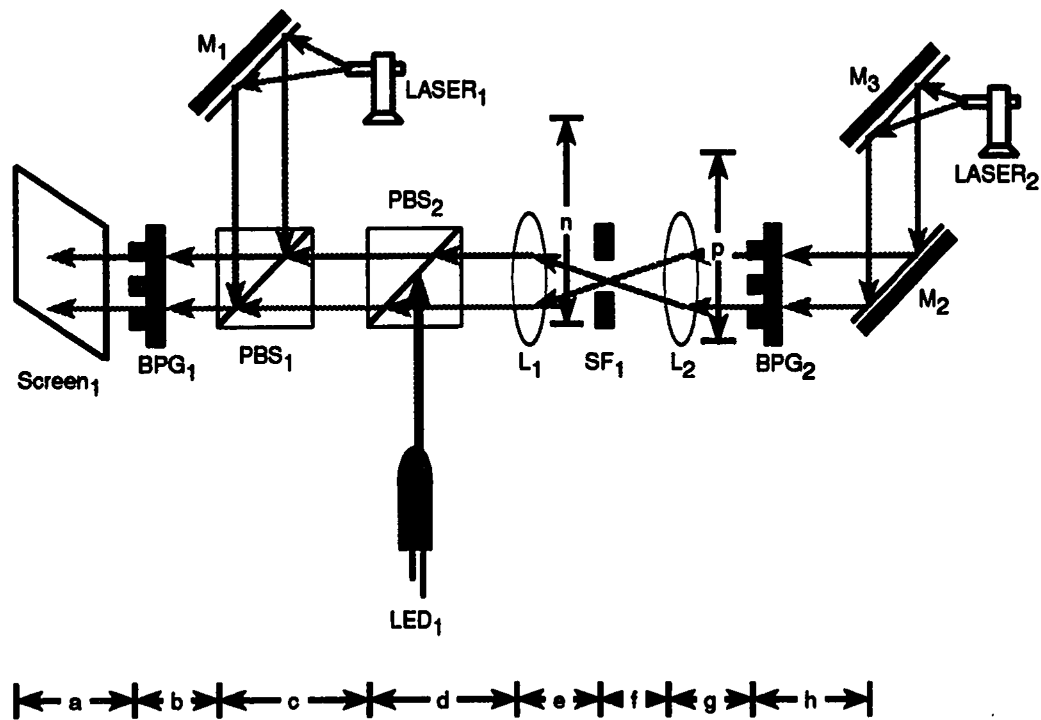


Figure 30.1: The Integrated Spot Array Generator.

30.3 OHDL Description of the Architecture

30.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

30.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System3: Integrated Spot array Generator";
Architect = "";
```

30.3.3 Components instantiation

Components instantiation

In[3]:=

```
{m1, m2} = GetObjects[2, "Mirror", CatalogNumber -> "M1"];

{l1, l2} = GetObjects[2, "PlanoConvexLens", CatalogNumber -> "PCL1"];

sf1 = GetObject["SpatialFilter", CatalogNumber -> "SF1"];

{bpg1, bpg2} = GetObjects[2, "BinaryPhaseGrating", CatalogNumber -> "BPG1"];

{pl1, pl2} = GetObjects[2, "PulsedLaser", CatalogNumber -> "PL1"];

led1 = GetObject["LED",  CatalogNumber -> "LED1"];

pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];

bs1 = GetObject["BeamSplitter", CatalogNumber -> "BS1"];

output = GetObject["Screen", CatalogNumber -> "S1"];
```

30.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
i = 0.1; k = 0.1; l = 0.1; m = 0.1; n = 0.1; o = 0.01; p = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[bs1]];

AddPlacementConstraints[{
relativeTo[bs1, m1, {0, n, 0}],
relativeTo[m1, pl1, {k, o, 0}],
relativeTo[bs1, bpg1, {-b, 0, 0}],
relativeTo[bpg1, output, {-a, 0, 0}],
relativeTo[bs1, pbs1, {c, 0, 0}],
relativeTo[bs1, led1, {i, -m, 0}],
relativeTo[pbs1, l1, {d, 0, 0}],
relativeTo[l1, sf1, {e, 0, 0}],
relativeTo[sf1, l2, {f, 0, 0}],
relativeTo[l2, bpg2, {g, 0, 0}],
relativeTo[bpg2, m2, {h, 0, 0}],
relativeTo[bpg2, pl2, {l, p, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

30.3.5 Orientation Constraints

Constraints

In[6]:=

```
AddOrientationConstraints[{
 orientationOf[output, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[m1, {angle[45 Degree], angle[90 Degree], angle[0]}],
 orientationOf[pl1, {angle[180 Degree], angle[0 Degree], angle[0]}],
 orientationOf[pl2, {angle[-90 Degree], angle[0 Degree], angle[0]}],
 orientationOf[led1, {angle[90 Degree], angle[0 Degree], angle[0]}],
 orientationOf[m2, {angle[45 Degree], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[output, bpg1],
 parallelTo[bpg1, l1],
 parallelTo[l1, sf1],
 parallelTo[sf1, l2],
 parallelTo[l2, bpg2]}
 ];
```

In[6]:=

```
ComputeOrientations[];
```

30.3.6 Layout

In[7]:=

```
ShowArchitecture[];
```

30.4 Conclusion

In this Chapter another method of generating spots to be used for clocking SEED elements was presented. This method is an improvement over the previous one described in Chapter 28, i.e., it can generate more spots of equal power. The architecture was described using $\mathcal{OHDL}$. Results were presented as a 3D-layout.

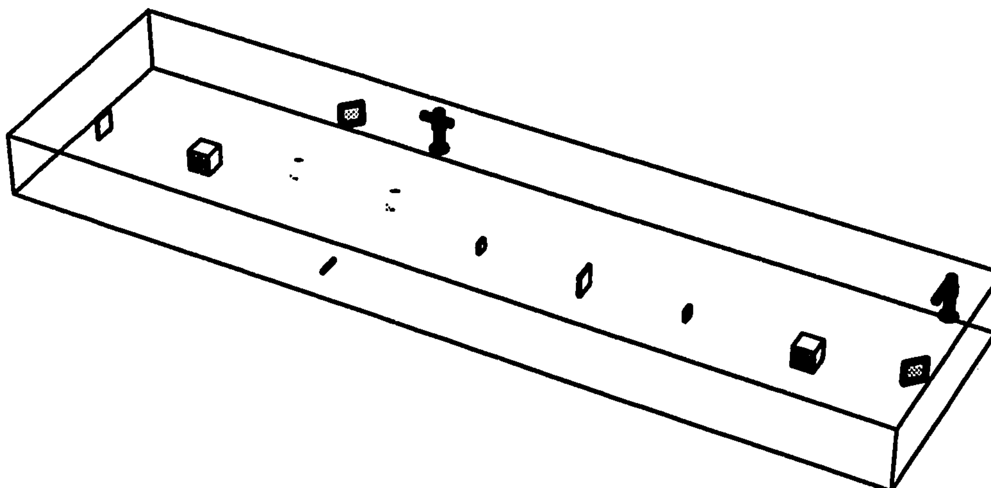


Figure 30.2: 3D-Layout of Integrated Spot Array Generator

Chapter 31

$System_3$ of AT&T Switching Fabrics

Chapter Abstract

The availability of the integrated spot array generator allows changes and modifications to $System_2$ of the AT&T switching fabrics [31] resulting in an improved system called $System_3$.

31.1 Introduction

Similar to $System_2$, $System_3$ also realizes a four-stage interconnection network. It is divided into submodules each of which has been described in previous Chapters (Chapters 26, 27 and 30).

Section 31.2.1 outlines the approach taken for the realization of this architecture. Section 31.2.2 discusses its operation.

31.2 Optical Implementation

31.2.1 Approach

Input:

8 × 8 bundle.

Output:

8 × 8 bundle.

Function:

3-Stage CLOS MIN.

The approach is to use existing components that were previously designed. These include the integrated spot-array generator, passive beam-combiner and the crossover interconnection network. Some additional components are used to guide the light through the various subsystems.

31.2.2 Operation

Figure 31.1 shows the schematic of $System_3$ of the AT&T switching fabrics. It is constructed from three sub-architectures. The clock is generated by the integrated spot-array generator which is given directly as one of the inputs to the passive beam-combiner. This also receives the input that has passed through the crossover interconnect to achieve the proper connectivity. The output of the passive beam-combiner is collected at the screen $SCREEN_1$.

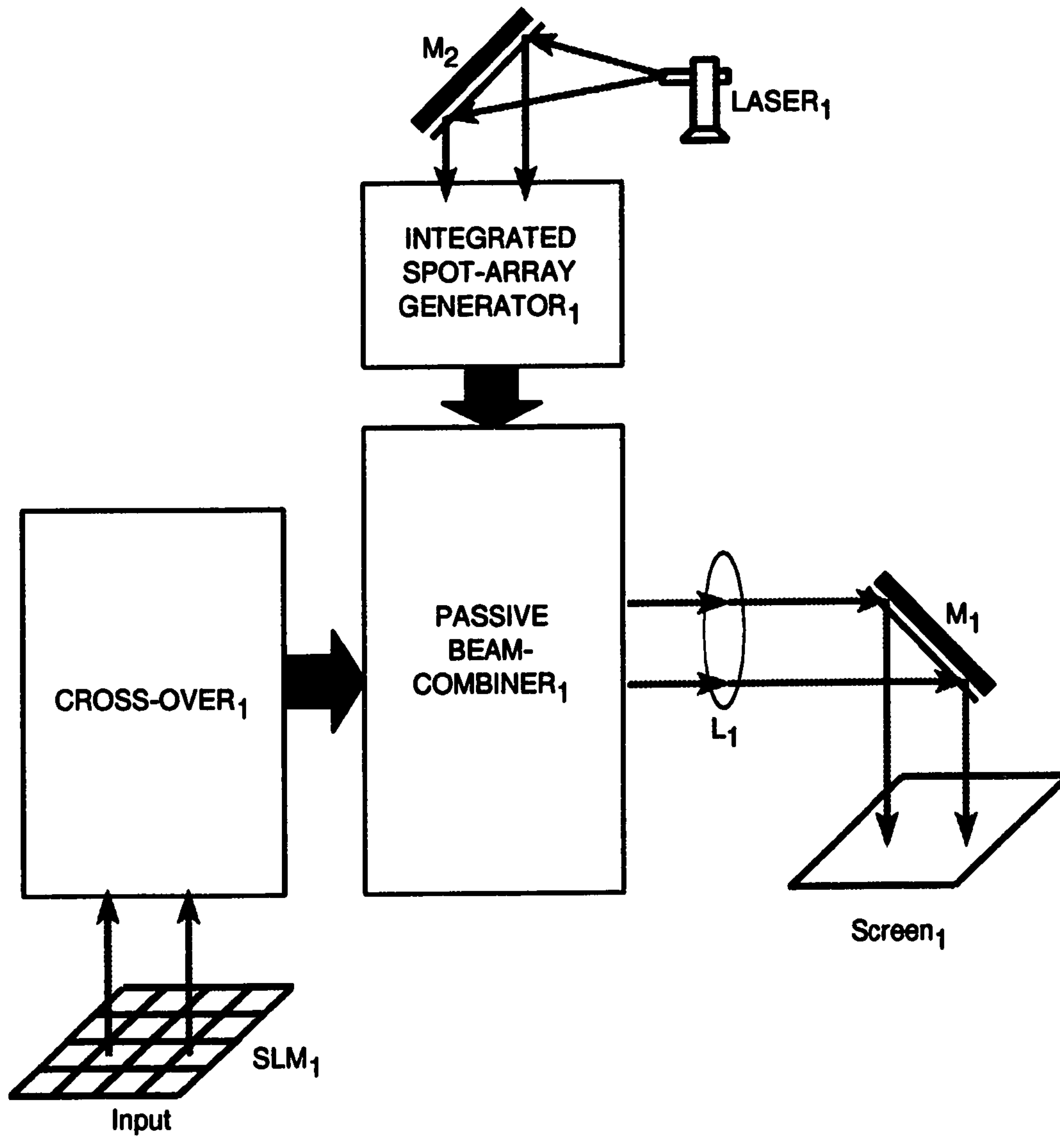


Figure 31.1: The Schematic of $System_3$ of the AT&T Switching Fabrics.

31.3 OHDL Description of the Architecture

31.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

31.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System3: OHM CrossOver";
Architect = "";
```

31.3.3 Components instantiation

Components instantiation

In[4]:=

```
l1 = GetObject["ConvexLens", CatalogNumber -> "CL1"];

m1 = GetObject["Mirror", CatalogNumber -> "M1"];

input = GetObject["SLM", CatalogNumber -> "SLM1"];

output = GetObject["Screen", CatalogNumber -> "S1"];

laser1 = GetObject["PulsedLaser", CatalogNumber -> "PL1"];

crossover = GetObject["Assembly", CatalogNumber -> "A1"];

spotarraygenerator = GetObject["Assembly", CatalogNumber -> "A1"];

passivebeamcombiner = GetObject["Assembly", CatalogNumber -> "A1"];
```

31.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.01; g = 0.1; h = 0.001; i =
0.1; j = 0.1;
```

Constraints

In[8]:=

```
AddPlacementConstraint[default[laser1]];

 AddPlacementConstraints[{
 relativeTo[laser1, spotarraygenerator, {h, -c, 0}],
 relativeTo[spotarraygenerator, passivebeamcombiner, {0, -b, 0}],
 relativeTo[passivebeamcombiner, crossover, {-g, a, 0}],
 relativeTo[crossover, input, {f, -a, 0}],
 relativeTo[passivebeamcombiner, l1, {i, d, 0}],
 relativeTo[l1, m1, {j, 0, 0}],
 relativeTo[m1, output,{0, -e, 0}]
 }];
```

In[11]:=

```
ComputePositions[];
```

31.3.5 Orientation Constraints

Constraints

In[12]:=

```
AddOrientationConstraints[{
 orientationOf[l1, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[input, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[laser1, {angle[-90 Degree], angle[0 Degree], angle[0]}],
 orientationOf[m1, {angle[-45 Degree], angle[90 Degree], angle[0]}]
 }];
 AddOrientationConstraints[{
 parallelTo[input, output]
 }];
```

In[14]:=

```
ComputeOrientations[];
```

31.3.6 Layout

In[16]:=

```
ShowArchitecture[];
```

31.4 Conclusion

In this Chapter improvements were made in *System*$_2$ of the AT&T switching fabrics. Three of the previously designed architectures were used to construct a four-stage interconnection network. The $\mathcal{OHDL}$ description was provided and the resulting 3D-layout was presented.

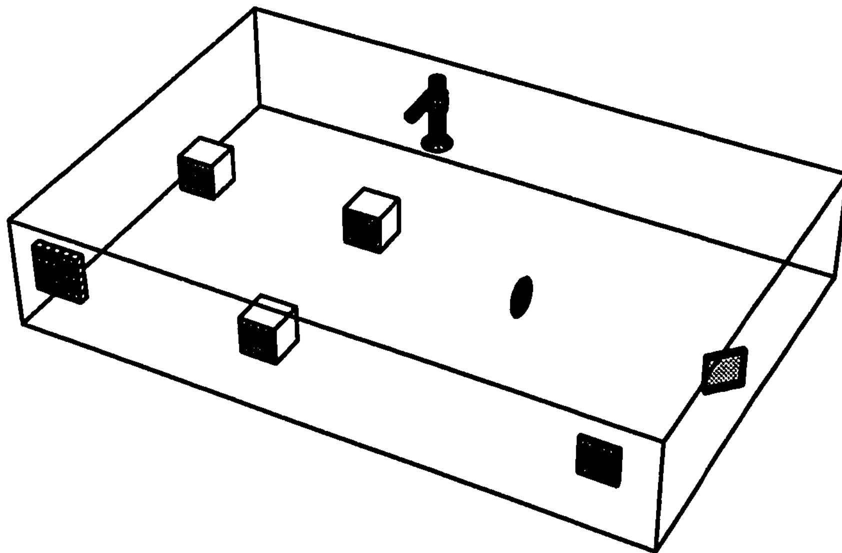


Figure 31.2: 3D-Layout of $System_3$ of AT&T Switching Fabrics

Chapter 32

1×3 BPG Interconnect for Banyan Connection

Chapter Abstract

This architecture provides Banyan connectivity between 2-matrices.

32.1 Introduction

The Banyan network is a multistage interconnection network connecting N inputs with the same number of outputs. It uses the minimum number of connections. The Banyan network consists of $\log_2(N)$ stages. Each node of a specific stage has 2 input connections and 2 output connections.

This Chapter describes one approach for realizing one stage of the Banyan interconnect. The $\mathcal{O}pti\mathcal{CAD}$ system is used to describe the architecture in $\mathcal{OHDL}$. Results

are obtained as a 3D-layout.

32.2 Optical Implementation

32.2.1 Approach

Input:

Input matrix.

Output:

Superimposed (3 signals) output matrix

Function:

Provides Banyan connectivity between two matrices

The $System_4$ optical interconnect is used to create a 3D-network composed of a 2D $N \times M$ network replicated X times to create X parallel $N \times M$ networks. This is equivalent to an $N \times M$ network that is X bits deep.

32.2.2 Operation

This type of network can be implemented with a simple 2D-interconnect. An example of the interconnect is shown in Figure 32.1, where the output of each node is directed to the pupil-plane where a binary phase grating (*BPG*) splits the signal into three equal parts.

These copies of the original signal are then directed to the inputs of the next stage of the network. Figure 32.1(b) illustrates how a Banyan network (thick lines) can

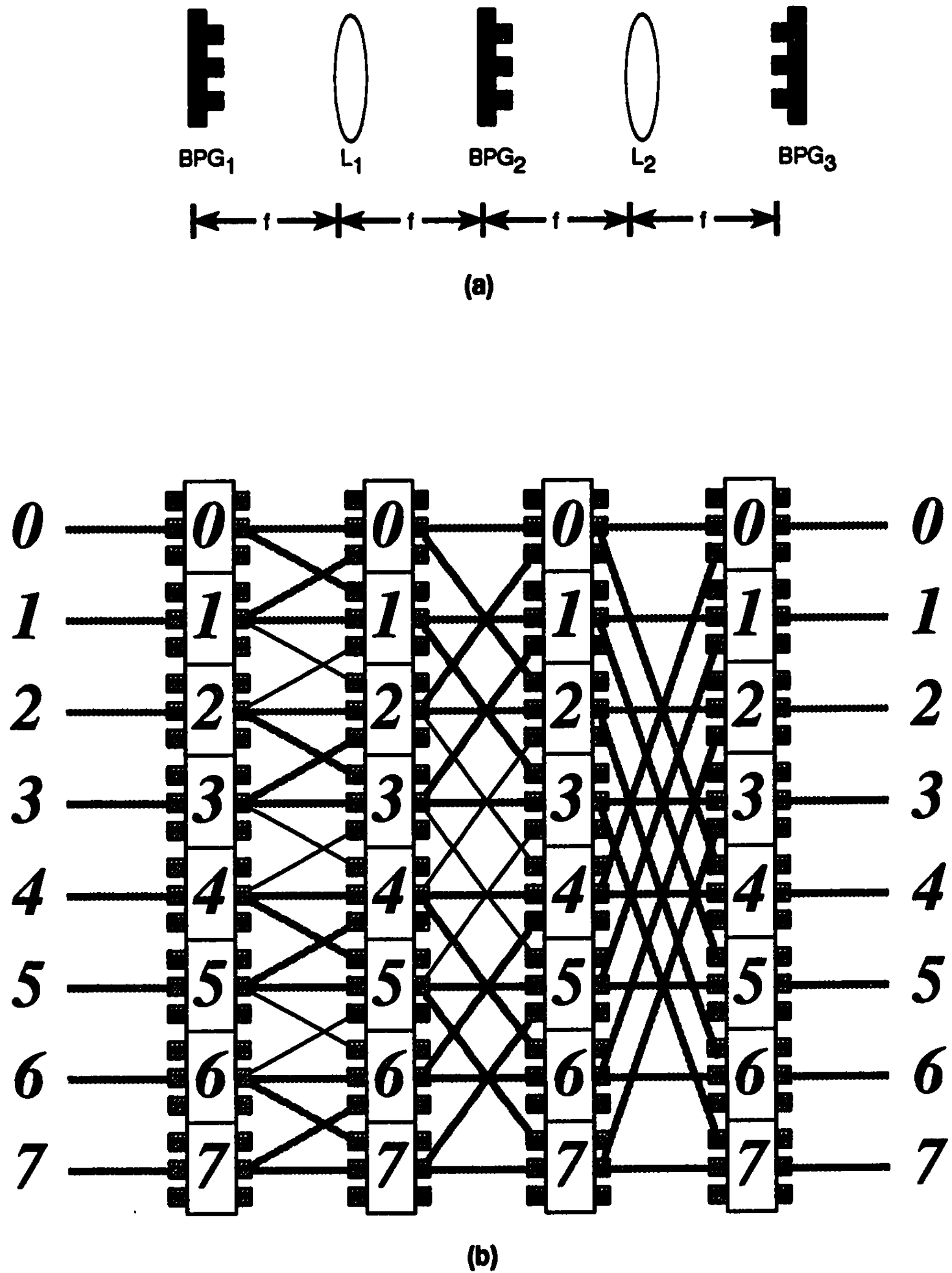


Figure 32.1: Pupil Plane Interconnection. (a) 1 × 3 BPG Interconnect. (b) Banyan Network using 1 × 3 Interconnect.

be created using these interconnects. The thick lines represent active connections between nodes in $stage_i$ and nodes in $stage_{i+1}$; the thin lines represent connections between stages in the network that are blocked by placing metal masks in front of the optical windows of specific nodes in $stage_{i+1}$ [15].

32.3 OHDL Description of the Architecture

32.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

32.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System 4: 1x3 BPG Interconnect for Banyan Connection";
Architect = "";
```

32.3.3 Components instantiation

Components instantiation

In[3]:=

```
{l1, l2} = GetObjects[2, "ConvexLens", CatalogNumber -> "CL1"];

{bpg1, bpg2, bpg3} = GetObjects[3, "BinaryPhaseGrating",
 CatalogNumber -> "BPG1", Size -> {0.02, 0.02, 0.002}];
```

32.3.4 Placement Constraints

Parameters

In[4]:=
```
f = 0.1;
```

Constraints

In[5]:=
```
AddPlacementConstraint[default[bpg1]];

AddPlacementConstraints[{
relativeTo[bpg1, l1, {f, 0, 0}],
relativeTo[l1, bpg2, {f, 0, 0}],
relativeTo[bpg2, l2, {f, 0, 0}],
relativeTo[l2, bpg3, {f, 0, 0}]}];
```

In[5]:=
```
ComputePositions[];
```

32.3.5 Orientation Constraints

Constraints

In[6]:=
```
AddOrientationConstraints[{
orientationOf[bpg1, {angle[0], angle[90 Degree], angle[0]}]}
];

AddOrientationConstraints[{
parallelTo[bpg1, l1],
parallelTo[bpg1, l2],
parallelTo[bpg1, bpg2],
parallelTo[bpg1, bpg3]}
];
```

In[7]:=
```
ComputeOrientations[];
```

32.3.6 Layout

In[8]:=
```
ShowArchitecture[];
```

32.4 Conclusion

In this Chapter an optical architecture for realizing one stage of the Banyan interconnect was described using $\mathcal{OHDL}$. The final output was obtained as a 3D-layout.

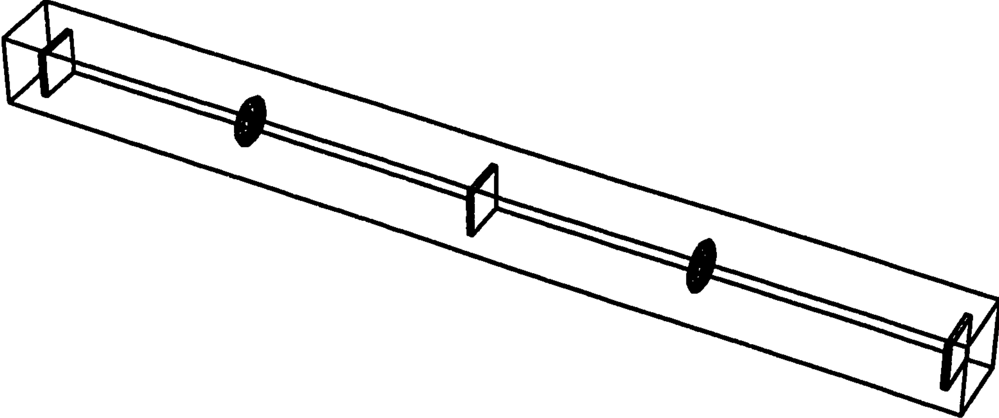


Figure 32.2: 3D-Layout of 1 × 3 BPG Interconnect for Banyan Connection

Chapter 33

Lossy Beam Combiner

Chapter Abstract

The simple but lossy beam-combination unit is used to combine several signals into only one signal.

33.1 Introduction

This Chapter describes a lossy beam-combination unit. It combines several beams into one but the result is lossy. It employs specifically made S-SEEDs.

Section 33.2.1 describes the approach used in designing the beam combiner. Section 33.2.2 discusses the working of the architecture. A description of the architecture is presented in $\mathcal{OHDL}$.

33.2 Optical Implementation

33.2.1 Approach

Input:

Input matrix (input image) and Clock: Input matrix (spot array)

Output:

Output image after an interconnect of one stage of Banyan

Function:

Combines two superimposed (actually three, but one of which is blocked out by reflectors in the optical windows of S-SEED array), and the clock to produce an output image which goes to a three way splitter and a superimposer to realize Banyan connectivity.

This subsystem combines two beams and the clock to provide an output image which will go to a three-way splitter and superimposer for Banyan connectivity.

33.2.2 Operation

The lossy beam combination (used in $System_4$) is a method to combine the two input signals, the clock, and the output signals (although the losses were similar to those of $System_3$). The beam-combination hardware included the polarization beam-splitter (PBS_1), and a partially reflecting mirror (OWM_1) (50:50), shown in Figure 33.1 [16].

The input image enters the beam-combination unit s-polarized, thus it is reflected up by PBS_1, passes through a quarter-wave plate (QWP_1) to OWM_1. Half of the power is reflected by OWM_1, passing through QWP_1 again, which will rotate the

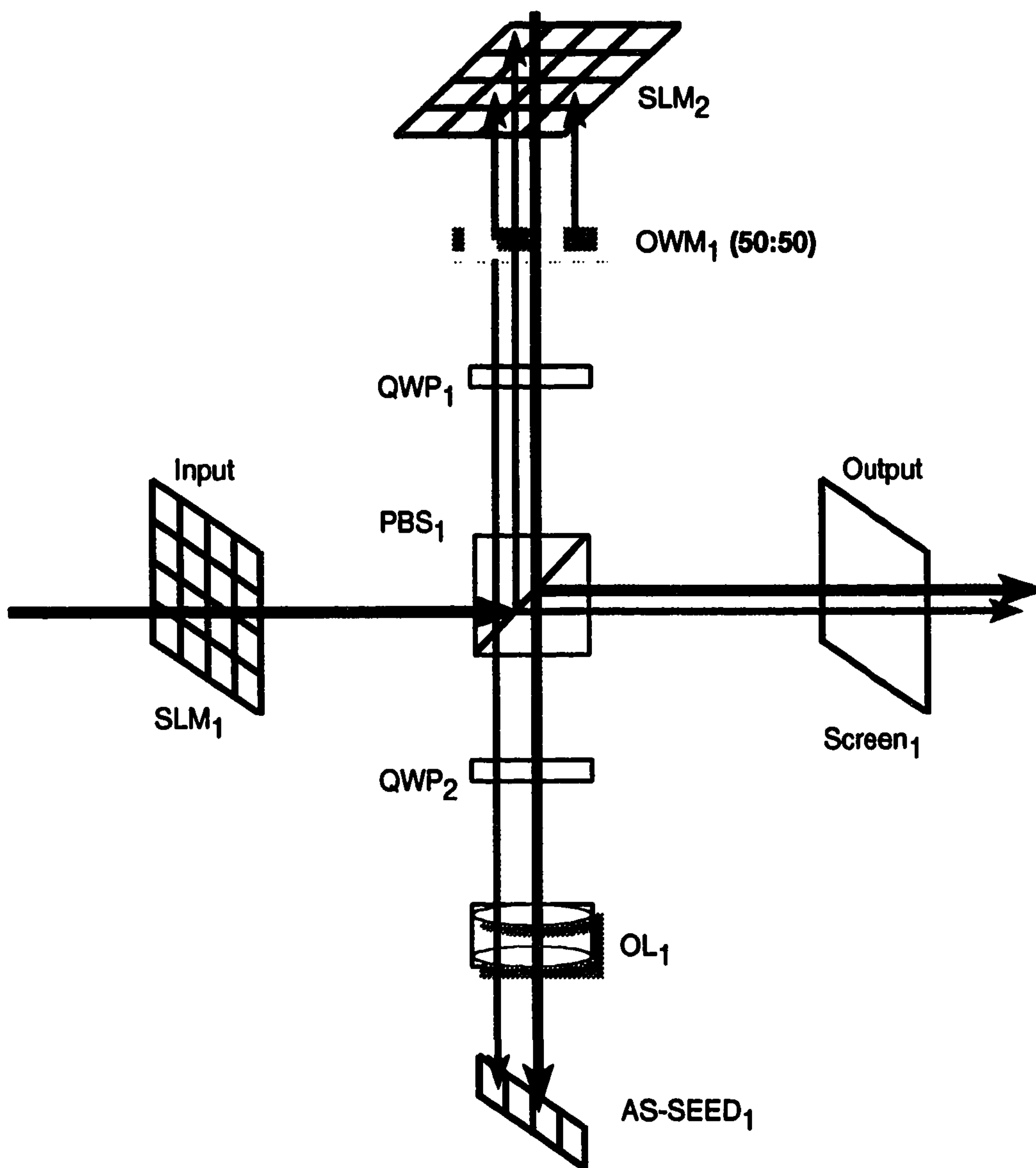


Figure 33.1: Lossy Beam Combination.

image to p-polarized light. This reflected image will then pass through PBS_1, through another quarter-wave plate (QWP_2), the objective lens OL_1, and onto the array of switching nodes. For the case S-SEED nodes, the information present on the input image is latched into the S-SEEDs. The circularly polarized *clock* can then pass through all the elements (half the clock power is reflected (lost) at $M(50:50)$), be modulated by the information stored in the switching node array, and then be reflected out of the beam-combination unit to provide the output image to the next part of the system.

33.3 OHDL Description of the Architecture

33.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

33.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System4: Lossy Beam Combination";
Architect = "";
```

33.3.3 Components instantiation

Components instantiation

In[3]:=

```
{input, clock} = GetObjects[2, "SLM", CatalogNumber -> "SLM1"];
pbs1 = GetObject["PolarizingBeamSplitter", CatalogNumber -> "PBS1"];
{qwp1, qwp2} = GetObjects[2, "QuarterWavePlate", CatalogNumber -> "QWP1"];
screen1 = GetObject["Screen", CatalogNumber -> "S1"];
owm1 = GetObject["OneWayMirror", CatalogNumber -> "OWM1"];
ol1 = GetObject["ObjectiveLens", CatalogNumber -> "OL1"];
sseed1 = GetObject["SSEED", CatalogNumber -> "SSEED1"];
```

33.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, input, {-a, 0, 0}],
 relativeTo[pbs1, screen1, {h, 0, 0}],
 relativeTo[pbs1, qwp1, {0, e, 0}],
 relativeTo[qwp1, owm1, {0, f, 0}],
 relativeTo[owm1, clock, {0, g, 0}],
 relativeTo[pbs1, qwp2, {0, -d, 0}],
 relativeTo[qwp2, ol1, {0, -c, 0}],
 relativeTo[ol1, sseed1, {0, -b, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

33.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[clock, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[input, screen1],
 parallelTo[clock, owm1],
 parallelTo[clock, qwp1],
 parallelTo[clock, qwp2],
 parallelTo[clock, ol1],
 parallelTo[clock, sseed1]}
 ];
```

In[8]:=

```
ComputeOrientations[];
```

33.3.6 Layout

In[9]:=

```
ShowArchitecture[];
```

33.4 Conclusion

This Chapter presented the design of a lossy beam combiner. An $\mathcal{OHDL}$description was given which produced a 3D-layout as output.

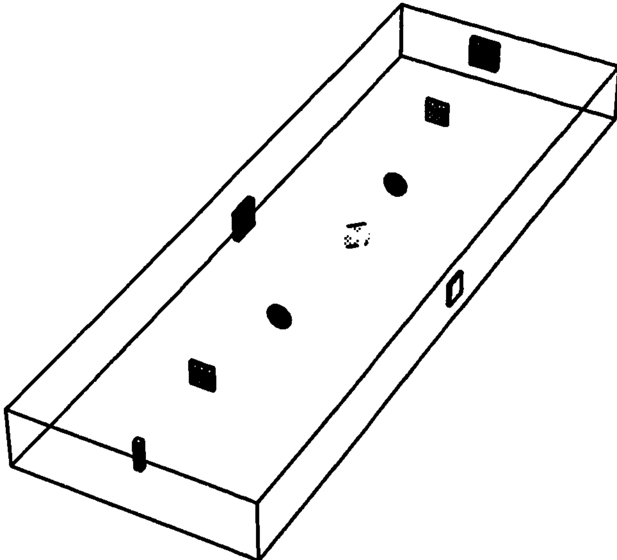


Figure 33.2: 3D-Layout of Lossy Beam Combiner

Chapter 34

Spot Array Generation

Chapter Abstract

As systems become more sophisticated and use more components, there is a need for a spot array generator that provides a large number of equal power spots of light. The architecture discussed in this Chapter redistributes the input optical power into 2048 *equal power spots of light.*

34.1 Introduction

Improving the spot-array generation to accommodate larger SEEDs led to the development of the architecture presented in this Chapter. It uses several custom made components fine-tuned to achieve the desired functionality.

Section 34.2.1 describes the approach employed in designing the generator. Section 34.2.2 discusses the working of the architecture.

34.2 Optical Implementation

34.2.1 Approach

Input:

None.

Output:

Equi-power spot array.

Function:

Generates spot arrays to be used as clock signals.

The approach followed in this architecture is similar to the one used in the spot-array generators described in Chapters 28 and 30. The spot array generation is accomplished by using binary phase grating (BPG). This will uniformly distribute the optical power to the output spots.

34.2.2 Operation

The spot-array generator hardware used by AT&T's $System_4$ is shown in Figure 34.1. The output light of the laser drive is initially collimated (by LC_1) and then circularized by the Brewster telescope (BT_1 and BT_2). The analyzer (A_1) and quarter-wave plate (QWP_1) are used as an isolator to reduce the anti-stablizing effects of backreflections on the laser. The light collimated by L_1 and L_2 passes through the Risley prisms (RP_1) which are used to register the spot-arrays on the SEED device photosensitive windows. Finally, the light passes through a 64×32 binary phase

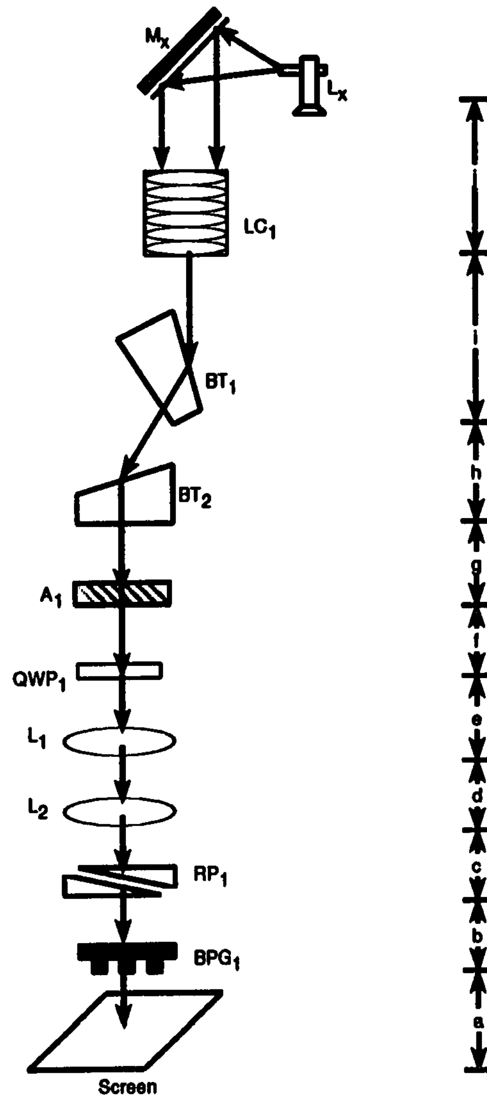


Figure 34.1: Spot Array Generation.

grating (BPG_1), which redistributes the input optical power into 2048 equal power spots of light.

34.3 OHDL Description of the Architecture

34.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

34.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System4: Spot Array Generation";
Architect = "";
```

34.3.3 Components instantiation

Components instantiation

In[3]:=

```
ld1 = GetObject["LaserDiode", CatalogNumber -> "LD1"];
lc1 = GetObject["LaserCollimatorLens", CatalogNumber -> "LC1"];
{bt1, bt2} = GetObjects[2, "BrewsterTelescope", CatalogNumber -> "BT1"];
an1 = GetObject["Analyser", CatalogNumber -> "AN1"];
qwp1 = GetObject["QuarterWavePlate", CatalogNumber -> "QWP1"];
l1 = GetObject["ConvexLens", CatalogNumber -> "CL1"];
l2 = GetObject["ConcaveLens", CatalogNumber -> "CCL1"];
rp1 = GetObject["RisleyPrism", CatalogNumber -> "RP1"];
bpg1 = GetObject["BinaryPhaseGrating", CatalogNumber -> "BPG1"];
screen1 = GetObject["Screen", CatalogNumber -> "S1"];
```

34.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
i = 0.1; j = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[screen1]];

 AddPlacementConstraints[{
 relativeTo[screen1, bpg1, {0, a, 0}],
 relativeTo[bpg1, rp1, {0, b, 0}],
 relativeTo[rp1, l2, {0, c, 0}],
 relativeTo[l2, l1, {0, d, 0}],
 relativeTo[l1, qwp1, {0, e, 0}],
 relativeTo[qwp1, an1, {0, f, 0}],
 relativeTo[an1, bt2, {0, g, 0}],
 relativeTo[bt2, bt1, {0, h, 0}],
 relativeTo[bt1, lc1, {0, i, 0}],
 relativeTo[lc1, ld1, {0, j, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

34.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
 orientationOf[screen1, {angle[90 Degree], angle[90 Degree], angle[0]}],
 orientationOf[bt1, {angle[45 Degree], angle[0 Degree], angle[0]}],
 orientationOf[ld1, {angle[-90 Degree], angle[0 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[screen1, bpg1],
 parallelTo[bpg1, rp1],
 parallelTo[rp1, l2],
 parallelTo[l1, l2],
 parallelTo[l1, an1],
 parallelTo[l1, bt2],
 parallelTo[l2, lc1]}
 ];
```

In[8]:=

```
ComputeOrientations[];
```

34.3.6 Layout

In[9]:=

```
ShowArchitecture[];
```

34.4 Conclusion

In this Chapter an optical architecture for generating larger number of spots was described using $\mathcal{OHDL}$. All the placement and orientation constraints were supplied to produce the final result as a 3D-layout of the architecture.

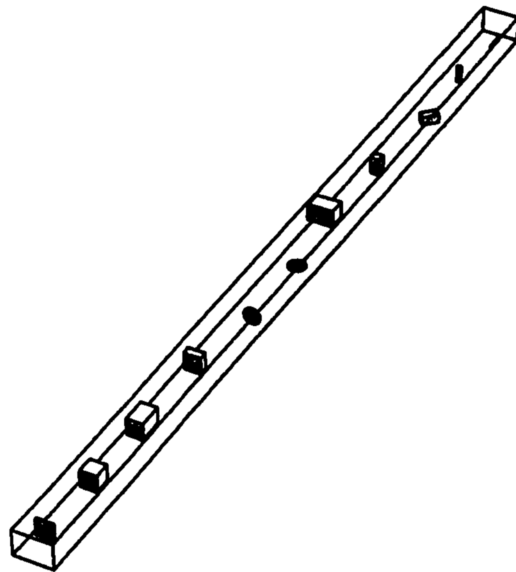


Figure 34.2: 3D-Layout of Spot Array Generation

Chapter 35

Single Stage of Banyan Network

Chapter Abstract

A single stage of the Banyan interconnect is realized by using some of the previously described components (Chapters 33 and 34). Some additional components such as prisms and gratings provide the necessary connectivity.

35.1 Introduction

This Chapter discusses the implementation of a single stage of the Banyan network using some components/architectures described earlier. This single stage is modular i.e., it can be used to build systems of multiple stages of the Banyan network.

Section 35.2.1 discusses the approach taken to describe the architecture. Section 35.2.2 outlines the working of the whole system. A description of the architecture is provided in $\mathcal{OHDL}$. Results are summarized as a 3D-layout.

35.2 Optical Implementation

35.2.1 Approach

Input:

Input matrix (superimposed images)

Output:

Output matrix (superimposed images)

Function:

One stage of a Banyan network. The approach is to latch the superimposed input image onto an AS-SEED array. The AS-SEEDs are application specific SEEDs which were specifically designed and fabricated SEEDs. They include small mirrors that were located at pre-determined locations on the optical windows to block unwanted signals. As part of the latching, undesirable connections will be blocked by the stage dependent metallic reflectors in the optical window of these AS-SEEDs.

The principle employed is to latch the super-imposed input image onto an AS-SEED array. Undesirable connections will be blocked out by the stage dependent metallic reflectors in the optical windows of the AS-SEED.

35.2.2 Operation

Figure 35.1 shows the block diagram of the hardware realizing a single stage of the Banyan network.

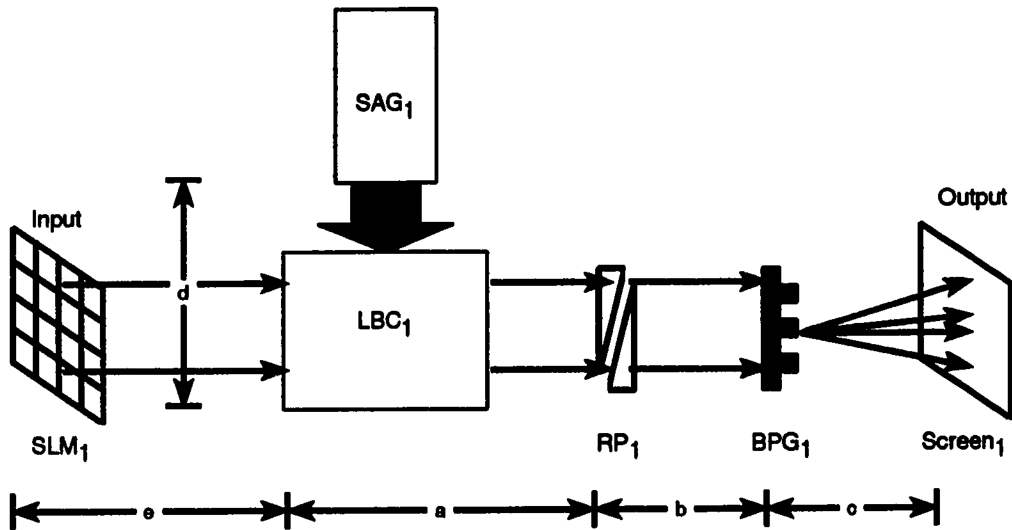


Figure 35.1: Block Diagram of the Hardware Realizing a Single Stage of the Banyan Network.

It uses instances of both the spot array generator and the lossy beam combiner in addition to a number of other optical components such as Risley prisms (RP_1), spatial light modulator (SLM_1) and binary phase grating (BPG_1). The input image is latched onto the AS-SEED array where metallic reflectors block out some of the undesirable signals. The signals travel to the lossy beam combiner LBC_1 where the input and the clock signals are merged. The output obtained at the screen $SCREEN_1$ after passing through RP_1 and BPG_1 achieves the functionality of a single stage of the Banyan interconnection network.

35.3 OHDL Description of the Architecture

35.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

35.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System4: OHM (1 Stage)";
Architect = "";
```

35.3.3 Components instantiation

Components instantiation

In[3]:=

```
input = GetObject["SLM", CatalogNumber -> "SLM1"];
rp1 = GetObject["RisleyPrism", CatalogNumber -> "RP1"];
bpg1 = GetObject["BinaryPhaseGrating", CatalogNumber -> "BPG1",
 Size -> {0.02, 0.02, 0.002}];
screen1 = GetObject["Screen", CatalogNumber -> "S1"];
{sag1, lbc1} = GetObjects[2, "Assembly", CatalogNumber -> "A1"];
```

35.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1; b = 0.1; c = 0.1; d = 0.1; e = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[lbc1]];

AddPlacementConstraints[{
relativeTo[lbc1, input, {-e, 0, 0}],
relativeTo[lbc1, sag1, {0, d, 0}],
relativeTo[lbc1, rp1, {a, 0, 0}],
relativeTo[rp1, bpg1, {b, 0, 0}],
relativeTo[bpg1, screen1, {c, 0, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

35.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
orientationOf[input, {angle[0], angle[90 Degree], angle[0]}]}
];

AddOrientationConstraints[{
parallelTo[input, rp1],
parallelTo[input, bpg1],
parallelTo[input, screen1]}
];
```

In[8]:=

```
ComputeOrientations[];
```

35.3.6 Layout

In[9]:=

```
ShowArchitecture[];
```

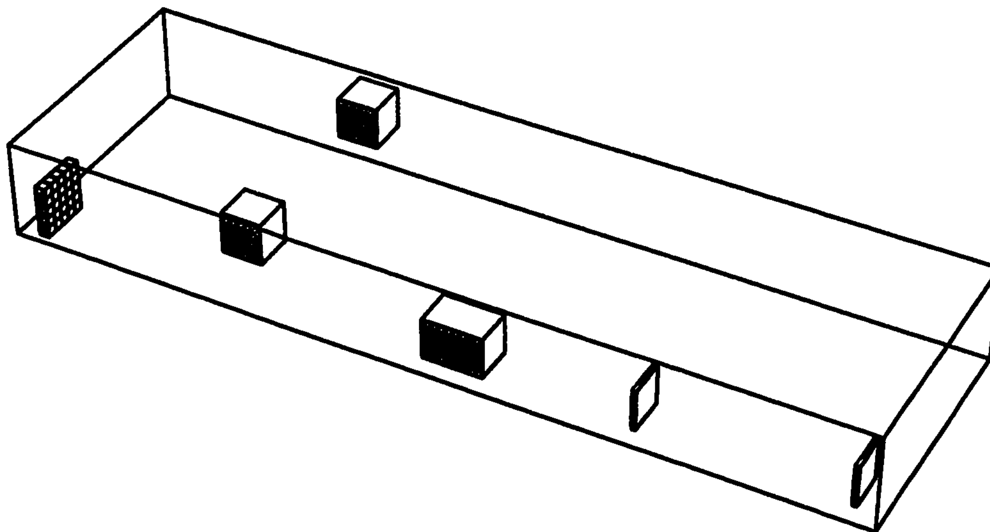


Figure 35.2: 3D-Layout of Single Stage of Banyan Network

35.4 Conclusion

This Chapter presented an optical architecture for realizing one stage of the Banyan interconnect. The architecture uses previously described sub-architectures to achieve the interconnection. The *OptiCAD* system was used to describe the architecture and obtain results. The description is given in *OHDL* and the results are obtained as a 3D-layout.

Chapter 36

$System_4$ of AT&T Switching Fabrics

Chapter Abstract

This architecture called $System_4$ achieves the functionality of a six-stage Banyan interconnection network. It is composed of a six stage 16×32 *Banyan network (32 bit wide data path) with 1024* (32×32 *array) AS-SEED per stage.*

36.1 Introduction

The single stage of the Banyan interconnection presented in the previous Chapter can be cascaded to realize a multiple stage interconnection network. The number of stages depends on the length of the input vector. This Chapter presents an architecture for a six stage Banyan interconnection network.

Section 36.2.1 describes the approach for realizing the architecture. Section 36.2.2 discusses its working. An $\mathcal{OHDL}$ description is provided and results are given in the form of a 3D-layout.

36.2 Optical Implementation

36.2.1 Approach

Input:

Input matrix (input image) (16x32).

Output:

Output image: 3-way split diverging and interleaved 16x32 matrix

Function:

Employs a 32x32 application specific S-SEED. There are 32 copies of planar six-stage Banyan. The approach used is to employ diffraction grating to split the beam and superimpose them on the image plane. Undesirable connections are masked out by special etched metallic reflectors on S-SEED arrays.

The single stage of the Banyan network was implemented in Chapter 35. Since the design was general, six stages can be constructed by simply cascading six instances of that architecture.

36.2.2 Operation

Figure 36.1 shows the block diagram of the six-stage architecture.

The control of the system was accomplished by electrically disabling rows of the AS-SEED arrays. When the AS-SEED were disabled, they absorbed the clocks on both inputs/outputs preventing the usable system from being passed onto the next stage of the system. When the voltage for the AS-SEED is "on", it behaves like two

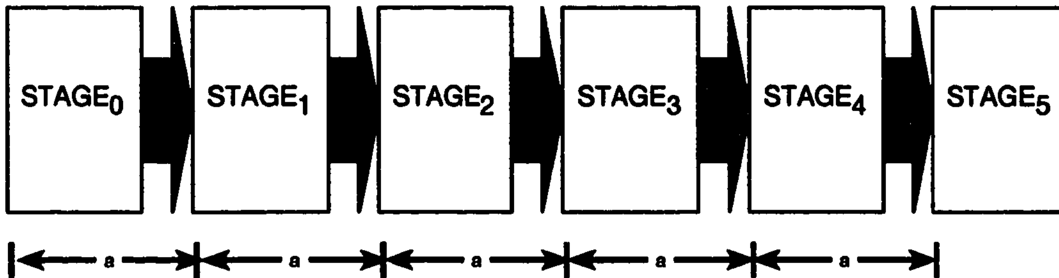


Figure 36.1: The Block Diagram of the Six Stage Banyan Network.

modules, capturing and storing the bits of information to pass onto the next stage when clocked.

The entire system is under interactive computer control allowing any number of paths to be set up through the network in real time.

36.3 OHDL Description of the Architecture

36.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

36.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System 4: Six-stage Banyan (16x32)x(16x32)";
Architect = "";
```

36.3.3 Components instantiation

Components instantiation

In[3]:=

```
{stage0, stage1, stage2, stage3, stage4, stage5} =
 GetObjects[6, "Assembly", CatalogNumber -> "A1"];
```

36.3.4 Placement Constraints

Parameters

In[4]:=

```
a = 0.1;
```

Constraints

In[5]:=

```
AddPlacementConstraint[default[stage0]];

 AddPlacementConstraints[{
 relativeTo[stage0, stage1, {a, 0, 0}],
 relativeTo[stage1, stage2, {a, 0, 0}],
 relativeTo[stage2, stage3, {a, 0, 0}],
 relativeTo[stage3, stage4, {a, 0, 0}],
 relativeTo[stage4, stage5, {a, 0, 0}]}];
```

In[6]:=

```
ComputePositions[];
```

36.3.5 Orientation Constraints

Constraints

In[7]:=

```
AddOrientationConstraints[{
 orientationOf[stage0, {angle[0], angle[0 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[stage1, stage0],
 parallelTo[stage2, stage0],
 parallelTo[stage3, stage0],
 parallelTo[stage4, stage0],
 parallelTo[stage5, stage0]}
 ];
```

In[7]:=

```
ComputeOrientations[];
```

36.3.6 Layout

In[8]:=

```
ShowArchitecture[];
```

36.4 Conclusion

In this Chapter the design of a six stage Banyan network was presented. This network was described in $\mathcal{OHDL}$ and the output was obtained as a 3D-layout.

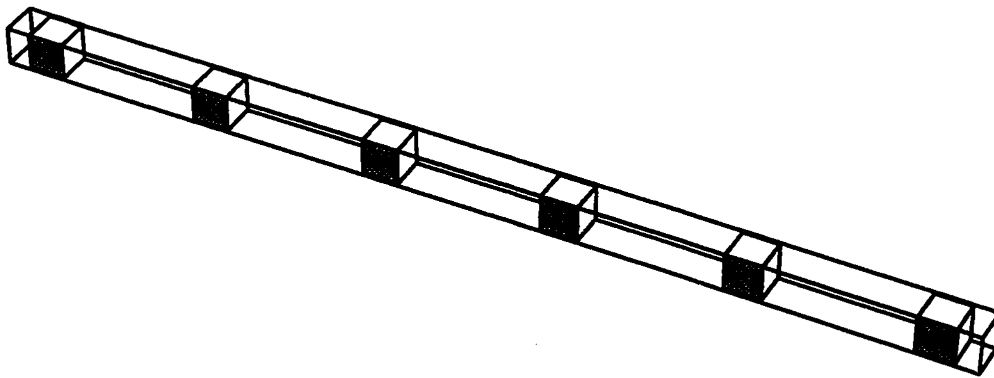


Figure 36.2: 3D-Layout of $System_4$ of AT&T Switching Fabrics

Chapter 37

Pupil Division Beam Combination

Chapter Abstract

A beam-combination unit is presented. This has the advantage of being less lossy than other approaches presented in Chapters 25 and 33.

37.1 Introduction

The beam combiner presented in this Chapter is based on the principle of pupil division beam combination. It has the advantage of not weakening the merged signals.

Section 37.2.1 discusses the approach employed and Section 37.2.2 describes the working of the architecture. An $\mathcal{OHDL}$ description of the architecture is provided.

37.2 Optical Implementation

37.2.1 Approach

Input:

Input matrix (input image) (16×32).

Output:

Output image: 3-way split diverging and interleaved 16×32 matrix

Objectives: Increased functionality and temporal bandwidth. Employs FET-SEED, smart pixels as the switching nodes which is an improvement of the hardware used in $System_4$. Computation of routing path using path-hunt algorithm and using electronic control.

Function:

Employs a 32×32 application specific S-SEED. There are 32 copies of planar six-stage Banyan. The approach used is to employ diffraction grating to split the beam and superimpose them on the image plane. Undesirable connections are masked out by special etched metallic reflectors on S-SEED arrays.

All the input signals are directed towards a FET-SEED array where they combine. The output is obtained from the FET-SEED array.

37.2.2 Operation

Figure 37.1 shows the components ans their placements in the architecture.

The input signals (s-polarized) are directed down through the objective lens (OL_1) to the FET-SEED array by PBS_1. The *Clock* signal, (p-polarized) passes through

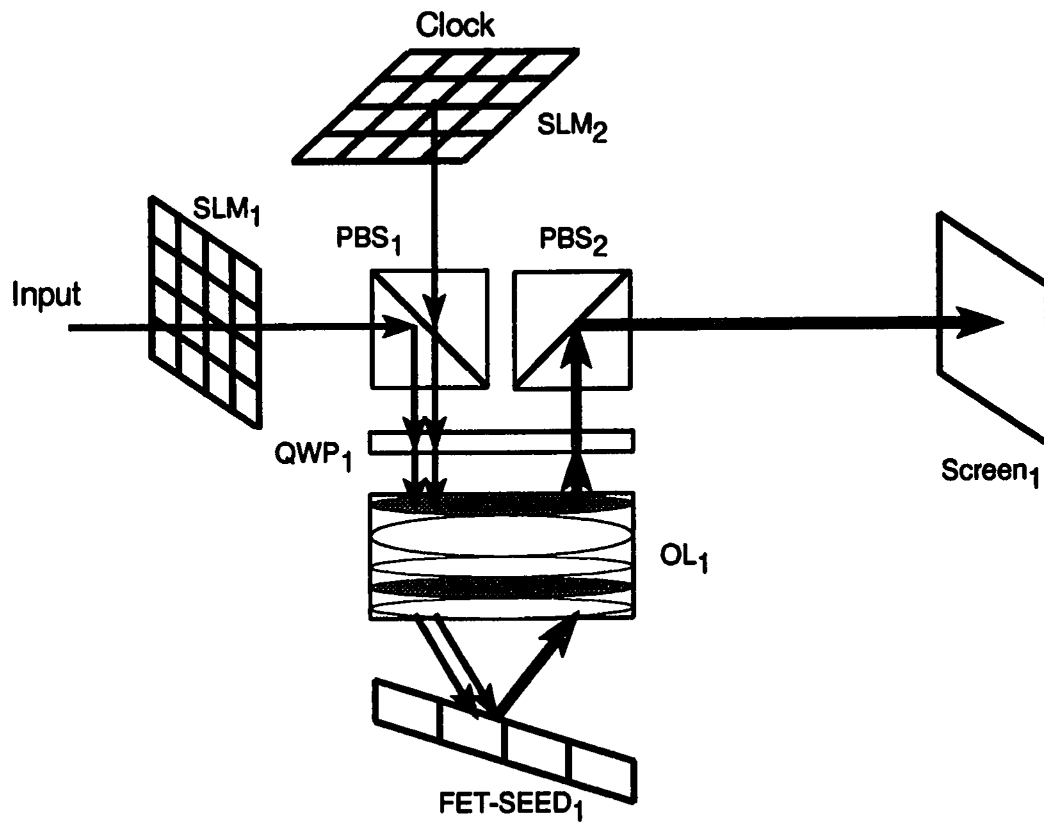


Figure 37.1: Pupil Division Beam Combination Unit.

PBS_1 and OL_1 and onto the FET-SEED array where it is reflection-modulated and directed up to PBS_2 where it is directed to $SCREEN_1$ as an output signal.

37.3 OHDL Description of the Architecture

37.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

37.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System5: Packet Switched System";
Architect = "";
```

37.3.3 Components instantiation

Components instantiation

In[4]:=

```
{input, clock} = GetObjects[2, "SLM", CatalogNumber -> "SLM1"];

{pbs1, pbs2} = GetObjects[2, "PolarizingBeamSplitter", CatalogNumber ->
"PBS1"];

screen1 = GetObject["Screen", CatalogNumber -> "S1"];

ol1 = GetObject["ObjectiveLens", CatalogNumber -> "OL1"];

qwp1 = GetObject["QuarterWavePlate", CatalogNumber -> "QWP1"];

fetseed1 = GetObject["FETSEED", CatalogNumber -> "FETSEED1"];
```

37.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1; b = 0.1; c = 0.06; d = 0.1; e = 0.1; f = 0.1; g = 0.1; h = 0.1;
```

Constraints

In[7]:=

```
AddPlacementConstraint[default[pbs1]];

 AddPlacementConstraints[{
 relativeTo[pbs1, input, {-a, 0, 0}],
 relativeTo[pbs1, pbs2, {c, 0, 0}],
 relativeTo[pbs2, screen1, {d, 0, 0}],
 relativeTo[pbs1, clock, {0, h, 0}],
 relativeTo[pbs1, qwp1, {0, -g, 0}],
 relativeTo[qwp1, ol1, {0, -f, 0}],
 relativeTo[ol1, fetseed1, {0, -e, 0}]}];
```

In[10]:=

```
ComputePositions[];
```

37.3.5 Orientation Constraints

Constraints

In[14]:=

```
AddOrientationConstraints[{
 orientationOf[pbs2, {angle[90 Degree], angle[0 Degree], angle[0]}],
 orientationOf[input, {angle[0], angle[90 Degree], angle[0]}],
 orientationOf[clock, {angle[90 Degree], angle[90 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[screen1, input],
 parallelTo[qwp1, clock],
 parallelTo[ol1, clock],
 parallelTo[fetseed1, clock]}
 ];
```

In[16]:=
```
ComputeOrientations[];
```

37.3.6 Layout

In[18]:=
```
ShowArchitecture[];
```

37.4 Conclusion

A new approach for beam combination was presented. The advantages of using this approach are as follows:

1. Four times more power can be imaged onto the FET-SEED arrays than the amplitude-division approach.
2. The number of aberrations are decreased because of the small beam-diameter.
3. A view-port is available at each stage to test and ease alignment.

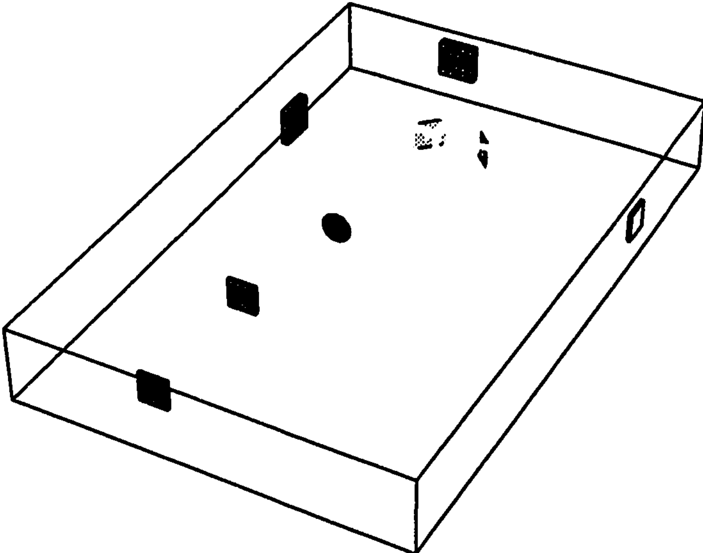


Figure 37.2: 3D-Layout of Pupil Division Beam Combination

Chapter 38

$System_5$ of AT&T Switching Fabrics

Chapter Abstract

$System_5$ is the last of the five switching fiber demonstrators implemented at AT&T. It has increased functionality and temporal bandwidth.

38.1 Introduction

The last of the AT&T switching fabrics is called $System_5$. This system implements a packet switched network. It uses FET-SEED smart pixels as switching nodes.

This Chapter describes the approach of the design in Section 38.2.1 and the detailed operation in Section 38.2.2. The architecture is also described using $\mathcal{OHDL}$.

38.2 Optical Implementation

38.2.1 Approach

Input:

Input matrix (input image) (16x32).

Output:

Output image: 3-way split diverging and interleaved 16x32 matrix

Function:

Achieves the same functionality as $System_4$ presented in Chapter 26. The only difference is the use of pupil division beam combination unit rather than the traditional amplitude-division approach.

The first stage of the system is used to latch and switch the incoming data that have been entered into the single mode fiber bundle. The last four stages are identical except for the optical interconnection which is a Banyan network. The output of the final stage is imaged onto a multi-mode fiber bundle to be delivered to the electronic output buffers.

38.2.2 Operation

Figure 38.1 shows the block diagram of the six-stage architecture.

The packet switching fabric ($System_5$), is designed to switch ATM like cells of data. The input cells initially enter and are stored by the electronic input buffers (double buffers). While in the input buffers, the destination address for all the cells are sent to the electronic control where a fast path-hunt algorithm developed for EGS networks

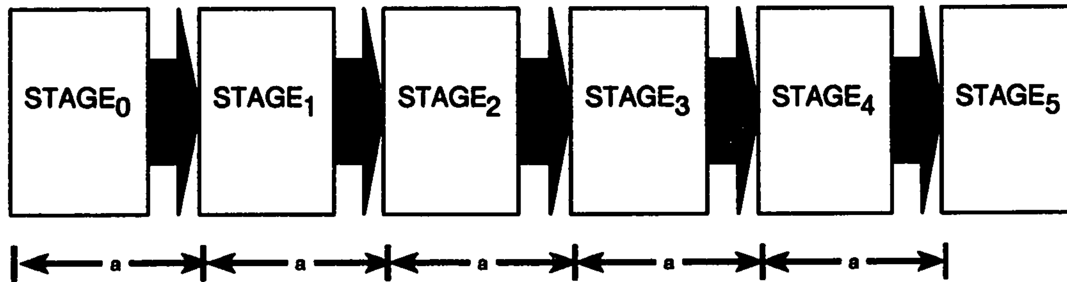


Figure 38.1: *System$_5$* of the AT&T Switching Fabric.

calculates the paths that need to be set up throughout the Banyan multistage network [15]. After the path-hunt is completed, a routing address is prepended onto each cell. This routing address includes $s-1$ "0" bits followed by s control bits. The transfer of information to and from the electronic control as well as the complete MIN path-hunt must be completed in one cell type (for ATM cells: 53 octets $\times$ 8 bits/octet/155 Mb/s $\sim 2.7\mu s$). Prior to sending the cells through the MIN, the paths need to be set up, this is accomplished by loading the s control bit into the control latches of the (2, 1, 1) smart pixels switching nodes [31]. This is accomplished by initially loading "0" bits, one stage at a time, into the control latches. This puts all the (2, 1, 1) nodes smart pixels into the stage where the A input channels are available to both smart pixel outputs and control latches. The s-embedded control bits are then serially shifted into the smart pixels of the s-stages. With this completed, all the paths are set up throughout the MIN allowing cells to be directed to their desired output buffers. This process is repeated for each cell entering the MIN.

38.3 OHDL Description of the Architecture

38.3.1 Initialization of the Simulator

In[1]:=

```
initialize[];
```

38.3.2 Architecture Information

In[2]:=

```
ArchitectureName = "System 5: Complete System";
Architect = "";
```

38.3.3 Components instantiation

Components instantiation

In[4]:=

```
{stage0, stage1, stage2, stage3, stage4, stage5} =
 GetObjects[6, "Assembly", CatalogNumber -> "A1"];
```

38.3.4 Placement Constraints

Parameters

In[5]:=

```
a = 0.1;
```

Constraints

In[6]:=
```
AddPlacementConstraint[default[stage0]];

 AddPlacementConstraints[{
 relativeTo[stage0, stage1, {a, 0, 0}],
 relativeTo[stage1, stage2, {a, 0, 0}],
 relativeTo[stage2, stage3, {a, 0, 0}],
 relativeTo[stage3, stage4, {a, 0, 0}],
 relativeTo[stage4, stage5, {a, 0, 0}]}];
```

In[8]:=
```
ComputePositions[];
```

38.3.5 Orientation Constraints

Constraints

In[9]:=
```
AddOrientationConstraints[{
 orientationOf[stage0, {angle[0], angle[0 Degree], angle[0]}]}
 ];

 AddOrientationConstraints[{
 parallelTo[stage1, stage0],
 parallelTo[stage2, stage0],
 parallelTo[stage3, stage0],
 parallelTo[stage4, stage0],
 parallelTo[stage5, stage0]}
 ];
```

In[11]:=
```
ComputeOrientations[];
```

38.3.6 Layout

In[13]:=
```
ShowArchitecture[];
```

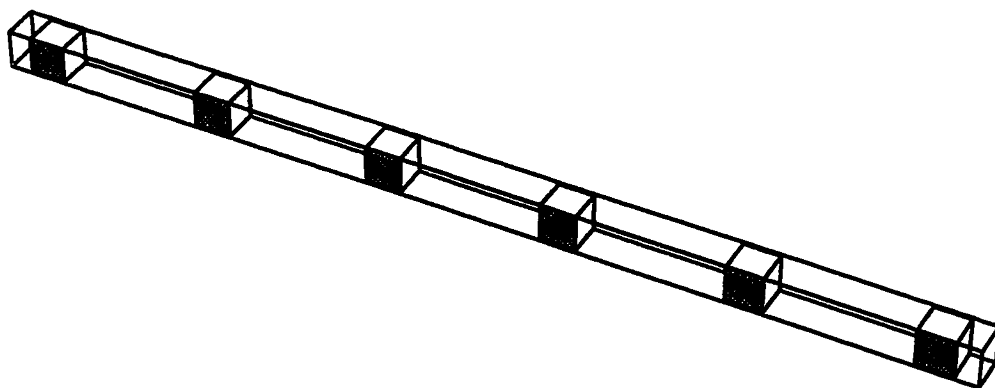


Figure 38.2: 3D-Layout of $System_5$ of AT&T Switching Fabrics

38.4 Conclusion

This Chapter presented the last of the five AT&T switching fabrics. This is called $System_5$. It was described using $\mathcal{OHDL}$ and the results were obtained as a 3D-layout.

A System for Prototyping Optical Architectures

Volume IV: Implementation

BY

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

December 1995

Preface

OptiCAD is a Computer Aided Design (CAD) system for designing optical architectures. It was designed and implemented to facilitate the design and verification of optical architectures.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system omitting the implementation details. It is however a complete document describing critical issues, design decisions and algorithms employed.

The second part contains several optical architectures designed and described using *OptiCAD*. These have been carefully chosen so as to illustrate most of the features of *OptiCAD*. This part essentially contains the user's view of the system.

The third part contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. Each notebook is presented as a self contained chapter. These include issues such as component modeling, component libraries, simulator and different kinds of analyses.

A rich collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. These utilities facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to be useful in other domains.

Reading through Parts III and IV is by itself expected to be an illustrative and interesting exercise in appreciating the power of functional programming in general and *Mathematica* in particular. The code is not necessarily efficient. Whenever there was a choice between efficiency and readability/clarity, the latter was preferred.

Part V is a reference manual for *OptiCAD* and the *OHDL* language.

Part III

An Implementation of OptiCAD in Mathematica

Chapter 39

Mathematica

Chapter Abstract
This Chapter attempts to provide various facets of this powerful tool. A study of this section should help in the understanding of the notebooks presented in this Volume.

Mathematica is a general software system for mathematical and other applications. Version 1 of *Mathematica* was announced during 1988. Even that version was heralded as a significant landmark in the history of computing tools. Based on the feedback received from a diversity of users, *Mathematica* was enhanced with a richer set of graphics primitives and other changes to make it Version 2, which was released in early 1991. At the time of this writing *Mathematica* is in its 2.2.2 stage.

Mathematica 2.2.2 is available on a number of platforms including Microsoft Windows, Windows NT, Apple Macintosh, numerous workstations (SUNs, $\mathcal{NEXT}$s, HP, IBM R6000 workstations, etc.), supercomputers like Cray, and parallel systems like Convex. Most of these versions support graphical front-ends with notebook (more

on this later) facility. The version under Microsoft DOS does not have a notebook facility.

It is not unreasonable to say that there is hardly a field of science/engineering which employs mathematical techniques that haven't been benefited by *Mathematica*. Even during the first two years of its public life, *Mathematica* was used in engineering, computer sciences, physical sciences, life sciences, business & social sciences in that order. Most of the use in the early stages is in the field of education in the above mentioned areas.

39.1 What is *Mathematica*?

As mentioned earlier *Mathematica* is a general purpose computer software system and a language intended for mathematical and other applications. *Mathematica* can be used as[84]:

- A **numerical** and **symbolic calculator** where one types in the questions and *Mathematica* prints out the answers.
- A **visualization system** for functions and data.
- A high-level **programming language** in which one can create programs, large and small.
- A **modeling** and **analysis** environment.
- A system for **representing knowledge** in scientific and technical fields.
- A **software platform** on which one can run packages built for specific applications.

- A way to create **interactive documents** that mix text, animated graphics, and sound with active formulae.
- A **control language** for external programs and processes.
- An **embedded system** called from within other programs.

These facets are elaborated and presented in some detail later in the subsequent sections.

One of the major assets of *Mathematica* is its choice of syntax and notation that by and large confirms to standard mathematical notation. The same set of conventions are methodically followed in all places. As a consequence, although it takes some time to get used to the style of *Mathematica*, it is remarkably consistent in the way it provides access to a number of builtin functions.

In general the computations of *Mathematica* can be divided into three classes: **numerical, symbolic**, and **graphical**. These three classes co-exist and to a user they appear in a unified manner. The following three sessions of *Mathematica* should illustrate these computations. All examples are run on *Mathematica* running on a $\mathcal{NEXT}$ machine.

39.2 Structure of *Mathematica*

Mathematica consists of two communicating modules, a front-end and a back-end. Normally, the user is unaware of the separation between the two as he/she will be communicating just with the front-end. The front-end is the one that translates user's requests to the language of *Mathematica* kernel (or back-end).

Example: Find the numerical value of log(4pi)

Log[4 Pi] is the *Mathematica* version of log(4pi). The N tells *Mathematica* that you want a numerical result.

```
In[1]:=
    N[ Log[ 4 Pi ] ]
Out[1]=
    2.53102
```

Here is log(4pi) to 40 decimal places.

```
In[2]:=
    N[ Log[ 4 Pi ], 40 ]
Out[2]=
    2.531024246969290792977891594269411847798
```

100%

Figure 39.1: Snapshot of *Mathematica* Session Illustrating its Computing Facilities

Mathematica implementations under advanced environments provide the ability to provide linking of several back-ends (kernels) that may be running on different machines with the front-end. This is illustrated in Figure 39.4.

More details about linking user developed front-ends or computational software in other languages under different platforms are discussed in a later subsection on distributed computation.

With respect to the structure of *Mathematica* back-end itself, one could imagine it to have four layers as shown in Figure 39.5.

Mathematica kernel is based on a rich collection of builtin functions. There are over a thousand builtin functions; this should be contrasted with no more than 50 mathematical routines provided by many high-level languages. In addition most of these builtin functions are designed to work together in a unified manner. They also

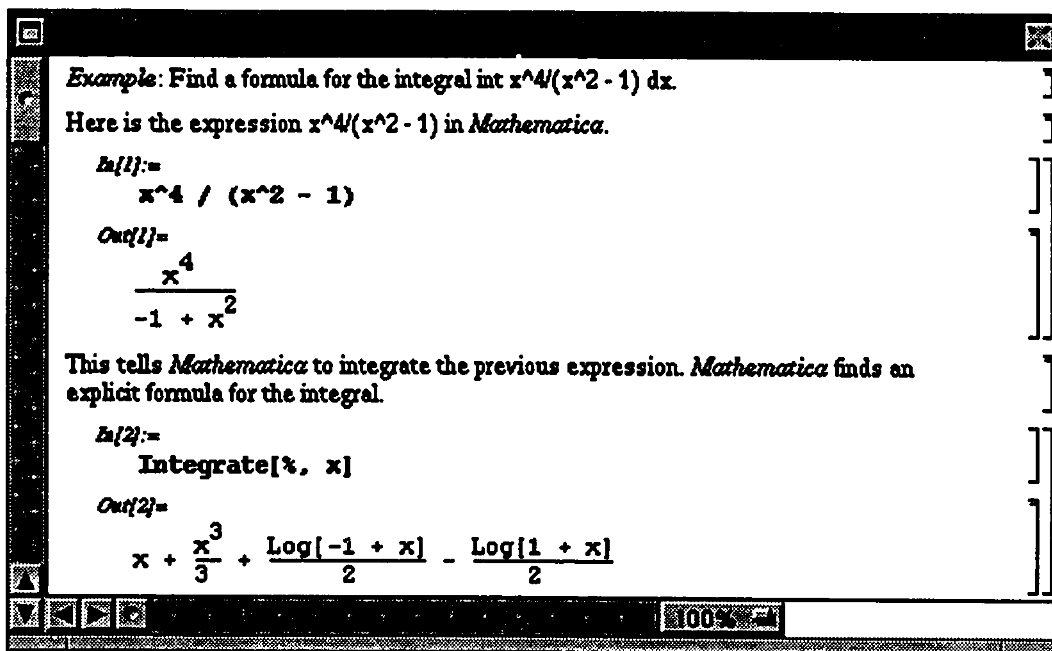


Figure 39.2: Snapshot of *Mathematica* Session Illustrating its Symbolic Computation Power

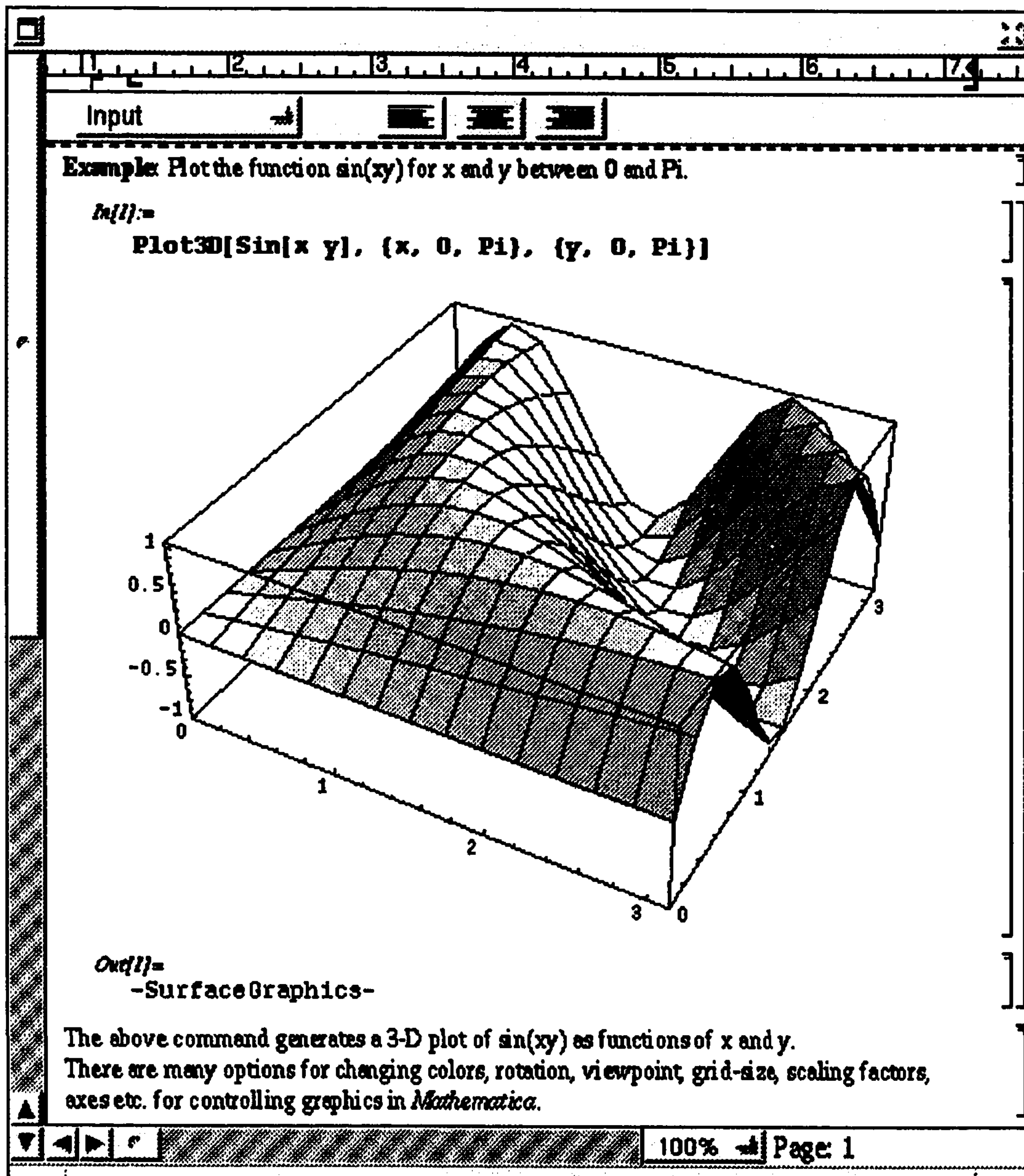


Figure 39.3: Snapshot of *Mathematica* Session Illustrating its Graphics Capabilities

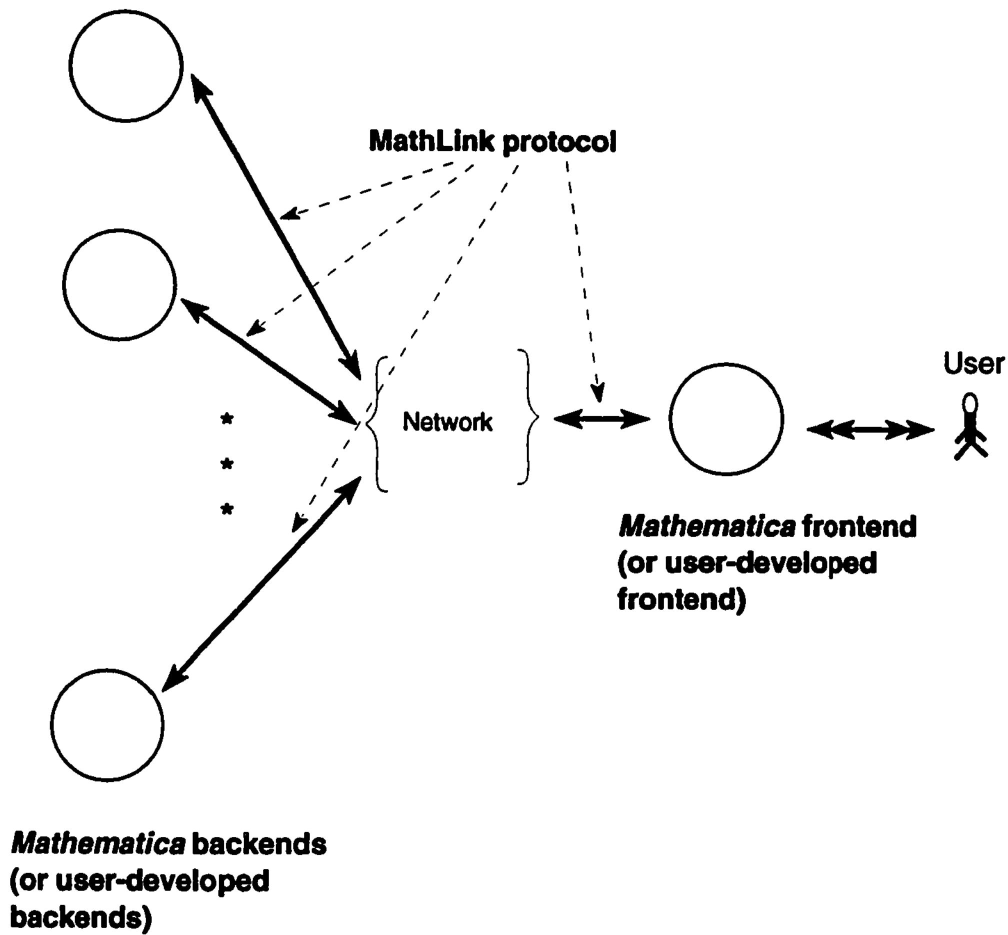


Figure 39.4: *Mathematica* Front-End, Back-Ends and their Interaction.

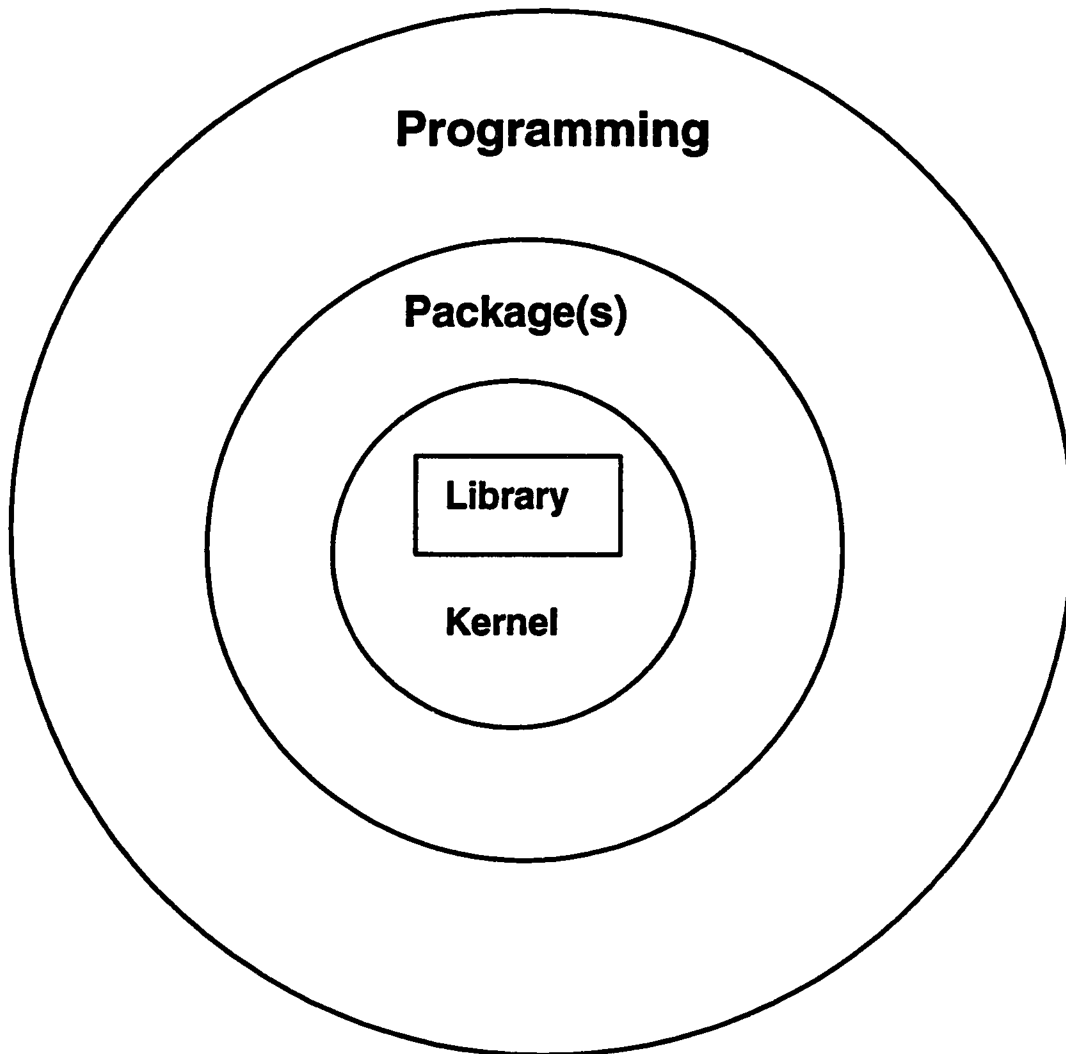


Figure 39.5: Hierarchy of Virtual *Mathematica* Back-End Servers.

integrate numerical, symbolic, and graphical computation in a smooth manner. The kernel itself consists of the builtin-function library and a very powerful language. The *Mathematica* language and the details of the packages are taken up in later sections.

39.3 *Mathematica* as an Interactive Calculator

In its simplest form of usage *Mathematica* can be thought of as an interactive calculator. The user types in expressions involving builtin functions, operands, control-structure-functions and *Mathematica* responds to that expression by its evaluation. The user submitted expressions could produce numerical results, which is what most users need; they could also produce symbolic results; or even graphical plots. The sessions shown earlier and Parts III and IV illustrate *Mathematica*'s usage as an interactive calculator.

39.4 *Mathematica*'s Packages

No system, however extensive it may be, can continue to be useful unless it is extensible. One of the major features of *Mathematica* system is its extensibility. In addition to providing a substantial base in the form of its kernel functions, it also provides a collection of packages.

A package is a collection of functions written in *Mathematica* language, neatly encapsulated into its own context. When *Mathematica* is loaded for execution, it has the functionality of its kernel. Should we require additional domain specific knowledge, we would have to "teach" it to *Mathematica*. This is often done by loading the appropriate package:

$<<$ *package-class`package-name`*

This will have the effect of reading and processing the definitions of functions and any other domain-specific knowledge from a file *package-name*.m in the directory *package-class* of the standard packages.

There are a number of packages. In order to have some structure, these packages are grouped together by their fields into the following classes: `Algebra`, `Calculus`, `DiscreteMath`, `Examples`, `Geometry`, `Graphics`, `LinearAlgebra`, `MathFunctions`, `Miscellaneous`, `NumberTheory`, `NumericalMath`, `ProgrammingExamples`, `Statistics`, `Utilities` .
Each of these categories/classes contain a number of related packages. For instance, the category `DiscreteMath` contains the packages `CombinatorialFunctions.m`, `CombinatorialSimplification.m`, `Combinatorica.m`, `ComputationalGeometry.m`, `Permutations.m`, `RSolve.m` , and `Tree.m`. Some of the packages are fairly extensive and implement a number of functions that would otherwise take years of development. For instance the package `Combinatorica.m` alone implements numerous functions related to combinatorics and almost all commonly used graph algorithms.

Once, a package is loaded into *Mathematica*, it learns the domain specific information. From that point on, the interaction with *Mathematica*, can be at a level where *Mathematica*, can be treated as a system implemented exclusively for the domain under consideration. A brief description of the functionality of each category is presented in Section 39.16. For a comprehensive view of the functionality of each package the reader is referred to [75].

39.5 *Mathematica* for Symbolic Computation

Most of *Mathematica*'s symbolic capabilities are based on algebraic simplification routine `Simplify`, discrete structure handling, and the knowledge of a number of asymptotic results.

39.6 Programming in *Mathematica*

There are four widely accepted programming paradigms. These are:

1. procedural or imperative
2. functional
3. logic programming
4. object-oriented

In `imperative` or `procedural` languages the user of the language specifies step by step computation that needs to be performed on the state of the machine to achieve the next step. Most conventional languages (e.g. Fortran, COBOL, ALGOL, PL/I, C, Pascal, Ada, Modula) belong to this class. One of the reasons for the preponderance of this class of languages is that they are fairly efficient, and they are tailored to the conventional machines almost all of which are based on Von Neumann architecture.

In `functional` languages like LISP, FP, the whole computation can be thought of as a sequence of function compositions or applications. The data supplied by users will be passed as arguments to functions, whose results are passed as arguments to other functions and so on, until the desired result is achieved. The primary motivation for the development of functional style languages is the clarity and ease with which

correctness-of-programs can be established, and more recently the ability to better utilize the hardware parallelism. These languages are growing in their importance with the availability of cheap parallel processor systems.

In `logic programming` paradigm, the whole computation can be seen as one of deducing facts and establishing proofs for claims using the available fact base. The primary candidate that belongs to this paradigm is `PROLOG`. These languages allow very high-level specification and provide powerful search strategies thus relieving the user from detailed sequencing, so that he can concentrate on specification.

In `Object-oriented` languages, the computation model is based on a collection of autonomous and anonymous objects that respond to a sequence of messages. For each message an object invokes an operation or computation that manipulates the private data held by that object and returns the result as a message. Objects of the same kind form a class. All objects of a class share the methods / procedures of that class. Reusability of code is encouraged and promoted by code-sharing achieved by inheritance. The three major features of object-orientation are: data abstraction, encapsulation, and inheritance. Even advanced imperative languages like Ada and Modula support only data abstraction. Languages like Smalltalk, EIFFEL, Objective C, *C++* are typical object-oriented languages.

It is not hard to see, why *Mathematica* is a precursor to the next generation high-level languages that incorporate different programming paradigms in a single language, making it natural to use for different kinds of problems. Beginners who are familiar with languages like C, tend to use *Mathematica* in an imperative style. As they use *Mathematica* more and more the usage slowly evolves to be one of functional style which is the most convenient style advocated by *Mathematica*. Yet *Mathematica* supports very powerful rewriting mechanisms and rule-processing, which makes it

very elegant to develop rule-based programs. The same can be used to realize object-oriented features or inheritance.

39.7 *Mathematica* as a Visualization System

Mathematica has a very powerful set of visualization primitives. The rendering features, extensive builtin mathematical functions, and the ability to realize arbitrary functions easily make *Mathematica* particularly suitable to display complex functions and manipulate them in different forms. There are also facilities for animating complex fields, and time-varying functions. The 2D and 3D graphics notebooks in Part IV attempt to give a flavor of this, although the best means of finding out the power is to actually use *Mathematica* for visualization purposes.

39.8 *Mathematica* for Distributed Computation

Most of the time users would probably be using *Mathematica* on a single machine. However, should it be necessary to split the computation on to several machines, it is possible to invoke several different programs on different machines in a network all running at the same time. All these programs could work on different portions of data; they can also work in a co-operative manner to perform the desired computation. A subset of these programs can be *Mathematica* kernels that can serve as general purpose algebraic, symbolic, and graphical calculators. Others could be customized modules.

In order to achieve this kind of co-operative computing, it is necessary for these various modules to communicate with each other. *MathLink* is a communication protocol that enables different modules to exchange expressions and messages. These

messages are encoded as ASCII text. *MathLink* allows the specification of data, arguments, commands, and programs to be passed among the modules. There are several levels of abstraction for the *MathLink* protocol. Although the higher level *MathLink* is easier to use, it is restrictive. Application developers who wish to develop distributed applications involving extensive mathematical computations would be benefited greatly by falling back on the extensive computational support of *Mathematica* kernel. However they need to follow the *MathLink* protocol as documented in [72].

One can use *MathLink* to do several things, some of which are[72]:

1. Call *Mathematica* from within an external program
2. Call an external program from within *Mathematica*
3. Implement one's own front-end for *Mathematica*
4. Exchange data and commands between *Mathematica* and external programs
5. Exchange expressions between concurrent/distributed *Mathematica* processes.

39.9 Integration and Interaction with Other Software Systems

Mathematica has rich 3D graphics facilities. So long as they are used within *Mathematica* the internal form of *Mathematica* is good enough. However, when it is necessary to port the graphics produced by *Mathematica* to other software packages or systems, it is necessary to generate them in some standard format. *Mathematica* can be used to

produce these graphics either in the postscript language or in a format called 3-Script that is developed by WRI[73].

Mathematica can also convert its internal representation of expressions into TEX for possible inclusion in documents, FORTRAN and C for possible inclusion in programs of the respective languages.

39.10 *Mathematica* as an Exploratory Tool for Scientists & Engineers

Traditionally there is a substantial gap between the knowledge representation as collection of formulae in books and research articles on one hand and programs that compute them on the other hand. Often a scientist or an engineer is at a disadvantage in trying to keep these two diverse ways of knowledge representation. Unless a scientist/engineer is well versed with the programming language, it is easy to get intimidated by the detail of the program. Likewise, unless the person is knowledgeable with the subject area, the codified knowledge in the form of formulae could be difficult to follow. *Mathematica* provides a new way of integrating and bridging the gaps between these two extreme ways of representing the knowledge, both of which are necessary.

First by providing a rich collection of builtin functions and a very high level language it substantially increases the level of abstraction in specifying computation. Secondly, by providing fine front-ends and notebook facility (see the subsection on Interfaces and Notebooks) it allows the documentation and verbal description in a supplementary role.

Traditional methods of using prepackaged software can give relations and results, but typically cannot capture the calculations that lead to them. In contrast, in a *Mathematica* program the computation is transparent. The computer model development, exploration, validation, and documentation of the system under consideration go hand in hand together. Models for components of a system are put together by experimenting with *Mathematica* by a sequence of computations and experiments. Once the proper computation is understood, they can be assembled to provide the computation model for the system under consideration.

39.11 *Mathematica* as a Multi-Media System

The ability to view dynamic effects with an animated sequence of graphics often enhances the comprehension. *Mathematica* achieves animation by producing different snapshots in quick succession. In particular a function `SpinShow` works by showing the image from different points of view thus enhancing the depth effect.

The ability to produce pure sounds at user specified frequencies, mixing them digitally, and playing different (mathematical) functions is an intriguing thought that was made into a reality in *Mathematica.* The *Mathematica* function `Play` is an analog of `Plot`. `Play` can play different functions, i.e., take different functions as arguments, compute them and play sounds according to the values of those functions. Sampling rate, tones of pitch, volume, frequencies of the sound can be varied. The ability to produce and synthesize sounds coupled with digital and mathematical editing facilities are expected to open up a new domain in incorporating the sound dimension into the realm of computation.

39.12 *Mathematica* as an Education Tool

There has been growing attention to put *Mathematica* to use in teaching a number of subjects. A number of courses in Mathematics and Engineering are being redesigned so that the emphasis is on learning through explorations with *Mathematica*. Books on calculus, differential equations, numerical analysis, differential geometry have been and are being written with *Mathematica*. There have even been attempts to introduce *Mathematica* for high-school kids and even six-year old students. The motivation for these attempts parallels that of Maria Montessori's contention that the human thinking and abstraction depend very much on the foundation and learning of the shapes and sizes of concrete objects in the childhood.

It remains to be seen to what extent these efforts contribute towards better learning.

39.13 *Mathematica* Interfaces & Notebooks

Mathematica has two kinds of interfaces. The first and simplest is based on character interface such as one provided by DOS. But the more widely used interface is one that is available on NEXTs, Macintoshes, and Suns (X-window based). This latter interface provides the notebook facility. The front-end is very much like a typical word processor. It keeps track of all input and output in various cells. It allows editing on the input cells. Once the user asks for the evaluation of an input cell, the front-end dispatches the contents of the input cell (using *MathLink*) to the back-end and gets the results back and displays them in the document. In a way the notebook can be thought of as a multi-media document, in the sense that it integrates text, graphics,

animations, and sound all into a single document. Such notebooks can be saved and transferred from one system to another with ease, because the description/contents are codified in ASCII.

39.14 Advanced Issues of *Mathematica*

In its normal mode of computation *Mathematica* interprets the programs. If some computation needs to be done several times or if the computation is taking too much time, then it is advantageous to make use of the compilation feature of *Mathematica*. There is a builtin function called `Compile[]` that takes expressions and returns a compiled object. The compiled object includes instructions close to the machine level. A number of builtin functions such as `NIntegrate[]` call `Compile[]` themselves.

The reader is referred to [74] for the instruction set used by the *Mathematica* compiler and for the representation details of *Mathematica* compiled expressions.

General computation principle of *Mathematica* is to: *take any expression and apply transformation rules until the result no longer changes.*

39.15 Limitations of *Mathematica*

All the aforementioned advantages of *Mathematica* do not come for free. When using *Mathematica* it is easy to get carried away and hope to achieve results that are way beyond the means of the hardware and software. Complex computations require significant amounts of computer time, memory or both. The user has to keep in mind the physical limitations of the underlying hardware in his/her expectations. For instance on a machine with 16MB RAM with a notebook interface, it is unreasonable

to expect to have several dozen 3D animations, or symbolic solutions to a system of several hundred equations with several hundred variables, or applying recursion to depths in several thousands, finding inverses for a thousand by thousand matrix. The precision of computation will also effect both the memory usage and computer time. The user has to develop a feel for what is "reasonable" to hope for in view of the nature of computations he/she is attempting to carry out using *Mathematica.*

In addition to the above limitations — and as is the case with any large software system — there are bugs in *Mathematica.* Most widely used cases are rather well debugged; but subtleties, and intricacies that arise in mathematical reasoning may not subject to themselves to algorithmic approaches all the time. It is perhaps this reason that contributes to most of *Mathematica*'s bugs. However, there is an active support by Wolfram Research Inc., in addressing bugs and responding to them. But, unlike other systems there are two distinct advantages in using *Mathematica* in this regard. First, due to the many possible ways in which one could express a computation in *Mathematica*, it is often very easy to get around one or more buggy builtin functions. Second, due to the interactive nature, it is very easy to use *Mathematica* to validate its own computation.

39.16 Standard Package Categories & Their Functionalities

The various categories of packages that come with the standard distribution are mentioned below alphabetically.

Packages from the category `Algebra` have functions that can: count the roots of

an arbitrary real polynomial in a given interval, carry out real/imaginary functional computation, carry out symbolic summation, and conduct trigonometric simplifications.

Packages from the category `Calculus` have functions that can compute/carry-out: Fourier transforms, Laplace transforms and their inverses, Pade approximations, and Vector analysis.

Packages from the category `Discrete Mathematics` have functions that can: compute Catalan Numbers, various kinds of factorials and other combinatorial functions, carry out combinatorial simplifications, realize enumerative combinatorics and graph theory, support computational geometry operations, permute objects, solve recurrence relations, and display graphical representations of trees and graphs.

Packages from the category `Geometry` have functions that can: give geometrical characteristics of regular polygons and polyhedra.

Packages from the category `Graphics` have functions that can: animate graphics, draw various kinds of 2D and 3D plots, and display surface and parametric plots.

Packages from the category `Linear Algebra` have functions that can: decompose matrices into a number of simpler matrices of various types (QR, Shur, Cholesky, LU, etc.), carry out Gaussian elimination, manipulate matrices, take projections and carry out normalization of vectors.

Packages from the category `Miscellaneous` have all sorts of functions that can: compute various kinds of calendar operations, provide basic properties of different chemical elements, illustrate database operations on city data, calculate distances between places on earth, provide numerous physical constants and the units of conversion, and even draw maps of various parts of the world in addition to providing statistical information about various countries.

Packages from the category `Number Theory` have functions that can: give decimal expansions of real numbers, factor integers by elliptic curve method, test for primality and generate certificates for primality, even compute some of Ramanujan's identities, find polynomials of integer coefficients with the specified approximate zero.

Packages from the category `Numerical Math` have functions that can: carry out interpolations, compute MiniMax approximations, realize higher order Runge-Kutta methods, simulate computer arithmetic, perform numerical integration (Gaussian, Newton-Cotes), conduct interval arithmetic, integrate lists, evaluate limits, and polynomially fit curves.

Packages from the category `Statistics` have functions that can: compute confidence intervals for various parameters such as means, variances, ratios of variance, compute densities, means, variances and related properties of continuous and discrete distributions, locate and disperse shapes of various distributions, test various hypotheses concerning mean and variance, generate various kinds of pseudo random numbers of various distributions, and fit least square approximations.

Packages from the category `Utilities` have functions that can: filter out and pass the correct parameters to a sequence of nested functions, provide support for restricted non-English alphabets, turn on/off the display of the timing information for each command.

Packages from the category `Examples` contain various examples of applications of *Mathematica* and styles of programming.

The various examples from the category `Programming Examples` are the programs from Maeder's second edition: "Programming in *Mathematica*".

In this Chapter we have attempted to provide a multi-faced view of the tool *Mathematica*. Each and every facet demands coverage that far exceeds a book on

its own. Whatever features one may like/dislike of *Mathematica*, it is hard to find a person who will not be benefitted by the power of *Mathematica*. In particular in scientific/engineering/educational fields where there is quite a bit of experimentation, *Mathematica* can save substantial amount of time. The remaining Chapters in this Part give a number of sessions that illustrate some features of *Mathematica*.

Chapter 40

Global Variables & System Initialization

Chapter Abstract

This notebook should be loaded immediately after loading the Mathematica kernel. It will load the necessary definition files and declare global variables to be used by $\mathcal{OHDL}$.

Global Variables & System Initialization

This notebook should be loaded immediately after loading the Mathematica kernel.
It will load the necessary definition files and declare global variables to be used by *OHDL*.

```
$PrePrint = (StringJoin["[ Memory in Use = <",
    ToString[MemoryInUse[]], ">]\n\n", ToString[#]])&
```

```
[ Memory in Use = <650208>]
[ Memory in Use = <<>ToString[MemoryInUse[]]<>>]<>ToString[#1] &
```

```
OHDL$Prefix = "OHDL";
OHDL$Prefix = "";
```

```
OHDL$PathPrefix = "~";
OHDL$PathPrefix = "/tmp/ocg";
```

```
OHDL$Debug = True;
```

```
AppendTo[$Path, StringJoin[OHDL$PathPrefix, "/Packages"]]
```

```
{., ~, ~/Library/Mathematica/Packages,
  /NextApps/Mathematica22.app/Library/Mathematica/Packages,
  /NextLibrary/Mathematica/Packages,
  /NextApps/Mathematica22.app/Install/Preload,
  /NextApps/Mathematica22.app/StartUp, /tmp/ocg/Packages}
```

```
Clear[OHDL$ContextNames, OHDL$FullList, OHDL$ContextDependencies]
```

```
OHDL$initialize[]:=
Module[{filesToLoad},
    filesToLoad = {"Utilities/OptionProcessing",
                   "Utilities/StringOperations",
                   "Utilities/SystemsProgramming",
                   "Translators/NotebookToPackage"
    };
    Map[Get[StringJoin[OHDL$PathPrefix, "/Packages/", #, ".m"]]&,
        filesToLoad]
]
```

```
OHDL$initialize[]
```

```
[ Memory in Use = <651780>]
OHDL$initialize[]
```

Chapter 41

Generic & Catalog Component Databases

Chapter Abstract

This notebook contains the two databases - one is the generic database which contains the attributes related to a component. The other database (vendor provided) contains fine tuned attributes as provided by the vendor.

Generic & Catalog Component Databases

This Notebook contains the two databases -- one is the generic database which contains the attributes related to a component.
The other database (vendor provided) contains fine tuned attributes as provided by the vendor.

Generic Data Bases

```
(* Data base of one entry for each component of a distinct type *)
genericComponentDataBase := {
    {ComponentType -> "BeamSplitter",
        Size -> {2 cm, 2 cm, 2 cm}, Angle -> 45 degree, Material -> "BK7",
        RefractiveIndex -> 1.5,
        Reflectance -> 60 percent, Transmission -> 40 percent},
    {ComponentType -> "ConvexLens",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 0.2 cm}, FocalLength -> 1 cm},
    {ComponentType -> "ESLM",
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz",
        RefractiveIndex -> 1.5,
        Reflectance -> 5 percent, Transmission -> 95 percent},
    {ComponentType -> "HalfWavePlate",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "Hologram",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "InterferenceFilter",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "LiquidCrystalTV",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "Mirror",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 0.2 cm}},
    {ComponentType -> "MemoryFilter",
```

```
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "PlanoConvexCylindricalLens",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 0.2 cm}, FocalLength -> 1 cm},
    {ComponentType -> "PolarizingBeamSplitter",
        Size -> {2 cm, 2 cm, 2 cm}, Angle -> 45 degree, Material -> "Quartz",
        RefractiveIndex -> 1.5,
        Reflectance -> 60 percent, Transmission -> 40 percent},
    {ComponentType -> "PulsedLaser",
        Size -> {2 inch, 2.5 inch, 7 inch}, OutputPower -> 0.5 mW,
        BeamDiameter -> 0.48 mm,
        RefractiveIndex -> 1.5,
        BeamDivergence -> 1.7 mrad, PulseDuration -> 2 ps, DutyRatio -> 0.1},
    {ComponentType -> "QuarterWavePlate",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {ComponentType -> "SEED",
        Size -> {2 cm, 2 cm, 2 cm}, Elements -> 4,
        Material -> "Quartz",
        RefractiveIndex -> 1.5,
        Reflectance -> 5 percent, Transmission -> 95 percent},
    {ComponentType -> "SLM",
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz",
        RefractiveIndex -> 1.5,
        Reflectance -> 5 percent, Transmission -> 95 percent},
    {ComponentType -> "SSEED",
        Size -> {2 cm, 2 cm, 2 cm}, Elements -> {4, 4},
        Material -> "Quartz",
        RefractiveIndex -> 1.5,
        Reflectance -> 5 percent, Transmission -> 95 percent},
    {ComponentType -> "WavePlate",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"}
};
```

Catalog Databases
One entry per catalog component

```
catalogComponentDataBase := {
    {CatalogNumber -> "BS1", ComponentType -> "BeamSplitter",
        Size -> {2 cm, 2 cm, 2 cm}, Angle -> 45 degree, Material -> "BK7",
        Reflectance -> 60 percent, Transmission -> 40 percent},
    {CatalogNumber -> "CL1", ComponentType -> "ConvexLens",
        Size -> {2 cm, 2 cm, 2 cm}, FocalLength -> 1 cm},
    {CatalogNumber -> "ESLM1", ComponentType -> "ESLM",
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz",
        Reflectance -> 5 percent, Transmission -> 95 percent},
    {CatalogNumber -> "HWP1", ComponentType -> "HalfWavePlate",
        Size -> {2 cm, 2 cm, 0.5 cm}, Material -> "Quartz"},
    {CatalogNumber -> "H1", ComponentType -> "Hologram",
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {CatalogNumber -> "IF1", ComponentType -> "InterferenceFilter",
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {CatalogNumber -> "M1", ComponentType -> "Mirror",
        Size -> {2 cm, 2 cm, 0.2 cm}},
    {CatalogNumber -> "MF1", ComponentType -> "MemoryFilter",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
    {CatalogNumber -> "LQTV", ComponentType -> "LiquidCrystalTV",
        RefractiveIndex -> 1.5,
        Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"},
```

```
{CatalogNumber -> "PCCL1", ComponentType -> "PlanoConvexCylindricalLens",
    Size -> {1 cm, 1 cm, 2 cm}, FocalLength -> 1 cm},
{CatalogNumber -> "PBS1", ComponentType -> "PolarizingBeamSplitter",
    Size -> {2 cm, 2 cm, 2 cm}, Angle -> 45 degree, Material -> "Quartz",
    Reflectance -> 60 percent, Transmission -> 40 percent},
{CatalogNumber -> "PL1", ComponentType -> "PulsedLaser",
    Size -> {3 cm, 2.5 cm, 4 cm}, OutputPower -> 0.5 mW,
    BeamDiameter -> 0.48 mm,
    BeamDivergence -> 1.7 mrad, PulseDuration -> 2 ps, DutyRatio -> 0.1},
{CatalogNumber -> "PL2", ComponentType -> "PulsedLaser",
    Size -> {3 cm, 2.5 cm, 4 cm}, OutputPower -> 1 mW,
    BeamDiameter -> 0.48 mm,
    BeamDivergence -> 1.7 mrad, PulseDuration -> 2 ps, DutyRatio -> 0.1},
{CatalogNumber -> "PL0", ComponentType -> "PulsedLaser",
    Size -> {2 inch, 2.5 inch, 7 inch}, OutputPower -> 1 mW,
    BeamDiameter -> 0.48 mm,
    BeamDivergence -> 1.7 mrad, PulseDuration -> 2 ps, DutyRatio -> 0.1},
{CatalogNumber -> "QWP1", ComponentType -> "QuarterWavePlate",
    Size -> {2 cm, 2 cm, 0.3 cm}, Material -> "Quartz"},
{CatalogNumber -> "SEED1", ComponentType -> "SEED",
    Size -> {2 cm, 2 cm, 2 cm}, Elements -> 4,
    Material -> "Quartz",
    Reflectance -> 5 percent, Transmission -> 95 percent},
{CatalogNumber -> "SLM1", ComponentType -> "SLM",
    Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz",
    Reflectance -> 5 percent, Transmission -> 95 percent},
{CatalogNumber -> "SSEED1", ComponentType -> "SSEED",
    Size -> {2 cm, 2 cm, 2 cm}, Elements -> {4, 4},
    Material -> "Quartz",
    Reflectance -> 5 percent, Transmission -> 95 percent},
{CatalogNumber -> "WP1", ComponentType -> "WavePlate",

    RefractiveIndex -> 1.5,
    Size -> {2 cm, 2 cm, 2 cm}, Material -> "Quartz"}

};
```

Chapter 42

Icons for Optical Components

Chapter Abstract

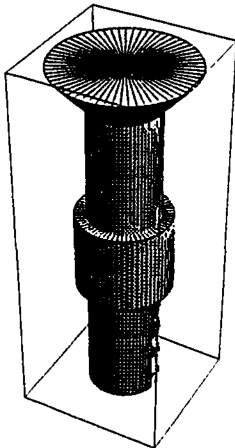
This notebook contains the necessary code to create icons for optical components. These are created by using the generic graphics shapes provided by other notebooks. The routines are general and the icons can be customized to a specific size, resolution and orientation.

Icons for Optical Components

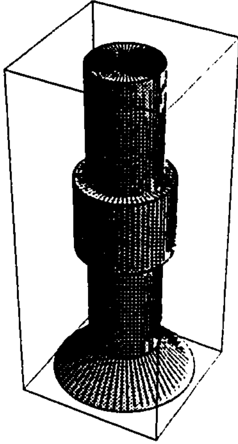
This notebook contains the necessary code to create icons for optical components.
These are created by using the generic graphics shapes provided by other notebooks.
The routines are general and the icons can be customized to a specific size, resolution and orientation.

```
(* Returns an icon of a stand constructed from
       Mathematica Graphics primitives *)
Clear[stand];
stand[height_:2, positionedHeight_:1.5, theta_:0.5]:=
Module[{verticalBar, base, sleeve},
    base = truncatedCone[height/5, height/10, height/9, theta,
            BottomCover -> True];
    verticalBar =  transformMma3D[
                       cylinder[height/10, 9/10 height, theta,
                              TopCover -> True],
                       {translateBy[{0, 0, height/10}]}
                 ];
    sleeve =    transformMma3D[
                   cylinder[height/7.5, 9/40 height, theta,
                       TopCover -> True, BottomCover -> True],
                   {translateBy[{0, 0, positionedHeight-1/2 9/40 height}]}
                 ];
    {base, verticalBar, sleeve}
]
```

```
Show[ Graphics3D[
        transformMma3D[ stand[],
                       {reflectThroughXY[]}
        ]
    ]
];
```



```
Show[ Graphics3D[stand[] ] ];
```



```
<<Users/Atif/OldStuff/Interpreter/ferozeobjects.m
```

```
(* Returns an icon of a laser constructed from Mathematica
       Graphics primitives *)
Clear[laser];
laser[height_:2, positionHeight_:1.5, laserLength_:1.5, theta_:0.5] :=
  Module[{},
    laserStand = stand[height, positionHeight, theta];
    laserBar = transformMma3D[ cylinder[laserLength/10, laserLength, theta,
                                       TopCover -> True, BottomCover -> True
                              ],
                              {rotateAroundYBy[angle[90 degree]],
                               translateBy[{-laserLength/3, -height/7.5,
                                       positionHeight}]
                              }
                 ];
    {laserStand, laserBar}
];

laser[{laserLength_, ignoredforthetimebeing_, height_}] :=
    laser[height, N[2/3 height], laserLength, 0.5]

Clear[PlanoConvexCylindricalLens];
PlanoConvexCylindricalLens[{length_, width_, height_}]:=
    slicedCylinder[length/2, height,
        N[Pi / 3],
        (* ArcCos[ Min[1, Abs[1-2(width/length)^2] ] ],  *)
        0.05]

Clear[SLM];
SLM[{length_, width_, height_}] :=
    seedgraphicsobject[ 5, 5, 0.001, 0.001, 0.02 ]
```

```
Clear[SSEED];
SSEED[{length_, width_, height_}] :=
    seedgraphicsobject[ 5, 1, 0.001, 0.001, 0.02 ]

Clear[SEED];
SEED[{length_, width_, height_}] :=
    seedgraphicsobject[ 4, 4, 0.001, 0.001, 0.02 ]

Clear[ConvexLens];
ConvexLens[{length_, width_, height_}] :=
    lensTwo[length/2, 0.5]

Clear[QuarterWavePlate];
QuarterWavePlate[{length_, width_, height_}] :=
    wavePlate[length/2, 0, 0.003, 0.5]

Clear[HalfWavePlate];
HalfWavePlate[{length_, width_, height_}] :=
    wavePlate[length/2, 0, 0.006, 0.5]
```

Chapter 43

draw[self] for different optical components

Chapter Abstract

Each optical component has a draw[self] definition that it calls to render a 3D icon. This notebook serves as a database that provides these definitions.

draw[self] for Different Optical Components

Each optical component has a draw[self] definition that it calls to render a 3D icon.
This notebook serves as a database that provides these definitions.

■ Draw Self Definitions for Library Components

```
Clear[draw];

draw[InterferenceFilter[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                        /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[interferenceFilter[size],
                                        N[{thetaX, thetaXY,
                                                thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]


draw[BeamSplitter[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                        /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[splitter[First[size]],
                                        N[{thetaX, thetaXY,
                                                thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]

draw[PolarizingBeamSplitter[object_]]:=
        draw[BeamSplitter[object]];
```

```
draw[Mirror[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[mirror[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[Hologram[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[hologram[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[Laser[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[laser[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ];
draw[PulsedLaser[object_]]:= draw[Laser[object]];
```

```
draw[SLM[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                        /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[SLM[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[ConvexLens[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                        /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];    *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[ConvexLens[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[HalfWavePlate[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];    *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[HalfWavePlate[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]

draw[QuarterWavePlate[object_]] :=
    draw[HalfWavePlate[object]]
```

```
draw[SSEED[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[SSEED[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[SEED[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[SEED[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
draw[PlanoConvexCylindricalLens[object_]] :=
    Module[{size, position, orientedObject},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                            /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[PlanoConvexCylindricalLens[size],
                            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

```
(* default draw for those objects that don't have any special shape *)
draw[fullObject_] /; Head[fullObject] =!= List :=
    Module[{size, position, orientedObject, object = headLess[fullObject]},
        size = Size /. object;
        (* lower left position *)
        position = (Variables /. object)[[Range[Length[Origin]]]]
                    /. Coordinates;
        (* extract the orientation angles *)
        {thetaX, thetaXY, thetaP} =
                (Variables /. object)[[3+Range[Length[Origin]]]]
                        /. Orientations;
            (* Print[{thetaX, thetaXY, thetaP}];   *)
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant
                        rule for some theta *)
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];
            (* Print[{thetaX, thetaXY, thetaP}]; *)
        orientedObject = orient[cuboid[{0, 0, 0}, size],
            N[{thetaX, thetaXY, thetaP}]];
        transformMma3D[orientedObject, {translateBy[position]}]
    ]
```

Chapter 44

Simulator

Chapter Abstract

This is an experimental notebook containing code from various other notebooks. This notebook provides the basic framework which was later used to design the simulator template.

Simulator

This is an experimental notebook containing code from various other notebooks.
This notebook provides the basic framework which was later used to design the simulator template.

Support functions for OHDL

Implementation Support Functions

```
(* The main driver that calls component specific draw[self]. *)
Clear[drawDriver];
drawDriver[object_]:=
    Module[{type},
        type = ComponentType /. object;
        type = ToExpression[type];
        draw[type[object]]
    ]
```

```
(* User called function that reads the objects' orientations and positions
    and generates the whole layout as a 3D graphics. *)
GenerateArchitecture[]:=
        If[Orientations =!= {} && Coordinates =!= {},
            GraphicsObject = Graphics3D[
                                Map[drawDriver[#]&, CurrentObjects ]
                               ];
            GraphicsObjectValid = True,
            Print["Error :: Could not Generate Architecture. "];
            Print["\t\t Insufficient Position and Orientation information"]
        ]
```

```
(* Calls GenerateArchitecture[] if no graphics has been generated. Then calls
    ViewArchitecture[]. *)
ShowArchitecture[options___]:=
Module[{},
    If[! GraphicsObjectValid,
        GenerateArchitecture[]
    ];
    If[! GraphicsObjectValid,
            Print["Error :: Could not Generate Valid Graphics Object"],
            ViewArchitecture[ GraphicsObject, options]
    ]
]
```

```
(* Shows the generated architecture using user specified options.
    Options[ViewArchitecture] = {AmbientLight -> GrayLevel[0.],
    AspectRatio -> Automatic, Axes -> False,
  AxesEdge -> Automatic, AxesLabel -> None, AxesStyle -> Automatic,
  Background -> Automatic, Boxed -> True, BoxRatios -> Automatic,
  BoxStyle -> Automatic, ColorOutput -> Automatic, DefaultColor -> Automatic,
  Epilog -> {}, FaceGrids -> None, Lighting -> True,
  LightSources ->    {{{1., 0., 1.}, RGBColor[1, 0, 0]}, {{1., 1., 1.},
  RGBColor[0, 1, 0]},
    {{0., 1., 1.}, RGBColor[0, 0, 1]}}, PlotLabel -> None,
  PlotRange -> Automatic, PlotRegion -> Automatic, Plot3Matrix -> Automatic,
  PolygonIntersections -> True, Prolog -> {}, RenderAll -> True,
  Shading -> True, SphericalRegion -> False, Ticks -> Automatic,
  ViewCenter -> Automatic, ViewPoint -> {1.3, -2.4, 2.},
  ViewVertical -> {0., 0., 1.}, DefaultFont :> $DefaultFont,
  DisplayFunction :> $DisplayFunction} *)
ViewArchitecture[Graphics3D[graphicsObject_], options___] :=
Module[{graphics3DOptions},
    graphics3DOptions = FilterOptions[Graphics3D, options];
    Show[ Graphics3D[ graphicsObject ], graphics3DOptions]
]
```

```
(* User called function. Adds a component into the global component database.
    Uses default attributes unless specified by user. *)
AddObject[objectAtrributes__]:=
    Module[{object, objNumber, vars},
        (* CurrentObjectNumber, CurrentObjects are global variables *)
        objNumber = (CurrentObjectNumber = CurrentObjectNumber + 1);
        (* Define a prefix list and append $ to it and make the code
            general *)
        vars = Map[ ToExpression[StringJoin[ToString[#], ToString[objNumber]]]&,
                {x$, y$, z$, r$, s$, t$}];
        object = Join[{ObjectNumber -> objNumber, Variables -> vars},
                    objectAtrributes];
        AppendTo[CurrentObjects, object];
        object
    ]
(* User called function. Returns a list of specific components of a specific
    type. Attributes may be provided by user. *)
GetObjects[numberOfObjects_Integer, componentType_String, attributes___] :=
    Table[GetObject[componentType, attributes], {numberOfObjects}]
```

```
(* User called function. Returns a specific components of a specific
    type. Attributes may be provided by user. *)
GetObject[componentType_String, attributes___] :=
    Module[ {genericAttributes, catalogAttributes},
            (* Get the list of rules (attr -> val) for componentTypes
               from generic database *)
        genericAttributes = First[ Select[genericComponentDataBase,
                    ((ComponentType /. #) === componentType) & ]];

            (* If CatalogNumber is specified, pick the record
               from the catalog database *)
        catalogAttributes =
            If[StringQ[CatalogNumber /. {attributes}],
                First[ Select[catalogComponentDataBase,
                    ((CatalogNumber /. #) ===
                    (CatalogNumber /. {attributes})) & ]],
                {},
            ];

            (* From the lists of rules, pick the first occurance of
               each rule. User specified attributes override catalog
               attributes which override generic attributes *)
        object = PickFirst[{{attributes},catalogAttributes,genericAttributes}];
        AddObject[object]
]
```

Constraint Generation

```
(* Adds multiple constraints calling AddPlacementConstraint[] on each one. *)
Clear[AddPlacementConstraints];
AddPlacementConstraints[constraints__] :=
    Map[AddPlacementConstraint, constraints]

(* User called function. Adds a placement constraint to a global database of
    constraints. The various constraints supported are:
        - relativeTo[firstObject_, secondObject_, offset__];
        - absolute[object_, position_];
        - default[object_] *)
Clear[AddPlacementConstraint];
```

```
AddPlacementConstraint[relativeTo[firstObject_, secondObject_, offset__]]:=
    Module[{l},
        PositionConstraints = Join[PositionConstraints,
                 l = Table[(Variables /. firstObject)[[i]] + offset[[i]] ==
                           (Variables /. secondObject)[[i]],
                       {i, Length[offset]}]
        ];
        l
    ];

AddPlacementConstraint[absolute[object_, position_]]:=
    Module[{l},
        PositionConstraints = Join[PositionConstraints,
                 l = Table[(Variables /. object)[[i]] == position[[i]],
                       {i, Length[position]}]
        ];
        l
    ];

(* INCORRECT :: Change according to TODO *)
AddPlacementConstraint[default[object_]]:=
    AddPlacementConstraint[absolute[object, Origin]];
```

```
coordinateRulesToQuadruples[coordinaterules_List] :=
Module[ {objectNumbers, quadruples, f},
    objectNumbers = Union[Map[
            ToExpression[StringDrop[ToString[First[#]], 1]] &, coordinaterules]]
    f = Function[objNumber,
                 Prepend[
                    Map[ ToExpression[
                            StringJoin[ToString[#], ToString[objNumber]]]&,
                        {x, y, z}],
                    objNumber
                 ]
        ];
    quadruples = Map[f, objectNumbers];
    quadruples /. coordinaterules
]

quadruplesToCoordinateRules[quadruples_List] :=
Module[ {f},
    f = Function[{o,x,y,z},
            {ToExpression[StringJoin["x",ToString[o]]] -> x,
             ToExpression[StringJoin["y",ToString[o]]] -> y,
             ToExpression[StringJoin["z",ToString[o]]] -> z}
        ];
    Flatten[ MapThread[f, Transpose[quadruples] ] ]
]

coordinateRules[coord_, axisLabel_String] :=
        Select[coord, (StringTake[ToString[First[#]],1] == axisLabel)&]
```

```
(* Translates a set of coordinates so that all lie in the first quadrant.  *)
handleDefaults[coord_] :=
    Module[{objects, xs, ys, zs},
        {objects, xs, ys, zs} = Transpose[ coordinateRulesToQuadruples[coord] ];
        xs = xs - Min[xs]; ys = ys - Min[ys]; zs = zs - Min[zs];
        quadruplesToCoordinateRules[ Transpose [ {objects, xs, ys, zs} ] ]
    ]

(* User called function. From the given position constraints, computes the
    position of each component in the database. *)
ComputePositions[] :=
    Module[{c},
        c = Solve[PositionConstraints];
        If[c == {},
            GraphicsObjectValid = False;
            Print["Error :: Infeasible Position Constraints"],
            c = First[c]; (* now update global variable Coordinates *)
            Coordinates = handleDefaults[c]
        ];

    ]
```

```
(* User called function. From the given orientation constraints, computes the
    orientation of each component in the database. *)
ComputeOrientations[] :=
    Module[{c},
        c = Solve[OrientationConstraints];

        If[c == {},
            GraphicsObjectValid = False;
            Print["Error :: Infeasible Orientation Constraints"],
            c = First[c]; (* now update global variable Coordinates *)
            Orientations = c
        ];

    ]
```

```
AddOrientationConstraint[parallelTo[object1, object2]];
AddOrientationConstraint[perpendicularTo[object1, object2]];
AddOrientationConstraint[atAnAngle[object1, object2, angle[] ]];
AddOrientationConstraint[relativeAngles[object1, object2, {angle[], angle[], angle[] }]];
```

```
(* Adds multiple constraints calling AddOrientationConstraint[] on each one. *)
AddOrientationConstraints[constraints__] :=
    Map[AddOrientationConstraint, constraints];

Clear[AddOrientationConstraint];
(* User called function. Adds an orientation constraint to a global database of
    constraints. The various constraints supported are:
        - parallelTo[firstObject_, secondObject_];
        - perpendicularTo[firstObject_, secondObject_];
        - atAnAngle[firstObject_, angle[theta_], secondObject_]
        - relativeAngles[firstObject_, secondObject_, angles__ ]
        - principalAxesAtAnglesOf[ firstObject_, secondObject_, angles__]
```

```
        - atAnAngleWithXAxis[object_, angle[thetaX_]]
        - atAnAngleWithYAxis[object_, angle[thetaY_]]
        - atAnAngleWithZAxis[object_, angle[thetaZ_]]
        - orientationOf[object_, angles__]
        *)

AddOrientationConstraint[parallelTo[firstObject_, secondObject_]] :=
    AddOrientationConstraint[relativeAngles[firstObject, secondObject,
        {angle[0], angle[0], angle[0]}]];

AddOrientationConstraint[perpendicularTo[firstObject_, secondObject_]] :=
    AddOrientationConstraint[relativeAngles[firstObject, secondObject,
        {angle[90 Degree]}]];

AddOrientationConstraint[atAnAngle[firstObject_, angle[theta_], secondObject_]]
    AddOrientationConstraint[relativeAngles[firstObject, secondObject,
        {angle[theta]}]];

AddOrientationConstraint[relativeAngles[firstObject_, secondObject_, angles__ ]
    Module[{i},
        OrientationConstraints = Join[OrientationConstraints,
                 i = Table[(Variables /. firstObject)[[i]] ==
                        N[ headLess[ angles[[i - 3]] ] ] +
                            (Variables /. secondObject)[[i]],
                        {i, 4, 3 + Length[angles]}]
        ];
        i
    ]

AddOrientationConstraint[principalAxesAtAnglesOf[
                firstObject_,
                secondObject_, angles__]] /; (Length[angles] == 2) :=
        AddOrientationConstraint[
            relativeAngles[firstObject, secondObject, angles ]];

AddOrientationConstraint[atAnAngleWithXAxis[object_, angle[thetaX_]]] :=
        AddOrientationConstraint[
                orientationOf[object,
                    {angle[thetaX], angle[0], angle[0]} ]];

AddOrientationConstraint[atAnAngleWithYAxis[object_, angle[thetaY_]]] :=
        AddOrientationConstraint[
                orientationOf[object,
                    {angle[Pi/2 + thetaY], angle[0], angle[0]} ]];

AddOrientationConstraint[atAnAngleWithZAxis[object_, angle[thetaZ_]]] :=
        AddOrientationConstraint[
                orientationOf[object,
                    {angle[0], angle[thetaZ], angle[0]} ]];

AddOrientationConstraint[orientationOf[object_, angles__]] :=
    Module[{i},
        OrientationConstraints = Join[OrientationConstraints,
                 i = Table[(Variables /. object)[[i]] ==
                        N[ headLess[ angles[[i - 3]] ] ],
                        {i, 4, 3 + Length[angles]}]
        ];
        i
    ]
```

NOTE: principalAxesAtAnAngleOf[firstObject_, secondObject_, angle[theta_]]
should express the directional cosines of the final principal axis in terms of thetaX and thetaXY of an object.
Generate a constraint with the dot product of the principal axes of the two objects to be equal to Cos[theta].

Support functions for OHDL

Global Variables

```
(* resets the users Global variables.*)
Clear[initialize];
initialize[]:=
    (CurrentObjectNumber = 0; (* # of components *)

    CurrentObjects = {}; (* The components in the architecture *)

    (* List of position constraints *)
    PositionConstraints = {};

    (* List of orientation constraints *)
    OrientationConstraints = {};

    Coordinates = {};
    Orientations = {};

    (* origin of the assembly *)
    Origin = {0, 0, 0};            (* ???? *)

    (* saved graphics object for the architecture *)
    GraphicsObject = Graphics3D[{}];
    GraphicsObjectValid = False;

    (* Number of messages generated so far *)
    CurrentMessageNumber = 0;

    (* Simulation Time *)
    CurrentTime = 0; )
(* Provides values for all user's global variables.
    Options[showState] = {FullInfo -> True/False}

    False (default) does not show the less important variables.
    True otherwise. *)
Clear[showState];
showState[options___] :=
Module[{fullInfo},
    fullInfo = If[{options} === {}, True, False];
    fullInfo = FullInfo /. {options, FullInfo -> fullInfo};

    Print["-----------------------------------------------------"];
    Print["CurrentObjectNumber = ", CurrentObjectNumber];
    Print["-----------------------------------------------------"];
    Print["CurrentMessageNumber = ", CurrentMessageNumber];
    Print["-----------------------------------------------------"];
    Print["CurrentTime = ", CurrentTime ];

If[ (PositionInfo /. {options, PositionInfo -> False}) || fullInfo,
    Print["-----------------------------------------------------"];
    Print["Coordinates = ", Sort[Coordinates]]
];
If[ (OrientationInfo /. {options, OrientationInfo -> False}) || fullInfo,
    Print["-----------------------------------------------------"];
    Print["Orientations = ", Sort[Orientations]];
];
If[ (ConstraintInfo /. {options, ConstraintInfo -> False}) || fullInfo,
    Print["-----------------------------------------------------"];
    Print["PositionConstraints = ", Sort[PositionConstraints]];
    Print["-----------------------------------------------------"];
    Print["OrientationConstraints = ", Sort[OrientationConstraints]]
];
If[ fullInfo,
    Print["-----------------------------------------------------"];
```

```
    Print["CurrentObjects = ", TableForm[CurrentObjects]]
];
Print["------------------------------------------------------"]
]
```

```
(* Returns the position of an object as stored in the global positions
    database *)
Clear[positionOf];
positionOf[ object_ ] :=
Module[{x, y, z},
    {x, y, z} = (Variables /. object)[[{1, 2, 3}]];
    {x, y, z} /. Join[Coordinates, {x -> 0, y -> 0, z -> 0}]
]
```

```
(* Returns the position of an object as stored in the global orientations
    database *)
Clear[orientationOf];
orientationOf[ object_ ] :=
Module[{r, s, t},
    {r, s, t} = (Variables /. object)[[{4, 5, 6}]];
    {r, s, t} /. Join[Orientations, {r -> 0, s -> 0, t -> 0}]
]
```

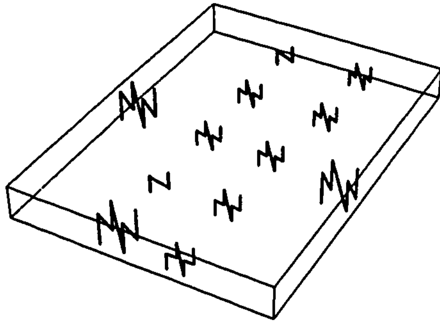
```
(* Returns a set of six equations each representing one side of the
    bounding box of a specific object. *)
Clear[boundingBoxEquations];
boundingBoxEquations[ object_ ]:=
Module[{xmin, ymin, zmin, xmax, ymax, zmax, r, s, t, size, boundingBox,
        newBoundingBox, newBoundingBoxPoints},
    {xmin, ymin, zmin} = positionOf[object];
    {r, s, t} = orientationOf[object];
    {xmax, ymax, zmax} = {xmin, ymin, zmin} + attributeOf[ object, Size ];
    boundingBox = cuboid[{xmin, ymin, zmin}, {xmax, ymax, zmax}];
    newBoundingBox = orient[boundingBox, {r, s, t}];
    newBoundingBoxPoints = Map[ headLess, newBoundingBox ];
    Map[ equationOfPlane, newBoundingBoxPoints]
]
```

```
(* Given an object and a message, it computes the coordinates of the output
    message. *)
Clear[ComputeOutputPosition];
ComputeOutputPosition[object_, message_]:=
Module[{x0, y0, z0, a, b, c, planes, line, tValues, tRules, tMin, exitFacePos,
        exitFace},
        (* Extract point of impact from the message *)
    {x0, y0, z0} = PointOfImpact /. message;
        (* Extract orientation of the incident ray from the message *)
    {a, b, c} = OrientationOfRay /. message;
        (* get all the six equations of the bounding planes of the object *)
    planes = boundingBoxEquations[object];
        (* Obtain the parametric representation of the ray. Parameter is 't', the
           distance from the place of impact.
           line is {x -> x0 + a t, y -> y0 + b t, z -> z0 + c t} *)
    line = MapThread[Rule[#1, #2 + #3 #4]&, {{x, y, z}, {x0, y0, z0}, {a, b, c}, ·
```

```
        (* Solve for the points of intersection of the ray with all the
           six bounding planes. *)
    tRules = Map[NSolve[{Evaluate[# /. line]}, {t}]&, planes];
        (* extract the 't' values. *)
    tValues = Map[ t /. # &, Flatten[tRules] ];
        (* find the Minimum, Positive 't' value *)
    tMin = Min[Select[ tValues, Positive ]];
        (* The face Index from which the ray leaves the box *)
    (* exitFacePos = Position[ tRules, t -> tMin][[1, 1]]; *)
        (* The equation of the plane corresponding to the exit face *)
    (* exitFace = planes[[exitFacePos]]; *)
        (* The point of exit from the box.
           - Find the point on the line at t = tMin.
           - Convert the list of rules to a point.*)
    {x, y, z} /. (line /. {t -> tMin})
]
```

Reading in coordinates of components.

```
fileName = "~/Packages/Examples/SerialAdder.coords";
fileName = "~/Packages/Examples/Michelson.coords";
fileName = "~/Packages/Examples/TripFlop.coords";
coords = Get[fileName];
Show[ Graphics3D[ Map[Line, coords] ] ];
```



Chapter 45

Results of Ray Tracing

Chapter Abstract

This notebook reads the log file generated by the simulator and displays it graphically in the form of individual events and accumulated events. The architecture information is also read from a file.

Results of Ray Tracing

This notebook reads the log file generated by the simulator and displays it graphically in the form of individual events and accumulated events.
The architecture information is also read from a file.

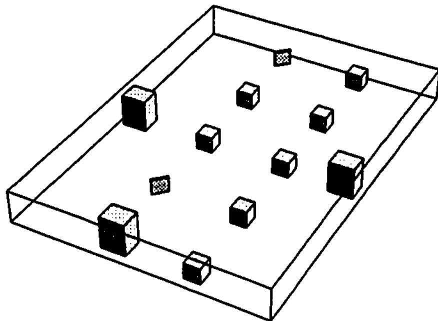
```
allEvents =
    Rest[ReadList["~/Packages/Simulator/log2", RecordSeparators -> {"\}\{"}]];

allEvents //TableForm
```

□ Reading in TripFlop

□ Drawing

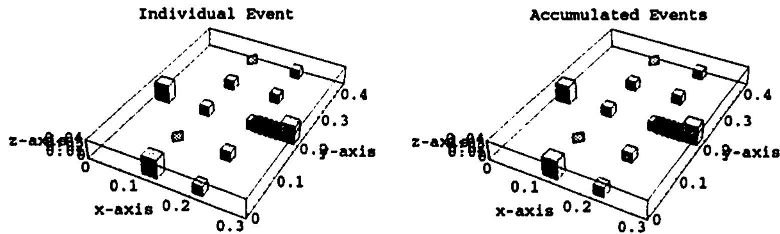
```
faces = Map[Faces /. # &, OHDL$CurrentObjects];
```

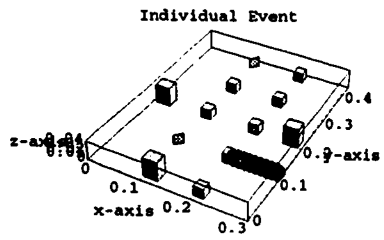
```
plot = Show[Graphics3D[Map[Polygon, faces , {2}]] ];
```

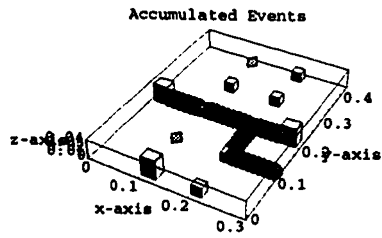


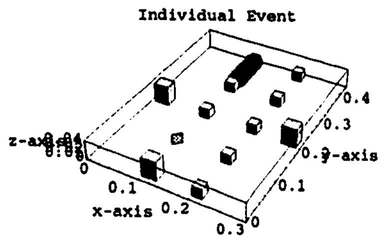
```
range = PlotRange /. FullOptions[plot];

pairs = {PlaceOfOrigin, PointOfImpact} /. allEvents
    /. {PlaceOfOrigin -> {0, 0, 0},
    PointOfImpact -> {0, 0, 0}};

allLines = Map[Line, Select[pairs, ((#[[1]] =!= #[[2]])
            && (#[[1]] =!= {0, 0, 0}))&] ]
```

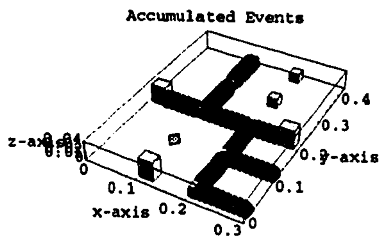
```
accumulated = Table[
Show[plot, Graphics3D[{AbsoluteThickness[7], RGBColor[1, 0, 0],
    allLines[[Range[i]]]}],
PlotRange -> range, Axes -> True, AxesLabel ->
            {"x-axis", "y-axis", "z-axis"},
    PlotLabel -> "Accumulated Events",
    DisplayFunction -> Identity],
{i, Length[allLines]}];

separate = Map[
    Show[plot, Graphics3D[{AbsoluteThickness[7], RGBColor[1, 0, 0], #}],
        PlotLabel -> "Individual Event",
            PlotRange -> range, DisplayFunction -> Identity,
    Axes -> True, AxesLabel -> {"x-axis", "y-axis", "z-axis"}
]&, allLines];

MapThread[Show[GraphicsArray[{#1, #2}], DisplayFunction -> $DisplayFunction]&,
    {separate[[{1, 5, 10, 15, 20, 25}]],
     accumulated[[{1, 5, 10, 15, 20, 25}]]}];
```

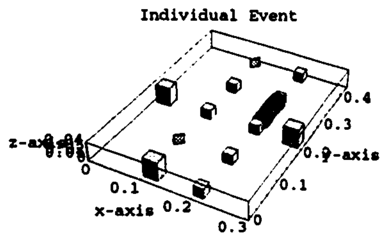


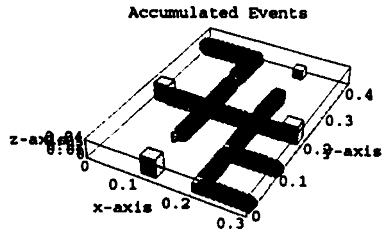

Individual Event
z-axis
y-axis
0.4
0.3
0.1
0
0.1
x-axis 0.2
0.3

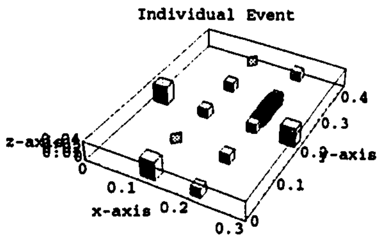

Accumulated Events
z-axis
y-axis
0.4
0.3
0.1
0
0.1
x-axis 0.2
0.3

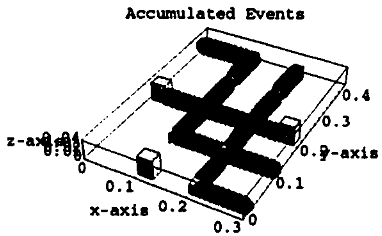

Individual Event
z-axis
y-axis
0.4
0.3
0.1
0
0.1
x-axis 0.2
0.3

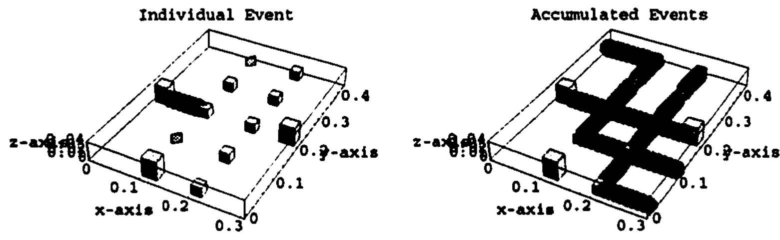

Accumulated Events
z-axis
y-axis
0.4
0.3
0.1
0
0.1
x-axis 0.2
0.3


Individual Event
0.4
0.3
0.1
0
0.1
x-axis
0.2
0.3


Accumulated Events
0.4
0.3
0.1
0
0.1
x-axis
0.2
0.3


Individual Event
0.4
0.3
0.1
0
0.1
x-axis
0.2
0.3


Accumulated Events
0.4
0.3
0.1
0
0.1
x-axis
0.2
0.3


Individual Event
Accumulated Events
x-axis
y-axis
z-axis
0.1
0.2
0.3
0.4

Chapter 46

Power & Delay Analyses

Chapter Abstract

This notebook contains the routines for carrying out power and delay analysis on a log file generated by the simulator. This log file corresponds to one simulation of a particular architecture. It first creates a signal flow graph and then produces bar graphs for each simple path.

Power & Delay Analyses

This notebook contains the routines for carrying out power and delay analysis on a log file generated by the simulator.
This log file corresponds to one simulation of a particular architecture.
It first creates a signal flow graph and then produces bar graphs for each simple path.

Initialization

```
<<DiscreteMath`Combinatorica`
```

```
<<Graphics`Graphics`
```

moveToUtilities

```
Clear[logicalAnd];
logicalAnd[any_Symbol]:=
    any === True

logicalAnd[l_List]:=
    Fold[And[#1, #2]&, True, l]

logicalAnd[seq___]:=
    Fold[And[#1, #2]&, True, {seq}]
```

```
logicalAnd[False]
False
```

```
logicalAnd[Any]
False
```

```
logicalAnd[True, False, True, True]
False
```

```
logicalAnd[True, True, True]
True
```

```
logicalAnd[{True, False, True, True}]
False
```

```
logicalAnd[{True, True, True}]
True
```

```
logicalAnd[False, False, False]
False
```

```
logicalAnd[{False, False, False}]
False
```

```
Clear[logicalOr];
logicalOr[any_Symbol]:=
    any === True

logicalOr[l_List]:=
    Fold[Or[#1, #2]&, False, l]

logicalOr[seq___]:=
    Fold[Or[#1, #2]&, False, {seq}]
```

```
logicalOr[True]
True
logicalOr[False]
False
logicalOr[Any]
False
logicalOr[True, False, True, True]
True
logicalOr[True, True, True]
True
logicalOr[False, False, False]
False
logicalOr[{True, False, True, True}]
True
logicalOr[{False, False, False}]
False
logicalOr[{True, True, True}]
True
```

Analysis -- Implementation

```
edgeQ[pair__]:=
    MemberQ[edges, pair]
pathQ[path__]:=
    logicalAnd[ Map[edgeQ, Partition[path, 2, 1]] ]
simplePathQ[path__]:=
    (Sort[path] === Union[path]) && pathQ[path]
completedPathQ[path_]:=
    If[pathQ[path] && MemberQ[ sinks, Last[path] ], Print[path]; True, False]
simpleCompletedPathQ[path_]:=
    (Length[path] == Length[Union[path]]) && completedPathQ[path]
allPaths = Flatten[Table[Backtrack[Table[vertices, {i}],
            pathQ, completedPathQ, All],
                {i, 3, 6}], 1]
{2, 8, 13}
{4, 1, 13}
{5, 6, 13}
{7, 8, 13}
{8, 9, 13}
{2, 4, 1, 13}
{2, 8, 9, 13}
{3, 2, 8, 13}
{3, 5, 6, 13}
{3, 7, 8, 13}
{7, 8, 9, 13}
```

```
{8, 2, 8, 13}
{10, 2, 8, 13}
{2, 3, 2, 8, 13}
{2, 3, 5, 6, 13}
{2, 3, 7, 8, 13}
{2, 8, 2, 8, 13}
{2, 10, 2, 8, 13}
{3, 2, 4, 1, 13}
{3, 2, 8, 9, 13}
{3, 7, 8, 9, 13}
{7, 8, 2, 8, 13}
{8, 2, 4, 1, 13}
{8, 2, 8, 9, 13}
{10, 2, 4, 1, 13}
{10, 2, 8, 9, 13}
{12, 3, 2, 8, 13}
{12, 3, 5, 6, 13}
{12, 3, 7, 8, 13}
{2, 3, 2, 4, 1, 13}
{2, 3, 2, 8, 9, 13}
{2, 3, 7, 8, 9, 13}
{2, 8, 2, 4, 1, 13}
{2, 8, 2, 8, 9, 13}
{2, 10, 2, 4, 1, 13}
{2, 10, 2, 8, 9, 13}
{3, 2, 3, 2, 8, 13}
{3, 2, 3, 5, 6, 13}
{3, 2, 3, 7, 8, 13}
{3, 2, 8, 2, 8, 13}
{3, 2, 10, 2, 8, 13}
{3, 7, 8, 2, 8, 13}
{3, 12, 3, 2, 8, 13}
{3, 12, 3, 5, 6, 13}
{3, 12, 3, 7, 8, 13}
{7, 8, 2, 4, 1, 13}
{7, 8, 2, 8, 9, 13}
{8, 2, 3, 2, 8, 13}
{8, 2, 3, 5, 6, 13}
{8, 2, 3, 7, 8, 13}
{8, 2, 8, 2, 8, 13}
{8, 2, 10, 2, 8, 13}
{10, 2, 3, 2, 8, 13}
{10, 2, 3, 5, 6, 13}
{10, 2, 3, 7, 8, 13}
{10, 2, 8, 2, 8, 13}
{10, 2, 10, 2, 8, 13}
{12, 3, 2, 4, 1, 13}
{12, 3, 2, 8, 9, 13}
{12, 3, 7, 8, 9, 13}
```

■ Power Analysis - Implementation

```
Clear[componentTransferFunction];
componentTransferFunction[componentNumber_, options___]:=
Module[{component, size, index, type, attributeName},
    {component} =
        Select[OHDL$CurrentObjects,
                    ((ObjectNumber /. #) == componentNumber) &, 1];
    attributeName = AttributeName /. {options, AttributeName -> Delay};
    size = Size /. component;
    index = RefractiveIndex /. component;
    type = ComponentType /. component;
    Switch[type,
        "BeamSplitter", scale = 0.5,
        "InterferenceFilter", scale = 0.99,
        "Mirror", scale = 0.99,
        "PolarizingBeamSplitter", scale = 0.5,
        "PulsedLaser", scale = 1.0,
        _, scale = 1.0
    ];
    scale
]
```

```
Clear[freeSpaceTransferFunction];
freeSpaceTransferFunction[source_, destination_, options___]:=
Module[{decay, sourcePos, destinationPos, distanceBetween},
    decay = 10^(-6);
    sourcePos = Fold[ Plus[#1, #2]&, {0, 0, 0},
                        Flatten[Faces /.
                            OHDL$CurrentObjects[[source]], 1] ] / 24;
    destinationPos = Fold[ Plus[#1, #2]&, {0, 0, 0},
                        Flatten[Faces /.
                            OHDL$CurrentObjects[[destination]], 1] ] / 24;
    distanceBetween = Function[{p1, p2}, Sqrt[Apply[Plus, (p1 - p2)^2]]];
    1 - distanceBetween[sourcePos, destinationPos] * decay
]
Clear[messageTransferFunction];
messageTransferFunction[source_, destination_, options___]:=
Module[{},
    componentTransferFunction[source, options] *
                                freeSpaceTransferFunction[source,
                                    destination, options]
]
Clear[pathTransferFunction];
pathTransferFunction[path__, input_, options___]:=
Module[{transferFunctions},
    transferFunctions =
        Map[messageTransferFunction[#[[1]], #[[2]], options]&,
            Partition[path, 2, 1]];
    FoldList[Times[#1, #2]&, input, transferFunctions]
]
```

```
powerTable[path_]:=
Module[{table},
    table = MapThread[Append, {Partition[path, 2, 1],
                    Rest[pathTransferFunction[path, 1,
            AttributeName -> Delay]]}];
    TableForm[table,
            TableHeadings -> {Automatic, {"Sender", "Receiver",
                        "Current Intensity"}},
            TableAlignments -> {Automatic, Center}
    ]
]

Print["\tPower Analysis of Trip-Flop Architecture"];
Print["\t=======================================\n\n"];
Do[
    Print["Power Analysis for path ", allPaths[[i]]];
    Print[powerTable[ allPaths[[i]] ]];
    Print["---------------------------------------\n"],
 {i, 1, 3}]
(* {i, 1, Length[allPaths]}] *)
```

```
    Power Analysis of Trip-Flop Architecture
    ========================================


Power Analysis for path {2, 3, 2, 4, 1, 13}
    Sender   Receiver   Current Intensity
1     2         3             0.5

2     3         2             0.25


3     2         4             0.125

4     4         1             0.12375

5     1         13            0.061875
---------------------------------------

Power Analysis for path {2, 3, 2, 8, 9, 13}
    Sender   Receiver   Current Intensity
1     2         3             0.5

2     3         2             0.25

3     2         8             0.125

4     8         9             0.0625

5     9         13            0.03125
---------------------------------------

Power Analysis for path {2, 3, 7, 8, 9, 13}
    Sender   Receiver   Current Intensity
1     2         3             0.5

2     3         7             0.25

3     7         8             0.2475

4     8         9             0.12375

5     9         13            0.061875
```

```
Clear[analysisGraph];
analysisGraph[path__, dataFunction_, heading_String]:=
Module[{data, table, names, numbers, bar, items, xmax, interval},
    data = dataFunction[path];
    table = Graphics[{ Text[FontForm[data, {"Times", 10}], {0, 0}] }];
    names = Map[StringJoin["Component #", ToString[#[[2]] ]]&,
            First[data]
          ];
    names = Transpose[{Range[Length[names]], names}];
    numbers = Map[#[[3]]&, First[ data ]];
    bar = BarChart[numbers, BarOrientation -> Horizontal,
        BarEdges -> False,
        PlotLabel -> FontForm[StringJoin[heading, ToString[path]],
              {"Helvetica", 10}],
        BarStyle -> (GrayLevel[0.5]&),
        Ticks -> {Automatic, names},
        DefaultFont -> {"Helvetica", 8},
        DisplayFunction -> Identity
    ];
    items = Length[names];
    xmax = Max[numbers];
    interval = xmax / 20;
    bar = Show[bar, Epilog -> {GrayLevel[1], AbsoluteThickness[0.25],
              Table[Line[{{i, 0.5}, {i, items + 0.5}}],
                  {i, interval, xmax, interval}]},
              DisplayFunction -> Identity
        ];
    Show[GraphicsArray[{table, bar}, Frame -> True]];
]
```

```
Clear[powerGraph];
powerGraph[path_]:=
    analysisGraph[path, powerTable, "Power Analysis of Path "]
```

Delay Analysis - Implementation

```
Clear[componentTransferFunction];
componentTransferFunction[componentNumber_, options___]:=
Module[{component, size, index, attributeName},
    {component} =
        Select[OHDL$CurrentObjects,
            ((ObjectNumber /. #) == componentNumber) &, 1];
    attributeName = AttributeName /. {options, AttributeName -> Delay};
    size = Size /. component;
    index = RefractiveIndex /. component;
    Max[size]/(3.0*10^8) * (index - 1)
]

Clear[freeSpaceTransferFunction];
freeSpaceTransferFunction[source_, destination_, options___]:=
Module[{sourcePos, destinationPos, distanceBetween},
    sourcePos = Fold[ Plus[#1, #2]&, {0, 0, 0},
                        Flatten[Faces
                            /. OHDL$CurrentObjects[[source]], 1] ] / 24;
    destinationPos = Fold[ Plus[#1, #2]&, {0, 0, 0},
                        Flatten[Faces
                            /. OHDL$CurrentObjects[[destination]], 1] ] / 24;
    distanceBetween = Function[{p1, p2}, Sqrt[Apply[Plus, (p1 - p2)^2]]];
    distanceBetween[sourcePos, destinationPos] / (3.0*(10^8))
]

Clear[messageTransferFunction];
messageTransferFunction[source_, destination_, options___]:=
Module[{},
    componentTransferFunction[source, options] +
                                freeSpaceTransferFunction[source,
                                    destination, options]
]

Clear[pathTransferFunction];
pathTransferFunction[path__, input_, options___]:=
Module[{transferFunctions},
    transferFunctions =
        Map[messageTransferFunction[#[[1]], #[[2]], options]&,
        Partition[path, 2, 1]];
    FoldList[Plus[#1, #2]&, input, transferFunctions]
]

delayTable[path_]:=
Module[{table},
    table = MapThread[Append, {Partition[path, 2, 1],
                (10^12)*Rest[pathTransferFunction[path, 0,
                        AttributeName -> Delay]]}];
    TableForm[table,
            TableHeadings -> {Automatic, {"Sender", "Receiver", "Delay (psec)"}}
            TableAlignments -> {Automatic, Center}
    ]
]
```

```
Print["\tDelay Analysis of Trip-Flop Architecture"];
Print["\t======================================\n\n"];
Do[
    Print["Delay Analysis for path ", allPaths[[i]]];
    Print[delayTable[ allPaths[[i]] ], "\n"];
    Print["---------------------------------------\n"],
{i, 1, 3}]
(* {i, 1, Length[allPaths]}] *)
```

```
    Delay Analysis of Trip-Flop Architecture
    ========================================


Delay Analysis for path {2, 3, 2, 4, 1, 13}
    Sender   Receiver   Delay (psec)
1     2         3        366.667

2     3         2        733.333

3     2         4         1100.

4     4         1        1466.67

5     1         13       2562.88
-----------------------------------------

Delay Analysis for path {2, 3, 2, 8, 9, 13}
    Sender   Receiver   Delay (psec)
1     2         3        366.667

2     3         2        733.333

3     2         8         1100.

4     8         9        1466.67

5     9         13       2111.69
-----------------------------------------

Delay Analysis for path {2, 3, 7, 8, 9, 13}
    Sender   Receiver   Delay (psec)
1     2         3        366.667

2     3         7        740.955

3     7         8        1119.82

4     8         9        1486.49

5     9         13       2131.51
-----------------------------------------
```

```
Clear[delayGraph];
delayGraph[path_]:=
    analysisGraph[path, delayTable, "Delay Analysis of Path "]
```

Analyses of Trip-Flop Architecture

Loading the Simulator Generated Data

Reading of the Simulation Results

```
allEvents =
    Rest[ReadList["~/Packages/Simulator/log2", RecordSeparators -> {"\}\{"}]];
```

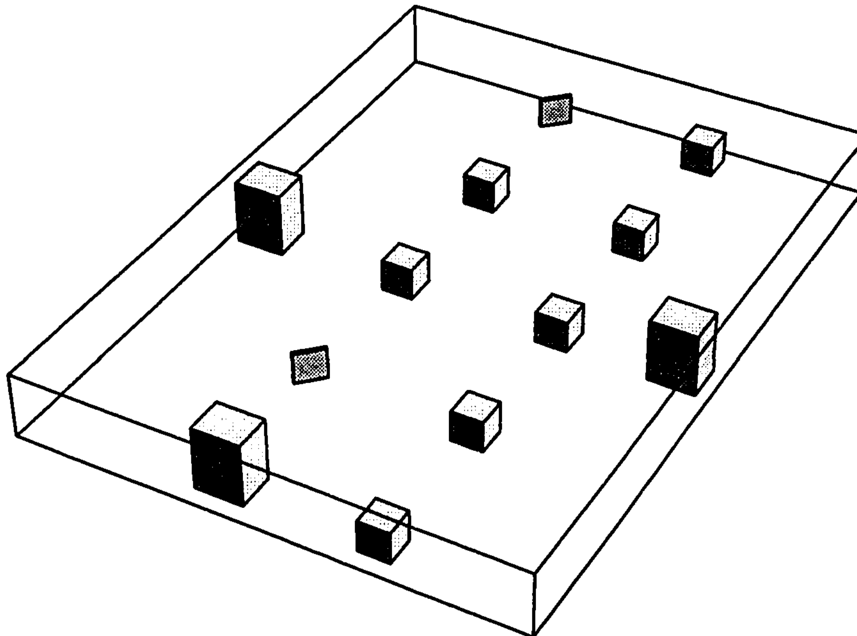
```
allEvents //TableForm
```

Reading in TripFlop

Drawing

```
faces = Drop[Map[Faces /. # &, OHDL$CurrentObjects], -1];
```

```
plot = Show[Graphics3D[Map[Polygon, faces , {2}]] ];
```



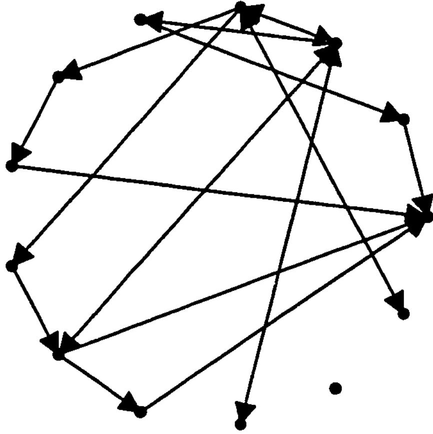
Signal-Flow Graph

```
edges = Select[{Sender, Receiver} /. allEvents,
        (((Head[#[[1]] ] === Integer) && (Head[#[[2]] ] === Integer)) &&
            (#[[1]] != #[[2]]))&];

vertices = Union[ Flatten[edges] ];

sinks = Complement[vertices, Union[ Transpose[edges][[1]]] ];
```

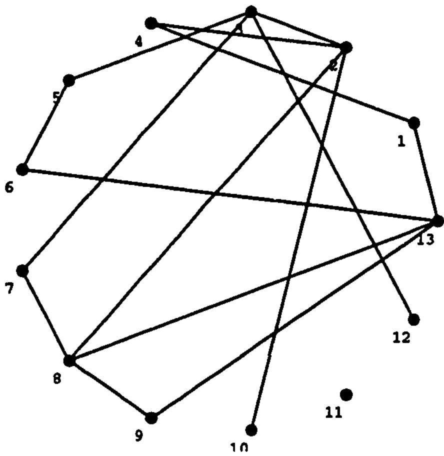
```
TableForm[({ObjectNumber, ComponentType} /. OHDL$CurrentObjects),
        TableHeadings -> {Automatic, {"ObjectNumber", "ComponentType"}}]
```

	ObjectNumber	ComponentType
1	1	PolarizingBeamSplitter
2	2	PolarizingBeamSplitter
3	3	PolarizingBeamSplitter
4	4	InterferenceFilter
5	5	InterferenceFilter
6	6	Mirror
7	7	Mirror
8	8	BeamSplitter
9	9	BeamSplitter
10	10	PulsedLaser
11	11	PulsedLaser
12	12	PulsedLaser
13	13	BoundingBox

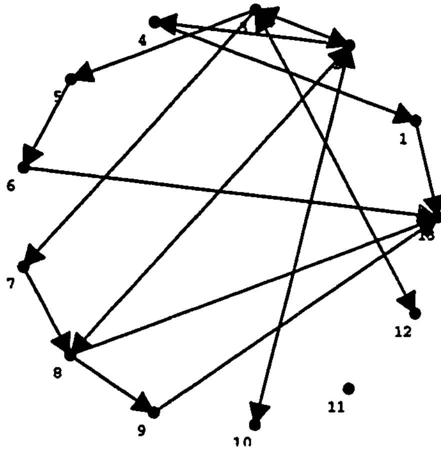
```
dir = ShowGraph[FromOrderedPairs[edges], Directed];
```



```
unDir = ShowLabeledGraph[FromOrderedPairs[edges]];
```



```
Show[dir, unDir];
```



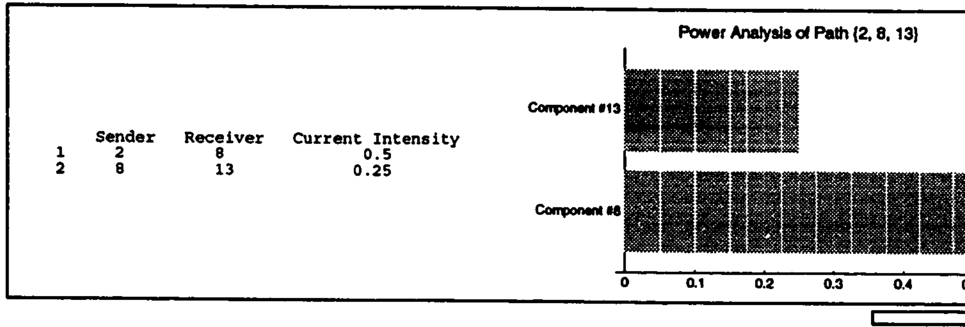

```
allPaths = Flatten[Table[Backtrack[Table[vertices, {i}],
                                    simplePathQ, simpleCompletedPathQ, All],
              {i, 3, 6}], 1]
```

```
{2, 8, 13}
{4, 1, 13}
{5, 6, 13}
{7, 8, 13}
{8, 9, 13}
{2, 4, 1, 13}

{2, 8, 9, 13}
{3, 2, 8, 13}
{3, 5, 6, 13}
{3, 7, 8, 13}
{7, 8, 9, 13}
{10, 2, 8, 13}
{2, 3, 5, 6, 13}
{2, 3, 7, 8, 13}
{3, 2, 4, 1, 13}
{3, 2, 8, 9, 13}
{3, 7, 8, 9, 13}
{8, 2, 4, 1, 13}
{10, 2, 4, 1, 13}
{10, 2, 8, 9, 13}
{12, 3, 2, 8, 13}
{12, 3, 5, 6, 13}
{12, 3, 7, 8, 13}
{2, 3, 7, 8, 9, 13}
{7, 8, 2, 4, 1, 13}
{8, 2, 3, 5, 6, 13}
{10, 2, 3, 5, 6, 13}
{10, 2, 3, 7, 8, 13}
{12, 3, 2, 4, 1, 13}
{12, 3, 2, 8, 9, 13}
{12, 3, 7, 8, 9, 13}
```

Power Analysis

```
Print["\tPower Analysis of Trip-Flop Architecture"];
Print["\t=======================================\n\n"];
Do[
   powerGraph[ allPaths[[i]] ],
{i, 1, Length[allPaths]}]
```

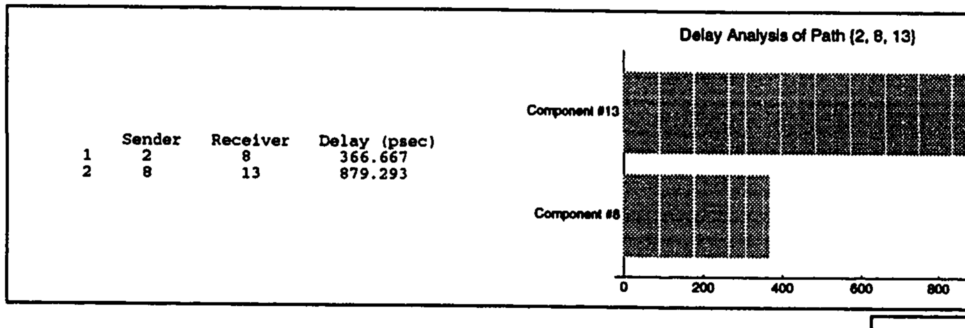
```
Power Analysis of Trip-Flop Architecture
========================================
```



Delay Analysis

```
Print["\tDelay Analysis of Trip-Flop Architecture"];
Print["\t=======================================\n\n"];
Do[
   delayGraph[ allPaths[[i]] ],
{i, 1, Length[allPaths]}]
```

```
Delay Analysis of Trip-Flop Architecture
========================================
```



Chapter 47

FrameWork of Simulator

Chapter Abstract

This notebook contains the complete code for the simulation of the Trip-Flop architecture. It was created automatically from the simulator template and the Trip-Flop architecture description. Some editing was done later. Execution of this notebook will create a log file that will record all the events of the simulation. The animation of the simulation may be seen in the notebook Simulator/ShowSimulation.ma.

FrameWork of Simulator

This notebook contains the complete code for the simulation of the Trip-Flop architecture.
It was created automatically from the simulator template and the Trip-Flop architecture description.
Some editing was done later.

Execution of this notebook will create a log file that will record all the events of the simulation.
The animation of the simulation may be seen in the notebook Simulator/ShowSimulation.ma.

Support functions for OHDL

Additional Library Routines

RuleProcessing

```
Clear[sortRuleLists];
sortRuleLists[ruleLists__, key_Symbol, predicate_:Less]:=
    Sort[ruleLists,
        predicate[(key /. Append[#1, key -> Infinity]),
                  (key /. Append[#2, key -> Infinity])
        ]&
    ]
```

```
Clear[sortRuleLists];
sortRuleLists[ruleLists__, key_Symbol, predicate_:Less]:=
Module[{},
    Map[ Rest, Sort[ Map[{key /. Append[#, key -> Infinity], #}&,
               ruleLists] ], predicate]


]
```

```
sortRuleLists[
    {{a -> 1, b -> 2, c -> 3}, {a -> -1, b -> 10, c -> 30},
          {a -> 12, b -> 24, c -> 13}},
    a]
```

```
{{a -> -1, b -> 10, c -> 30}, {a -> 1, b -> 2, c -> 3},
 {a -> 12, b -> 24, c -> 13}}
```

```
sortRuleLists[
    {{a -> 1, b -> 2, c -> 3}, {a -> -1, b -> 10, c -> 30},
          {a -> 12, b -> 24, c -> 13}}, a, Greater]
```

```
{{a -> 12, b -> 24, c -> 13}, {a -> 1, b -> 2, c -> 3},
 {a -> -1, b -> 10, c -> 30}}
```

```
sortRuleLists[
    {{a -> 1, b -> 2, c -> 3}, {a -> -1, b -> 0, c -> 30},
          {a -> 12, b -> 24, c -> 13}}, b]
```

```
{{a -> -1, b -> 0, c -> 30}, {a -> 1, b -> 2, c -> 3},
 {a -> 12, b -> 24, c -> 13}}
```

```
sortRuleLists[
    {{a -> 1, b -> 2, c -> 3}, {a -> -1, b -> 0, c -> 30},
            {a -> 12, b -> 24, c -> 13}}, c]
{{a -> 1, b -> 2, c -> 3}, {a -> 12, b -> 24, c -> 13},
  {a -> -1, b -> 0, c -> 30}}
```

Geometry

```
(* Finds the square of the distance between two points *)
squaredDistanceBetween[point1__, point2__]:=
    N[ Apply[ Plus, Map[ (# * #)&, point1 - point2] ] ]
squaredDistanceBetween[{1, 1, 1}, {1, 1, 1}]
0
squaredDistanceBetween[{1, 1, 1}, {2, 2, 2}]
3.
(* Finds the shortest distance between two points *)
distanceBetween[point1_, point2_]:=
    Abs[ N[ Sqrt[ squaredDistanceBetween[ point1, point2 ] ] ] ]
distanceBetween[{1, 1, 1}, {1, 1, 1}]
0
distanceBetween[{1, 1, 1}, {2, 2, 2}]
1.73205
(* FInds the point of intersection between a vector and a polygon *)
Clear[pointOfIntersection];
pointOfIntersection[

            line[point[startingPoint_], directionCosines[orientation_]],
            polygon[points__]
        ]:=
Module[{plane, t, lineEquation, tRules, tValues, tMin, foundValidPoint},
    EntryPrint["pointOfIntersection", "Called with Arguments ",
            line[point[startingPoint], directionCosines[orientation]],
            polygon[points]
    ];
    plane = equationOfPlane[points];
        (* Obtain the parametric representation of the ray.
            Parameter is 't', the
           distance from the place of impact.
           line is {x -> x0 + a t, y -> y0 + b t, z -> z0 + c t} *)
    lineEquation = MapThread[Rule[#1, #2 + #3 #4]&, {{x, y, z},
        startingPoint, orientation, {t, t, t}}];
    DPrint["lineEquation = ", lineEquation];
    tRules = NSolve[{Evaluate[plane /. lineEquation]}, {t}];
        (* extract the 't' values. *)
    DPrint["tRules = ", tRules];
    tRules = Select[tRules,
            ((Positive[t /. #]))&
    ];
    DPrint["tRules = ", tRules];
    tRules = Select[tRules,
            ((pointInPolygonQ[ point[{x, y, z} /. (lineEquation /. #)],
                            polygon[points]]))&
    ];
    DPrint["tRules = ", tRules];
    foundValidPoint = If[tRules === {}, {},
        tValues = t /. tRules;
        DPrint["tValues = ", tValues];
```

```
        (* find the Minimum, Positive 't' value *)
      tMin = Min[tValues];
      {x, y, z} /. (lineEquation /. {t -> tMin})
    ];
    ExitPrint["pointOfIntersection", "foundValidPoint = ", foundValidPoint];
    foundValidPoint
]

pointOfIntersection[
    line[point[{0, 0, 0}], directionCosines[normalize[{1, 1, 1}]]],
    polygon[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}]
]
{}

pointOfIntersection[
    line[point[{2, 0, 0}], directionCosines[normalize[{1, 0, 0}]]],
    polygon[{{1, 0, 1}, {1, 1, 0}, {1, 1, 1}}]
]
{}
```

```
(* Boolean function returns true if a point lies in the Polygon *)
Clear[pointInPolygonQ];
pointInPolygonQ[point[p_], polygon[points_] ]:=
Module[{bool, x, y, z, xmin, xmax, ymin, ymax, zmin, zmax},
    EntryPrint["pointInPolygonQ", point[p], polygon[points]];
    {x, y, z} = p;
    {{xmin, xmax}, {ymin, ymax}, {zmin, zmax}} =
        Map[{Min[#], Max[#]}&, Transpose[points] ];
    bool = (leQ[xmin, x] && leQ[x, xmax] && leQ[ymin, y] &&
                    leQ[y, ymax] && leQ[zmin, z] && leQ[z, zmax]);
    ExitPrint["pointInPolygonQ", bool];
    bool
]

pointInPolygonQ[_] := (Print["Error::pointInPolygonQ:Default Case"];False)
(* Returns the area of a triangle *)
areaOfTriangle[{p1_, p2_, p3_}]:=
Module[{a, b, c, s},
    a = distanceBetween[p1, p2];
    b = distanceBetween[p3, p2];
    c = distanceBetween[p1, p3];
    s = (a + b + c)/2.0;
    Chop[Abs[Sqrt[s (s - a) (s - b) (s - c)]]]
]
areaOfTriangle[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}]
0.866025
areaOfTriangle[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}}]
0.5
```

```
(* Boolean function returns True if a point lies in the Triangle *)
Clear[pointInTriangleQ];
pointInTriangleQ[point[p_], polygon[{p1_, p2_, p3_}] ]:=
Module[{bool, fullArea, area1, area2, area3},
    EntryPrint["pointInTriangleQ", point[p], polygon[points]];
    fullArea = areaOfTriangle[{p1, p2, p3}];
    DPrint["fullArea ", fullArea];
    area1 = areaOfTriangle[{p, p2, p3}];
    area2 = areaOfTriangle[{p, p1, p3}];
    area3 = areaOfTriangle[{p, p2, p1}];
    DPrint["{fullArea, area1, area2, area3, Sum} ",
        {fullArea, area1, area2, area3}, area1 + area2 + area3];
    bool = eqQ[fullArea, area1 + area2 + area3];
    ExitPrint["pointInTriangleQ", bool];
    bool
]

pointInTriangleQ[_] := (Print["Error::pointInTriangleQ:Default Case"];False)

pointInTriangleQ[point[{1, 1, 1}], polygon[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}}]]
False

pointInTriangleQ[point[{0.5, 0, 0}],
    polygon[{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}}]]
True

pointInTriangleQ[point[{0.22, 0.21, 0.01}],
polygon[{{0.22, 0.2000, 0}, {0.22, 0.2000, 0.02000}, {0.22, 0.2200, 0.02000}}]
]
True

pointInTriangleQ[point[{0.22, 0.21, 0.011}],
    polygon[{{0.22, 0.2000, 0}, {0.22, 0.2200, 0.02000}, {0.22, 0.2200, 0}}]
 ]
False

(* Boolean function returns True if the Point lies in the Polygon *)
Clear[pointInPolygonQ];
pointInPolygonQ[point[p_], polygon[points_] ] /; (Length[points] >= 3) :=
Module[{bool, bools, p1, ps, triangles},
    EntryPrint["pointInPolygonQ", point[p], polygon[points]];
    {p1, ps} = {First[points], Rest[points]};
    triangles =
        Map[ polygon[Prepend[#, p1]]&, Partition[ps, 2, 1] ];
    bools = Map[pointInTriangleQ[point[p], #]&, triangles];
    bool = Fold[Or, False, bools];
    ExitPrint["pointInPolygonQ", bool];
    bool
]

pointInPolygonQ[_] := (Print["Error::pointInPolygonQ:Default Case"];False)

pointInPolygonQ[point[{0.22, 0.21, 0.01}], polygon[coords[[2, 3]] ] ]
True

Table[pointOfIntersection[line[point[{0.3, 0.21, 0.01}],
                        directionCosines[{-1, 0, 0}]],
    polygon[coords[[2, i]]]], {i, 6}]
```

```
(* A simplified function different from the one in Simulator, based on Faces
    info in object. Returns a set of six equations each representing one
    side of the bounding box of a specific object. *)
Clear[boundingBoxEquations];
boundingBoxEquations[ object_ ]:=
Module[{},
    Map[ equationOfPlane, Faces /. object]
]
```

Misc

```
(* Returns the position of an object as stored in the global positions
    database *)
Clear[positionOf];
positionOf[ object_ ] :=
Module[{x, y, z},
    {x, y, z} = (Variables /. object)[[{1, 2, 3}]];
    {x, y, z} /. Join[Coordinates, {x -> 0, y -> 0, z -> 0}]
]
```

```
(* Returns the position of an object as stored in the global orientations
    database *)
Clear[orientationOf];
orientationOf[ object_ ] :=
Module[{r, s, t},
    {r, s, t} = (Variables /. object)[[{4, 5, 6}]];
    {r, s, t} /. Join[Orientations, {r -> 0, s -> 0, t -> 0}]
]
```

```
(* Returns a set of six equations each representing one side of the bounding
    box of a specific object. *)
Clear[boundingBoxEquations];
boundingBoxEquations[ object_ ]:=
Module[{xmin, ymin, zmin, xmax, ymax, zmax, r, s, t, size, boundingBox,
        newBoundingBox, newBoundingBoxPoints},
    {xmin, ymin, zmin} = positionOf[object];
    {r, s, t} = orientationOf[object];
    {xmax, ymax, zmax} = {xmin, ymin, zmin} + attributeOf[ object, Size ];
    boundingBox = cuboid[{xmin, ymin, zmin}, {xmax, ymax, zmax}];
    newBoundingBox = orient[boundingBox, {r, s, t}];
    newBoundingBoxPoints = Map[ headLess, newBoundingBox ];
    Map[ equationOfPlane, newBoundingBoxPoints]
]
```

Init

```
OHDL$FunctionsToTrace = {
    {"DPrint", False},
    {"processMessage", False},
    {"pointOfIntersection", False},
    {"pointOfImpact", False},
    {"simulateArchitecture", True},
    {"pointInTriangleQ", False},
    {"performAction", False},
    {"ComputeOutputPosition", False},
    {"pointInPolygonQ", False},
    {"simulateComponent", False},
    {"equationOfPlane", False}
};

OHDL$CallStack = {};
```

Utilities - Debug

```
Clear[debugOnQ];
debugOnQ[functionName_String]:=
    Select[Append[OHDL$FunctionsToTrace, {functionName, False}],
              (#[[1]] == functionName)&][[1, 2]]
```

```
Clear[DPrint];
DPrint[args___]:=
    If[(OHDL$CallStack =!= {}) && (debugOnQ[First[OHDL$CallStack]]),
        Print["Debug::", First[OHDL$CallStack], ":", args]
    ]

Clear[EntryPrint];
EntryPrint[functionName_String, args___]:=
Module[{},
    OHDL$CallStack = Prepend[OHDL$CallStack, functionName];
    DPrint[args]
]

Clear[ExitPrint];
ExitPrint[functionName_String, args___]:=
Module[{},
    DPrint[args];
    If[First[OHDL$CallStack] != functionName,
        Print["Error::ExitPrint[]:Corrupted Call Stack "]
    ];
    OHDL$CallStack = Rest[OHDL$CallStack]
]
```

Cache

```
OHDL$MaxCacheSize = 100;
OHDL$CurrentCacheSize = 0;
OHDL$CacheEnabled = False;
```

```
Clear[initializeCache];
initializeCache[options___]:=
Module[{},
    OHDL$MaxCacheSize = MaxCacheSize /. {options, MaxCacheSize -> 200};
    OHDL$CacheEnabled = CacheEnabled /. {options, CacheEnabled -> True};
    OHDL$Cache = {};
    OHDL$CurrentCacheSize = 0;
]

initializeCache[CacheEnabled -> False];

Clear[cacheLookup];
cacheLookup[partialEntry__]:=
Module[{heads, values, entry},
    If[OHDL$CacheEnabled,
        {heads, values} = Transpose[Map[Apply[List, #]&, partialEntry]];
        entry = Select[OHDL$Cache, ((heads /. #) === values)&, 1];
        If[entry === {}, {}, First[entry]],
        {}
    ]
]
```

```
Clear[cacheInstall];
cacheInstall[entry__]:=
Module[{},
    If[OHDL$CacheEnabled,
        OHDL$Cache = Prepend[OHDL$Cache, entry];
        OHDL$CurrentCacheSize = OHDL$CurrentCacheSize + 1;
        If[OHDL$CurrentCacheSize > OHDL$MaxCacheSize,
            OHDL$Cache = Take[OHDL$Cache, OHDL$MaxCacheSize];
            OHDL$CurrentCacheSize = OHDL$MaxCacheSize
        ];
        entry,
        {}
    ]
]

cacheLookup[{A -> a, B -> b}]

{}

cacheInstall[{A -> a, B -> b}]

{A -> a, B -> b}

{OHDL$CurrentCacheSize, OHDL$MaxCacheSize, OHDL$Cache}

{3, 3, {{C -> c, D -> d, A -> a, Y -> y}, {A -> a, B -> b},
   {C -> c, D -> d, A -> a, Y -> y}}}

cacheInstall[{C -> c, D -> d, A -> a, Y -> y}]

{C -> c, D -> d, A -> a, Y -> y}

cacheInstall[{A -> a, B -> b}]

{A -> a, B -> b}
```

```
cacheLookup[{A -> a, Y -> y}]
{C -> c, D -> d, A -> a, Y -> y}
```

Simulator Code

```
(* Given an object and a message, it computes the coordinates of the output
    message. *)
Clear[ComputeOutputPosition];
ComputeOutputPosition[object_, message_]:=
Module[{x0, y0, z0, a, b, c, planes, line, tValues, tRules, tMin, exitFacePos,
        exitFace, t, results},
    EntryPrint["ComputeOutputPosition", object, message];
        (* Extract point of impact from the message *)
    {x0, y0, z0} = PointOfImpact /. message;
        (* Extract orientation of the incident ray from the message *)
    {a, b, c} = OrientationOfRay /. message;
        (* get all the six equations of the bounding planes of the object *)
    planes = boundingBoxEquations[object];
        (* Obtain the parametric representation of the ray. Parameter is 't',
            the distance from the place of impact.
            line is {x -> x0 + a t, y -> y0 + b t, z -> z0 + c t} *)
    DPrint["BoundingBox Planes ", planes];
    line = MapThread[Rule[#1, #2 + #3 #4]&, {{x, y, z}, {x0, y0, z0},
                                                        {a, b, c}, {t, t, t}}];
    DPrint["line = ", line];
        (* Solve for the points of intersection of the ray with all the
           six bounding planes. *)
    tRules = Map[NSolve[{Evaluate[# /. line]}, {t}]&, planes];
    DPrint["tRules = ", tRules];
        (* extract the 't' values. *)
    tValues = Map[ t /. # &, Flatten[tRules] ];
    DPrint["tValues = ", tValues];
        (* find the Minimum, Positive 't' value *)
```

```
    tMin = Min[Select[ tValues, Positive ]];
    DPrint["tMin = ", tMin];
        (* The face Index from which the ray leaves the box *)
    (* exitFacePos = Position[ tRules, t -> tMin][[1, 1]]; *)
        (* The equation of the plane corresponding to the exit face *)
    (* exitFace = planes[[exitFacePos]]; *)
        (* The point of exit from the box.
           - Find the point on the line at t = tMin.
           - Convert the list of rules to a point.*)
    results = {x, y, z} /. (line /. {t -> tMin});
    ExitPrint["ComputeOutputPosition", results];
    results
]

(* The main module that takes an object and a message. It extracts the type of
    message and applies the corresponding simulate[self] function. *)
Clear[performActions];
performActions[object_, message_]:=
Module[ {actions},
    actions = Action /. message;
    Map[performAction[object, message, #]&, actions]
]

Clear[performAction];       (* Check for Unit length vector in Action & Message *)
performAction[object_, message_, action_] /; (Head[action] === absolute) :=
Module[{mn, poi, newOrientation, newMessage, pos, index, distance, exitTime,
        results, cacheEntry, objectNumber},
    EntryPrint["performAction"];
    mn = (OHDL$MessageNumber = OHDL$MessageNumber + 1);
    poi = PointOfImpact /. message;
    objectNumber = ObjectNumber /. object;
        (* Extract the direction cosines of the new ray *)
    newOrientation = removeHead[action];
    newMessage = PickFirst[{{OrientationOfRay -> newOrientation}, message}];

    cacheEntry =
        cacheLookup[{StartPosition -> poi, Orientation -> newOrientation}];

    If[cacheEntry === {},
        pos = ComputeOutputPosition[object, newMessage];
        DPrint["absolute: Not in Cache, Calculated OutputPos = ", pos];
        distance = distanceBetween[pos, poi];
        cacheInstall[{StartPosition -> poi, Orientation -> newOrientation,
                EndPosition -> pos, Distance -> distance,
                EndObject -> objectNumber}],
        {pos, distance} = {EndPosition, Distance} /. cacheEntry
    ];

    index = RefractiveIndex /. Join[object, {RefractiveIndex -> 1}];
    exitTime =
        (TimeOfImpact /. message) + distance/(VelocityOfLight / 1(*index*));
    results = PickFirst[{
                {   Sender -> objectNumber,
                    MessageNumber -> mn,
                    TimeOfExit -> exitTime,
                    PlaceOfExit -> pos
                },
                newMessage
              }
    ];
    ExitPrint["performAction", results];
    results
]
```

```
performAction[object_, message_, action_]
    /; (Head[action] === relativeOrientation) :=
Module[{relativeMatrix, current, new},
    relativeMatrix = removeHead[action];
    current = OrientationOfRay /. message;
    new = current . relativeMatrix;
    performAction[object, message, absolute[new]]
]

performAction[args___]:=
    Print["Error :: performAction[] : called with arguments: [", args, "]"]
(* resets all global variables related to simulation statistics. *)
Clear[initializeStatistics];
initializeStatistics[]:=
Module[{},
    OHDL$ProcessedEvents = 0;
]
```

```
(*  *)
Clear[startSimulationEvent];
startSimulationEvent[object_]:=
Module[{objectNumber},
    objectNumber = ObjectNumber /. object;
    OHDL$MessageNumber = OHDL$MessageNumber + 1;
    OHDL$EventNumber = OHDL$EventNumber+1;
    {MessageType -> Start, EventNumber -> OHDL$EventNumber,
        MessageNumber -> OHDL$MessageNumber,
        EventTime -> 0, Sender -> -1, Receiver -> objectNumber
    }
]
showEvents[events__]:=
    Print[events]
scheduleEvents[events_]:=
Module[{eventTimes},
    eventTimes = EventTime /. events;
    If[Min[eventTimes] < OHDL$SimulationClock,
        Print[
            "Error::scheduleEvents[]: Attempt to schedule an event of the past "];
        showEvents[events],
        (* else *)
        OHDL$EventList = sortRuleLists[Join[events, OHDL$EventList], EventTime]
    ]
]
scheduleEvent[event_]:=
    scheduleEvents[{event}]
```

```
(*  *)
Clear[initializeSimulation];
initializeSimulation[]:=
Module[{},
    Put[Null, OHDL$EventLogFile];
    initializeCache[];
    OHDL$SimulationClock = 0;
    OHDL$EventNumber = 0;
    OHDL$MessageNumber = 0;
    OHDL$EventList = {};
    initializeStatistics[];
    OHDL$StartCPUTime = TimeUsed[];
    OHDL$SystemMemoryUsed = MemoryInUse[];
    scheduleEvents[Map[startSimulationEvent, OHDL$CurrentObjects]];
    (* for every component in the architecture, generate a start event put it
       in the event list scheduled for time 0 *)
]
(*  *)
Clear[getNextEvent];
getNextEvent[]:=
Module[{first},
    first = First[OHDL$EventList];
    OHDL$EventList = Rest[OHDL$EventList];
    PutAppend[first, OHDL$EventLogFile];
    first
]
```

```
(*  *)
Clear[updateStatistics];
updateStatistics[]:=
Module[{},
    OHDL$ProcessedEvents = OHDL$ProcessedEvents + 1;
]
(*  *)
Clear[interpretSimulationResults];
interpretSimulationResults[]:=
Module[{},
    Print["Number of processed events = ", OHDL$ProcessedEvents];
    Print["Simulation ended at Time = ", OHDL$SimulationClock];
    Print["CPU Time for simulation = ", TimeUsed[] - OHDL$StartCPUTime];
    Print["Memory used in Simulation = ", MemoryInUse[] - OHDL$SystemMemoryUsed];

]
(*  *)
Clear[showSimulationConfiguration];
showSimulationConfiguration[heading_String:""]:=
Module[{},
    Print[heading];
    Print["Complete Configuration:"];
    Print["Current Event List is:\n", OHDL$EventList];
    Print["States of Objects:\n",
          Map[({ObjectNumber, State} /. #)&, OHDL$CurrentObjects ]];
    Print["Current Simulation time = ",    OHDL$SimulationClock];
    Print["CPU Time used so far = ", TimeUsed[] ];
    Print["Memory used so far = ", MemoryInUse[] ];
]
```

```
Clear[modifyAttribute];
modifyAttribute[objectNumber_, attribute_, newValue_]:=
Module[{f},
    f = Function[object,
            If[(ObjectNumber /. object) != objectNumber, object,
                    PickFirst[{{attribute -> newValue}, object}]
            ]
        ];
    OHDL$CurrentObjects = Map[f, OHDL$CurrentObjects];
]

Clear[changeState];
changeState[objectNumber_, newState_]:=
    modifyAttribute[objectNumber, State, newState]
```

```
Clear[pointOfImpact];
pointOfImpact[object_, startingPoint_, orientation_]:=
Module[{faces, objectNumber, pointsOfIntersection, nonEmptyPoints,
        distancePointPairs, distance, pnt, dist},
    EntryPrint["pointOfImpact", "Args = ", object, startingPoint, orientation];
    faces = Faces /. object;
    objectNumber = ObjectNumber /. object;
    pointsOfIntersection = Map[
        pointOfIntersection[
            line[point[startingPoint], directionCosines[orientation]],
            polygon[#]
        ]&,
        faces
    ];
    DPrint["points of Intersection = ", pointsOfIntersection];
    nonEmptyPoints = Select[pointsOfIntersection, (# =!= {})&];
    DPrint["nonEmptyPoints = ", nonEmptyPoints];
    {distance, pnt, objectNumber} =
        If[nonEmptyPoints === {}, {Infinity, {-1, -1, -1}, objectNumber},
            distancePointPairs =
                Sort[ Map[{squaredDistanceBetween[#, startingPoint], #}&,
                                                    nonEmptyPoints] ];
            DPrint["distances = ", distancePointPairs];
            {dist, pnt} = First[distancePointPairs];
            { Chop[N[ Sqrt[dist] ]], pnt, objectNumber}
    ];
    ExitPrint["pointOfImpact", "Output = ", distance, pnt, objectNumber];
    {distance, pnt, objectNumber}
]
```

```
(*  *)
Clear[processMessage];
processMessage[message_]:=
Module[{messageType, orientation, startingPoint, startingTime, sender,
        distancePointObjectNumberTriplets, distance, point, receiver,
        timeOfImpact, newEvent, cacheEntry},
    EntryPrint["processMessage", message];
    messageType = MessageType /. message;
    orientation = OrientationOfRay /. message;
    startingPoint = PlaceOfExit /. message;
    startingTime = TimeOfExit /. message;
    sender = Sender /. message;

    cacheEntry =
        cacheLookup[{StartPosition -> startingPoint, Orientation -> orientation

    If[cacheEntry === {},
        distancePointObjectNumberTriplets =
            Sort[Map[pointOfImpact[#, startingPoint, orientation]&,
                    OHDL$CurrentObjects
                ]
            ];
        DPrint["Not in Cache, Calculated distancePointObjectNumberTriplets ",
            distancePointObjectNumberTriplets];
        {distance, point, receiver} = First[distancePointObjectNumberTriplets];
        cacheInstall[{StartPosition -> startingPoint, Orientation -> orientatic
                EndPosition -> point, Distance -> distance,
                EndObject -> receiver}],
        (* else *)
        {distance, point, receiver} =

            {Distance, EndPosition, EndObject} /. cacheEntry
    ];


    timeOfImpact = N[distance/VelocityOfLight + startingTime];
        (* Change for different refractive Index *)

    newEvent = PickFirst[{{ Sender -> sender,
                            Receiver -> receiver,
                    MessageNumber ->
                        (OHDL$MessageNumber = OHDL$MessageNumber + 1),
                            (* MessageType -> ?, *)
                            StartingTime -> startingTime,
                            PlaceOfOrigin -> startingPoint,
                            EventTime -> timeOfImpact,
                            TimeOfImpact -> timeOfImpact,
                            PointOfImpact -> point,
                            OrientationOfRay -> orientation (* ,
                            Action -> ? *)

                            },
                        message}
                ];

    If[distance =!= Infinity,
        scheduleEvent[newEvent],
        DPrint["Error::processMessage[]:Message escaping system ", message]
    ];
    ExitPrint["processMessage", newEvent];
    newEvent
    (* Switch[eventType,
```

```
        _, Print[
            "Error::processMessage[]:Unknown Type of event. Ignored ", event]
    ]
        *)
]
(*  *)
Clear[processMessages];
processMessages[messages__]:=
Module[{},
    Map[processMessage, messages]
]
Clear[simulateComponent];
```

```
simulateComponent[BoundingBox[object_], message_]:=
Module[{state, messageType},
    DPrint["Simulate BoundingBox called"];
    DPrint["Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;
    Switch[messageType,
        Start,
            Switch[state,
                Down,
                    changeState[ObjectNumber /. object, Up];
                    DPrint["Initialized BoundingBox"],
                _,
                    Print["Error::SimulateComponent[]:BoundingBox already Up"]
            ],
        Stop,
            Switch[state,
                Up,
                    changeState[ObjectNumber /. object, Down];
                    DPrint["Shutting off BoundingBox"],
                _,
                    Print["Error::SimulateComponent[]:BoundingBox already Down"]
            ],
        _,
            Print["Warning::Ray escaping into free space ",
                                        PointOfImpact /. message]
    ];
    {}
]
```

```
simulateComponent[InterferenceFilter[object_], message_]:=
Module[{state, messageType, none, action, illegalAction},
    DPrint["Simulate InterferenceFilter called"];
    DPrint["Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;
    Switch[messageType,
        Start,
            Switch[state,
                Down,
                    changeState[ObjectNumber /. object, Up];
                    DPrint["Initialized InterferenceFilter"],
                _,
                    Print[
                    "Error::SimulateComponent[]:InterferenceFilter already Up"]
            ],
        Stop,
            Switch[state,
                Up,
                    changeState[ObjectNumber /. object, Down];
                    DPrint["Shutting off InterferenceFilter"],
                _,
                    Print[
                    "Error::SimulateComponent[]:InterferenceFilter already Down
            ],
        Normal,
            Switch[state,
                Up,
                    action = {relativeOrientation[IdentityMatrix[3]]};
                    DPrint["InterferenceFilter Normal message in Up state",

                            action],
                Down,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:InterferenceFilter Down"]
            ],
        _,
            Print["Error::Unknown Message Type "]
    ];
    Switch[action,
        illegalAction,
            Print["simulateComponent[InterferenceFilter[]]: Unexpected message ",
                    object, message]; {},
        none, {},
        _,
            performActions[object, PickFirst[Prepend[message, Action -> action]]]
    ]
]

simulateComponent[Mirror[object_], message_]:=
Module[{state, messageType, action, illegalAction, none},
    DPrint["Simulate Mirror called"];
    DPrint["Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;

    Switch[messageType,
        Start,
            Switch[state,
                Down,
```

```
                    changeState[ObjectNumber /. object, Up];
                    action = none;
                    DPrint["Initialized Mirror"],
                Up,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:Mirror already Up"]
            ],
        Stop,
            Switch[state,
                Up,
                    changeState[ObjectNumber /. object, Down];
                    action = none;
                    DPrint["Shutting off Mirror"],
                Down,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:Mirror already Down"]
            ],
        Normal,
            Switch[state,
                Up,
                action =
                    {relativeOrientation[{{0, 1., 0}, {-1., 0, 0}, {0, 0, 1.}}]
                    DPrint["Mirror Normal message in Up state", action],
                Down,
                    action = illegalAction,
                _,
                    action = none;

                    Print["Error::SimulateComponent[]:Mirror Down"]
            ],


        _,
            Print["Warning::simulateComponent[Mirror[]]:Unknown Message Type",
                                                    message]
    ];

    Switch[action,
        illegalAction,
            Print["simulateComponent[BeamSplitter[]]: Unexpected message ",
                    object, message]; {},
        none, {},
        _,
            performActions[object, PickFirst[Prepend[message, Action -> action]]]
    ]
]

simulateComponent[BeamSplitter[object_], message_]:=
Module[{state, messageType, action, illegalAction, none, action1, action2,
        results, size, orientationOfRay, newPointOfImpact, pointOfImpact},
    EntryPrint["simulateComponent", "Object = ", object, "Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;

    size = Size /. object;
    orientationOfRay = OrientationOfRay /. message;
    pointOfImpact = (PointOfImpact /. message);
    newPointOfImpact = (size orientationOfRay)/2 + pointOfImpact;
    DPrint["Changing pointOfImpact from ", pointOfImpact, " to ",
```

```
                newPointOfImpact];

Switch[messageType,
    Start,
        Switch[state,
            Down,
                changeState[ObjectNumber /. object, Up];
                action = none;
                DPrint["Initialized BeamSplitter"],
            Up,
                action = illegalAction,
            _,
                action = none;
                Print["Error::SimulateComponent[]:BeamSplitter already Up"]
        ],
    Stop,
        Switch[state,
            Up,
                changeState[ObjectNumber /. object, Down];
                action = none;
                DPrint["Shutting off BeamSplitter"],
            Down,
                action = illegalAction,
            _,
                action = none;
                Print["Error::SimulateComponent[]:BeamSplitter already Down
        ],
    Normal,
        Switch[state,
            Up,
                action1 = relativeOrientation[IdentityMatrix[3]];
                action2 = If[eqQ[First[orientationOf[object]], 0.0],
                        (* -90 Degree *)
                  relativeOrientation[{{0, -1., 0}, {1., 0, 0}, {0, 0, 1.}}
                        (* +90 Degree *)
                  relativeOrientation[{{0, 1., 0}, {-1., 0, 0}, {0, 0, 1.}}
                ];
                action  = {action1, action2};
                DPrint["Beam Splitter Normal message in Up state ", action]
            Down,
                action = illegalAction,
            _,
                action = none;
                Print["Error::SimulateComponent[]:BeamSplitter Down"]
        ],


    _,
        Print[
         "Warning::simulateComponent[BeamSplitter[]]:Unknown Message Type",
                                            message]
];

results = Switch[action,
    illegalAction,
        Print["simulateComponent[BeamSplitter[]]: Unexpected message ",
                object, message]; {},
    none, {},
    _,
        performActions[object,
            PickFirst[{{Action -> action, PointOfImpact -> newPointOfImpact
                                            message}}]]
```

```
    ];
    ExitPrint["simulateComponent", results];
    results
]

simulateComponent[PolarizingBeamSplitter[object_], message_]:=
    simulateComponent[BeamSplitter[object], message]

simulateComponent[Laser[object_], message_]:=
Module[{state, messageType, action, illegalAction, none},
    DPrint["Simulate Laser called"];
    DPrint["Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;

    Switch[messageType,
        Start,
            Switch[state,
                Down,
                    changeState[ObjectNumber /. object, Up];
                    action = none;
                    DPrint["Initialized Laser"],
                Up,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:Laser already Up"]
            ],
        Stop,
            Switch[state,
                Up,
                    changeState[ObjectNumber /. object, Down];
                    action = none;
                    DPrint["Shutting off Laser"],
                Down,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:Laser already Down"]
            ],
        Normal,
            Switch[state,
                Up,
                    action = If[eqQ[First[orientationOf[object]], 0.0],
                      {absolute[{1, 0, 0}]},
                      {absolute[{-1, 0, 0}]}
                    ];
                    DPrint["Laser ", action],
                Down,
                    action = illegalAction,
                _,
                    action = none;
                    Print["Error::SimulateComponent[]:Laser Down"]
            ],


        _,
            Print["Warning::simulateComponent[Laser[]]:Unknown Message Type",
                                                  message]
    ];

    Switch[action,
        illegalAction,
```

```
            Print["simulateComponent[Laser[]]: Unexpected message ",
                  object, message]; {},
        none, {},
        _,
            performActions[object, PickFirst[Prepend[message, Action -> action]]]
    ]
]

simulateComponent[PulsedLaser[object_], message_]:=
    simulateComponent[Laser[object], message]

simulateComponent[receivingObject_, message_]:=
Module[{type, state, messageType, action, object, illegalAction, none},
    type = ToString[Head[receivingObject]];
    object = removeHead[receivingObject];
    DPrint["Default Simulate component called for ObjectType = ", type];
    DPrint["Message = ", message];
    state = State /. object;
    messageType = MessageType /. message;
    Switch[type,
        "HalfWavePlate",
            action = illegalAction,
        "QuarterWavePlate",
            action = illegalAction,
        "Screen",
            action = illegalAction,
        "Sink",
            action = illegalAction,
        _,
            Print["Error::simulateComponent[] called with unknown component ",
                                                                object];
            action = illegalAction
    ];
    Switch[action,
        illegalAction,
            DPrint[
            "simulateComponent[]: Unexpected message ", object, message]; {},
        none, {},
        _,
            performActions[removeHead[object],
                        PickFirst[Prepend[message, Action -> action]]]
    ]
]

simulateDriver[receiver_, message_]:=
Module[{object, type},
    DPrint["Arguments::simulateDriver[]", receiver, message];
    object = Select[OHDL$CurrentObjects, ((ObjectNumber /. #) == receiver)&];
    If[object =!= {}, {object} = object,
        showSimulationConfiguration[
            listToString[
                {"Error::simulateDriver[]: Called with illegal object Number ",
                    receiver}
            ]   (* Check listToString *)
        ]
    ];
    type = ToExpression[ComponentType /. object];
    simulateComponent[type[object], message]
]

(* The top level routine for simulating an architecture. *)
Clear[simulateArchitecture];
simulateArchitecture[options___]:=
Module[{simulationDone, showSteps, event, receiver, newMessageList,
```

```
        maxSimulationTime},
    EntryPrint["simulateArchitecture"];
    showSteps = ShowSteps /. {options, ShowSteps -> False};
    maxSimulationTime =
        MaxSimulationTime /. {options, MaxSimulationTime -> Infinity};

    initializeSimulation[];
    simulationDone = False;
    If[showSteps, showSimulationConfiguration["Initial Config:"]];
    While[! simulationDone,
        event = getNextEvent[];
        OHDL$SimulationClock = EventTime /. event;
        receiver = Receiver /. event;
        newMessageList = simulateDriver[receiver, event];
        newMessageList = Map[
            PickFirst[
                {{PredecessorMessageNumber ->
                    (MessageNumber /. Append[event, MessageNumber -> 0])}}, #}
            ]&,
            newMessageList
        ];
        DPrint["Message List = ", newMessageList]
        processMessages[newMessageList];
        updateStatistics[];
        If[showSteps, showSimulationConfiguration["Current Config:"]];
        simulationDone = (OHDL$EventList === {}) ||
                        (OHDL$SimulationClock >= maxSimulationTime)
    ];
    interpretSimulationResults[];
    Print["Simulation done ... "];
    ExitPrint["simulateArchitecture"];

]
```

Generic Object Simulation

■ Simulation

```
initialize[];

object = GetObject["BeamSplitter"]

object = {ObjectNumber -> 1, Variables -> {x$1, y$1, z$1, r$1, s$1, t$1},
  Angle -> 45*Degree, ComponentType -> "BeamSplitter", Material -> "BK7",
  Reflectance -> 0.6, RefractiveIndex -> 1.5, Size -> {0.02, 0.02, 0.02},
  Transmission -> 0.4}

{ObjectNumber -> 1, Variables -> {x$1, y$1, z$1, r$1, s$1, t$1},
  Angle -> 45 Degree, ComponentType -> BeamSplitter, Material -> BK7,
  Reflectance -> 0.6, RefractiveIndex -> 1.5, Size -> {0.02, 0.02, 0.02},
  Transmission -> 0.4}

Coordinates = {}

{}

Orientations = N[{r$1 -> 30 Degree, s$1 -> 45 Degree, t$1 -> 60 Degree}]

{r$1 -> 0.523599, s$1 -> 0.785398, t$1 -> 1.0472}

message = {  Sender ->?, (* who is the originator of this message? *)
             Receiver -> ?, (* the receiving object *)
             MessageNumber ->, (* sequence number -- unique *)
             MessageType -> Test, (* what type of message is this? an Event, Control, Test ... *)
             MessageContents -> Ray, (* what are the contents of the message ? *)
```

```
        StartingTime -> ?, (* When did the message start *)
        PlaceOfOrigin -> {0.1, 0.1, 0.1}, (* where is the ray coming from? *)
        TimeOfImpact -> ?, (* for an Event type message when did the event occur? *)
        PointOfImpact -> {0, 0, 0}, (* for an Event type what is the reference point? *)
        OrientationOfRay -> {1, 1, 1} / N[Sqrt[3]], (* unit vector expressed in the
                                        direction cosines in the direction of the ray *)
        Action -> PassItThrough (* type of action to be carried out in response to this message *)

        PlaceOfExit -> ?, (* Sender's view of message. Changes to PlaceOfOrigin *)
        TimeOfExit -> ?, (* Sender's view of message. Changes to StartingTime *)

        EventTime -> ?, (* when is the event going to occur *)
};
```

```
message = {
Sender -> CurrentObjectNumber, (* who is the originator of this message? *)
Receiver -> ?
MessageNumber -> CurrentMessageNumber, (* sequence number  -- unique *)
MessageType -> Test, (* what type of message is this? an Event,
                    Control, Test ... *)
MessageContents -> Ray, (* what are the contents of the message ? *)
TimeOfImpact -> CurrentTime, (* for an Event type
                        message when did the event occur? *)
PointOfImpact -> {0, 0, 0}, (* for an Event type
                        what is the reference point? *)
PlaceOfOrigin -> {0.1, 0.1, 0.1}, (* where is the ray coming from? *)
OrientationOfRay -> {1, 1, 1} / N[Sqrt[3]], (* unit vector expressed in the
        direction cosines in the direction of the ray *)
Action -> PassItThrough (* type of action to be carried
                        out in response to this message *)



 };

showState[PositionInfo -> True, OrientationInfo -> False,
        ConstraintInfo -> False, FullInfo -> True]

orientationOf[object]

{0, 0, 0}

eqs = boundingBoxEquations[ object ]

points = Union[ Flatten[newBoundingBoxPoints, 1] ]
```

```
Table[Map[ pointOnThePlaneQ[#, eqs[[i]] ]&, points],
        {i, Length[eqs]}]//TableForm

Table[Map[ pointOnThePlaneQ[points[[i]], # ]&, eqs],
        {i, Length[points]}]//TableForm

out = ComputeOutputPosition[ object, message ]

Map[pointOnThePlaneQ[ out, #]&, eqs]

simulate[object, message]
```

Simulator Action

■ SerialAdder

■ Michelson

Initialization

```
initialize[]
```

■ TripFlop

□ From OldStuff

□ Modified

□ Current

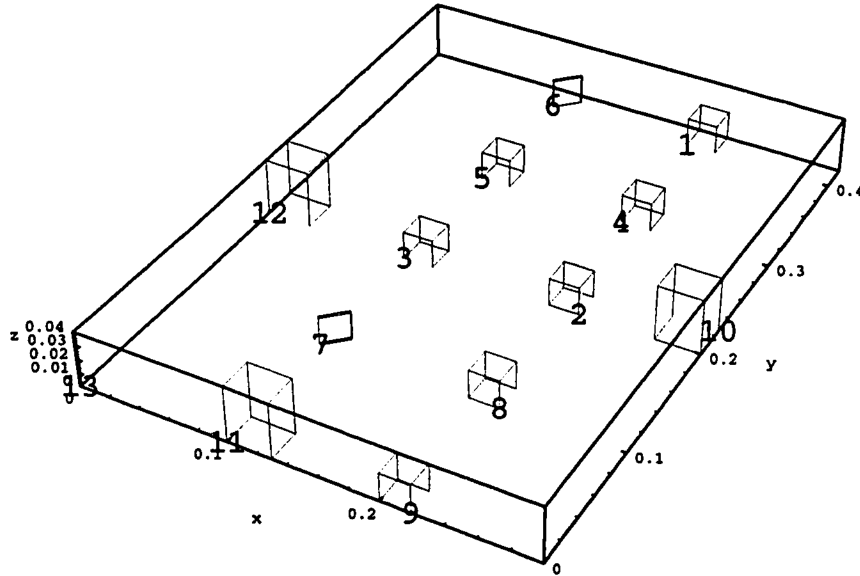
```
fileName = "~/Packages/Examples/TripFlop.coords";

coords = SetPrecision[Get[fileName], $MachinePrecision - 4];

shiftObject[objectNumber_, axisNumber_, displacement_]:=
Module[{vars, vals, f},
    vars = Map[
            ToExpression[
                StringJoin[#, "$", ToString[objectNumber]]
            ]&,
            {"x", "y", "z", "r", "s", "t"}
        ];
    vals = vars /. Coordinates;
    vals[[axisNumber]] = vals[[axisNumber]] + displacement;
    Coordinates =
        PickFirst[{vars[[axisNumber]] -> vals[[axisNumber]], Coordinates}];

    f = Function[{p, d},
            Table[If[i != axisNumber, p[[i]], p[[i]] + d], {i, Length[p]}]];
    coords[[objectNumber]] =
        Map[f[#, displacement]&, coords[[objectNumber]], {2}]
]

shiftObject[2, 1, 0.02];
shiftObject[8, 1, 0.02];
shiftObject[9, 1, 0.02];
shiftObject[10, 2, 0.025];

OHDL$CurrentObjects =
    MapThread[Join[#1,
            {State -> Down, Faces -> #2}]&, {OldCurrentObjects, coords}];
```

```
coords //TableForm

boundingBoxFaces =
    Map[removeHead,
        cuboid[
            Apply[Sequence,
                Transpose[corners =
                    Map[{Min[#], Max[#]}&,
                        Transpose[ Flatten[coords, 2] ]
                    ]
                ]
            ]
        ]
    ]
{{{0, 0, 0}, {0, 0, 0.04}, {0, 0.42, 0.04}, {0, 0.42, 0}},
  {{0.3, 0, 0}, {0.3, 0, 0.04}, {0.3, 0.42, 0.04}, {0.3, 0.42, 0}},
  {{0, 0, 0}, {0, 0, 0.04}, {0.3, 0, 0.04}, {0.3, 0, 0}},
  {{0, 0.42, 0}, {0, 0.42, 0.04}, {0.3, 0.42, 0.04}, {0.3, 0.42, 0}},
  {{0, 0, 0}, {0, 0.42, 0}, {0.3, 0.42, 0}, {0.3, 0, 0}},
  {{0, 0, 0.04}, {0, 0.42, 0.04}, {0.3, 0.42, 0.04}, {0.3, 0, 0.04}}}

CurrentObjectNumber = Length[OHDL$CurrentObjects]
12
```

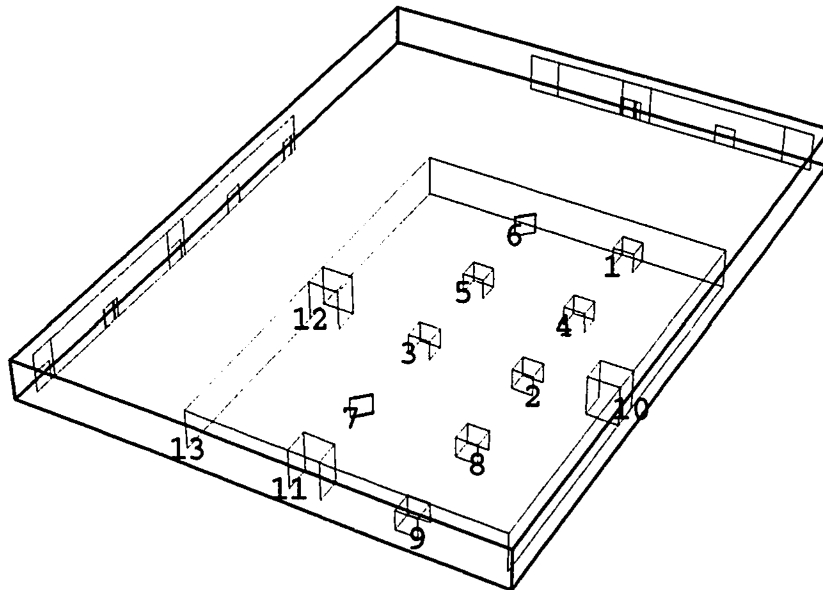
```
createObject = Function[{faces, type},
        CurrentObjectNumber=CurrentObjectNumber+1;
        {ObjectNumber -> CurrentObjectNumber, Faces -> faces,
        State -> Down,
        Variables ->
            Map[
                ToExpression[
                    StringJoin[#, "$", ToString[CurrentObjectNumber]]
                ]&,
                {"x", "y", "z", "r", "s", "t"}
            ],
        ComponentType -> type (* ,Material -> "BK7",
        Size -> {0.02, 0.02, 0.02}, Transmission -> 0.4 *) }]
Function[{faces, type}, CurrentObjectNumber = CurrentObjectNumber + 1;
   {ObjectNumber -> CurrentObjectNumber, Faces -> faces, State -> Down,
    Variables ->
     (ToExpression[#1<>$<>ToString[CurrentObjectNumber]] & ) /@
      {x, y, z, r, s, t}, ComponentType -> type}]

OHDL$CurrentObjects =
    Join[OHDL$CurrentObjects,
        {createObject[boundingBoxFaces, "BoundingBox"]}
    ];

boundingBoxVariables = Variables /. Last[OHDL$CurrentObjects]
{x$13, y$13, z$13, r$13, s$13, t$13}
```

```
Coordinates =
Flatten[
    Append[Coordinates, Map[ Rule[#, 0]&,
            boundingBoxVariables[[{1, 2, 3}]] ] ]
]
{x$1 -> 0.2, x$10 -> 0.3, x$11 -> 0.1, x$12 -> 0., x$2 -> 0.22, x$3 -> 0.1,
  x$4 -> 0.2, x$5 -> 0.1, x$6 -> 0.1, x$7 -> 0.1, x$8 -> 0.22, x$9 -> 0.22,
  y$1 -> 0.4, y$10 -> 0.225, y$11 -> 0., y$12 -> 0.2, y$2 -> 0.2,
  y$3 -> 0.2, y$4 -> 0.3, y$5 -> 0.3, y$6 -> 0.4, y$7 -> 0.1, y$8 -> 0.1,
  y$9 -> 0., z$1 -> 0., z$10 -> 0., z$11 -> 0., z$12 -> 0., z$2 -> 0.,
  z$3 -> 0., z$4 -> 0., z$5 -> 0., z$6 -> 0., z$7 -> 0., z$8 -> 0.,
  z$9 -> 0., x$13 -> 0, y$13 -> 0, z$13 -> 0}
Orientations =
Flatten[
    Append[Orientations, Map[ Rule[#, 0]&,
            boundingBoxVariables[[{4, 5, 6}]] ]]
]
{r$10 -> -3.141592653589793, r$7 -> -0.7853981633974483,
  r$2 -> 1.570796326794897, r$9 -> 1.570796326794897, s$5 -> 0.,
  s$6 -> 1.570796326794897, s$7 -> -1.570796326794897, s$2 -> 0.,
  s$9 -> 0., t$5 -> 0., t$6 -> 0., t$7 -> 0., t$2 -> 0., t$9 -> 0.,
  r$11 -> 0., r$4 -> 0., r$5 -> 0., r$6 -> 0.7853981633974483, s$11 -> 0.,
  s$12 -> 0., s$4 -> 0., t$11 -> 0., t$12 -> 0., t$4 -> 0., r$12 -> 0.,
  r$3 -> 0., s$1 -> 0., s$3 -> 0., s$8 -> 0., t$1 -> 0., t$3 -> 0.,
  t$8 -> 0., r$1 -> 0., r$8 -> 1.570796326794897, r$13 -> 0, s$13 -> 0,
  t$13 -> 0}
CurrentObjectNumber
13

Coordinates
{x$1 -> 0.2, x$10 -> 0.3, x$11 -> 0.1, x$12 -> 0., x$2 -> 0.22, x$3 -> 0.1,
  x$4 -> 0.2, x$5 -> 0.1, x$6 -> 0.1, x$7 -> 0.1, x$8 -> 0.22, x$9 -> 0.22,
  y$1 -> 0.4, y$10 -> 0.225, y$11 -> 0., y$12 -> 0.2, y$2 -> 0.2,
  y$3 -> 0.2, y$4 -> 0.3, y$5 -> 0.3, y$6 -> 0.4, y$7 -> 0.1, y$8 -> 0.1,
  y$9 -> 0., z$1 -> 0., z$10 -> 0., z$11 -> 0., z$12 -> 0., z$2 -> 0.,
  z$3 -> 0., z$4 -> 0., z$5 -> 0., z$6 -> 0., z$7 -> 0., z$8 -> 0.,
  z$9 -> 0., x$13 -> 0, y$13 -> 0, z$13 -> 0}
Orientations
{r$10 -> -3.141592653589793, r$7 -> -0.7853981633974483,
  r$2 -> 1.570796326794897, r$9 -> 1.570796326794897, s$5 -> 0.,
  s$6 -> 1.570796326794897, s$7 -> -1.570796326794897, s$2 -> 0.,
  s$9 -> 0., t$5 -> 0., t$6 -> 0., t$7 -> 0., t$2 -> 0., t$9 -> 0.,
  r$11 -> 0., r$4 -> 0., r$5 -> 0., r$6 -> 0.7853981633974483, s$11 -> 0.,
  s$12 -> 0., s$4 -> 0., t$11 -> 0., t$12 -> 0., t$4 -> 0., r$12 -> 0.,
  r$3 -> 0., s$1 -> 0., s$3 -> 0., s$8 -> 0., t$1 -> 0., t$3 -> 0.,
  t$8 -> 0., r$1 -> 0., r$8 -> 1.570796326794897, r$13 -> 0, s$13 -> 0,
  t$13 -> 0}
OHDL$CurrentObjects

numberedWireFrame = Function[object,
        {Map[Line, Faces /. object],
        Text[FontForm[ToString[ObjectNumber /. object],
                {$DefaultFont[[1]], 20.0}],
            ((Variables /. object)[[{1, 2, 3}]]) /. Coordinates
        ]}
    ];
objectList = Map[numberedWireFrame, OHDL$CurrentObjects];
```

```
(* Draws all objects in separate plots *)
plot = Show[ Map[
Graphics3D[{AbsolutePointSize[0.2], Thickness[0.0001], #}]&, objectList] ,
    PlotRange -> corners, Axes -> True, AxesLabel -> {"x", "y", "z"}];
```

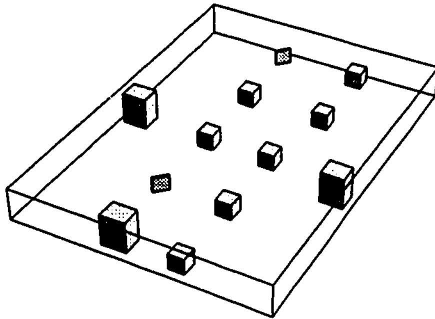



```
<<Graphics/Graphics3D.m;
Shadow[plot, ZShadow -> False];
```



```
(* Draws all planes for a component in separate plots *)
fullRange = Transpose[Transpose[corners] + {-0.05, 0.05}];
Map[ Show[ Graphics3D[{#}], PlotRange -> fullRange]&,
    Map[Polygon, coords[[11]] ] ]
```

```
Show[ Graphics3D[ Map[Polygon, Flatten[coords, 1]] ] ];
```



```
simulateArchitecture[]
```

Testing

```
    initializeSimulation[]
```

```
showSimulationConfiguration[]
```

```
Complete Configuration:

Current Event List is:{{MessageType -> Start, EventNumber -> 13,

    MessageNumber -> 13, EventTime -> 0, Sender -> -1, Receiver -> 13},

   {MessageType -> Start, EventNumber -> 12, MessageNumber -> 12,

    EventTime -> 0, Sender -> -1, Receiver -> 12},

   {MessageType -> Start, EventNumber -> 11, MessageNumber -> 11,

    EventTime -> 0, Sender -> -1, Receiver -> 11},

   {MessageType -> Start, EventNumber -> 10, MessageNumber -> 10,

    EventTime -> 0, Sender -> -1, Receiver -> 10},

   {MessageType -> Start, EventNumber -> 9, MessageNumber -> 9,

    EventTime -> 0, Sender -> -1, Receiver -> 9},

   {MessageType -> Start, EventNumber -> 8, MessageNumber -> 8,

    EventTime -> 0, Sender -> -1, Receiver -> 8},

   {MessageType -> Start, EventNumber -> 7, MessageNumber -> 7,

    EventTime -> 0, Sender -> -1, Receiver -> 7},

   {MessageType -> Start, EventNumber -> 6, MessageNumber -> 6,
```

```
    EventTime -> 0, Sender -> -1, Receiver -> 6},

   {MessageType -> Start, EventNumber -> 5, MessageNumber -> 5,

    EventTime -> 0, Sender -> -1, Receiver -> 5},

   {MessageType -> Start, EventNumber -> 4, MessageNumber -> 4,

    EventTime -> 0, Sender -> -1, Receiver -> 4},

   {MessageType -> Start, EventNumber -> 3, MessageNumber -> 3,

    EventTime -> 0, Sender -> -1, Receiver -> 3},

   {MessageType -> Start, EventNumber -> 2, MessageNumber -> 2,

    EventTime -> 0, Sender -> -1, Receiver -> 2},

   {MessageType -> Start, EventNumber -> 1, MessageNumber -> 1,

    EventTime -> 0, Sender -> -1, Receiver -> 1}}
States of Objects:{{1, Down}, {2, Down}, {3, Down}, {4, Down}, {5, Down},


   {6, Down}, {7, Down}, {8, Down}, {9, Down}, {10, Down}, {11, Down},

   {12, Down}, {13, Down}}
Current Simulation time = 0
CPU Time used so far = 44.3
Memory used so far = 1164264
```

```
scheduleEvents[{
{MessageType -> Normal, OrientationOfRay -> {-1, 0, 0}, Receiver -> 10,
    PointOfImpact -> {0.3, 0.21, 0.01}, EventTime -> 16, TimeOfImpact -> 16},
{MessageType -> Normal, OrientationOfRay -> {1, 0, 0}, Receiver -> 11,
    PointOfImpact -> {0.1, 0.01, 0.01}, EventTime -> 36, TimeOfImpact -> 36},
{MessageType -> Normal, OrientationOfRay -> {1, 0, 0}, Receiver -> 12,
    PointOfImpact -> {0, 0.21, 0.02}, EventTime -> 66, TimeOfImpact -> 66}
}
]
```

```
{{MessageType -> Start, EventNumber -> 1, MessageNumber -> 1, EventTime -> 0,
   Sender -> -1, Receiver -> 1},
  {MessageType -> Start, EventNumber -> 2, MessageNumber -> 2,
   EventTime -> 0, Sender -> -1, Receiver -> 2},
  {MessageType -> Start, EventNumber -> 3, MessageNumber -> 3,
   EventTime -> 0, Sender -> -1, Receiver -> 3},
  {MessageType -> Start, EventNumber -> 4, MessageNumber -> 4,
   EventTime -> 0, Sender -> -1, Receiver -> 4},
  {MessageType -> Start, EventNumber -> 5, MessageNumber -> 5,
   EventTime -> 0, Sender -> -1, Receiver -> 5},
  {MessageType -> Start, EventNumber -> 6, MessageNumber -> 6,
   EventTime -> 0, Sender -> -1, Receiver -> 6},
  {MessageType -> Start, EventNumber -> 7, MessageNumber -> 7,
   EventTime -> 0, Sender -> -1, Receiver -> 7},
  {MessageType -> Start, EventNumber -> 8, MessageNumber -> 8,
   EventTime -> 0, Sender -> -1, Receiver -> 8},
  {MessageType -> Start, EventNumber -> 9, MessageNumber -> 9,
   EventTime -> 0, Sender -> -1, Receiver -> 9},
  {MessageType -> Start, EventNumber -> 10, MessageNumber -> 10,
   EventTime -> 0, Sender -> -1, Receiver -> 10},
  {MessageType -> Start, EventNumber -> 11, MessageNumber -> 11,
   EventTime -> 0, Sender -> -1, Receiver -> 11},
  {MessageType -> Start, EventNumber -> 12, MessageNumber -> 12,
   EventTime -> 0, Sender -> -1, Receiver -> 12},
  {MessageType -> Start, EventNumber -> 13, MessageNumber -> 13,
   EventTime -> 0, Sender -> -1, Receiver -> 13},
  {MessageType -> Normal, OrientationOfRay -> {-1, 0, 0}, Receiver -> 10,
   PointOfImpact -> {0.3, 0.21, 0.01}, EventTime -> 16, TimeOfImpact -> 16},
  {MessageType -> Normal, OrientationOfRay -> {1, 0, 0}, Receiver -> 11,
   PointOfImpact -> {0.1, 0.01, 0.01}, EventTime -> 36, TimeOfImpact -> 36},
  {MessageType -> Normal, OrientationOfRay -> {1, 0, 0}, Receiver -> 12,
   PointOfImpact -> {0, 0.21, 0.02}, EventTime -> 66, TimeOfImpact -> 66}}
```

```
simulateArchitecture[ShowSteps -> True]
```

```
OHDL$Debug = False;
Do[Print[count];
    event = getNextEvent[];
    OHDL$SimulationClock = EventTime /. event;
    receiver = Receiver /. event;
    newMessageList = simulateDriver[receiver, event];
        newMessageList = Map[
            PickFirst[
                {{PredecessorMessageNumber ->
                    (MessageNumber /. Append[event, MessageNumber -> 0])}, #}
            ]&,
            newMessageList
        ];
    processMessages[newMessageList];
    updateStatistics[],
{count, 25}];
OHDL$Debug = True
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

■ Loop

```
event = getNextEvent[]
{MessageType -> Start, EventNumber -> 1, MessageNumber -> 1,
  EventTime -> 0, Receiver -> 1, Sender -> GlobalSimulator}
showSimulationConfiguration[]

Complete Configuration:
Current Event List is:{{MessageType -> Start, EventNumber -> 2,

    MessageNumber -> 2, EventTime -> 0, Receiver -> 2,

    Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 3, MessageNumber -> 3,
    EventTime -> 0, Receiver -> 3, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 4, MessageNumber -> 4,
    EventTime -> 0, Receiver -> 4, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 5, MessageNumber -> 5,
    EventTime -> 0, Receiver -> 5, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 6, MessageNumber -> 6,
    EventTime -> 0, Receiver -> 6, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 7, MessageNumber -> 7,
    EventTime -> 0, Receiver -> 7, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 8, MessageNumber -> 8,
    EventTime -> 0, Receiver -> 8, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 9, MessageNumber -> 9,
    EventTime -> 0, Receiver -> 9, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 10, MessageNumber -> 10,
```

```
    EventTime -> 0, Receiver -> 10, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 11, MessageNumber -> 11,
    EventTime -> 0, Receiver -> 11, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 12, MessageNumber -> 12,
    EventTime -> 0, Receiver -> 12, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 13, MessageNumber -> 13,
    EventTime -> 0, Receiver -> 13, Sender -> GlobalSimulator},
   {MessageType -> Normal, OrientationOfRay -> {-1, 0, 0}, Receiver -> 10,
    PointOfImpact -> {0.3, 0.21, 0.01}, EventTime -> 16, TimeOfImpact -> 16}
    , {MessageType -> Normal, OrientationOfRay -> {1, 0, 0},
    Receiver -> 11, PointOfImpact -> {0.1, 0.01, 0.01}, EventTime -> 36,
    TimeOfImpact -> 36}, {MessageType -> Normal,
    OrientationOfRay -> {1, 0, 0}, Receiver -> 12,
    PointOfImpact -> {0, 0.21, 0.02}, EventTime -> 66, TimeOfImpact -> 66}}
States of Objects:{{1, Down}, {2, Down}, {3, Down}, {4, Down}, {5, Down},

   {6, Down}, {7, Down}, {8, Down}, {9, Down}, {10, Down}, {11, Down},
   {12, Down}, {13, Down}}
Current Simulation time = 0
CPU Time used so far = 32.1167
Memory used so far = 977284
```

```
    OHDL$SimulationClock = EventTime /. event
0
    receiver = Receiver /. event
1
    newMessageList = simulateDriver[receiver, event]
{}
    processMessages[newMessageList]
{}
showSimulationConfiguration[]
```

```
Complete Configuration:
Current Event List is:{{MessageType -> Start, EventNumber -> 2,

    MessageNumber -> 2, EventTime -> 0, Receiver -> 2,
    Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 3, MessageNumber -> 3,
```

```
    EventTime -> 0, Receiver -> 3, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 4, MessageNumber -> 4,
    EventTime -> 0, Receiver -> 4, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 5, MessageNumber -> 5,
    EventTime -> 0, Receiver -> 5, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 6, MessageNumber -> 6,
    EventTime -> 0, Receiver -> 6, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 7, MessageNumber -> 7,
    EventTime -> 0, Receiver -> 7, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 8, MessageNumber -> 8,
    EventTime -> 0, Receiver -> 8, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 9, MessageNumber -> 9,
    EventTime -> 0, Receiver -> 9, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 10, MessageNumber -> 10,
    EventTime -> 0, Receiver -> 10, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 11, MessageNumber -> 11,
    EventTime -> 0, Receiver -> 11, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 12, MessageNumber -> 12,
    EventTime -> 0, Receiver -> 12, Sender -> GlobalSimulator},
   {MessageType -> Start, EventNumber -> 13, MessageNumber -> 13,
    EventTime -> 0, Receiver -> 13, Sender -> GlobalSimulator},
   {MessageType -> Normal, OrientationOfRay -> {-1, 0, 0}, Receiver -> 10,
    PointOfImpact -> {0.3, 0.21, 0.01}, EventTime -> 16, TimeOfImpact -> 16}
    , {MessageType -> Normal, OrientationOfRay -> {1, 0, 0},
    Receiver -> 11, PointOfImpact -> {0.1, 0.01, 0.01}, EventTime -> 36,
    TimeOfImpact -> 36}, {MessageType -> Normal,
    OrientationOfRay -> {1, 0, 0}, Receiver -> 12,
    PointOfImpact -> {0, 0.21, 0.02}, EventTime -> 66, TimeOfImpact -> 66}}
States of Objects:{{1, Up}, {2, Down}, {3, Down}, {4, Down}, {5, Down},

   {6, Down}, {7, Down}, {8, Down}, {9, Down}, {10, Down}, {11, Down},
   {12, Down}, {13, Down}}
Current Simulation time = 0
```

```
CPU Time used so far = 32.55
Memory used so far = 1013852
```

```
updateStatistics[]
```

```
MemoryInUse[]
```

```
1014540
```

Chapter 48

Code Generation

Chapter Abstract

This notebook contains all the routines related to code generation based on templates. It is used by OptiCAD to generate analyzers and simulators.

Code Generation

This notebook contains allthe routines related to code generation based on templates.
It is used by OptiCAD to generate analyzers and simulators.

```
Clear[fileToString];
fileToString[file_String]:=
Module[{elt},
    {elt} = ReadList[file, Word, WordSeparators -> {}, RecordSeparators -> {}];
    elt
]

Clear[filesToString];
filesToString[files__]:=
    StringJoin[Map[fileToString, files]]

OHDL$DistinctComponents =
    {"PolarizingBeamSplitter",
    "Mirror", "BoundingBox",
    "PulsedLaser", "InterferenceFilter"};

Clear[GetComponentSimulators];
GetComponentSimulators[]:=
Module[{},
    filesToString[Map[StringJoin[
                "~/Packages/OpticalComponents/", #, "Simulator.maCell"]&,
                OHDL$DistinctComponents
            ]
    ]
]


Clear[GenerateSimulator];
GenerateSimulator[options___]:=
Module[{template, outFile, targetLanguage},
    template = SimulatorTemplate /. {options,
        SimulatorTemplate -> "~/Packages/Simulator/SimulatorTemplate.ma"};
    outFile = OutputFile /. {options, OutputFile -> "/tmp/GeneratedTool.ma"};
    targetLanguage = TargetLanguage /.
            {options, TargetLanguage -> "Mathematica"};
    Switch[targetLanguage,
            "Mathematica",
                Splice[template, outFile, FormatType -> OutputForm],
            _,
                Print[
                    "Warning::GenerateSimulator[]:Translations into
                        the target language ",
                        targetLanguage, " are not supported yet"]
    ]
]

GenerateSimulator[SimulatorTemplate ->
    "~/Packages/Simulator/SimulatorTemplate.ma",
    TargetLanguage -> "C", OutputFile -> "/tmp/GeneratedSimulator.ma"]

Warning::GenerateSimulator[]:Translations into the target language C

   are not supported yet
```

```
GenerateSimulator[SimulatorTemplate ->
        "~/Packages/Simulator/SimulatorTemplate.ma",
    TargetLanguage -> "Mathematica", OutputFile ->
            "/tmp/GeneratedSimulator.ma"]
~/Packages/Simulator/SimulatorTemplate.ma
```

Chapter 49

3D Graphics Operations on Mathematica 3D graphics Objects

Chapter Abstract

Contains generic routines for carrying out various tranformations on 3D-graphics objects in general and Mathematica graphics objects in particular.

Operations on *Mathematica* 3D Graphics Objects

Contains generic routines for carrying out various tranformations on Mathematica graphics objects.

```
(* Translator function. Takes a Mathematica graphics object and
    translates it to a form suitable for transform3D[]. It takes the
    result of the transformation and translates it back to a Mathematica
    primitive object represenatation. Can work with:
        - Point
        - Line
        - Cuboid
        - Text
        - Graphics3D
        - and other higher level list representations of all the above. *)
Clear[transformMma3D];
transformMma3D[Point[coords_], operations__, options___]:=
Module[{p},
    {{p}} = transform3D[{{coords}}, operations, options ];
    Point[p]
]

transformMma3D[Line[points__], operations__, options___]:=
Module[{pts},
    {pts} = transform3D[{points}, operations, options ];
    Line[pts]
]


transformMma3D[Polygon[points__], operations__, options___]:=
Module[{pts},
    {pts} = transform3D[{points}, operations, options ];
    Polygon[pts]
]

transformMma3D[Cuboid[minPoint_], operations__, options___]:=
Module[{p},
    {{p}} = transform3D[{{minPoint}}, operations, options ];
    Cuboid[p]
]

transformMma3D[Cuboid[minPoint_, maxPoint_], operations__, options___]:=
Module[{minP, maxP},
    {{minP, maxP}} = transform3D[{{minPoint, maxPoint}}, operations, options ];
    Cuboid[minP, maxP]
]

transformMma3D[Text[expr_, coords_], operations__, options___]:=
Module[{p},
    {{p}} = transform3D[{{coords}}, operations, options ];
    Text[expr, p]
]

transformMma3D[Graphics3D[objects__], operations__, options___]:=
Module[{},
    Graphics3D[ Map[ transformMma3D[#, operations, options]&, objects] ]
]
```

```
transformMma3D[default_, operations_, options___] /; Head[default] =!= List :=
Module[{otherPrimitives},
    otherPrimitives = {AbsoluteDashing, AbsolutePointSize,
                       AbsoluteThickness, CMYKColor, Dashing,
                       EdgeForm, FaceForm, FontForm, GrayLevel, Hue,
                       PointSize, PostScript, RGBColor, Scaled,
                       SurfaceColor, Thickness};
    If[! MemberQ[ otherPrimitives, Head[default] ],
        Print["Warning :: transformMma3D[]: called with Default = ", default]
    ];
    default
]

transformMma3D[objects__, operations__, options___]:=
Module[{},
    Map[ transformMma3D[#, operations, options]&, objects]
]
```

Chapter 50

Architecture Translation to Chapter

Chapter Abstract

This notebook in conjuction with many of the utility files, takes an $\mathcal{OHDL}$ description of an architecture and generates the corresponding LaTeX*document.*

Architecture Translation to Chapter

PSCell is a global variable. The following assignment is a temporary fix for architectures without graphics.

```
PSCell = SinPSCell = ReadList["~/Packages/Misc/Sin.ps",
    Word, WordSeparators -> {}, RecordSeparators -> {}][[1]];

    translationsQ =
        (Head[#] === List) && (Length[Dimensions[#]] >= 2) &&
        ((Dimensions[#][[2]] == 4) && Dimensions[#][[1]] > 0)&;

    translationsQ[{{"", "input", Identity, {Echo -> True, Echo -> True}},
                {"", "input", Identity, {Echo -> True, Echo -> True}}}]
    True

    translationsQ[{{"", "input", Identity},
                {"", "input", Identity}}]
    False

    Clear[translateNotebook];
    (* Takes a notebook file and a table of Translations of the form
        {outputFile, cellType, translationFunction, translationSpecificOptions}
        Applies translationFunction to all cells of "cellType"
        within the context of a window and stores results in
        the file "outputFile" *)
    translateNotebook[notebookFile_String,
        translations_?translationsQ, options___] :=
    Module[{outputFile, cellType, translationFunction, translationOptions, f},
        f = Function[{translation},
                {outputFile, cellType, translationFunction,
                    translationOptions} = translation;

                translateNotebook[
                    notebookFile,
                    outputFile, {{cellType, translationFunction}},
                    Apply[Sequence, Join[translationOptions, {options}]]]
        ];
        Map[f, translations];
    ]

    (* Applies translationFunction to all cells of type cellType and stores results
        in outputFile *)
    translateNotebook[notebookFile_String, outputFile_String, cellType_String,
        translationFunction_, options___] /; (Head[translationFunction] =!= Rule):=
        translateNotebook[notebookFile, outputFile,
                {{cellType, translationFunction}}, options];

    (* Takes a notebook file and a table of Translations of the form
            {cellType, translationFunction}
        Applies translationFunction to all cells of "cellType".
        Stores all results in one file "outputFile"
        Options[translateNotebook] = {CellSeparator, PreserveCellHeaders,
                        OutputPageWidth, PreserveFontInfo, Echo, AppendToOutputFile
        CellSeparator (default = ""), Specify separator bw cells in output file.
        PreserveCellHeaders (default = If one cell then False else True), Have
            separate cells in output file or all merged into one.
        OutputPageWidth (default = Infinity), The word wrap width of output file.
        PreserveFontInfo (default = True), Preserve italics, bold, colors etc.
        Echo (default = False), echo results to screen too.
        AppendToOutputFile (default = False), do not delete output file if it exist
        WindowSize (default = 1), the context size of the sliding window in
            terms of number of cells. Assert[WindowSize > 0]
        WindowAnchor (default = 1), the cell on which the function is to be appliec
```

```
        within the window. Assert[1 <= WindowAnchor <= windowSize]
    NullCellsAtBoundaries (default = True), when True, the function is applied
        to each and every cell in the input file -- by providing Null entries
        in the buffer if needed;
        when False, function application is deffered until the buffer is
        filled with cells from the input file.
*)
translateNotebook[notebookFile_String, outputFile_String,
            translations_?MatrixQ, (* {{cellType, translationFunction}, ...} *)
            options___] :=
Module[{inFile, outFile, record, pos, cell, mmaCellHeaderPrefix,
        translatedCell, cellSeparator, validTypes, mmaCellHeaderSuffix,
        preserveCellHeaders, echo, preservedHeader, wordSeparators,
        currentCellType, cellTypes, translationFunction, appendFlag,
        outputPageWidth, fontInfoSeparator, preserveFontInfo, notebookEndMarker
        windowSize, windowAnchor, windowBuffer, nullCellsAtBoundaries,
        endOfFile, cellsToRead, bufferIndex},
    validTypes = {"clipboard", "completions", "footer", "header", "help",
                  "info", "input", "leftfooter", "leftheader", "message",
                  "name", "output", "postscript", "print", "section",
                  "smalltext", "special1", "special2", "special3", "special4"
                  "special5", "subsection", "subsubsection", "subsubtitle",
                  "subtitle", "text", "title"};
    wordSeparators = {";", "]"}; nullCell = "";
    mmaCellHeaderPrefix = "\n:[font = "; mmaCellHeaderSuffix = "]\n";
    fontInfoSeparator = "\n;[s]\n"; notebookEndMarker = "\n^*)";

    cellSeparator = CellSeparator /. {options, CellSeparator -> ""};
    preserveCellHeaders = PreserveCellHeaders /. {options,
                              PreserveCellHeaders -> (Length[cellTypes] > 1)};

    outputPageWidth = OutputPageWidth /. {options, OutputPageWidth -> Infinity}

    preserveFontInfo = PreserveFontInfo /. {options, PreserveFontInfo -> True};

    echo = Echo /. {options, Echo -> False};
    appendFlag = AppendToOutputFile /. {options, AppendToOutputFile -> False};

    windowSize = WindowSize /. {options, WindowSize -> 1};
    windowAnchor = WindowAnchor /. {options, WindowAnchor -> 1};
    nullCellsAtBoundaries =
        NullCellsAtBoundaries /. {options, NullCellsAtBoundaries -> True};

    cellTypes = Map[First, translations];
    If[Length[ Complement[cellTypes, validTypes] ] == 0,
        inFile = OpenRead[notebookFile];
        outFile = If[appendFlag,
                        OpenAppend[outputFile,
                              FormatType -> OutputForm,
                              PageWidth -> outputPageWidth],
                        OpenWrite[outputFile,
                              FormatType -> OutputForm,
                              PageWidth -> outputPageWidth]
                    ];
        windowBuffer = Table[nullCell, {windowSize}];
        endOfFile = False;
        bufferIndex = If[nullCellsAtBoundaries,
            windowAnchor,
            1
        ];
        While[(!endOfFile) && (bufferIndex < windowSize),
            windowBuffer[[bufferIndex]] = record = Read[inFile, Record,
```

```
                RecordSeparators -> {mmaCellHeaderPrefix}];
    endOfFile = (record === EndOfFile);
    If[endOfFile,
        windowBuffer[[bufferIndex]] = nullCell,
        bufferIndex++
    ];
];
While[(!endOfFile),
    windowBuffer[[windowSize]] = Read[inFile, Record,
                RecordSeparators -> {mmaCellHeaderPrefix}];
    endOfFile = (windowBuffer[[windowSize]] === EndOfFile);
    If[endOfFile,
        windowBuffer[[windowSize]] = nullCell,
    (* else *)
    record = windowBuffer[[windowAnchor]];
    (* Print[record]; *)
    currentCellType = frontToken[record, wordSeparators];
    If[ MemberQ[cellTypes, currentCellType],
            pos = First[Flatten[StringPosition[record,
                mmaCellHeaderSuffix] ] ];
            cell = StringDrop[record, pos+1];
            cell = If[preserveFontInfo,
                        cell,
                        frontToken[cell, {fontInfoSeparator,
                            notebookEndMarker}]
                    ];
                    (* Take care of the case where the fontInfoSeparato:
                        might appear in the cell itself *)
            preservedHeader =   If[preserveCellHeaders,
                                    StringJoin[ mmaCellHeaderPrefix,
                                                StringTake[record, pos]

                                    ],
                                    ""
                                ];
            translationFunction = Flatten[Select[translations,
                                            (#[[1]] == currentCellType)&
                                            ]
                                    ][[2]];
            translatedCell = If[windowSize == 1,
                    translationFunction[cell],
                    translationFunction[cell, windowBuffer, windowAncho:
            ];
            If[translatedCell != "",
                Map[If[# != "", Write[outFile, #]]&,
                    {preservedHeader, translatedCell, cellSeparator}];
                If[echo,
                    Map[Print, {preservedHeader,
                        translatedCell, cellSeparator}];
                ]
            ];
    ];
    windowBuffer = RotateLeft[windowBuffer, 1];
  ]; (* end of If[endOfFile] *)
];
If[nullCellsAtBoundaries,
 Do[ (* eliminate the duplication of the following code with the previc
        loop *)
    windowBuffer[[windowSize]] = nullCell;
    record = windowBuffer[[windowAnchor]];
    currentCellType = frontToken[record, wordSeparators];
    If[ MemberQ[cellTypes, currentCellType],
            pos = First[Flatten[StringPosition[record,
```

```
                        mmaCellHeaderSuffix] ] ];
                    cell = StringDrop[record, pos+1];
                    cell = If[preserveFontInfo,
                              cell,
                              frontToken[cell, {fontInfoSeparator,
                                      notebookEndMarker}]
                           ];
                           (* Take care of the case where the fontInfoSeparato:
                              might appear in the cell itself *)
                    preservedHeader =  If[preserveCellHeaders,
                                          StringJoin[ mmaCellHeaderPrefix,
                                                      StringTake[record, pos]
                                          ],
                                          ""
                                       ];
                    translationFunction = Flatten[Select[translations,
                                                   (#[[1]] == currentCellType)&
                                                   ]
                                          ][[2]];
                    translatedCell = If[windowSize == 1,
                           translationFunction[cell],
                           translationFunction[cell, windowBuffer, windowAncho:
                    ];
                    If[translatedCell != "",
                        Map[If[# != "", Write[outFile, #]]&,
                            {preservedHeader, translatedCell, cellSeparator}];
                        If[echo,
                            Map[Print, {preservedHeader,
                                    translatedCell, cellSeparator}];
                        ]
                    ];
            ];
            windowBuffer = RotateLeft[windowBuffer, 1],
         {bufferIndex - windowAnchor}];
        ];
         Close[inFile]; Close[outFile];
         Null,

         (* else *)
         Print["Warning :: translateNotebook[] : called with Unknown cell Type(s):
             Complement[cellTypes, validTypes]]
    ]
]

f = Function[{x, y, z}, (*Print[x, y, z];*)x];
translateNotebook["~/Packages/Misc/test.ma",
    {{"/tmp/ocg/output.ma",
    "input", f, {Echo -> True, WindowSize -> 20,
        NullCellsAtBoundaries -> True, WindowAnchor -> 3}}}]

textToTeXTable = {
    {"&", "\\&"},
    {" ", "\ "}};

Clear[TeXPattern];
TeXPattern[textPattern_String] :=
Module[{rules},
    rules = Map[Rule[#[[1]], #[[2]]]&, textToTeXTable];
    StringReplace[textPattern, rules]
]

TeXPattern["Vector-Op of AT&T Switching Fabrics"]
Vector-Op\ of\ AT\&T\ Switching\ Fabrics
```

```
Clear[chapterTranslations];
chapterTranslations[] :=
Module[(* {validTypes, oneParameterTeXMacro, nullStringFunction,
        commentedOneParameterTeXMacro, encapsulateFunction,
        titleTranslationFunction, sectionTranslationFunction,
        subsectionTranslationFunction, subsubsectionTranslationFunction,
        inputTranslationFunction, outputTranslationFunction,
        nameTranslationFunction, special1TranslationFunction,
        special2TranslationFunction, special3TranslationFunction,
        special4TranslationFunction, special5TranslationFunction,
        translationParametersTranslationFunction,
        abstractTranslationFunction, includeFileTranslationFunction,
        textTranslationFunction, smalltextTranslationFunction,
        clipboardTranslationFunction, completionsTranslationFunction,
        footerTranslationFunction, headerTranslationFunction,
        helpTranslationFunction,
        infoTranslationFunction, leftfooterTranslationFunction,
        leftheaderTranslationFunction, messageTranslationFunction,
        postscriptTranslationFunction, printTranslationFunction,
        subsubtitleTranslationFunction,
        subtitleTranslationFunction,
        translations
        }, *)
        {validTypes, oneParameterTeXMacro, nullStringFunction,
        commentedOneParameterTeXMacro, encapsulateFunction,
        translationParametersTranslationFunction,
        abstractTranslationFunction, includeFileTranslationFunction,
        translations
        },
    validTypes = {"clipboard", "completions", "footer", "header", "help",

            "info", "input", "leftfooter", "leftheader", "message",
            "name", "output", "postscript", "print", "section",
            "smalltext", "special1", "special2", "special3", "special4",
            "special5", "subsection", "subsubsection", "subsubtitle",
            "subtitle", "text", "title"};
    oneParameterTeXMacro[macroName_String] :=
            StringJoin["\\", macroName, "{", #, "}"] &;
    oneParameterTeXMacroWithTranslation[macroName_String] :=
            StringJoin["\\", macroName, "{", TeXPattern[#], "}"] &;
    twoParameterTeXMacro[macroName_String] :=
            StringJoin["\\", macroName, "{", #1, "}", "{", #2, "}"] &;
    nullStringFunction[] := "" &;
    commentedOneParameterTeXMacro[macroName_String] :=
        StringJoin["% \\", macroName, "{",
            StringReplace[#, "\n" -> "\n%"],
            "}"
        ] &;
    encapsulateFunction[texCommand_String] :=
        If[texCommand == "",
            StringJoin["{", #, "\n}\n"] &,
            StringJoin["{\\", texCommand, " ", #, "\n}\n"] &
        ];
    titleTranslationFunction = (chapTitle=TeXPattern[#];
                                StringJoin["\\chapter{", TeXPattern[#], "}"])&;
    sectionTranslationFunction = oneParameterTeXMacroWithTranslation["section"]
    subsectionTranslationFunction =
        oneParameterTeXMacroWithTranslation["subsection"];
    subsubsectionTranslationFunction =
        oneParameterTeXMacroWithTranslation["subsubsection"];
    (* acknowledgementTranslationFunction =
        oneParameterTeXMacro["section*"]; *)
```

```
        inputTranslationFunction =
            StringJoin["\\begin{verbatim}\n", #, "\n\\end{verbatim}\n"]&;
        inputTranslationFunction = StringJoin["\\MRun{", #, "}{Caption}"] &;
        inputTranslationFunction =
            If[StringPosition[#, "ShowArchitecture[]"] === {},
                StringJoin["\\MRun{", #, "}{Caption}"],
                StringJoin["\\MRun{", #, "}{Caption}\n",
                    "\\ArchFigure{",prefixPath,
                    "Figures/Architecture.eps}{3D-Layout of ", chapTitle,"}\n"]
        ]&;
        outputTranslationFunction = nullStringFunction[];
        postscriptTranslationFunction =
            Function[{cell},
                PSCell = cell;
                Splice["~/Packages/Templates/EPSTemplate",
                    StringJoin["~/", prefixPath, "Figures/Architecture.eps"],
                    FormatType -> TextForm];""
            ];
        nameTranslationFunction = commentedOneParameterTeXMacro["comment"];
        special1TranslationFunction =
            translationParametersTranslationFunction = (ToExpression[#];"") &;
        special2TranslationFunction =
            (translationEffective =
                If[translationEffective === False, True, False])&;
        special3TranslationFunction =
            StringJoin["\\FIGURE{", prefixPath, "}{", #, "}"] &;
        abstractTranslationFunction =
         special4TranslationFunction =
            Function[string,"\\chapterabstract{" <> string <> "}\n"];
        includeFileTranslationFunction =
         special5TranslationFunction = StringJoin["\\input ", prefixPath, #] &;
        textTranslationFunction = Identity;
        smalltextTranslationFunction = encapsulateFunction["small"];
        clipboardTranslationFunction = completionsTranslationFunction =
        footerTranslationFunction = headerTranslationFunction =
        helpTranslationFunction =
        infoTranslationFunction = leftfooterTranslationFunction =
        leftheaderTranslationFunction = messageTranslationFunction =
        printTranslationFunction =
        subsubtitleTranslationFunction = subtitleTranslationFunction =
            nullStringFunction[];
        filter = Function[string,
                If[translationEffective === False,
                    "",
                    string
                ]
            ];
        translations =
         Map[{#, Composition[filter,
                ToExpression[StringJoin[#, "TranslationFunction"]]]}&, validTypes]

    ]
```

Clear[chapterTranslations];
chapterTranslations[] :=
Module[(* {validTypes, oneParameterTeXMacro, nullStringFunction,
commentedOneParameterTeXMacro, encapsulateFunction,
titleTranslationFunction, sectionTranslationFunction,
subsectionTranslationFunction, subsubsectionTranslationFunction,
inputTranslationFunction, outputTranslationFunction,
nameTranslationFunction, special1TranslationFunction,
special2TranslationFunction, special3TranslationFunction,

```
        special4TranslationFunction, special5TranslationFunction,
        translationParametersTranslationFunction,
        abstractTranslationFunction, includeFileTranslationFunction,
        textTranslationFunction, smalltextTranslationFunction,
        clipboardTranslationFunction, completionsTranslationFunction,
        footerTranslationFunction, headerTranslationFunction,
        helpTranslationFunction,
        infoTranslationFunction, leftfooterTranslationFunction,
        leftheaderTranslationFunction, messageTranslationFunction,
        postscriptTranslationFunction, printTranslationFunction,
        subsubtitleTranslationFunction,
        subtitleTranslationFunction,
        translations
        }, *)
        {validTypes, oneParameterTeXMacro, nullStringFunction,
        commentedOneParameterTeXMacro, encapsulateFunction,
        translationParametersTranslationFunction,
        abstractTranslationFunction, includeFileTranslationFunction,
        translations
        },
validTypes = {"clipboard", "completions", "footer", "header", "help",
                "info", "input", "leftfooter", "leftheader", "message",
                "name", "output", "postscript", "print", "section",
                "smalltext", "special1", "special2", "special3", "special4",
                "special5", "subsection", "subsubsection", "subsubtitle",
                "subtitle", "text", "title"};
oneParameterTeXMacro[macroName_String] :=
                StringJoin["\\", macroName, "{", #, "}"] &;
nullStringFunction[] := "" &;
commentedOneParameterTeXMacro[macroName_String] :=
        StringJoin["% \\", macroName, "{",
                StringReplace[#, "\n" -> "\n%"],
                "}"
        ] &;
encapsulateFunction[texCommand_String] :=
        If[texCommand == "",
                StringJoin["{", #, "\n}\n"] &,
                StringJoin["{\\", texCommand, " ", #, "\n}\n"] &
        ];
titleTranslationFunction = oneParameterTeXMacro["chapter"];
sectionTranslationFunction = oneParameterTeXMacro["section"];
subsectionTranslationFunction = oneParameterTeXMacro["subsection"];
subsubsectionTranslationFunction = oneParameterTeXMacro["subsubsection"];
(* acknowledgementTranslationFunction = oneParameterTeXMacro["section*"]; *)
inputTranslationFunction = oneParameterTeXMacro["MRun"];
inputTranslationFunction =
        StringJoin["\\begin{verbatim}\n", #, "\n\\end{verbatim}\n"]&;
outputTranslationFunction = nullStringFunction[];
nameTranslationFunction = commentedOneParameterTeXMacro["comment"];
special1TranslationFunction =
        translationParametersTranslationFunction = (ToExpression[#];"") &;
special2TranslationFunction =
        (translationEffective = If[translationEffective === False, True, False])&;
special3TranslationFunction = StringJoin["\\FIGURE{", prefixPath, "}{", #, "}"] &;
abstractTranslationFunction =
 special4TranslationFunction = Function[string,
"\\noindent\n{\\bf Abstract}: \n" <>
        StringReplace[
                FixedPoint[
                        StringReplace[#, "\n\n\n\n" -> "\n\n\n"] &,
                        string
                ],
```

```
                "\n\n\n" -> "\n\n\n\\noindent \n{\bf Keywords}:"
            ]
        ];
        includeFileTranslationFunction =
         special5TranslationFunction = StringJoin["\\input ", prefixPath, #] &;
        textTranslationFunction = Identity;
        smalltextTranslationFunction = encapsulateFunction["small"];
        clipboardTranslationFunction = completionsTranslationFunction =
        footerTranslationFunction = headerTranslationFunction =
        helpTranslationFunction =
        infoTranslationFunction = leftfooterTranslationFunction =
        leftheaderTranslationFunction = messageTranslationFunction =
        postscriptTranslationFunction = printTranslationFunction =
        subsubtitleTranslationFunction = subtitleTranslationFunction =
            nullStringFunction[];
        filter = Function[string,
                    If[translationEffective === False,
                        "",
                        string
                    ]
            ];
        translations =
         Map[{#, Composition[filter, ToExpression[StringJoin[#, "TranslationFunction"]]]}&, validTypes]
]

chapterTranslations[]

(* For a notebook.ma file, returns the notebook.examples.ma file name. *)
Clear[notebookNameToSourceFileName];
notebookNameToSourceFileName[notebookName_String] :=
    StringJoin[StringDrop[notebookName, -3], ".s"]

notebookNameToSourceFileName["string.ma"]
string.s

(*  From a notebook file that has a description of
    an architecture according to chapter template,
    creates a TeXPrep source file. *)
createSourceFile[notebookName_String]:=
Module[{sourceFileName, translations},
    sourceFileName = notebookNameToSourceFileName[notebookName];
    Print["Creating ", sourceFileName, "....."];
    translations = chapterTranslations[];
    translateNotebook[notebookName, sourceFileName,
        translations, Echo -> False,
        PreserveCellHeaders -> False,
        PreserveFontInfo -> False
    ]
]

(*  From a notebook file that has a description of
    an architecture according to chapter template,
    creates a TeX chapter by running it through
    createSourceFile and TeXPrep. *)
Clear[GenerateChapter];
GenerateChapter[notebookName_String]:=
Module[{chapterFileName, sourceFileName},
    createSourceFile[notebookName];
    chapterFileName =
        StringJoin[StringDrop[notebookName, -3], ".tex"];
    sourceFileName = notebookNameToSourceFileName[notebookName];
    Run["echo", sourceFileName, ">", chapterFileName]
]
```

```
GenerateChapter["/tmp/ocg/TripFlop/TripFlop.ma"];

Clear[GenerateDummyFiles];
GenerateDummyFiles[notebookName_String]:=
Module[{files, path},
    files = {"abstract.tex", "introduction.tex", "approach.tex",
             "operation.tex", "conclusion.tex",
             "references.tex"};
    path = StringJoin[dirName[notebookName], "/", #]&;
    Map[If[!(existFileQ[path[#]]),
            Run["cp", "/dev/null", path[#]]
        ]&,
        files
    ]
]

GenerateDummyFiles["/tmp"]

{0, 0, 0, 0, 0, 0}

SetDirectory["~/Thesis/Part2"];
unixLikeFind[Directory[], notebookQ, GenerateChapter];
unixLikeFind[Directory[], notebookQ, GenerateDummyFiles]

/a/nargis/export/home/ocg/Users/Atif/Thesis/Part2

createSourceFile["~/Thesis/Part2/Banyan/Banyan.ma"]

Creating ~/Thesis/Part2/Banyan/Banyan.s.....

SetDirectory["~/Thesis/Part2"];
unixLikeFind[Directory[], notebookQ, createSourceFile]

SetDirectory["~/Thesis/Part2"];
unixLikeFind[Directory[], notebookQ, Print];

createSourceFile[
"~/Thesis/Part2/06.SerialAdder/SerialAdder.ma"]

Creating ~/Thesis/Part2/29.system5Complete/system5Complete\

   .s.....
```

Part IV

Supporting Implementation in Mathematica

Chapter 51

Units used in the system

Chapter Abstract

This notebook contains all global units, constants and values used in the user interface. It should be loaded before loading the front end.

Units used in the system

This notebook contains all global Units, Constants and Values used in the User Interface. It should be loaded before loading the Front End.

■ Units

```
meter = 1;
watt = 1;
radian = 1;
second = 1;
degree = Degree;
percent = 0.01;
inch = 2.54 10^-2 meter;
cm = 0.01 meter;
mm = 0.001 meter;
mw = 0.001 watt;
mrad = 0.001 radian;
ps = 10^-12 second;
SpeedOfLight = 3 * 10^8;
```

Chapter 52

Dependency Generation

Chapter Abstract

This notebook contains routines to extract dependencies from functions. These dependencies are later used (among other things) to find context dependencies in the generated packages.

Dependency Generation

This notebook contains routines to extract dependencies from functions.
These dependencies are later used (among other things) to find context dependencies in the generated Packages.

The code is generic and may be used for languages other than *Mathematica*.

```
(* Reads a specified .functions file and returns a structure of the form
    {contextName, listOfFunctionsInFile} *)
Clear[functionPairs];
functionPairs[functionsFileName_String]:=
Module[{contextName, functions, repeatsRemoved},
    contextName = functionsFileNameToContextName[functionsFileName];
    functions = ReadList[functionsFileName, String];
    repeatsRemoved = Union[Map[frontToken[#, {"["}]&, functions]];
    {contextName, repeatsRemoved}
]
```

```
functionPairs["OptiCAD/Packages/Databases/Dependencies.functions"]
```

```
{Databases`Dependencies`, {functionPairs, getAllFunctionNames, inputCellToPairs,
   obtainDependencies, obtainDependenciesForFile, selfDependencyQ, uses}}
```

```
functionPairs["OptiCAD/Packages/Simulator/Simulator.functions"]
```

```
{Simulator`Simulator`, {AddObject, AddOrientationConstraint,
   AddOrientationConstraints, AddPlacementConstraint, AddPlacementConstraints,
   boundingBoxEquations, ComputeOrientations, ComputeOutputPosition,
   ComputePositions, coordinateRules, coordinateRulesToQuadruples, drawDriver,
   GenerateArchitecture, GetObject, GetObjects, handleDefaults, initialize,
   orient, orientationOf, orientBoundingBox, positionOf,
   quadruplesToCoordinateRules, ShowArchitecture, showState, simulate,
   ViewArchitecture}}
```

```
(* Starting from the given path, it returns the list of structures returned by
    functionPairs[]. All subdirectories are also explored.
    Options[getAllFunctionNames] = {RegenerateAllFunctions -> True/False}.
        True (default) forces regeneration of the resulting file.
        False returns a previously generated file.*)
Clear[getAllFunctionNames];
getAllFunctionNames[startDirectory_String, options___]:=
Module[{allFunctions, f, temp, regenerateAllFunctions, file},
    regenerateAllFunctions =
        RegenerateAllFunctions /. {options, RegenerateAllFunctions -> True};
    file = StringJoin[OHDL$PathPrefix,
                    "/Packages/Databases/AllFunctions.database"];
    If[regenerateAllFunctions,
        allFunctions = {};
        f = Function[fn,
                If[(temp=functionPairs[fn]) =!= {},
                    allFunctions = Append[allFunctions, temp]
                ]
            ];      (* Change to operateByIndex some day *)
        unixLikeFind[startDirectory, functionsFileNameQ, f];
        Put[allFunctions, file];
        allFunctions,
        allFunctions = Get[file]
    ]
];
```

```
getAllFunctionNames["OptiCAD/Packages/Graphics"]

{{Graphics`Shapes`, {AffineShape, BeginPackage, cone, Cone, cuboid,
    cuboidtolines, cylinder, Cylinder, disk, DoubleHelix, ellipsoid, Helix,
    hologram, interferenceFilter, lensTwo, mirror, RotateShape,
    seedgraphicsobject, slicedCylinder, slicedTruncatedCone, Sphere, splitter,
    Torus, TranslateShape, truncatedCone, wavePlate, WireFrame}},
  {Graphics`Misc`, {displayObject}},
  {Graphics`Graphics3D`, {operationToMatrix, transform3D}}}

getAllFunctionNames["OptiCAD/Packages/Geometry"]

{{Geometry`Predicates`, {isTheLineAnAxis, isTheLineXAxis, isTheLineYAxis,
    isTheLineZAxis, validLine}},
  {Geometry`Geometry`, {colinearQ, equationOfPlane, normalize, pointInPlane,
    pointOnThePlaneQ, rayAtBoundary,  vectorAtAnAngle}}}

(* Takes a pair of the form
    {{contextOfFunctionName, functionName}, pairsOfAllContextFunctions}
        and returns True if
    functionName \in pairsOfAllContextFunctions *)
Clear[selfDependencyQ];
selfDependencyQ[dependency__] :=
Module[{},
    StringMatchQ[frontToken[dependency[[1, 2]], {"["}], dependency[[2,2]] ]
]

dependency = {{"Context", "ComputeOrientations[]"} ,
            {"OHDLSimulator`Simulator`", "ComputeOrientations"}};
selfDependencyQ[dependency]

True

dependency = {{"Context", "ComputeOrientations[]"} ,
            {"OHDLSimulator`Simulator`", "selfDependencyQ"}};
selfDependencyQ[dependency]

False

(* returns from functionLists all elements used in functionName of contextName.
    Options[uses] = {RecursionInfo -> True/False}.
    True (default) forces check for recursion dependency;
    False otherwise *)
Clear[uses];
uses[contextName_String, {functionName_String, functionBody_String},
                functionLists__, options___]:=
Module[{dependencies, f, recursionInfo, f1},
    recursionInfo = RecursionInfo /. {options, RecursionInfo -> True};
    f = Function[pair, If[hasTokenQ[functionBody, pair[[2]] ],
                        {{quote[contextName], quote[functionName]}, quote[pair]},
                        {{}, {}}
                       ]
        ];
    f1 = Function[fp,
            Select[ Map[ f[{fp[[1]], #}]&, fp[[2]] ], (# =!= {{}, {}})&]
         ];
    dependencies = If[recursionInfo,
        Select[ Join[ Apply[Sequence, Map[f1, functionLists] ] ], (# != {})&],
        Select[ Join[ Apply[Sequence, Map[f1, functionLists] ] ],
                ( (# != {}) && !selfDependencyQ[#])&]
    ];
    If[dependencies == {}, "", ToString[dependencies]]
]
```

```
uses["myContext", {"f[]", "a = p[q]; b = m[1]; c = s[p]; d = f[b]"},
    {{"cp", {"p"}}, {"cms", {"m", "s"}}, {"cxy", {"x", "y"}},
        {"myContext", {"f"}}}]
{{{"myContext", "f[]"}, {"cp", "p"}}, {{"myContext", "f[]"}, {"cms", "m"}},\
  {{"myContext", "f[]"}, {"cms", "s"}}, {{"myContext", "f[]"}, {"myContext",\
  "f"}}}
uses["myContext", {"f[]", "a = p[q]; b = m[1]; c = s[p]; d = f[b]"},
    {{"cp", {"p"}}, {"cms", {"m", "s"}}, {"cxy", {"x", "y"}}},
        RecursionInfo -> False]
{{{"myContext", "f[]"}, {"cp", "p"}}, {{"myContext", "f[]"}, {"cms", "m"}},\
  {{"myContext", "f[]"}, {"cms", "s"}}}
cell = "simulate[object_, message_]:= \n
Module[ {},\n
    Switch[ Action /. message,\n
        PassItThrough, \n
            mn = (CurrentMessageNumber = CurrentMessageNumber + 1);\n
            poi = PointOfImpact /. message;\n
            pos = ComputeOutputPosition[object,message];\n
            index = RefractiveIndex /. Join[object, {RefractiveIndex -> 1}];\n
            distance = Abs [ Sqrt[ Apply[Plus, MapThread[(#1 - #2)^2 &, {poi, pos}
            exitTime = (TimeOfImpact /. message) + distance/(SpeedOfLight / index)
            PickFirst[{\n
                        {   Sender -> ObjectNumber /. object, \n
                            MessageNumber -> mn,\n
                            TimeOfExit -> exitTime,\n
                            PlaceOfExit -> pos\n
                        }, \n
                        message\n
                      }\n\n

            ],\n
        _,\n
            Print[\"Warning :: simulate called with an unknown type of message\"]
    ]\n
]\n\nsimulateobject[message_]:= \n
Module[ {},\n
    Switch[ Action /. message,\n
        PassItThrough, \n
            mn = (CurrentMessageNumber = CurrentMessageNumber + 1);\n
            poi = PointOfImpact /. message;\n
            pos = ComputeOutputPosition1[object,message];\n
            pos = simulate[object,message];\n
            index = RefractiveIndex /. Join[object, {RefractiveIndex -> 1}];\n
            distance = Abs [ Sqrt[ Apply[Plus, MapThread[(#1 - #2)^2 &, {poi, pos}
            exitTime = (TimeOfImpact /. message) + distance/(SpeedOfLight / index)
            PickFirst[{\n
                        {   Sender -> ObjectNumber /. object, \n
                            MessageNumber -> mn,\n
                            TimeOfExit -> exitTime,\n
                            PlaceOfExit -> pos\n
                        }, \n
                        message\n
                      }\n
            ],\n
        _,\n
            Print[\"Warning :: simulate called with an unknown type of message\"]
    ]\n
]";
(* Breaks an input definition cell into {functionName, Body} Pairs.
    All comments and quoted strings are removed first.
```

```
    Options[inputCellToPairs] = {HeadBodySeparator, FunctionSeparator,
                                                 FunctionsInFullForm}.

    HeadBodySeparator (default = ":="): string between the head and body of the
                                   definition.
    FunctionSeparator (default = "\n\n"): separator between functions.
    FunctionsInFullForm (default = False): not supported yet. If True return on
                    symbol of the function header (no args, conditions).
*)
Clear[inputCellToPairs];
inputCellToPairs[inputCell_String, options___]:=
Module[{headBodySeparator, functionSeparator,
        quotedStringsRemoved, commentsRemoved,
         functions, pairs, functionsInFullForm, clearRemoved, tabsReplaced,
         whiteSpaceSquashed},
    headBodySeparator =
        HeadBodySeparator /. {options, HeadBodySeparator -> ":="};
    functionSeparator =
        FunctionSeparator /. {options, FunctionSeparator -> "\n\n"};
    functionsInFullForm =
        FunctionsInFullForm /. {options, FunctionsInFullForm -> False};

    quotedStringsRemoved = removeQuotedStrings[inputCell];
    commentsRemoved = removePattern[quotedStringsRemoved, "(*", "*)"];

    clearRemoved = removePattern[commentsRemoved, "Clear[", "];"];
    tabsReplaced = StringReplace[ clearRemoved, "\t" -> " "];

    whiteSpaceSquashed =
        StringReplace[ tabsReplaced,
            Join[Table[StringJoin[nChars[i, "\n"], " "] -> " ", {i, 8, 2, -1}],
                Table[nBlanks[i] -> " ", {i, 8, 2, -1}]
            ]
        ];

    If[definitionCell[whiteSpaceSquashed] == "", {},
        functions =
            splitRepeated[whiteSpaceSquashed,
                {{headBodySeparator}, {functionSeparator}}];
        pairs = Map[{frontToken[StringReplace[#[[1]],
                                {"\n" -> " "}], {"["}], #[[2]]} &,
                        Partition[functions, 2]
                ];
        Map[{   StringReplace[#[[1]], {" " -> ""}],
                StringDrop[#[[2]], StringLength[headBodySeparator]]}&,
            pairs
        ]
    ]
]
inputCellToPairs["a := \n\n b\n\nc:=d"] //InputForm
{{"a", "  b"}, {"c", "d"}}
```

```
cell2 = "draw[InterferenceFilter[object_]] := \n
    Module[{size, position, orientedObject},\n
        size = Size /. object;\n
\n
\t\t(* lower left position *)\n
\t\tposition = (Variables /. object)[[Range[Length[Origin]]]] /. Coordinates;\n
        (* extract the orientation angles *)\n
        {thetaX, thetaXY, thetaP} = \n
                (Variables /. object)[[3+Range[Length[Origin]]]] /. Orientations;\
            (* Print[{thetaX, thetaXY, thetaP}];   *)          \n
        {thetaX, thetaXY, thetaP} = (* to take care of non-existant\n
                        rule for some theta *)\n
            Map[ If[Head[#] === Symbol, 0, #]&, {thetaX, thetaXY, thetaP}];\n
            (* Print[{thetaX, thetaXY, thetaP}]; *)                \n
\n
        orientedObject = orient[interferenceFilter[size], \n
            N[{thetaX, thetaXY, thetaP}]];\n
    \n
        transformMma3D[orientedObject, {translateBy[position]}]\n
    ]\n";

inputCellToPairs[cell];
```

```
(* Obtains all the function dependencies in a specific file. *)
Clear[obtainDependenciesForFile];
obtainDependenciesForFile[ notebookName_String,
    outputFile_String, allFunctionLists__]:=
Module[{f, a, contextName},
    contextName = notebookNameToContextName[notebookName];
    f = Function[cell, a = Map[uses[contextName, #, allFunctionLists]&,
                                        inputCellToPairs[cell] ];
            listToString[a, "\n"]
        ];
    translateNotebook[notebookName, outputFile,
            "input", f, CellSeparator -> "",
            Echo -> False, PreserveFontInfo -> False, AppendToOutputFile -> True]
]

CloseAllStreams[];
obtainDependenciesForFile["/tmp/ocg/Packages/Simulator/Simulator.ma",
    "/tmp/output", allFunctionPairs]
```

```
(* For a specified path, obtains all function dependencies withen and across file.
    Stores the dependency information  in a specified file *)
Clear[obtainDependencies];
obtainDependencies[ startDirectory_String, outputFile_String]:=
Module[{allFunctions},
    allFunctions =
        getAllFunctionNames[startDirectory, RegenerateAllFunctions -> False];
    operateByIndex[startDirectory, notebookQ,
        (Print["Processing dependencies for ", #, " ", MemoryInUse[]];
         obtainDependenciesForFile[StringJoin[Directory[], "/", #], outputFile,
               allFunctions] )&
    ]
]

CloseAllStreams[]; backupAndDelete["OptiCAD/output"];
obtainDependencies["OptiCAD/Packages/Utilities", "OptiCAD/output"];
Print["Done ....", MemoryInUse[]]

Processing dependencies for ListManipulation.ma 910240
Processing dependencies for OptionProcessing.ma 916124
Processing dependencies for RealArithmetic.ma 957252
Processing dependencies for ReWriteRules.ma 995376
Processing dependencies for StringOperations.ma 1022944
Processing dependencies for SystemsProgramming.ma 1406848
Done ....
```

Chapter 53

Dependency Information Processing

Chapter Abstract

This Notebook assumes the availability of a raw functional dependency file. It processes this information to produce a Context Dependency file. This context dependency file is later used to generate Needs[] information in the generated Packages.

Dependency Information Processing

This Notebook assumes the availability of a raw functional dependency file.
It processes this information to produce a Context Dependency file.
This context dependency file is later used to generate Needs[] information in the generated Packages.

```
(* Reads the already generated
    OHDL$PathPrefix <> "/Packages/Databases/FunctionDependencies.database" file
    and produces a flat list of {currentContexts, allUsedContexts}. It saves this
    informtion in a global variable OHDL$ContextDependencies
    Options[] = {RegenerateContextDependencyDatabase -> True/False}
    True (default): forces generation of new file.
    False : returns the global variable OHDL$ContextDependencies *)
Clear[generateContextDependencyDatabase];
generateContextDependencyDatabase[options___]:=
Module[{fullList, functionDependenciesFileName, f, contextDependencyFile,
        regenerateContextDependencyDatabase},
    contextDependencyFile =
        StringJoin[OHDL$PathPrefix,
            "/Packages/Databases/ContextDependencies.database"];
    functionDependenciesFileName =
        StringJoin[OHDL$PathPrefix,
            "/Packages/Databases/FunctionDependencies.database"];
    regenerateContextDependencyDatabase =
        RegenerateContextDependencyDatabase /.
                    {options, RegenerateContextDependencyDatabase -> True};
    If[regenerateContextDependencyDatabase,
        fullList = ReadList[functionDependenciesFileName,
                        String, RecordLists -> True];
        fullList = Map[StringReplace[#, {"\n" -> ""}]&, Select[Flatten[fullList],

                (# != "")&] ];
        OHDL$FullList = Flatten[Map[ToExpression, fullList], 1];
        OHDL$ContextNames = Union[Map[First, OHDL$FullList, 2]];
        f = Function[contextName,
            Union[
                Prepend[
                        Map[
                            #[[2,1]]&,
                            Select[OHDL$FullList, (#[[1,1]] == contextName) &]
                        ],
                        contextName
                ]
            ]
        ];
        OHDL$ContextDependencies = Map[Flatten[{#, Complement[f[#], {#}]}]&,
                OHDL$ContextNames];

        Put[OHDL$ContextDependencies, contextDependencyFile];
        Clear[OHDL$FullList, OHDL$ContextNames],
        (* else load the database *)
        OHDL$ContextDependencies = Get[contextDependencyFile]
    ]
]
generateContextDependencyDatabase[];
OHDL$ContextDependencies
```

```
{{OHDLDatabases'Dependencies', OHDLTranslators'NotebookToPackage',
   OHDLUtilities'StringOperations', OHDLUtilities'SystemsProgramming'},
  {OHDLDatabases'FunctionsToContexts'},
  {OHDLGeometry'Geometry', OHDLUtilities'RealArithmetic'},
  {OHDLGraphics'Graphics3D', OHDLGeometry'Geometry', OHDLGeometry'Predicates'},
  {OHDLGraphics'Misc', OHDLUtilities'OptionProcessing'},
  {OHDLGraphics'Shapes', OHDLTranslators'TransformMma3D'},
  {OHDLOpticalComponents'draw', OHDLGraphics'Shapes',
   OHDLOpticalComponents'Icons', OHDLSimulator'Simulator',
   OHDLTranslators'TransformMma3D', OHDLUtilities'ReWriteRules'},
  {OHDLOpticalComponents'Icons', OHDLGraphics'Shapes',
   OHDLTranslators'TransformMma3D'},
  {OHDLSimulator'Simulator', OHDLGeometry'Geometry', OHDLGraphics'Graphics3D',
   OHDLGraphics'Shapes', OHDLOpticalComponents'draw',
   OHDLTranslators'TransformMma3D', OHDLUtilities'OptionProcessing',
   OHDLUtilities'ReWriteRules'},
  {OHDLTranslators'NotebookToPackage', OHDLUtilities'StringOperations'},
  {OHDLTranslators'TransformMma3D', OHDLGraphics'Graphics3D'},
  {OHDLUtilities'OptionProcessing'}, {OHDLUtilities'RealArithmetic'},
  {OHDLUtilities'StringOperations', OHDLUtilities'OptionProcessing'},
  {OHDLUtilities'SystemsProgramming', OHDLDatabases'Dependencies',
   OHDLTranslators'NotebookToPackage', OHDLUtilities'RealArithmetic',
   OHDLUtilities'StringOperations'}}
```

```
(* Returns a list of all uses contexts for a specifed context. If the global
    variable OHDL$ContextDependencies is empty, will regenerate it. *)
Clear[dependencyContexts];
dependencyContexts[contextName_String]:=
Module[{list},
    If[Head[OHDL$ContextDependencies] === Symbol,
        generateContextDependencyDatabase[]
    ];
    list = Select[OHDL$ContextDependencies, (#[[1]] == contextName)&, 1];
    If[list === {},
        Print["Warning::dependencyContexts[]: ", contextName,
                " is Null. Returning Identity "];
        {contextName},
        (* else *)
        First[list]
    ]
]
```

```
dependencyContexts["OHDLDatabases'Units'"]
```

```
Warning::dependencyContexts[]: OHDLDatabases'Units' is Null. Returning Identity
{OHDLDatabases'Units'}
```

```
dependencyContexts["OHDLUtilities'SystemsProgramming'"]
```

```
{OHDLUtilities'SystemsProgramming', OHDLDatabases'Dependencies',
  OHDLTranslators'NotebookToPackage', OHDLUtilities'RealArithmetic',
  OHDLUtilities'StringOperations'}
```

```
(* Reads the file OHDL$PathPrefix <>"/Packages/Databases/FunctionUsage.database
    and returns in a global variable OHDL$FunctionUsageDatabase a database
    containing records of the form {{contextName, functionName}, "Usage Message
    Options[generateUsageDatabase] =
        {RegenerateFunctionUsageDatabase -> True/False}
    True (default): Forces reading of the file.
    False: Returns the veriable OHDL$FunctionUsageDatabase.
*)
Clear[generateUsageDatabase];
generateUsageDatabase[options___]:=
Module[{usageDatabaseFile, regenerateFunctionUsageDatabase, rawInput},
    usageDatabaseFile =
        StringJoin[OHDL$PathPrefix, "/Packages/Databases/FunctionUsage.db"];
    regenerateFunctionUsageDatabase =
        RegenerateFunctionUsageDatabase /.
                        {options, RegenerateFunctionUsageDatabase -> True};
    If[regenerateFunctionUsageDatabase,
        rawInput = ReadList[usageDatabaseFile, Word, RecordLists -> True,
            RecordSeparators -> {"\n\n"}, WordSeparators -> {" ", "\t", "\n"}];
        OHDL$FunctionUsageDatabase =
            Map[{{#[[1]], #[[2]]}, listToString[Drop[#, 2], " "]}&, rawInput],
        (* else load the database *)
        OHDL$FunctionUsageDatabase
    ]
]
```

```
(* For a given contextName and functionName, returns a Usage message from the
    database *)
Clear[functionNameToUsage];
functionNameToUsage[longFunctionName_String, contextName_String]:=
Module[{list, message, functionName},
    functionName = frontToken[longFunctionName, {"["}];
    If[Head[OHDL$FunctionUsageDatabase] === Symbol,
        generateUsageDatabase[]
    ];
    list = Select[OHDL$FunctionUsageDatabase,
            (#[[1]] === {contextName, functionName})&, 1];
    If[list === {},
        Print["Warning::functionNameToUsage[]: ", {contextName, functionName},
                " is Null. Returning Reverse "];
        StringReverse[functionName],
        (* else *)
        {message} = Rest[First[list]];
        message
    ]
]
```

```
functionNameToUsage["unixLikeFind[]", "Utilities`SystemsProgramming`"]
```

```
Usage Message for unixLikeFind
```

```
functionNameToUsage["find[args]", "Utilities`SystemsProgramming`"]
```

```
Warning::functionNameToUsage[]: {Utilities`SystemsProgramming`, find}

   is Null. Returning Reverse
dnif
```

Chapter 54

Predicates for Geometry

Chapter Abstract

This notebook contains the necessary predicates for geometry. These are not only used by Geometry.ma but by several other notebooks.

Predicates for Geometry

This Notebook contains the necessary predicates for geometry. These are not only used only by Geometry.ma but by sevaral other Notebooks.

```
(* Checks whether the given line is the same as one of the principal axes *)
Clear[axisQ];
axisQ[line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ]] :=
    (   ((x1 =!= x2) && (y1 == y2 == 0.0) && (z1 == z2 == 0.0)) ||
        ((y1 =!= y2) && (x1 == x2 == 0.0) && (z1 == z2 == 0.0)) ||
        ((z1 =!= z2) && (y1 == y2 == 0.0) && (x1 == x2 == 0.0)) ) &&
    ! ((x1 === x2) && (y1 === y2) && (z1 === z2))

axisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
False

axisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]
True

(* Checks whether the given line is the same as the x-axis *)
Clear[xAxisQ];
xAxisQ[line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ]] :=
    ((x1 =!= x2) && (y1 == y2 == 0.0) && (z1 == z2 == 0.0)) &&
    ! ((x1 === x2) && (y1 === y2) && (z1 === z2))

xAxisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
False

xAxisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]
True

(* Checks whether the given line is the same as the y-axis *)
Clear[yAxisQ];
yAxisQ[line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ]] :=
    ((y1 =!= y2) && (x1 == x2 == 0.0) && (z1 == z2 == 0.0)) &&
    ! ((x1 === x2) && (y1 === y2) && (z1 === z2))

yAxisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
False

yAxisQ[line[point[{0, 0, 0}], point[{0, 1, 0}]]]
True

(* Checks whether the given line is the same as the z-axis *)
Clear[zAxisQ];
zAxisQ[line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ]] :=
    ((z1 =!= z2) && (y1 == y2 == 0.0) && (x1 == x2 == 0.0)) &&
    ! ((x1 === x2) && (y1 === y2) && (z1 === z2))

zAxisQ[line[point[{0, 0, 0}], point[{0, 0, -1}]]]
True

zAxisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]
False

(* checks whether a specific line is valid i.e. not a point *)
Clear[lineQ];
lineQ[line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ] ] :=
    ! ((x1 === x2) && (y1 === y2) && (z1 === z2))

lineQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
True
```

```
lineQ[line[point[{0, 0, 0}], point[{0, 0, 0}]]]
False

(* Checks whether the given point lies on the given plane *)
Clear[pointOnThePlaneQ];
pointOnThePlaneQ[{x_, y_, z_}(*point__*), {a_, b_, c_, d_} (*plane__*)]:=
    eqQ[Dot[ N[{x, y, z, 1}], N[{a, b, c, d}]], 0.0];

pointOnThePlaneQ[{x0_, y0_, z0_}(*point__*), Equal[lhs_, rhs_] (*plane__*)]:=
    eqQ[(lhs - rhs) /. {x -> x0, y -> y0, z -> z0}, 0.0]

pointOnThePlaneQ[{0, 0, 0}, {1, 0, 0, 0}]
True

pointOnThePlaneQ[{0, 0, 0}, {1, 1, 1, 1}]
False

(* Checks whether a set of points is colinear.
    NOTE: Not yet implemented. Current status = (Always returns False) *)
Clear[colinearQ];
colinearQ[points_] := False
```

Chapter 55

Commonly used Geometry Functions

Chapter Abstract

This notebook contains the routines necessary for Geometrical computations. They are extensively used by Graphics3D, Simulation and several other Notebooks.

Commonly used Geometry Functions

This Notebook contains the routines necessary for Geometrical computations. They are extensively used by Graphics3D, Simulation and several other Notebooks.

```
(* returns a point on the given plane *)
Clear[pointInPlane];
pointInPlane[ plane[a_, b_, c_, d_] ]:=
        Which[  a =!= 0, {-d/a, 0, 0},
                b =!= 0, {0, -d/b, 0},
                c =!= 0, {0, 0, -d/c}
        ] /;  ((a =!= 0) || (b =!= 0) || (c =!= 0))
```

```
N[pointInPlane[plane[2, 4, 5, 7]]]
{-3.5, 0, 0}
```

```
(* Given a vector, it returns its normalized vactor *)
Clear[normalize];
normalize[vector__] /; (1 < Length[vector] <=3) :=
Module[{squareOfMagnitude},
    squareOfMagnitude = N[Apply[Plus, Map[Times[#, #]&, vector]]];
    If[neQ[squareOfMagnitude, 0],
        N[vector / Sqrt[squareOfMagnitude]],
        Print["Warning:: normalize[]:Normalize ",
              "called with a zero length vector. Returning Unit vector "];
        Prepend[Table[0, {Length[vector] - 1}], 1]
    ]
]
```

```
normalize[{2, 3, 4}]
{0.371391, 0.557086, 0.742781}
```

```
normalize[{0, 0, 0}]
Warning:: normalize[]:Normalize called with a zero length vector. Returning\

  Unit vector
{1, 0, 0}
```

```
(* function that takes a ray, and an angle,
    computes and returns unit rays that make an angle given
    with the given vector.
    {{x0,y0}, {a,b}}    :  input vector starting from {x0,y0}
               with direction of the unit vector {a,b}.
 *)
Clear[vectorAtAnAngle];
vectorAtAnAngle[v_, angle_] := Module[ {theta},
    theta = ArcTan[ v[[2]][[2]] / v[[2]][[1]] ];
    {
        {v[[1]], {Cos[theta+angle], Sin[theta+angle]}},
        {v[[1]], {Cos[theta-angle], Sin[theta-angle]}}
    }
]
```

```
N[ vectorAtAnAngle[{{1, 1}, {1, 1}}, 90 Degree] ]
{{{1., 1.}, {-0.707107, 0.707107}}, {{1., 1.}, {0.707107, -0.707107}}}
```

NOTE: Let the equation of a plane be $ax + by + cz + d = 0$. This is represented by the coefficient list $\{a, b, c, d\}$.

TODO: Check for colinearity of the given points. Define a predicate colinearQ[] in predicates that
checks for the colinearity. The definition of the equation of plane is valid only when colinearQ[points] is False.

```
(* Given a set of points (at least three), it returns an equation of the plane
    on which those points lie. *)
Clear[equationOfPlane];
equationOfPlane[points__] /; (Length[points] >= 3) && !colinearQ[points] :=
Module[{p1, p2, p3, x1, y1, z1, x2, y2, z2, c1, c2, c3, c4, unknowns,
        augmentedPoints, equations, results, coefficients, choppedPoints},
    choppedPoints = points;
    {p1, p2} = Select[ choppedPoints, # =!= {0.0, 0.0, 0.0} &, 2];
    p3 = First[ Complement[choppedPoints, {p1, p2}] ];
    augmentedPoints = Map[Append[#, 1.0]&, {p1, p2, p3}];
    {x1, y1, z1} = p1;
    {x2, y2, z2} = p2;
    Which[
        neQ[Det[{p1, p2, p3}], 0.0],
            c4 = 1.0;
            unknowns = {c1, c2, c3};
            equations = Map[Equal[Chop[ N[ Dot[{c1, c2, c3, 1}, #] ] ], 0.0]&,
                                          augmentedPoints],

        neQ[x1 y2, x2 y1],
            c3 = 1.0; c4 = 0.0;
            unknowns = {c1, c2};
            equations = Map[Equal[Chop[ N[ Dot[{c1, c2, 1}, #] ] ], 0.0]&,
                                          {p1, p2}],

        neQ[x1 z2, x2 z1],

            c2 = 1.0; c4 = 0.0;
            unknowns = {c1, c3};
            equations = Map[Equal[Chop[ N[ Dot[{c1, 1, c3}, #] ] ], 0.0]&,
                                          {p1, p2}],

        neQ[z1 y2, z2 y1],
            c1 = 1.0; c4 = 0.0;
            unknowns = {c2, c3};
            equations = Map[Equal[Chop[ N[ Dot[{1, c2, c3}, #] ] ], 0.0]&,
                                          {p1, p2}],

        True,
            Print["Error :: equationOfPlane[]: None of the cases ... "];
            Print["\t", {p1, p2, p3}]
    ];
    results = Chop[ Solve[ equations ] ];
    {coefficients} = {c1, c2, c3, c4} /. results;
    If[ (* all the points are on the computed plane *)
            Apply[And, Map[ pointOnThePlaneQ[#, coefficients ]&,
                        choppedPoints ] ],
            Simplify[ Equal[ Dot[{x, y, z, 1.0}, coefficients], 0.0] ],
            Print["Error :: equationOfPlane[] : Some points are not on the plane"]
    ]
]

equationOfPlane[{{0, 1, 0}, {1, 0, 0}, {9, 9, 9}}]
1. - 1. x - 1. y + 1.88889 z == 0.
```

Ray Intersecting with a Boundary between Two Media

```
(* Input:
    {{x0,y0}, {a,b}, n} :  input vector starting from {x0,y0}
                  with direction of the unit vector {a,b}.
    surface      : a list of constraints & refractive indices.
                  {f(x,y) == 0, xl <= x <= xh &&, yl <= y <= yh, ni, no}
                        ni  :  refractive index of the first medium
                        no  :  refractive index of the second medium

Output:
    {{xi,yi}, {c,d}, no}    : a ray going through {xi, yi}, i.e.,
    the point of intersection of input ray and surface, parallel to
    the unit vector {c,d}.
    reflection/refraction   : a boolean quantity indicating whether
    the outgoing ray is a reflected one or refracted one.
    *)
rayAtBoundary[incomingRay_, surface_] :=
(* incomingRay == {{xstart, ystart}, {xdir, ydir}, refractiveIndex}
   surface == { surface_expression, boundary_predicate,
            refractive_index_1, refractive_index_2 } *)
Module[
(* local vars... too many. *)
{a, b, bc, critical, dotprod, dotprods, fx, fy,
    max, n, ni, no, normalVec,
    rayVecList, reflection,
    t, thetai, thetar, tMin, tList, where, x, y, x0, y0, xi, yi},

(* initialize temporary local vars *)

critical = dotprod = fx = fy = thetai = thetar = Infinity;
reflection = True;
dotprods = rayVecList = tList = {};
where = 0;
max = -Infinity;
{{x0,y0}, {a, b}, n} = incomingRay;
{f, bc, ni, no} = surface;
{xi, yi} = {x0, y0};
normalVec = {{x0,y0},{a,b}};

(* check for consistency between the refractive indices mentioned in
    the incomingRay and the surface. *)
If[eqQ[n,no], (* coming from the second side, so swap the ni and no *)
    {ni, no} = {no, ni},
    If[neQ[n,ni],
        Print["Check parameter consistency to RayAtBoundary"]
    ]
];

(* compute the list of values for the parameter t, corresponding to
    the points of intersection of incomingRay with the surface *)
tList = t /.
    NSolve[{ Evaluate[f[x,y] /.
            {x -> x0 + a * t,
             y -> y0 + b * t}
          ] == 0},
        {t}];
(* filter out those t values that are neither real,
     nor are in the forward direction *)
tList = Sort[Flatten[
        Map[ If[ltQ[#,0] ||
```

```
                        neQ[Im[#],0], {}, #]&, tList ]
    ]];
(* further filter the ones that do not satisfy the boundary conditions,
    and then find the minimum positive t
    that satisfies boundary conditions. *)
tMin = Min[Map[
    If[bc[x0 + a * #, y0 + b * #],#,Infinity]&, tList]
];
(* at last, compute the point of intersection of the surface,
    by the incoming ray *)
{xi, yi} = With[{t = tMin}, {x0 + a * t, y0 + b * t}];

(* compute the unit normal to the surface at the point of contact *)
{fx, fy} = normalize[
        {D[f[x,y], x], D[f[x,y], y]} /.
            {x -> xi, y -> yi}
    ];
(* make sure that the incoming ray makes an acute angle with the normal *)
If[ gtQ[(dotprod = {a,b}.{fx,fy}), 0],
    thetai = N[ArcCos[dotprod]] ,
    (* else *)
    thetai = N[Pi - ArcCos[dotprod]];
    {fx, fy} = {-fx, -fy}
];
(* compute the normal vector at the point of intersection *)
normalVec = {{xi, yi}, With[{x = xi, y = yi}, Evaluate[{fx,fy}] ] };

(* determine if the ray is going to undergo total internal reflection,
    or not (refraction).
    In either case,
        obtain a list of candidate outgoing rays in rVecList *)
critical = If[gtQ[ni,no], N[ArcSin[no / ni]], Infinity ];
If[ gtQ[ni,no] &&
    gtQ[thetai,critical],
    (* total internal reflection *)
    reflection = True;
    rVecList = vectorAtAnAngle[normalVec, N[Pi - thetai]],
    (* refraction *)
    reflection = False; (* compute refraction angle by Snell's law *)
    thetar = N[ArcSin[ni Sin[thetai] / no]];
    rVecList = vectorAtAnAngle[normalVec, thetar]
];
(* from the candidate rays, determine the correct outgoing ray,
    by picking the one that has a maximum dot product value with
    the incoming ray. *)
dotprods = Map[(#[[2]].{a,b})&, rVecList];
max = Max[dotprods];
{where} = First[Position[dotprods, max]];

(* return the winning candidate ray as the output ray,
    together with an indicator that tells if it is a refracted ray,
    or a totally internally reflected one. *)
{ {{xi,yi}, rVecList[[where]][[2]], If[reflection,ni,no]},
    If[reflection,"Reflection","Refraction"] }
]
```

```
ni = 1;
no = 3;
x0 = -1; y0 = -3;
{a, b} = normalize[{1,3}];

f = ((#1-3)^2 + #2^2 - 9) &;

bc = (  leQ[0,#1] &&
    leQ[#1,3] &&
    leQ[-3,#2] &&
    leQ[#2,3]    ) &;
icr = {{-1,-3},normalize[{1,3}],ni};
s = {f, bc, ni, no};

rayAtBoundary[icr,s]
```

```
{{{0., -4.44089 10^-16}, {0.948683, 0.316228}, 3}, Refraction}
```

Trace of the Function Using an Example

```
critical = dotprod = fx = fy = thetai = thetar = Infinity;
reflection = True;
dotprods = rayVecList = tList = {};
where = 0;
max = -Infinity;
{{x0,y0}, {a, b}, n} = icr;
{f, bc, ni, no} = s;
{xi, yi} = {x0, y0};
normalVec = {{x0,y0},{a,b}};
tList = t /.
    NSolve[{ Evaluate[f[x,y] /.
            {x -> x0 + a * t,
             y -> y0 + b * t}
          ] == 0},
        {t}];
tList = Sort[Flatten[
        Map[ If[ltQ[#,0] ||
                        neQ[Im[#],0], {}, #]&, tList ]
    ]];
tMin = Min[Map[
    If[bc[x0 + a * #, y0 + b * #],#,Infinity]&, tList]
];
(* at last, compute the point of intersection of the surface,
    by the incoming ray *)
{xi, yi} = With[{t = tMin}, {x0 + a * t, y0 + b * t}];
```

```
(* compute the unit normal to the surface at the point of contact *)
{fx, fy} = normalize[
        {D[f[x,y], x], D[f[x,y], y]} /.
            {x -> xi, y -> yi}
    ];
(* make sure that the incoming ray makes an acute angle with the normal *)
If[ gtQ[(dotprod = {a,b}.{fx,fy}), 0],
    thetai = N[ArcCos[dotprod]] ,
    (* else *)
    thetai = N[Pi - ArcCos[dotprod]];
    {fx, fy} = {-fx, -fy}
];

(* compute the normal vector at the point of intersection *)
normalVec = {{xi, yi}, With[{x = xi, y = yi}, Evaluate[{fx,fy}] ] };


(* determine if the ray is going to undergo total internal reflection,
    or not (refraction).
    In either case,
        obtain a list of candidate outgoing rays in rVecList *)
critical = If[gtQ[ni,no], N[ArcSin[no / ni]], Infinity ];
If[ gtQ[ni,no] &&
    gtQ[thetai,critical],
    (* total internal reflection *)
    reflection = True;
    rVecList = vectorAtAnAngle[normalVec, N[Pi - thetai]],
    (* refraction *)
    reflection = False; (* compute refraction angle by Snell's law *)
    thetar = N[ArcSin[ni Sin[thetai] / no]];
    rVecList = vectorAtAnAngle[normalVec, thetar]
];
(* from the candidate rays, determine the correct outgoing ray,
    by picking the one that has a maximum dot product value with
    the incoming ray. *)
dotprods = Map[(#[[2]].{a,b})&, rVecList];

max = Max[dotprods];

{where} = First[Position[dotprods, max]];

(* return the winning candidate ray as the output ray,
    together with an indicator that tells if it is a refracted ray,
    or a totally internally reflected one. *)
{ {{xi,yi}, rVecList[[where]][[2]], If[reflection,ni,no]}, reflection }
```

$\{\{\{-2.22045\ 10^{-16},\ -8.88178\ 10^{-16}\},$
$\{0.948683,\ 0.316228\},\ 3\},\ \mathrm{False}\}$

```
(* initialize temporary local vars *)
incomingRay = {{-1,-3},normalize[{1,3}],ni};
surface = {f, bc, ni, no};
critical = dotprod = fx = fy = thetai = thetar = Infinity;
reflection = True;
dotprods = rayVecList = tList = {};
where = 0;
max = -Infinity;
{{x0,y0}, {a, b}, n} = incomingRay;
{f, bc, ni, no} = surface;
{xi, yi} = {x0, y0};
normalVec = {{x0,y0},{a,b}};
```

```
{"ni =", ni, "no =", no, "n = ", n}
{ni =, 1, no =, 3, n = , 1}

(* check for consistency between the refractive indices mentioned in
    the incomingRay and the surface. *)
If[eqQ[n,no], (* coming from the second side, so swap the ni and no *)
    {ni, no} = {no, ni},
    If[neQ[n,ni],
        Print["Check parameter consistency to RayAtBoundary"]
    ]
]

eqQ[n,no]
False

{"ni =", ni, "no =", no}
{ni =, 1, no =, 3}

(* compute the list of values for the parameter t, corresponding to
    the points of intersection of incomingRay with the surface *)
tList = t /.
    NSolve[{ Evaluate[f[x,y] /.
            {x -> x0 + a * t,
             y -> y0 + b * t}
          ] == 0},
        {t}]
{3.16228, 5.05964}
```

```
(* filter out those t values that are neither real,
     nor are in the forward direction *)
tList = Sort[Flatten[
        Map[ If[ltQ[#,0] ||
                         neQ[Im[#],0], {}, #]&, tList ]
    ]]
{3.16228, 5.05964}

(* further filter the ones that do not satisfy the boundary conditions,
    and then find the minimum positive t
    that satisfies boundary conditions. *)
tMin = Min[Map[
    If[bc[x0 + a * #, y0 + b * #],#,Infinity]&, tList]
]
3.16228

(* at last, compute the point of intersection of the surface,
    by the incoming ray *)
{xi, yi} = With[{t = tMin}, {x0 + a * t, y0 + b * t}]
{-2.22045 10^-16, -8.88178 10^-16}
```

```
(* compute the unit normal to the surface at the point of contact *)
{fx, fy} = normalize[
        {D[f[x,y], x], D[f[x,y], y]} /.
            {x -> xi, y -> yi}
    ]

{-1., -2.96059 10^-16}

(* make sure that the incoming ray makes an acute angle with the normal *)
If[ gtQ[(dotprod = {a,b}.{fx,fy}), 0],
    thetai = N[ArcCos[dotprod]] ,
    (* else *)
    thetai = N[Pi - ArcCos[dotprod]];
    {fx, fy} = {-fx, -fy}
]

{1., 2.96059 10^-16}

thetai /Degree //N

71.5651

(* compute the normal vector at the point of intersection *)
normalVec = {{xi, yi}, With[{x = xi, y = yi}, Evaluate[{fx,fy}] ] }

{{-2.22045 10^-16, -8.88178 10^-16},
 {1., 2.96059 10^-16}}
```

```
(* determine if the ray is going to undergo total internal reflection,
    or not (refraction).
    In either case,
        obtain a list of candidate outgoing rays in rVecList *)
critical = If[gtQ[ni,no], N[ArcSin[no / ni]], Infinity ];
If[ gtQ[ni,no] &&
    gtQ[thetai,critical],
    (* total internal reflection *)
    reflection = True;
    rVecList = vectorAtAnAngle[normalVec, N[Pi - thetai]],
    (* refraction *)
    reflection = False; (* compute refraction angle by Snell's law *)
    thetar = N[ArcSin[ni Sin[thetai] / no]];
    rVecList = vectorAtAnAngle[normalVec, thetar]
]

{{{-2.22045 10^-16, -8.88178 10^-16},
  {0.948683, 0.316228}},
 {{-2.22045 10^-16, -8.88178 10^-16},
  {0.948683, -0.316228}}}

{"ni =", ni, "no =", no}

{ni =, 1, no =, 3}

gtQ[ni,no]

False

gtQ[thetai,critical]

False
```

```
If[reflection,{"Reflection", rVecList},{"Refraction",thetar, rVecList}]
```

```
{Refraction, 0.321751,
 {{{-2.22045 10^-16, -8.88178 10^-16},
   {0.948683, 0.316228}},
  {{-2.22045 10^-16, -8.88178 10^-16},
   {0.948683, -0.316228}}}}
```

```
(* from the candidate rays, determine the correct outgoing ray,
    by picking the one that has a maximum dot product value with
    the incoming ray. *)
dotprods = Map[(#[[2]].{a,b})&, rVecList]
```

```
{0.6, 3.88578 10^-16}
```

```
max = Max[dotprods]
```

```
0.6
```

```
{where} = First[Position[dotprods, max]]
```

```
{1}
```

```
(* return the winning candidate ray as the output ray,
    together with an indicator that tells if it is a refracted ray,
    or a totally internally reflected one. *)
{ {{xi,yi}, rVecList[[where]][[2]], If[reflection,ni,no]},
    If[reflection,"Reflection","Refraction"] }
```

```
{{{-2.22045 10^-16, -8.88178 10^-16},
  {0.948683, 0.316228}, 3}, Refraction}
```

Chapter 56

Miscellaneous Graphics Routines

Chapter Abstract

This notebook contains miscallenous routines for graphics. These are used by the other graphics notebooks.

Miscellaneous Graphics Routines

This notebook contains miscellaneous routines for graphics.
These are used by the other graphics notebooks.

```
(* Displays in 3D a wire graphics object represented by an edge list
    Options[displayObject] = {Functionality}
    If (Functionality -> Identity) then return edgeList. *)
Clear[displayObject];
displayObject[edgeList__, options___]:=
Module[{r, obj},
    r = ({-0.5, 0.5} + #)& /@ (* extend the boundaries by 0.5 *)
        FullOptions[
            Show[obj = Graphics3D[{Thickness[0.01], Map[Line, edgeList] }],
                DisplayFunction -> Identity
            ],
            PlotRange
        ];
    showOptions = FilterOptions[Graphics3D, options];
    If[(Functionality /. {options}) === Identity,
            edgeList,
        Show[obj,
             showOptions, DisplayFunction -> $DisplayFunction,
             PlotRange -> r, Boxed -> False,
            Axes -> True, AxesLabel -> {"x, x*", "y, y*", "z, z*"},
            AspectRatio -> 1
        ]
    ]
]
```

Chapter 57

2D Graphics Operations

Chapter Abstract

This notebook contains the necessary routines to display, position and orient graphics objects in 2D. Numerous operations are provided such as:

- translateBy[point[{m, n}]]
- rotateAround[angle[theta]]
- rotateAround[point[{m, n}], angle[theta]]
- rotateBy[angle[theta]]
- reflectThrough[line[point[{x1, y1}], point[{x2, y2}]]]
- scaleLocal[{xScale, yScale}]
- scaleOverall[scaleFactor]

2D Graphics Operations

This notebook contains the necessary routines to display, position and orient Graphics Objects in 2D. Numerous operations are provided such as:

```
- translateBy[point[{m, n}]]
- rotateAround[ angle[theta] ]
- rotateAround[ point[{m, n}], angle[theta] ]
- rotateBy[ angle[theta] ]
- reflectThrough[ line[ point[{x1, y1}], point[{x2, y2}] ] ]
- scaleLocal[ {xScale, yScale} ]
- scaleOverall[ scaleFactor ]
```

```
Clear[displayPolygon];
displayPolygon[polygonList__, options___]:=
Module[{r, obj},
    r = ({-0.5, 0.5} + #)& /@ (* extend the boundaries by 0.5 *)
        FullOptions[
            Show[obj = Graphics[{Polygon[polygonList]}],
                DisplayFunction -> Identity
            ],
            PlotRange
        ];
    showOptions = FilterOptions[Graphics, options];

    Show[Graphics[{Polygon[polygonList]}],
        DisplayFunction -> $DisplayFunction, showOptions,
        PlotRange -> r,
        Axes -> True, AxesLabel -> {"x, x*", "y, y*"},
        AspectRatio -> 1
    ]
]
```

2D Graphics Primitives

Point in 2D is represented as {x, y}

Transformations and Matrices: treat [T] as a geometric operator. An object [A] can be operated on by [T] to give [B] = [A] [T]

Transformation of Points

Transformation of the point {x, y} to {a x + c y, b x + d y} can be accomplished by the transformation Matrix T = {{a, b}, {c, d}}

```
Clear[a, b, c, d];
T = {{a, b}, {c, d}}; MatrixForm[T]
```

```
a  b
c  d
```

Special Cases

Identity Transformation

```
a = d = 1;
b = c = 0;
```

```
{x, y} . T
```

```
{x, y}
```

Scale x by a.

```
d = 1;
b = c = 0;
Clear[a];
```

```
{x, y} . T
```

```
{a x, y}
```

Scale x and y by scaling factors a and d.

```
b = c = 0;
Clear[a, d];
```

```
{x, y} . T
```

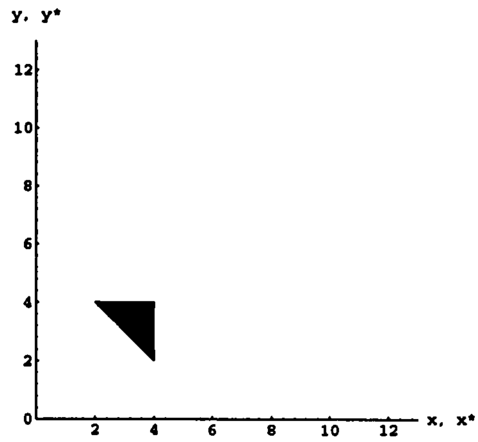
```
{a x, d y}
```

(LocalScalingFactors < 1) => Shrinking.
(LocalScalingFactors > 1) => Enlargement.

Example

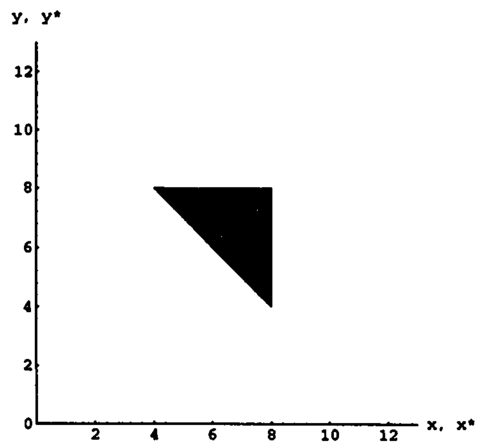
```
pa = {4, 2};
pb = {4, 4};
pc = {2, 4};
triangle = {pa, pb, pc};
Clear[pa, pb, pc];
```

```
displayPolygon[triangle, PlotRange -> {{0, 13}, {0, 13}}];
```



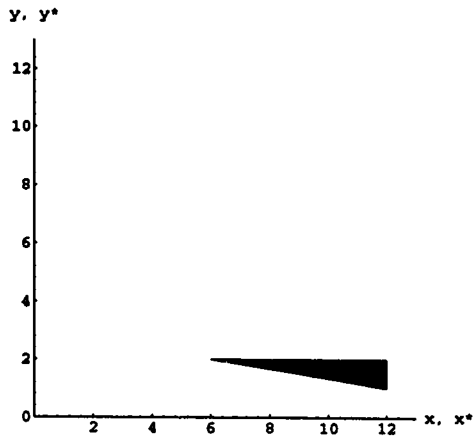

```
a = d = 2;
b = c = 0;
```

```
displayPolygon[Map[# . T &, triangle], PlotRange -> {{0, 13}, {0, 13}}];
```




```
d = 0.5;
a = 3;
b = c = 0;
```

```
displayPolygon[Map[# . T &, triangle], PlotRange -> {{0, 13}, {0, 13}}];
```



a (d) is negative => scaled reflection wrt x-axis (y-axis).

```
b = c = 0;
d = 1;
a = -1;
```

```
{x, y} . T
{-x, y}
```

NOTE : Both reflection and scaling of the coordinates involve only the diagonal terms of [T].

Shearing in the x direction.

```
a = d = 1;
b = 0;
Clear[c];
```

```
{x, y} . T
{x + c y, y}
```

Shearing in the y direction.

```
a = d = 1;
c = 0;
Clear[b];
```

```
{x, y} . T
{x, b x + y}
```

Transformation of Straight Lines

A straight line can be represented by its two end points.
L = {p1_Point, p2_Point}

Transformed line, L* according to the transformation T is [L . T]

```
NOTE : The transformation of the line preserves
1) the midpoint according to [T].
2) parallelism between two lines.
3) Intersection of two lines.
```

TODO: Illustrate the above using MMA 2D graphics (pg 65, Fig 1.2).

Transformations of Parallel and Intersecting Lines

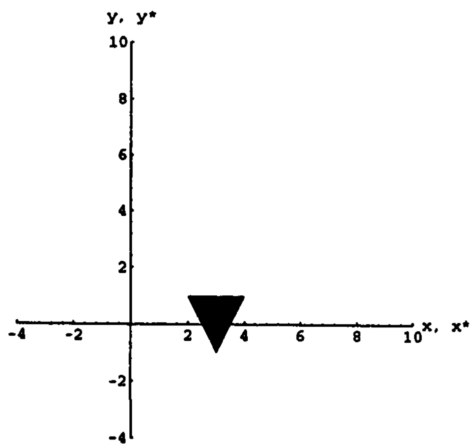
TODO: Illustrate the above using MMA 2D graphics (pg 70, Fig 1.2).

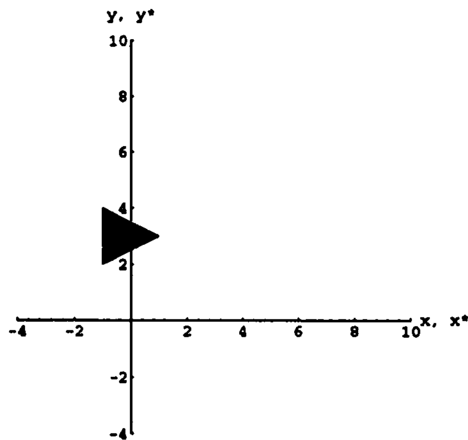
Rotation

```
rotateByTheta[polygon__, theta_]:=
    Module[{t = {{Cos[theta], Sin[theta]}, {-Sin[theta], Cos[theta]}} },
        polygon . t
    ]
```

```
pa = {3, -1};
pb = {4, 1};
pc = {2, 1};
triangle = {pa, pb, pc};
Clear[pa, pb, pc];
```

```
displayPolygon[triangle, PlotRange -> {{-4, 10}, {-4, 10}}];
displayPolygon[rotateByTheta[triangle, 90 Degree],
                PlotRange -> {{-4, 10}, {-4, 10}}];
```





Reflection

```
reflection[polygon__, axis_] /; (Det[axis] == -1) :=
    Module[{t = axis},
    rotateByTheta[polygon, ]
```

Scaling

TODO: Illustrate the above using MMA 2D graphics (pg 76, Fig 2.6).

TODO: Illustrate the above using MMA 2D graphics (pg 80, Fig 2.9).

Transformation Matrices for Different Primitive 2D-Operations

Operation representation

```
translateBy[point[{m, n}]]
rotateAround[ angle[theta] ]
rotateAround[ point[{m, n}], angle[theta] ]
rotateBy[ angle[theta] ]
reflectThrough[ line[ point[{x1, y1}], point[{x2, y2}] ] ]
scaleLocal[ {xScale, yScale} ]
scaleOverall[ scaleFactor ]
```

```
operationToMatrix[ translateBy[point[{m_, n_}]] ]:=
    {   { 1, 0, 0},
        { 0, 1, 0},
        { m, n, 1} }
```

```
operationToMatrix[ translateBy[point[{3, 5}]] ] //MatrixForm
```

```
1  0  0
0  1  0
3  5  1
```

Transformation matrix for rotation by an angle theta around the origin.

```
operationToMatrix[ rotateBy[ angle[theta_] ] ]:=
    operationToMatrix[ rotateAround[ angle[theta] ] ];

operationToMatrix[ rotateAround[ angle[theta_] ] ]:=
    Chop[ N[
        {   { Cos[theta],   Sin[theta], 0},
            { -Sin[theta],  Cos[theta], 0},
            { 0,            0,          1} }
    ]];
```

Transformation matrix for rotation by an angle theta around the given point.

1) Translate so that the given point becomes the origin.
2) Rotate by theta.
3) Translate back to original place.

```
operationToMatrix[ rotateAround[ point[{m_, n_}], angle[theta_] ] ]:=
    Dot[
        operationToMatrix[ translateBy[ point[{-m, -n}] ] ],
        operationToMatrix[ rotateAround[ angle[theta] ] ],
        operationToMatrix[ translateBy[ point[{m, n}] ] ]
    ]

operationToMatrix[ rotateBy[ angle[ 45 Degree ] ] ] //MatrixForm

0.707107    0.707107    0
-0.707107   0.707107    0
0           0           1.

operationToMatrix[ rotateAround[ angle[ 45 Degree ] ] ] //MatrixForm

0.707107    0.707107    0
-0.707107   0.707107    0
0           0           1.


operationToMatrix[ rotateAround[ point[{4, 3}], angle[90 Degree] ] ]
    //MatrixForm

0    1.   0
-1.  0    0
7.   -1.  1.

operationToMatrix[ flipAroundXaxis[ ] ] :=
        {   { 1,  0, 0},
            { 0, -1, 0},
            { 0,  0, 1} };
operationToMatrix[ flipAroundYaxis[ ] ] :=
        {   {-1, 0, 0},
            { 0, 1, 0},
            { 0, 0, 1} };
```

Transformation matrix for reflection through a given line.

1) Translate so that the given line passes through the origin.
2) Rotate so that the translated line coincides with the x-axis.
3) Reflect through the x-axis.
4) Rotate to restore the original orientation. (inverse of step 2)
5) Translate to the original place. (inverse of step 1).

```
operationToMatrix[ reflectThrough[ line[ point[{x1_, y1_}], point[{x2_, y2_}] ] :
Module[{c = (x1 y2 - x2 y1) / (x1 - x2),
        theta = N[ArcTan[(y2 - y1)/(x2 - x1)]]},

    Dot[
        operationToMatrix[ translateBy[ point[{0, -c}] ] ],
        operationToMatrix[ rotateBy[ angle[-theta] ] ],
        operationToMatrix[ flipAroundXaxis[ ] ],
        operationToMatrix[ rotateBy[ angle[theta] ] ],
        operationToMatrix[ translateBy[ point[{0, c}] ] ]
    ]
]
```

```
operationToMatrix[ reflectThrough[ line[ point[{2, 3}], point[{4, 4}] ] ] ]
     //MatrixForm
```

```
0.6    0.8    0.
0.8    -0.6   0.
-1.6   3.2    1.
```

Transformation matrix for scaling all x-coordinates by xScale and y-coordinates by yScale.

```
operationToMatrix[ scaleLocal[ {xScale_, yScale_}  ] ]:=
    {   { xScale,  0,      0},
        { 0,       yScale, 0},
        { 0,       0,      1}  }
```

```
operationToMatrix[ scaleLocal[ {1/2, 2} ] ] //MatrixForm
```

$$\begin{matrix} \frac{1}{2} & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Transformation matrix for scaling all coordinates by scaleFactor.

```
operationToMatrix[ scaleOverall[ scaleFactor_ ] ]:=
    {   { 1, 0, 0},
        { 0, 1, 0},
        { 0, 0, scaleFactor}   }
```

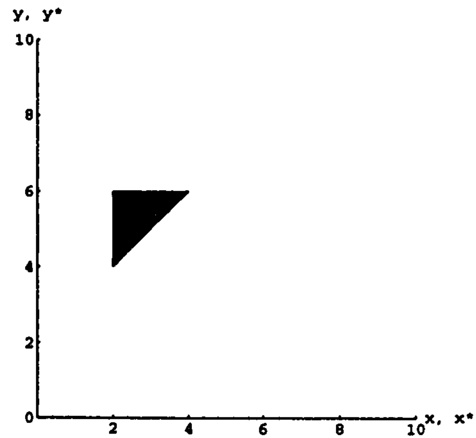
```
operationToMatrix[  scaleOverall[ 2 ] ] //MatrixForm
```

```
1   0   0
0   1   0
0   0   2
```

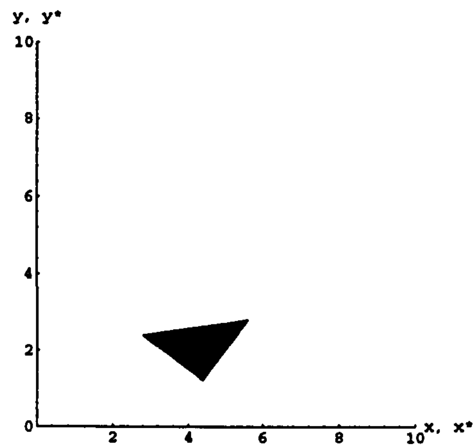
```
transform[object_, operations__, options___] :=
    Module[ {transformationMatrices, matrixChain, temp,
             homogenizedObject, objectSteps},
            homogenizedObject = Map[Append[#, 1]&, object];
            transformationMatrices = Map[ operationToMatrix, operations];
            matrixChain =
                FoldList[Dot, IdentityMatrix[3], transformationMatrices];
            If[! (ShowSteps /. Append[List[options], ShowSteps -> False]),
                    matrixChain = {Last[matrixChain]}
            ];
            objectSteps = Table[
                temp = Map[ Dot[#, matrixChain[[i]]]&,  homogenizedObject];
                Map[ Drop[#, -1] / If[ Last[#]==0, 1, Last[#] ] &, temp],
                {i, Length[matrixChain]}
            ];
            If[ShowSteps /. Append[List[options], ShowSteps -> False],
                    objectSteps,
                    First[objectSteps]
            ]
    ]
{x1, y1} = {2, 3};
{x2, y2} = {4, 4};
c = (x1 y2 - x2 y1) / (x1 - x2);
theta = N[ArcTan[(y2 - y1)/(x2 - x1)]];
obj = {{0,0}};
```

```
tempOut1 = transform[obj, {translateBy[ point[{0, -c}] ],
        rotateBy[ angle[-theta] ],
        flipAroundXaxis[ ],
        rotateBy[ angle[theta] ],
        translateBy[ point[{0, c}] ]}]

{{-1.6, 3.2}}

Clear[obj, theta, c, x1, x2, y1, y2];

tempOut2 = {0,0,1} .
    operationToMatrix[ reflectThrough[ line[ point[{2, 3}], point[{4, 4}] ] ] ]
{-1.6, 3.2, 1.}

tempOut1[[1]] == tempOut2[[{1, 2}]]
True

Clear[tempOut1, tempOut2];

pa = {2, 4};
pb = {4, 6};
pc = {2, 6};
triangle = {pa, pb, pc};
Clear[pa, pb, pc];
```

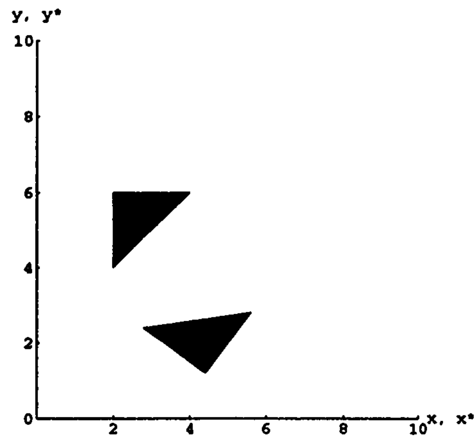
```
tempPlot1 = displayPolygon[triangle, PlotRange -> {{0, 10}, {0, 10}}];
```




```
tempPlot2 = displayPolygon[
transform[ triangle,
        { reflectThrough[ line[ point[{2, 3}], point[{4, 4}] ] ]}],
PlotRange -> {{0, 10}, {0, 10}}];
```



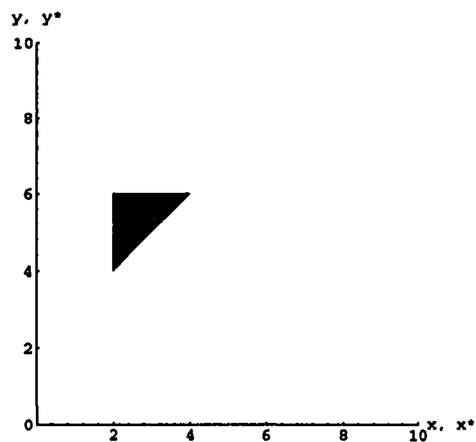
```
Show[tempPlot1, tempPlot2];
Clear[tempPlot1, tempPlot2];
```

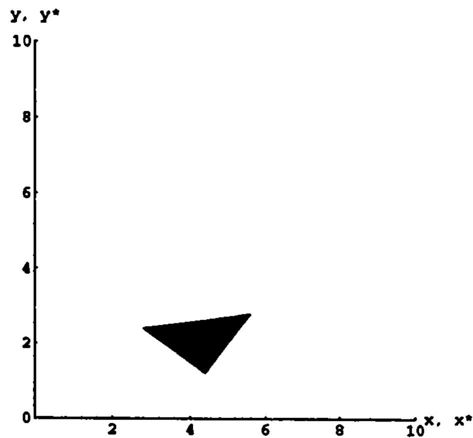



```
tempObjects = transform[ triangle,
                { reflectThrough[ line[ point[{2, 3}], point[{4, 4}] ] ] },
                        ShowSteps -> True]
```

```
{{{2, 4}, {4, 6}, {2, 6}}, {{2.8, 2.4}, {5.6, 2.8}, {4.4, 1.2}}}
```

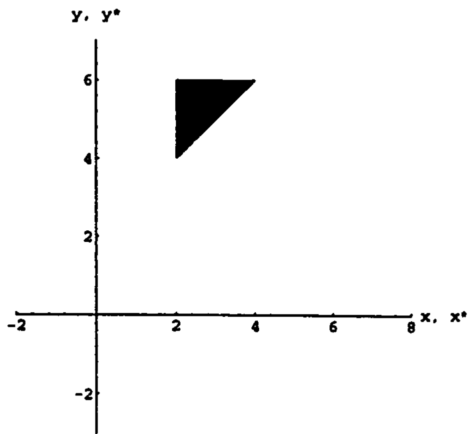
```
Map[displayPolygon[#, PlotRange -> {{0, 10}, {0, 10}}]&, tempObjects];
Clear[tempObjects];
```

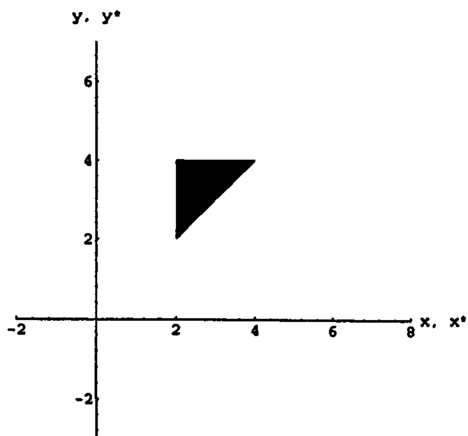


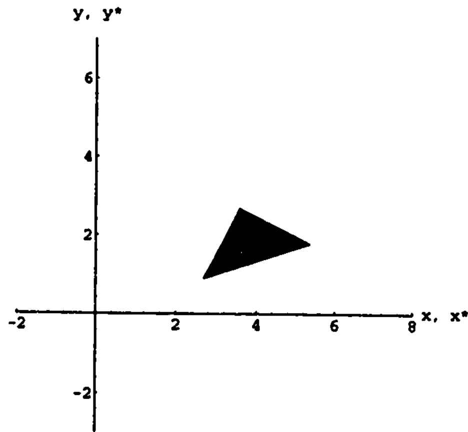


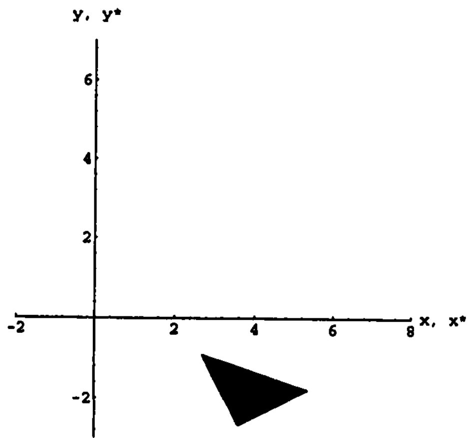

```
{x1, y1} = {2, 3};
{x2, y2} = {4, 4};
c = (x1 y2 - x2 y1) / (x1 - x2);
theta = N[ArcTan[(y2 - y1)/(x2 - x1)]];
```

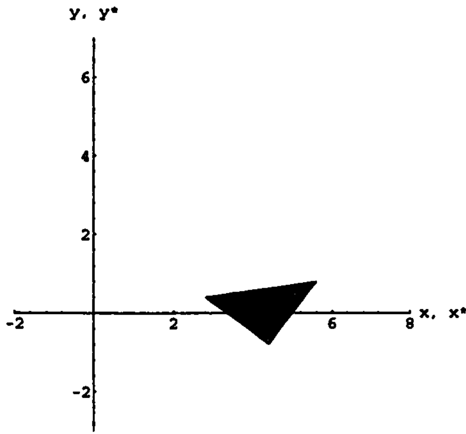
```
tempObject = transform[triangle, {translateBy[ point[{0, -c}] ],
        rotateBy[ angle[-theta] ],
        flipAroundXaxis[ ],
        rotateBy[ angle[theta] ],
        translateBy[ point[{0, c}] ]},
        ShowSteps -> True]
{{{2, 4}, {4, 6}, {2, 6}}, {{2, 2}, {4, 4}, {2, 4}},
  {{2.68328, 0.894427}, {5.36656, 1.78885}, {3.57771, 2.68328}},
  {{2.68328, -0.894427}, {5.36656, -1.78885}, {3.57771, -2.68328}},
  {{2.8, 0.4}, {5.6, 0.8}, {4.4, -0.8}}, {{2.8, 2.4}, {5.6, 2.8}, {4.4, 1.2}}}
Clear[obj, theta, c, x1, x2, y1, y2];
tempObjects =
    Map[displayPolygon[#, PlotRange -> {{-2, 8}, {-3, 7}}]&, tempObject];
Clear[tempObject];
```

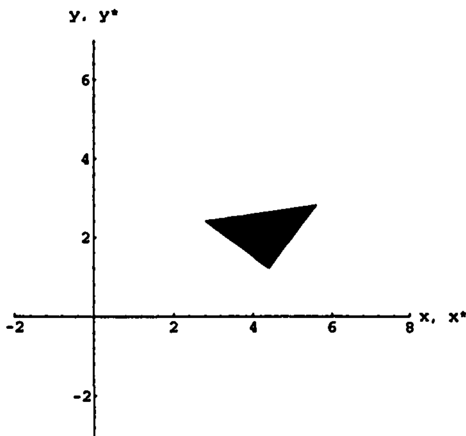

y, y*
6
4
2
-2
2
4
6
8
x, x*
-2


y, y*
6
4
2
-2
2
4
6
8
x, x*
-2

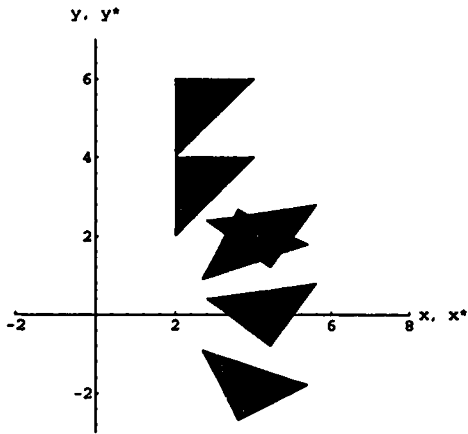

y, y*
6
4
2
-2
2
4
6
8
x, x*
-2


y, y*
6
4
2
-2
2
4
6
8
x, x*
-2


y, y*
6
4
2
-2
-2
2
6
8
x, x*


y, y*
6
4
2
-2
-2
2
4
6
8
x, x*

```
Show[tempObjects]; Clear[tempObjects];
```



Chapter 58

3D Graphics Operations

Chapter Abstract

This notebook contains the necessary code for 3D graphics operations.

3D Graphics Operations

This Notebook contains the necessary code for 3D graphics operations.
The various operations supported are:

```
reflectThrough[plane[a, b, c, d]]
rotateAround[
        line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ],
        angle[ theta ]
    ]
translateBy[ {l, m, n} ]
reflectThroughXY[ ]
reflectThroughYZ[ ]
reflectThroughXZ[ ]
rotateAroundXBy[ angle[theta] ]
rotateAroundYBy[ angle[theta] ]
rotateAroundZBy[ angle[theta] ]
shear[{b, c, d, f, g, i}]
scaleOverall[ scaleFactor ]
scaleLocal[ {xScale, yScale, zScale} ]
```

Point in 3D is represented as {x, y, z}
Transformations and Matrices: treat [T] as a geometric operator. An object [A] can be operated on by [T] to give [B] = [A] [T]

Operation representation


```
reflectThrough[plane[a, b, c, d]]
rotateAround[
        line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ],
        angle[ theta ]

    ]
translateBy[ {l, m, n} ]
reflectThroughXY[ ]
reflectThroughYZ[ ]
reflectThroughXZ[ ]
rotateAroundXBy[ angle[theta] ]
rotateAroundYBy[ angle[theta] ]
rotateAroundZBy[ angle[theta] ]
shear[{b, c, d, f, g, i}]
scaleOverall[ scaleFactor ]
scaleLocal[ {xScale, yScale, zScale} ]
```


```
Mma's 3D graphics primitives.

Point[coords]
Line[{pt1, pt2, ...}]
Polygon[{pt1, pt2, ...}]
Cuboid[{xmin, ymin, zmin}]
Text[expr, coords]
```

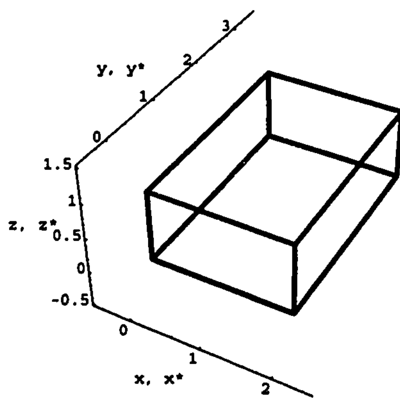
Point in 3D is represented as {x, y, z}

Transformations and Matrices: treat [T] as a geometric operator. An object [A] can be operated on by [T] to give [B] = [A] [T]

□ **Example: Drawing a parallelopiped using displayObject[].**

```
pa = {0, 0, 1};
pb = {2, 0, 1};
pc = {2, 3, 1};
pd = {0, 3, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 3, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},           (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

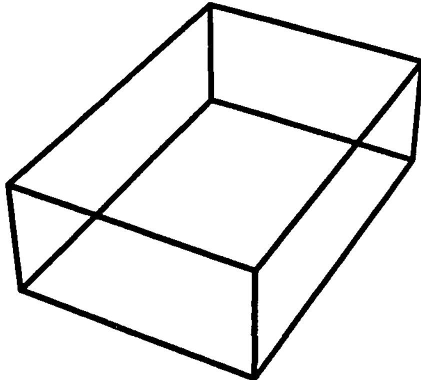
```
displayObject[object];
Clear[object];
```



□ **Example: Drawing a parallelopiped using 3D graphics primitives.**

```
pa = {0, 0, 1};
pb = {2, 0, 1};
pc = {2, 3, 1};
pd = {0, 3, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 3, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},           (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

```
Show[Graphics3D[{Thickness[0.01], Map[Line, object] }], Boxed -> False];
Clear[object];
```



Transformation Matrices for Different Primitive 3D-Operations

Generalized Transformation Matrix for 3D operations.

```
        | a b c p |
[T] =   | d e f q |
        | g i j r |
        | l m n s |
```

Notes:

```
A point [x, y, z] in 3D space is represented as a four dimensional position
vector [x' y' z' h] = [x y z 1] . [T]

Transformation from homogenous coordinates to ordinary coordinates is:
[x* y* z* 1] = [x'/h y'/h z'/h 1]

        | a b c |
        | d e f |   produces a linear transformation like scaling, shearing,
        | g i j |   reflection and rotation.

        [ l m n ]   produces translation.

        | p |
        | q |       produces a perspective transformation.
        | r |

        [ s ]       produces overall scaling.
```

Operation representation

```
reflectThrough[plane[a, b, c, d]]
rotateAround[
        line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ],
        angle[ theta ]
    ]
translateBy[ {l, m, n}  ]
reflectThroughXY[ ]
reflectThroughYZ[ ]
reflectThroughXZ[ ]
rotateAroundXBy[ angle[theta] ]
rotateAroundYBy[ angle[theta] ]
rotateAroundZBy[ angle[theta] ]
shear[{b, c, d, f, g, i}]
scaleOverall[ scaleFactor ]
scaleLocal[ {xScale, yScale, zScale}  ]
```

```
(* apply the operation in sequence to an edge list of the object.
   Options[transform3D] = {ShowSteps -> True/False}.
   False (default) intermediate versions of the object are not returned.
   True otherwise *)
Clear[transform3D];
transform3D[object_, operations__, options___] :=
   Module[ {transformationMatrices, matrixChain, temp,
            homogenizedObject, objectSteps, showSteps},
           showSteps = ShowSteps /. {options, ShowSteps -> False};
           homogenizedObject = Map[Append[#, 1]&, object, {2}];
           transformationMatrices = Map[ operationToMatrix, operations];
           matrixChain =
              FoldList[Dot, IdentityMatrix[4], transformationMatrices];
           If[! showSteps, matrixChain = {Last[matrixChain]}];
           objectSteps = Table[
              temp = Map[ Dot[#, matrixChain[[i]]]&,  homogenizedObject, {2}];
              Map[ Drop[#, -1] / If[ Last[#]==0, 1, Last[#] ] &, temp, {2}],
              {i, Length[matrixChain]}
           ];
           If[ showSteps, objectSteps, First[objectSteps] ]
   ]
```

Local Scaling

Transformation matrix for scaling all x-coordinates by xScale, y-coordinates by yScale and z-coordinates by zScale.

```
operationToMatrix[ scaleLocal[ {xScale_, yScale_, zScale_}  ] ]:=
   {  { xScale,  0,       0,      0},
      { 0,       yScale,  0,      0},
      { 0,       0,       zScale, 0},
      { 0,       0,       0,      1}  }
```

Example : Local Scaling

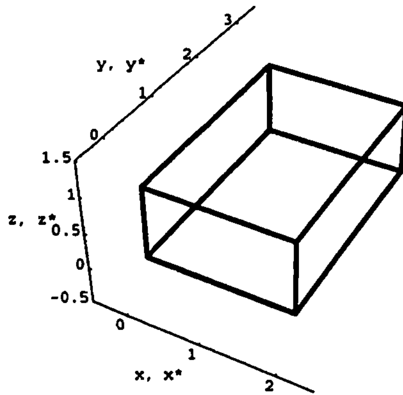
```
operationToMatrix[ scaleLocal[ {1/2, 1/3, 1} ] ] // MatrixForm
```

$$\begin{matrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

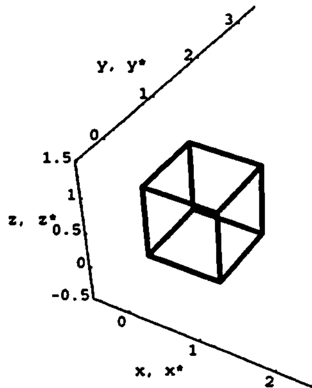
Example 3.1, Fig 3.1 from Rogers & Adams

```
pa = {0, 0, 1};
pb = {2, 0, 1};
pc = {2, 3, 1};
pd = {0, 3, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 3, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

```
displayObject[object, PlotRange -> {{-0.5, 2.5}, {-0.5, 3.5}, {-0.5, 1.5}}];
```



```
displayObject[transform3D[ object, {scaleLocal[ {1/2, 1/3, 1} ]}],
            PlotRange -> {{-0.5, 2.5}, {-0.5, 3.5}, {-0.5, 1.5}}];
Clear[object];
```



Overall Scaling

Transformation matrix for scaling all coordinates by scaleFactor.

```
operationToMatrix[ scaleOverall[ scaleFactor_ ] ]:=
    {   { 1, 0, 0, 0},
        { 0, 1, 0, 0},
        { 0, 0, 1, 0},
        { 0, 0, 0, 1/scaleFactor}   }
```

Example : Overall Scaling

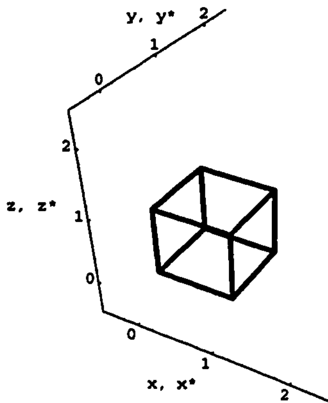
```
operationToMatrix[  scaleOverall[ 2 ] ] //MatrixForm
```

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{matrix}$$

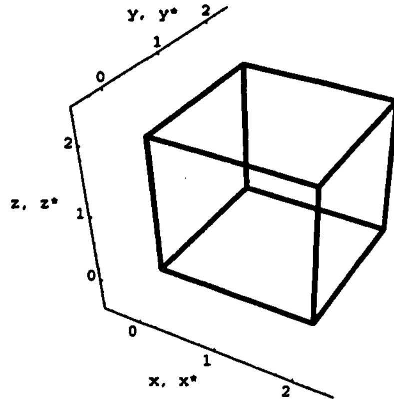
Example 3.1, Fig 3.1 from Rogers & Adams

```
pa = {0, 0, 1};
pb = {1, 0, 1};
pc = {1, 1, 1};
pd = {0, 1, 1};
pe = {0, 0, 0};
pf = {1, 0, 0};
pg = {1, 1, 0};
ph = {0, 1, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

```
displayObject[object, PlotRange -> {{-0.5, 2.5}, {-0.5, 2.5}, {-0.5, 2.5}}];
```



```
displayObject[transform3D[ object, {scaleOverall[ 2 ]}],
              PlotRange -> {{-0.5, 2.5}, {-0.5, 2.5}, {-0.5, 2.5}}];
Clear[object];
```



Shearing

Transformation matrix for scaling all coordinates by scaleFactor.

```
operationToMatrix[ shear[{b_, c_, d_, f_, g_, i_}] ]:=
    {   { 1, b, c, 0},
        { d, 1, f, 0},
        { g, i, 1, 0},
        { 0, 0, 0, 1}   }
```

Example : Overall Scaling

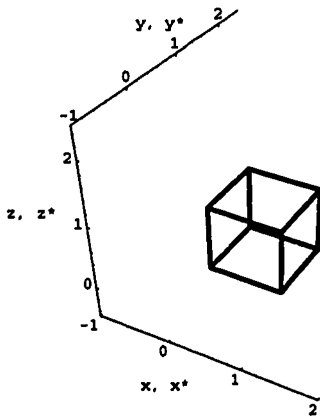
```
operationToMatrix[  shear[ {-0.85, 0.25, -0.75, 0.7, 0.5, 1} ] ] //MatrixForm
```

```
1       -0.85   0.25    0
-0.75   1       0.7     0
0.5     1       1       0
0       0       0       1
```

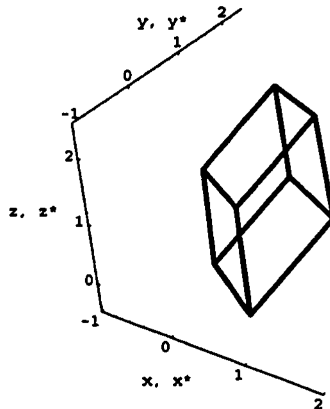
Example 3.3, Fig 3.1 (d) from Rogers & Adams

```
pa = {0, 0, 1};
pb = {1, 0, 1};
pc = {1, 1, 1};
pd = {0, 1, 1};
pe = {0, 0, 0};
pf = {1, 0, 0};
pg = {1, 1, 0};
ph = {0, 1, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

```
displayObject[object, PlotRange -> {{-1, 2}, {-1, 2.5}, {-0.5, 2.5}}];
```



```
displayObject[transform3D[object, {shear[ {-0.85, 0.25, -0.75, 0.7, 0.5, 1} ] :
            PlotRange -> {{-1, 2}, {-1, 2.5}, {-0.5, 2.5}}];
Clear[object];
```



Rotation Around the Coordinate Axes

Transformation matrix for rotation by an angle theta around the x-axis.

```
operationToMatrix[ rotateAroundXBy[ angle[theta_] ] ]:=
        Chop[ N[
        {   {1, 0,          0,          0},
            {0, Cos[theta], Sin[theta], 0},
            {0, -Sin[theta],Cos[theta], 0},
            {0, 0,          0,          1}}
    ]];
```

Transformation matrix for rotation by an angle theta around the y-axis.

```
operationToMatrix[ rotateAroundYBy[ angle[theta_] ] ]:=
        Chop[ N[
        {   {Cos[theta],    0,  -Sin[theta],    0},
            {0,             1,  0,              0},
            {Sin[theta],    0,  Cos[theta],     0},
            {0,             0,  0,              1}}
    ]];
```

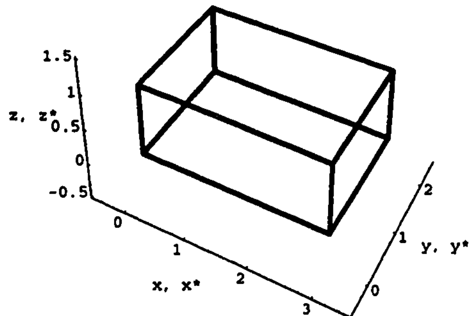
Transformation matrix for rotation by an angle theta around the z-axis.

```
operationToMatrix[ rotateAroundZBy[ angle[theta_] ] ]:=
        Chop[ N[
        {   {Cos[theta],    Sin[theta], 0,  0},
            {-Sin[theta],   Cos[theta], 0,  0},
            {0,             0,          1,  0},
            {0,             0,          0,  1}}
    ]];
```

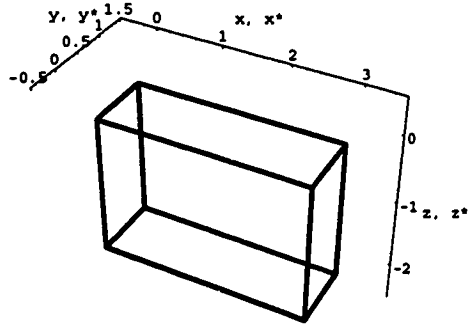
□ **Example: Rotation** *Fig 3.2.* **Example 3.4.**

```
pa = {0, 0, 1};     (* Fig 3.2 (a) *)
pb = {3, 0, 1};
pc = {3, 2, 1};     (* A in the figure *)
pd = {0, 2, 1};
pe = {0, 0, 0};
pf = {3, 0, 0};
pg = {3, 2, 0};
ph = {0, 2, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

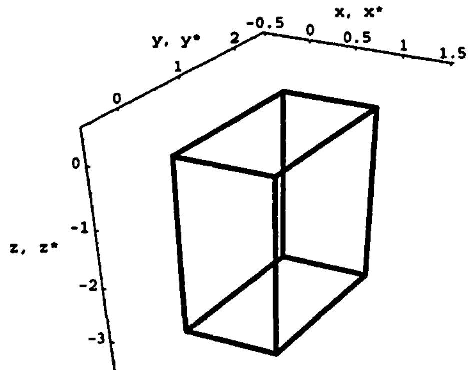
```
displayObject[object];
```



```
displayObject[transform3D[ object, {rotateAroundXBy[ angle[ -90 Degree ]]} ] ];
```




```
displayObject[transform3D[ object, {rotateAroundYBy[ angle[ 90 Degree ]]} ] ];
Clear[object];
```

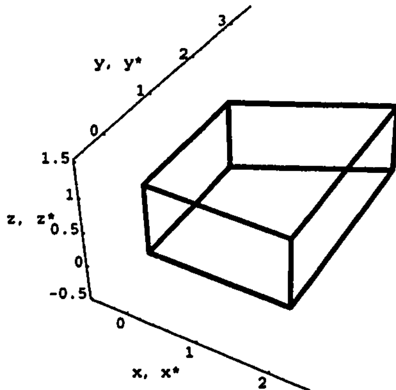


□ **Example: Illustration of non-commutativity of rotations** *Fig 3.3* **Example 3.5.**

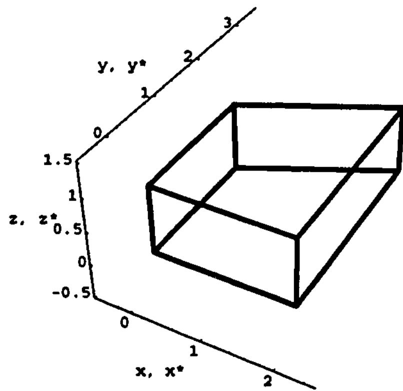
NOTE: Since 3D rotations are obtained by matrix multiplication, they are not commutative.

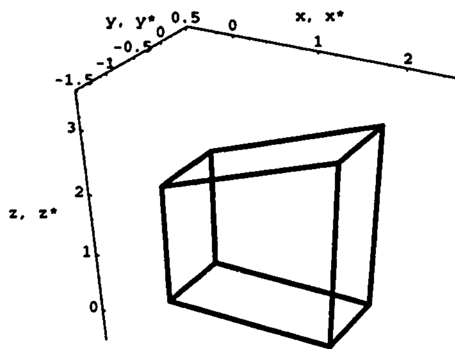
```
pa = {0, 0, 1};     (* Fig 3.3 *)
pb = {2, 0, 1};
pc = {2, 3, 1};     (* A in the figure *)
pd = {0, 2, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 2, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

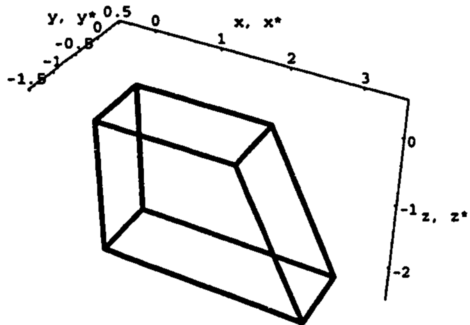
```
displayObject[object];
```



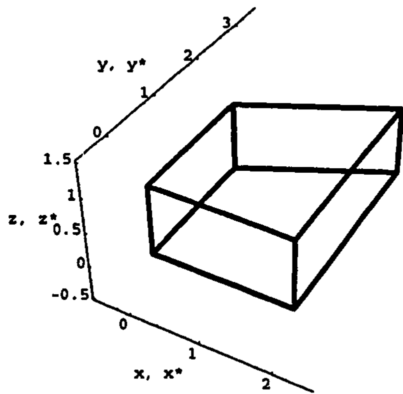

```
Map[displayObject,
        transform3D[object,
                  {   rotateAroundXBy[ angle[ 90 Degree ]],
                      rotateAroundYBy[ angle[ 90 Degree ]]
                  },
                  ShowSteps -> True
        ]
];
```

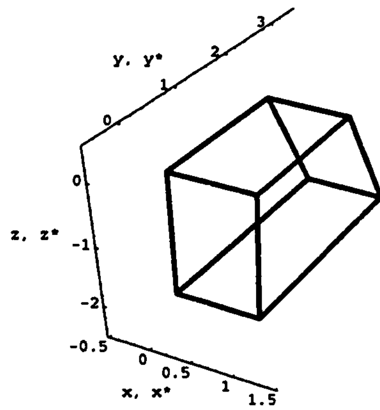

y, y*
3
2
1
0
1.5
1
z, z*
0.5
0
-0.5
0
1
x, x*
2

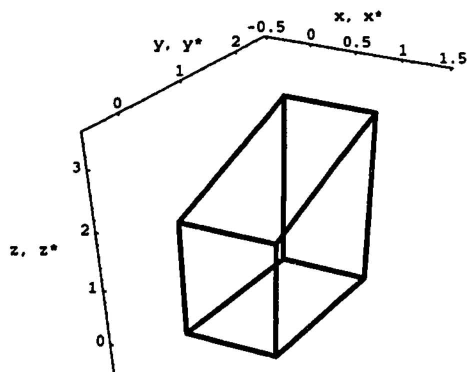

y, y*
0.5
0
-0.5
-1
-1.5
x, x*
0
1
2
3
2
z, z*
1
0



```
Map[displayObject,
        transform3D[object,
                    {   rotateAroundYBy[ angle[ 90 Degree ]],
                        rotateAroundXBy[ angle[ 90 Degree ]]
                    },
                    ShowSteps -> True
        ]
];
Clear[object];
```







Reflection Through Planes

Transformation matrix for reflection through the xy-plane.

```
operationToMatrix[ reflectThroughXY[ ] ]:=
{   {1, 0,  0, 0},
    {0, 1,  0, 0},
    {0, 0, -1, 0},
    {0, 0,  0, 1}}
```

Transformation matrix for reflection through the yz-plane.

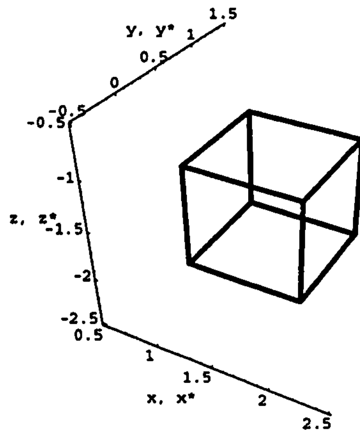
```
operationToMatrix[ reflectThroughYZ[ ] ]:=
{   {-1, 0, 0, 0},
    { 0, 1, 0, 0},
    { 0, 0, 1, 0},
    { 0, 0, 0, 1}}
```

Transformation matrix for reflection through the xz-plane.

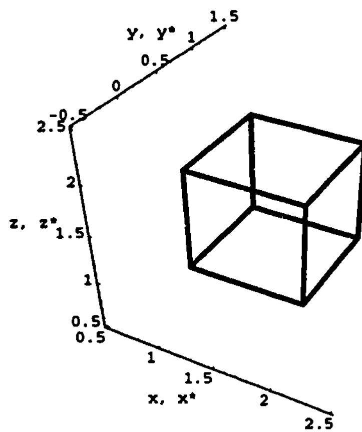
```
operationToMatrix[ reflectThroughXZ[ ] ]:=
{   {1,  0, 0, 0},
    {0, -1, 0, 0},
    {0,  0, 1, 0},
    {0,  0, 0, 1}}
```

□ **Example: Reflection through XY-plane** *Fig 3.4* **Example 3.6.**

```
pa = {1, 0, -1};        (* Fig 3.4 *)
pb = {2, 0, -1};
pc = {2, 1, -1};
pd = {1, 1, -1};
pe = {1, 0, -2};
pf = {2, 0, -2};
pg = {2, 1, -2};
ph = {1, 1, -2};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];

displayObject[object];
```



```
displayObject[transform3D[ object, {reflectThroughXY[]}]];
Clear[object];
```



■ Translation

Transformation matrix for translation by {l, m, n}.

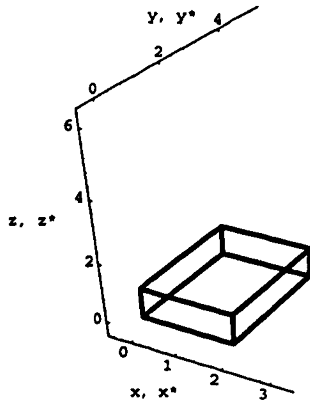
```
operationToMatrix[ translateBy[ {l_, m_, n_}  ] ]:=
    {   { 1, 0, 0, 0},
        { 0, 1, 0, 0},
        { 0, 0, 1, 0},
        { l, m, n, 1}   }
```

□ **Example : Translation**

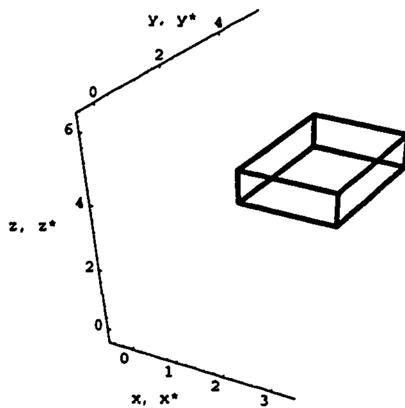
```
operationToMatrix[ translateBy[ {1, 2, 3} ] ] // MatrixForm
1   0   0   0
0   1   0   0
0   0   1   0
1   2   3   1
```

```
pa = {0, 0, 1};
pb = {2, 0, 1};
pc = {2, 3, 1};
pd = {0, 3, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 3, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pf},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pg},
  {pd, ph},
  {pe, pf}, {pe, ph},
  {pf, pg},
  {pg, ph} };
Clear[pa, pb, pc, pd, pe, pf, pg, ph];
```

```
displayObject[object, PlotRange -> {{-0.5, 3.5}, {-0.5, 5.5}, {-0.5, 6.5}}];
```




```
displayObject[transform3D[ object, {translateBy[ {1, 2, 3} ]}],
              PlotRange -> {{-0.5, 3.5}, {-0.5, 5.5}, {-0.5, 6.5}}];
Clear[object];
```



Rotation Around an Arbitrary Axis in Space

Transformation matrix for rotation about an arbitrary axis in space by a given angle.

1) Translate so that the starting point of the axis coincides with the origin.
2) Rotate the unit vector {cx, cy, cz} (that starts at the origin in the direction of the axis) around the x-axis by an angle so that it lies in the xy-plane.
3) Rotate around the y-axis by an angle so that the vector coincides with the z-axis.
4) Rotate around z-axis by the given angle.
5) Rotate around y-axis. (inverse of step 3).
6) Rotate around x-axis. (inverse of step 2).
7) Translate to the original place. (inverse of step 1).

```
operationToMatrix[
    rotateAround[
        line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ],
        angle[ theta_ ]
    ]
] :=
    operationToMatrix[
        rotateAroundXBy[angle[ theta ]]
    ] /; xAxisQ[line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ]]
```

```
operationToMatrix[
    rotateAround[
        line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ],
        angle[ theta_ ]
    ]
] :=
    operationToMatrix[
        rotateAroundYBy[angle[ theta ]]
    ] /; yAxisQ[line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ]]
```

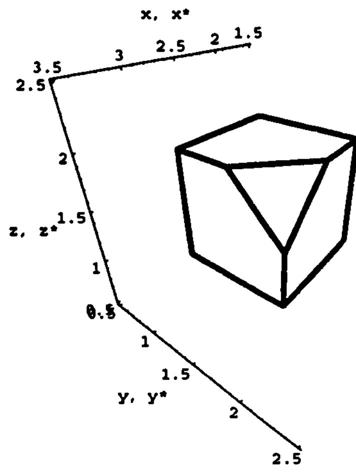
```
operationToMatrix[
    rotateAround[
        line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ],
        angle[ theta_ ]
    ]
] :=
    operationToMatrix[
        rotateAroundZBy[angle[ theta ]]
    ] /; zAxisQ[line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ]]
```

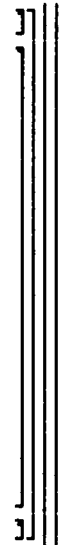
```
(* Apply only when the two end points of the line are distinct
    and the line is not any of the axes *)
operationToMatrix[
    rotateAround[
        line[ point[{x1_, y1_, z1_}], point[{x2_, y2_, z2_}] ],
        angle[ theta_ ]
    ]
] :=
Module[{cx, cy, cz, d, alpha, beta},
        {cx, cy, cz} = {x2-x1, y2-y1, z2-z1} /
                            Sqrt[(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2];
        d = Sqrt[cy^2 + cz^2];
        alpha = Chop[N[ ArcCos[cz/d] ]];
        beta = Chop[N[ ArcCos[d] ]];
        Dot[
            (* shift the reference point to origin *)
            operationToMatrix[ translateBy[ {-x1, -y1, -z1} ] ],
            (* project into xz-plane *)
            operationToMatrix[ rotateAroundXBy[ angle[alpha] ] ],
            (* project onto z-axis = align to z-axis *)
            operationToMatrix[ rotateAroundYBy[ angle[-beta] ] ],
            (* carry out the intended rotation around the given line,
              which is now the z-axis *)
            operationToMatrix[ rotateAroundZBy[ angle[theta] ] ],
            operationToMatrix[ rotateAroundYBy[ angle[beta] ] ],
            operationToMatrix[ rotateAroundXBy[ angle[-alpha] ] ],
            operationToMatrix[ translateBy[ {x1, y1, z1} ] ]
        ]
] /; (!axisQ[line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ]]) &&
        lineQ[line[ point[{x1, y1, z1}], point[{x2, y2, z2}] ] ]
```

□ **Example:** *Fig 3.8 (a)* **Example 3.8.**

```
pa = {2,   1,   2};     (* Fig 3.8 (a) *)
pb = {3,   1,   2};
pc = {3,   1.5, 2};
pd = {2.5, 2,   2};
pe = {2,   2,   2};
pf = {2,   1,   1};
pg = {3,   1,   1};
ph = {3,   2,   1};
pi = {2,   2,   1};
pj = {3,   2,   1.5};
object = { {pa, pb},  {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pg},           (* Those adjacent to pb *)
  {pc, pd}, {pc, pj},
  {pd, pe}, {pd, pj},
  {pe, pi},
  {pg, ph},
  {ph, pi}, {ph, pj}
 };
Clear[pa, pb, pc, pd, pe, pf, pg, ph, pi, pj];
```

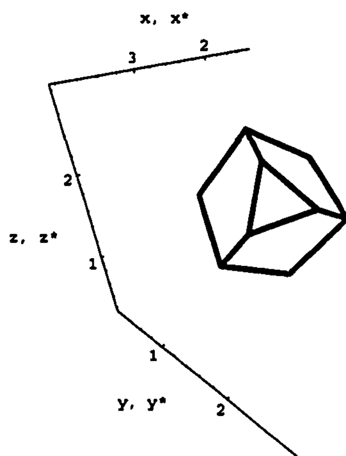
```
displayObject[object, ViewPoint -> {1, 3/4, 3/4}]
```





```
-Graphics3D-
```

```
displayObject[
    transform3D[
        object,
        {rotateAround[
            line[point[{2, 1, 1}], point[{3, 2, 2}] ],
            angle[-45 Degree] ]
        }
    ],
    ViewPoint -> {1, 3/4, 3/4}
];
Clear[object];
```



Reflection Through an Arbitrary Plane in Space

Representation of a plane = plane[a, b, c, d].
NOTE: This represents a plane governed by the equation ax+by+cz+d = 0. For such a plane, the normal is the unit normal ({a, b, c}/Sqrt[a^2 + b^2 + c^2]). These are the directional cosines {cx, cy, cz}.

Transformation matrix for reflection through an arbitrary plane in space.

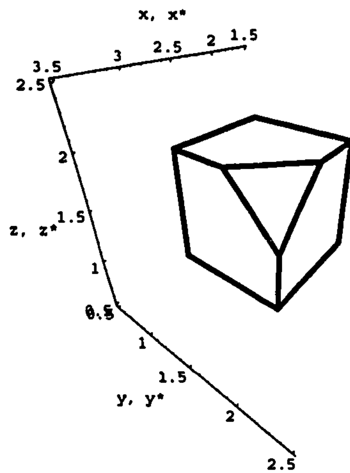
1) Translate so that a reference point in the plane coincides with the origin.
2) Rotate the unit normal vector {cx, cy, cz} of the plane so that it coincides (first rotate around x-axis and then around y-axis) with the z-axis.
3) Reflect through the given plane (which is now the xy-plane).
4) Rotate to restore the original inclination of the plane (Inverse of step (2)). (rotate around y-axis and then around x-axis)
5) Translate so that the reference point is brought back to its original place.

```
operationToMatrix[
    reflectThrough[
        plane[a_, b_, c_, d_]
    ]
] /; ((a =!= 0) || (b =!= 0) || (c =!= 0)) :=
Module[{x0, y0, z0, cx, cy, cz, len, alpha, beta},
        {x0, y0, z0} = pointInPlane[plane[a, b, c, d]];

        {cx, cy, cz} = {a, b, c}/Sqrt[a^2 + b^2 + c^2];
        len = Sqrt[cy^2 + cz^2];
        alpha = Chop[N[ ArcCos[cz/len] ]];
        beta = Chop[N[ ArcCos[len] ]];
        Dot[
            (* shift the reference point to origin *)
            operationToMatrix[ translateBy[ {-x0, -y0, -z0} ] ],
            (* project into xz-plane *)
            operationToMatrix[ rotateAroundXBy[ angle[alpha] ] ],
            (* project onto z-axis = align to z-axis *)
            operationToMatrix[ rotateAroundYBy[ angle[-beta] ] ],
            (* reflect through the given plane which is now the
               xy-plane *)
            operationToMatrix[ reflectThroughXY[ ] ],
            operationToMatrix[ rotateAroundYBy[ angle[beta] ] ],
            operationToMatrix[ rotateAroundXBy[ angle[-alpha] ] ],
            operationToMatrix[ translateBy[ {x0, y0, z0} ] ]
        ]
]
```

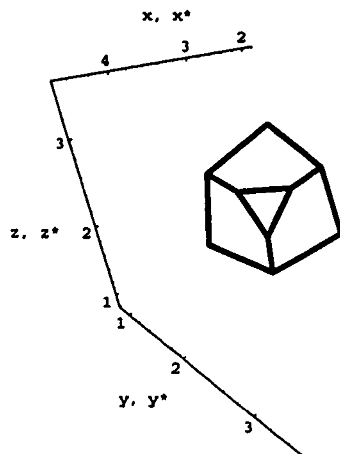
□ **Example:** *Fig 3.8 (a)* **Example 3.8.**

```
pa = {2,   1,   2};    (* Fig 3.8 (a) *)
pb = {3,   1,   2};
pc = {3,   1.5, 2};
pd = {2.5, 2,   2};
pe = {2,   2,   2};
pf = {2,   1,   1};
pg = {3,   1,   1};
ph = {3,   2,   1};
pi = {2,   2,   1};
pj = {3,   2,   1.5};
object = { {pa, pb},  {pa, pe}, (* Those adjacent to pa *)
  {pb, pc}, {pb, pg},          (* Those adjacent to pb *)
  {pc, pd}, {pc, pj},
  {pd, pe}, {pd, pj},
  {pe, pi},
  {pg, ph},
  {ph, pi}, {ph, pj}
 };
Clear[pa, pb, pc, pd, pe, pf, pg, ph, pi, pj];
```

```
displayObject[object, ViewPoint -> {1, 3/4, 3/4}];
```



```
displayObject[
    transform3D[
        object,
        {reflectThrough[ (* plane through the triangle (c, d, j) *)
            plane[0.1538461538461537, 0.1538461538461539,
                    0.1538461538461539, -1]
         ]
        }
    ] ,
    ViewPoint -> {1, 3/4, 3/4}
];
Clear[object];
```



Orienting an Object in Space

Representation for the Orientation of an object in space.

Every object has a bounding box. Initially, the bounding box is placed in the first octant with edges aligned to the coordinate axes.

Every bounding box has a reference point (every other measure of the component is relative to this point) and a principal axis (A unit vector in the x-direction of the local coordinate system) whose tail is the reference point.

The orientation is represented by {thetaX, thetaXY, thetaP}. The final orientation of the object is determined by the following steps.

-- The object is assumed to be in the first octant with its reference point at the origin.
-- Rotate the object around z-axis by an angle of thetaX. (0 < = thetaX < 2 Pi).
-- Let L be a line passing through the origin in the xz-plane and perpendicular to the rotated principal axis.
-- Rotate the object around L by thetaXY (0 <= thetaXY < 2 Pi). Hence thetaXY is the angle between the principal axis in its final orientation and the xy-plane.
-- Rotate the object around the principal axis by thetaP (0 <= thetaP < 2 Pi).

The function orient[] will return the oriented object for a given object.

```
Clear[orient];
orient[ object_, {thetaX_:0, thetaXY_:0, thetaP_:0}]:=
Module[{originalTip, pointOnYAxis, rotate1Tip, finalTip},
    (* The tip of the principal axis *)
    originalTip = {1, 0, 0};

    (* The unit reference vector on the y-axis chosen to determine L *)
    pointOnYAxis = {0, 1, 0};

    (* determine the new position of the principal axis and
       the reference vector on the y-axis after the rotation
        by thetaX around z-axis *)
    {{rotate1Tip}, {rotate1PointOnYAxis}} =
            transform3D[ {{originalTip}, {pointOnYAxis}},
                        {rotateAroundZBy[ angle[thetaX] ]}
            ];

    (* principal axis after the object is oriented *)
    {{finalTip}} = transform3D[
                    {{rotate1Tip}},
                    {rotateAround[
                        line[   point[{0, 0, 0}],
                                point[rotate1PointOnYAxis]
                        ],
                        angle[thetaXY]
                      ]
                    }
                ];

    (* carry out the three rotations namely
        1) rotation around z by thetaX
        2) rotation around L by thetaXY
        3) rotation around the final principal axis by thetaP
       on the object *)
    transformMma3D[ object,
        {   rotateAroundZBy[ angle[thetaX] ],
            rotateAround[
                line[ point[{0, 0, 0}], point[rotate1PointOnYAxis] ],
                angle[thetaXY] ],
            rotateAround[ line[ point[{0, 0, 0}], point[finalTip] ],
```

```
                                        angle[thetaP] ]
            }
        ]
    ]
```

Initially, the bounding box is placed in the first octant with edges aligned to the coordinate axes.

Every bounding box has a reference point (every other measure of the component is relative to this point) and a principal axis (A unit vector in the x-direction of the local coordinate system) whose tail is the reference point.

The orientation is represented by {thetaX, thetaXY, thetaP}. The final orientation of the object is determined by the following steps.

-- The object is assumed to be in the first octant with its reference point at the origin.
-- Rotate the object around z-axis by an angle of thetaX. (0 < = thetaX < 2 Pi).
-- Let L be a line passing through the origin in the xz-plane and perpendicular to the rotated principal axis.
-- Rotate the object around L by thetaXY (0 <= thetaXY < 2 Pi). Hence thetaXY is the angle between the principal axis in its final orientation and the xy-plane.
-- Rotate the object around the principal axis by thetaP (0 <= thetaP < 2 Pi).

The function `orientBoundingBox[]` will return the oriented bounding Box.

```
Clear[orientBoundingBox];
orientBoundingBox[ object_, {thetaX_:0, thetaXY_:0, thetaP_:0}]:=
Module[{transformed, originalTip, pointOnYAxis, rotate1Tip, finalTip},
    (* The tip of the principal axis *)
    originalTip = {1, 0, 0};

    (* The unit reference vector on the y-axis chosen to determine L *)
    pointOnYAxis = {0, 1, 0};

    (* determine the new position of the principal axis and
       the reference vector on the y-axis after the rotation by thetaX around
       z-axis *)
    {{rotate1Tip}, {rotate1PointOnYAxis}} =
            transform3D[ {{originalTip}, {pointOnYAxis}},
                    {rotateAroundZBy[ angle[thetaX] ]}
            ];

    (* principal axis after the object is oriented *)
    {{finalTip}} = transform3D[
                    {{rotate1Tip}},
                    {rotateAround[
                        line[   point[{0, 0, 0}],
                                point[rotate1PointOnYAxis]
                        ],
                        angle[thetaXY]
                      ]
                    }
                ];

    (* carry out the three rotations namely
        1) rotation around z by thetaX
        2) rotation around L by thetaXY
        3) rotation around the final principal axis by thetaP
            on the object *)
    {transformed} = transform3D[ {object},
        {   rotateAroundZBy[ angle[thetaX] ],
            rotateAround[
                line[ point[{0, 0, 0}], point[rotate1PointOnYAxis] ],
                angle[thetaXY] ],
            rotateAround[ line[ point[{0, 0, 0}], point[finalTip] ],
                    angle[thetaP]
            ]
```

```
        }
    ];
    transformed
]
object = cuboid[{0,0,0}, {1, 2, 3}];
Do[
    Show[   Graphics3D[{    Thickness[0.01],
                    orient[
                        {   object,
                            Line[{{0, 0, 0}, {1, 0, 0}}]
                        },
                        {thetaX, thetaXY, thetaP}
                    ]
                }
            ],
            Axes -> True, AxesLabel -> {"x", "y", "z"},
            PlotRange -> {{-5, 7}, {-5, 7}, {-7, 9}}
    ],
{thetaX, 0, 1/3 Pi, Pi/3},
{thetaXY, 0, 0 Pi, Pi/3},
{thetaP, 0, 0 Pi, Pi/3}
];
Clear[object];
```

Chapter 59

Notebook Translation

Chapter Abstract

This notebook in conjuction with many of the utility files, takes a Mathematica notebook and generates the corresponding Mathematica Package. It removes all the irrelevant information such as example sessions. In a more general setting it can be used to carry out a number of general translations. The notebook also contains various illustrative examples.

Notebook Translation

This notebook in conjuction with many of the utility files, takes a Mathematica notebook and generates the corresponding Mathematica Package.
It removes all the irrelevant information such as example sessions.
In a more general setting it can be used to carry out a number of general translations.
The notebook also contains various illustrative examples.

```
Clear[translateNotebook];
(* Takes a notebook file and a table of Translations of the form
      {outputFile, cellType, translationFunction}
   Applies translationFunction to all cells of "cellType" and stores results in
   the file "outputFile" *)
translateNotebook[notebookFile_String, translations_?MatrixQ, options___] :=
   Map[translateNotebook[notebookFile, #[[1]],
         {{#[[2]], #[[3]]}}, options]&, translations];

(* Applies translationFunction to all cells of type cellType and stores results
   in outputFile *)
translateNotebook[notebookFile_String, outputFile_String, cellType_String,
   translationFunction_, options___] /; (Head[translationFunction] =!= Rule):=
   translateNotebook[notebookFile, outputFile,
         {{cellType, translationFunction}}, options];

(* Takes a notebook file and a table of Translations of the form
      {cellType, translationFunction}
   Applies translationFunction to all cells of "cellType".
   Stores all results in one file "outputFile"
   Options[translateNotebook] = {CellSeparator, PreserveCellHeaders,
               OutputPageWidth, PreserveFontInfo, Echo, AppendToOutputFile}.

   CellSeparator (default = ""), Specify separator bw cells in output file.
   PreserveCellHeaders (default = If one cell then False else True), Have
      separate cells in output file or all merged into one.
   OutputPageWidth (default = Infinity), The word wrap width of output file.
   PreserveFontInfo (default = True), Preserve italics, bold, colors etc.
   Echo (default = False), echo results to screen too.
   AppendToOutputFile (default = False), do not delete output file if it exist
*)
translateNotebook[notebookFile_String, outputFile_String,
         translations_?MatrixQ, (* {{cellType, translationFunction}, ...} *)
         options___] :=
Module[{inFile, outFile, record, pos, cell, mmaCellHeaderPrefix,
      translatedCell, cellSeparator, validTypes, mmaCellHeaderSuffix,
      preserveCellHeaders, echo, preservedHeader, wordSeparators,
      currentCellType, cellTypes, translationFunction, appendFlag,
      outputPageWidth, fontInfoSeparator, preserveFontInfo, notebookEndMarker
   validTypes = {"clipboard", "completions", "footer", "header", "help",
               "info", "input", "leftfooter", "leftheader", "message",
               "name", "output", "postscript", "print", "section",
               "smalltext", "special1", "special2", "special3", "special4"
               "special5", "subsection", "subsubsection", "subsubtitle",
               "subtitle", "text", "title"};
   wordSeparators = {";", "]"};
   mmaCellHeaderPrefix = "\n:[font = "; mmaCellHeaderSuffix = "]\n";
   fontInfoSeparator = "\n;[s]\n"; notebookEndMarker = "\n^*)";

   cellSeparator = CellSeparator /. {options, CellSeparator -> ""};
   preserveCellHeaders = PreserveCellHeaders /. {options,
                     PreserveCellHeaders -> (Length[cellTypes] > 1)};

   outputPageWidth = OutputPageWidth /. {options, OutputPageWidth -> Infinity}
```

```
preserveFontInfo = PreserveFontInfo /. {options, PreserveFontInfo -> True};

echo = Echo /. {options, Echo -> False};
appendFlag = AppendToOutputFile /. {options, AppendToOutputFile -> False};

cellTypes = Map[First, translations];
If[Length[ Complement[cellTypes, validTypes] ] == 0,
    inFile = OpenRead[notebookFile];
    outFile = If[appendFlag,
                    OpenAppend[outputFile,
                            FormatType -> OutputForm,
                            PageWidth -> outputPageWidth],
                    OpenWrite[outputFile,
                            FormatType -> OutputForm,
                            PageWidth -> outputPageWidth]
                ];
    record = " ";
    While[record =!= EndOfFile,
        record = Read[inFile, Record,
                    RecordSeparators -> {mmaCellHeaderPrefix}];
        currentCellType = frontToken[record, wordSeparators];
        If[ MemberQ[cellTypes, currentCellType],
                pos = First[Flatten[StringPosition[record,
                        mmaCellHeaderSuffix] ] ];
                cell = StringDrop[record, pos+1];
                cell = If[preserveFontInfo,
                            cell,
                            frontToken[cell, {fontInfoSeparator,
                                                notebookEndMarker}]
                        ];
                        (* Take care of the case where the fontInfoSeparato:
                            might appear in the cell itself *)
                preservedHeader =   If[preserveCellHeaders,
                                        StringJoin[ mmaCellHeaderPrefix,
                                                StringTake[record, pos]
                                        ],
                                        ""
                                    ];
                translationFunction = Flatten[Select[translations,
                                                (#[[1]] == currentCellType)&
                                                ]
                                        ][[2]];
                translatedCell = translationFunction[cell];
                If[translatedCell != "",
                    Map[If[# != "", Write[outFile, #]]&,
                        {preservedHeader, translatedCell, cellSeparator}];
                    If[echo,
                        Map[Print, {preservedHeader, translatedCell,
                                        cellSeparator}];
                    ]
                ];
        ]
    ];
    Close[inFile]; Close[outFile];
    Null,

    (* else *)
    Print["Warning :: translateNotebook[] : called with Unknown cell Type(s
        Complement[cellTypes, validTypes]]
]
]
```

```
(* Extract all input cells from 3Doperations.ma file,
    converting all characters to upper case. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", "input", ToUpperCase,
    Echo -> False, CellSeparator -> "\n"]

(* Extract all input cells from Simulator/Simulator.ma file,
    converting all characters to lower case & removing Font Info and
    restricting output page width to 50 chars. *)
translateNotebook["OptiCAD/Packages/Simulator/Simulator.ma",
    "OptiCAD/tmp/ocg/Packages/Simulator/Simulator.m", "input", ToLowerCase,
    Echo -> False, CellSeparator -> "\n", PreserveFontInfo -> False,
    OutputPageWidth -> 50]

(* List the sizes of all input cells from 3Doperations.ma file. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", "input", StringLength,
    Echo -> False, CellSeparator -> "\n"]

(* Extract all graphics cells from 3Doperations.ma file. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", "postscript", Identity,
    Echo -> False, CellSeparator -> "\n", PreserveCellHeaders->True]

(* No such cell type called "out" *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", {{"input",ToUpperCase}, {"out", ToUpperCase}},
    Echo -> False, CellSeparator -> "\n"]

Warning :: translateNotebook[] : called with Unknown cell Type(s): {out}
```

```
(* Extract all input and output cells from 3Doperations.ma file,
    converting all characters to upper case in both kinds of cells. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", {{"input",ToUpperCase}, {"output", ToUpperCase}},
    Echo -> False, CellSeparator -> "\n"]

(* Extract all subsection cells from 3Doperations.ma file,
    printing characters in the reverse direction. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", "subsection", StringReverse,
    Echo -> True, CellSeparator -> "\n"]

(* Extract all subsection cells from 3Doperations.ma file,
    and print all words of subsection title in sorted order. *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", "subsection", Sort[tokenize[#]]&,
    Echo -> True, CellSeparator -> "\n"]

(* Produce a LaTeX specification for all the titles, subsections,
    and subsubsections from the notebook 3Doperations.ma. *)
titleTrans = {"title", StringJoin["\\title{", #, "}"]&};
subsectionTrans = {"subsection", StringJoin["\\subsection{", #, "}"]&};
subsubsectionTrans = {"subsubsection", StringJoin["\\subsubsection{", #, "}"]&};

translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
    "OptiCAD/tmp/ma2m.m", {subsectionTrans, titleTrans, subsubsectionTrans},
    Echo -> True, CellSeparator -> ""]
```

```
(* Returns cellContents if the cell contains a definition *)
Clear[definitionCell];
definitionCell[cellContents_String] :=
    If[StringMatchQ[removeQuotedStrings[cellContents], "*:=*"],
            cellContents,
            ""
    ];

definitionCell[args___]:=
    Print["Warning :: definitionCell[] : called with arguments: [", args, "]"]

definitionCell["def : body;:="]
```

```
(* Takes an input cell and returns as a string all the function heads.  *)
extractFunctionLines[input_String, options___]:=
Module[{firstFunctionName, rest, functions, headBodySeparator, functionSeparatc
    headBodySeparator = HeadBodySeparator /. {options, HeadBodySeparator -> ":=
    functionSeparator =
        FunctionSeparator /. {options, FunctionSeparator -> "\n\n"};
    If[definitionCell[input] != "",
        {firstFunctionName, rest} = split[input, {{headBodySeparator}}];
        If[rest =!= EndOfFile,
            functions = Drop[splitRepeated[rest,
                        {{functionSeparator}, {headBodySeparator}}], -1];
            functions = Prepend[functions, firstFunctionName];
            functions = Select[functions,
                    (StringTake[#, StringLength[headBodySeparator]] !=
                                                    headBodySeparator)&
                ];
            StringDrop[
                StringJoin[
                    Map[ StringJoin[#, "\n"]&, StringReplace[functions, "\n" ->
                ],
                -1
            ],
            (* else *)
            ""
        ],
        (* else *)
        ""
    ]
]
```

```
(* From an input cell, returns all function names after removing all comments,
    Clears[] and white spaces. *)
Clear[ extractFunctionNames ];
extractFunctionNames[input_String, options___]:=
Module[{functionLines, commentsRemoved,
        clearRemoved, tabsReplaced, blanksSquashed},
    functionLines = extractFunctionLines[input, options];
    commentsRemoved = removePattern[functionLines, "(*", "*)"];
    clearRemoved = removePattern[commentsRemoved, "Clear[", "];"];
    tabsReplaced = StringReplace[ clearRemoved, "\t" -> " "];
    blanksSquashed =
        StringReplace[ tabsReplaced, Table[nBlanks[i] -> " ", {i, 8, 2, -1}] ]
]

extractFunctionNames[
        "Clear[operation];\n(*Comment1*)\noperation[]:=\nPrint[]"
]

operation[]

(*  Extract all the defined function names from Simulator.ma. *)
CloseAllStreams[];
translateNotebook["OptiCAD/Packages/Simulator/Simulator.ma",
    "OptiCAD/tmp/ma2m.m", "input", extractFunctionNames,
    Echo -> False, CellSeparator -> ""]

(* Returns a .functions file name from a given notebook name. *)
Clear[notebookNameTofunctionsFileName];
notebookNameTofunctionsFileName[notebookName_String] :=
    StringJoin[StringDrop[notebookName, -3], ".functions"]


notebookNameTofunctionsFileName["string.ma"]

string.functions

(* Returns a package file name from a given notebook name. *)
Clear[notebookNameToPackageFileName];
notebookNameToPackageFileName[notebookName_String] :=
    StringDrop[notebookName, -1]

notebookNameToPackageFileName["string.ma"]

string.m

(* For a notebook.ma file, extracts all defined functions and stores them in a
    notebook.functions file. *)
Clear[createFunctionsFile];
createFunctionsFile[noteBookName_String] :=
    translateNotebook[noteBookName,
        notebookNameTofunctionsFileName[noteBookName],
        "input",
        extractFunctionNames]

createFunctionsFile["OptiCAD/Packages/Simulator/Simulator.ma"]

(* Appends to fileName a translation of contents. Translation depends on the
    Contents -> contentsType specified in options. *)
Clear[writeTo];
writeTo[fileName_, contents_, options___] :=
Module[{outFile, section},
    outFile =
        OpenAppend[fileName, FormatType -> OutputForm, PageWidth -> Infinity];
    section = Section /. {options, Section -> "Default"};
    Switch[section,
        (* For Package.m Files *)
```

```
        "PackagePreamble",
            Write[outFile,
                "(* \tCopyRight, \n\t Version 2.2\n",
                    "\t Title \n\t Author: \n\t Requirements: \n",
                    "\t Warnings: \n\t Sources:  \n\t Summary: *)"
            ],
        "PackageKeywords",
            Write[outFile, "(* \tKeywords:\n\t ", listToString[contents,
                    ", "], " *)"],
        "ContextName",
            Write[outFile, "BeginPackage[\"", listToString[contents,
                "\", \""], "\"]\n"],
        "UsageSection",
            MapThread[Write[outFile, StringJoin[#1, "::usage = \"",
                #2, ":", #3,"\"\n"]]&,
                        contents],
        "ProtectedSymbols",
            Write[outFile, "Protect[", listToString[contents, ", "], "]\n"],

        (* For Notebook.examples.ma Files *)
        "GetPackageCell",
            inputCellHeader = "\n:[font = input; preserveAspect]\n";
            Write[outFile, StringJoin[inputCellHeader,
                "Get[\"", contents, "\"]"]],

        "Default",
            Write[outFile, contents]
    ];
    Close[outFile]
]

(* Given a full pathName for notebook name, returns the Context name. *)
Clear[notebookNameToContextName];
notebookNameToContextName[notebookName_String]:=
Module[{prefix, directory, fileName},
        {directory, fileName} = splitPathName[notebookName, ".ma"];
        StringJoin[OHDL$Prefix, directory, "`", fileName, "`"]
]

notebookNameToContextName["OptiCAD/Packages/Utilities/SystemsProgramming.ma"]

(* Given a full pathName for .functions file name, returns the Context name. *)
Clear[functionsFileNameToContextName];
functionsFileNameToContextName[functionsFileName_String]:=
Module[{prefix, directory, fileName},
        {directory, fileName} = splitPathName[functionsFileName, ".functions"];
        StringJoin[OHDL$Prefix, directory, "`", fileName, "`"]
]

functionsFileNameToContextName[
        "OptiCAD/Packages/Utilities/SystemsProgramming.functions"
]
OHDLUtilities`SystemsProgramming`
```

```
(* From a notebook file, creates a Mathematica Package file. The .functions fi
    is created as a side effect. *)
Clear[createPackage];
createPackage[notebookName_String] :=
Module[{prefix, directory, fileName, packageName, contextName, functions,
            functionNames, usages},
    packageName = notebookNameToPackageFileName[notebookName];
    contextName = notebookNameToContextName[notebookName];
    writeTo[packageName, "", Section -> "PackagePreamble"];
    createFunctionsFile[notebookName];
    functions = ReadList[notebookNameTofunctionsFileName[notebookName],
        String];
    usages = Map[functionNameToUsage[#, contextName]&, functions];
    functionNames = Map[frontToken[#, {"["}]&, functions];
    writeTo[packageName, Union[functionNames], Section -> "PackageKeywords"];
    writeTo[packageName,
        dependencyContexts[contextName], Section -> "ContextName"];
    writeTo[packageName,
        {functionNames, functions, usages}, Section -> "UsageSection"];
    writeTo[packageName, "Begin[\"`Private`\"]\n"];
    translateNotebook[notebookName, packageName,
            "input", definitionCell, AppendToOutputFile -> True,
            PreserveFontInfo -> False];
    writeTo[packageName, StringJoin["\nEnd[] (* ", contextName, "Private` *)\n"
    writeTo[packageName, Union[functionNames], Section -> "ProtectedSymbols"];
    writeTo[packageName, StringJoin["\nEndPackage[] (* ", contextName, " *)\n"]
];

createPackage[args___]:=
    Print["Warning :: createPackage[] : called with arguments: [", args, "]"]
```

```
createPackage["OptiCAD/Packages/Utilities/SystemsProgramming.ma"]
```

```
(* For a notebook.ma file, returns the notebook.examples.ma file name. *)
Clear[notebookNameToExamplesFileName];
notebookNameToExamplesFileName[notebookName_String] :=
    StringJoin[StringDrop[notebookName, -3], ".examples.ma"]
```

```
notebookNameToExamplesFileName["string.ma"]
```

```
string.examples.ma
```

```
(* Returns contents of a cell that does not contain any definition. *)
Clear[nonDefinitionCells];
nonDefinitionCells[cellContents_String]:=
    If[definitionCell[cellContents] == "", cellContents, ""]
```

```
(* for a notebook.ma file, extracts all the example cells i.e. input, output pair.
    and stores them in a notebook.examples.ma file *)
Clear[createExamplesNotebook];
createExamplesNotebook[notebookName_String] :=
Module[{examplesFile},
    examplesFile = notebookNameToExamplesFileName[notebookName];
    writeTo[examplesFile, notebookNameToPackageFileName[notebookName],
            Section -> "GetPackageCell"];
    translateNotebook[notebookName,
        examplesFile,
        {{"title", Identity}, {"input", nonDefinitionCells}, {"output", Identity},
        {"postscript", Identity}},
        CellSeparator -> "", PreserveCellHeaders->True, PreserveFontInfo -> True,
        AppendToOutputFile -> True]
]
```

```
createExamplesNotebook["OptiCAD/Packages/Utilities/StringOperations.ma"]
```

```
(* For a notebook file, generates all the derived files i.e. package, examples
    and .functions files *)
Clear[GenerateDerivedFiles];
GenerateDerivedFiles[notebookName_?notebookQ]:=
Module[{},
    createPackage[notebookName];    (* createFunctionsFile[] is called from this *
    createExamplesNotebook[notebookName];
]

GenerateDerivedFiles[args___]:=
    Print["Warning :: GenerateDerivedFiles[] : called with arguments: [", args, "]

GenerateDerivedFiles["OptiCAD/Packages/Utilities/SystemsProgramming.ma"]

(* Returns all derived files (package, .functions, examples) names for a notebook
    file. *)
Clear[derivedFileNames];
derivedFileNames[notebookName_String]:=
    Map[#[notebookName]&, {notebookNameToExamplesFileName,
            notebookNameToPackageFileName,
            notebookNameTofunctionsFileName}];

derivedFileNames["OptiCAD/Packages/Utilities/StringOperations.ma"]

{"OptiCAD/Packages/Utilities/StringOperations.examples.ma",
  "OptiCAD/Packages/Utilities/StringOperations.m",
  "OptiCAD/Packages/Utilities/StringOperations.functions"}
```

```
(*  Removes all derived files (package, .functions, examples) names for a noteboo.
    file. *)
Clear[removeDerivedFiles];
removeDerivedFiles[notebookName_String]:=
    Map[If[existFileQ[#], DeleteFile]&, derivedFileNames[notebookName]]

removeDerivedFiles["OptiCAD/Packages/Utilities/StringOperations.ma"]

{Null, Null, Null}

(*  Recreates all derived files (package, .functions, examples) names for a
    notebook file. *)
reGenerateDerivedFiles[notebookName_String]:=
Module[{},
    removeDerivedFiles[notebookName];
    GenerateDerivedFiles[notebookName]
]

SetDirectory["/tmp/ocg/Packages"];
unixLikeFind[Directory[],  notebookQ, removeDerivedFiles];
unixLikeFind[Directory[],  notebookQ, GenerateDerivedFiles]

removeDerivedFiles["OptiCAD/Packages/Graphics/Graphics2D.ma"];
GenerateDerivedFiles["OptiCAD/Packages/Graphics/Graphics2D.ma"]
```

Chapter 60

Option Processing

Chapter Abstract

Contains many useful functions related to manipulating Mathematica options.

Option Processing

Contains many useful functions related to manipulating Mathematica options.

```
(* From ProgrammingExamples'FilterOptions' of Maeder *)
Clear[FilterOptions];
FilterOptions[ command_Symbol, opts___ ] :=
Module[{keywords = First /@ Options[command]},
    Sequence @@ Select[ {opts}, MemberQ[keywords, First[#]]& ]
]
```

```
FilterOptions[Graphics3D, PlotRange -> {2, 2, 2}, AnyOther -> Value]
```

```
Sequence[PlotRange -> {2, 2, 2}]
```

```
(* Pick Unique First Rules *)
Clear[PickFirst];
PickFirst[ruleLists__] :=
Module[ {l},
    l = Flatten[ruleLists];
    Map[Rule[#,#/.l]&,Union[Map[First, l]]]
]
```

```
PickFirst[{{A -> a, B -> b, X -> x}, {A -> aa, C -> cc, D -> dd}, {A -> aaa}}]
```

```
{A -> a, B -> b, C -> cc, D -> dd, X -> x}
```

```
(* Pick the last occurance of each rule *)
Clear[PickLast];
PickLast[ruleLists__] :=
Module[ {},
    PickFirst[Reverse[Map[Reverse,ruleLists]]]
]
```

```
PickLast[{{A -> a, B -> b, X -> x}, {A -> aa, C -> cc, D -> dd}, {A -> aaa}}]
```

```
{A -> aaa, B -> b, C -> cc, D -> dd, X -> x}
```

Chapter 61

Rewrite Rules

Chapter Abstract

This notebook contains some of the most widely used functions for processing re-write rules. This was created primarily to minimize the dependence on Mathematica's re-writing capabilities and confine the scope to a small group of functions. These can be reimplemented to reflect any changes if any are made in subsequent versions of Mathematica.

Rewrite Rules

This notebook contains some of the most widely used functions for processing Rewrite rules.
This was created primarily to minimize the dependence on Mathematica's rewrining capabilities and confine the scope to a small group of functions.
These can be reimplemented to reflect any changes if any are made in subsequent versions of Mathematica.

```
(*  removes the head *)
Clear[removeHead];
removeHead[head_[x___]] := x
```

```
removeHead[List[1, 2, 3]]
```

```
Sequence[1, 2, 3]
```

```
removeHead[Graphics3D[{}]]
```

```
{}
```

```
removeHead[Abcd -> abCD]
```

```
Sequence[Abcd, abCD]
```

```
(* returns the replacement pattern of attributeName as given in listOfRules *)
Clear[attributeOf];
attributeOf[ listOfRules_, attributeName_ ] :=
Module[{},
        attributeName /. listOfRules
]
```

```
attributeOf[{SoftwareName -> "Mathematica", NumberOfDisks -> 4}, NumberOfDisks]
```

```
4
```

Chapter 62

Finite Precision Comparisions

Chapter Abstract

Predicates for the relational operators == , ! = , >, < , <=, for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

Finite Precision Comparisions

Predicates for the relational operators == , != , > , < , <= ,
for the comparison of finite precision real numbers.
To have a controlled complexity over the thorny problem of
having to deal with finite precision computations.

```
tolerance = 10^-($MachinePrecision - 2);
```

```
(* Predicates for the relational operators == , != , > , < , <= ,
   for the comparison of finite precision real numbers.
   To have a controlled complexity over the thorny problem of
      having to deal with finite precision computations. *)
Clear[eqQ, neQ, gtQ, geQ, ltQ, leQ];

eqQ[x_,y_, tol_:tolerance] :=
    Module[{}, (N[x] - N[y] <= N[Abs[tol]]) && (N[y] - N[x] <= N[Abs[tol]]) ];

neQ[x_,y_, tol_:tolerance] :=
    Module[{}, (Abs[N[x] - N[y]] > N[Abs[tol]]) ];

gtQ[x_,y_, tol_:tolerance] :=
    Module[{}, (N[x] - N[y] > N[Abs[tol]]) ];

geQ[x_,y_, tol_:tolerance] :=
    gtQ[] || eqQ[];

ltQ[x_,y_, tol_:tolerance] :=
    Module[{}, (N[x] - N[y] < N[Abs[tol]]) ];

leQ[x_,y_, tol_:tolerance] :=
    Module[{}, (N[x] - N[y] <= N[Abs[tol]]) ];
```

Chapter 63

General String Manipulation Routines

Chapter Abstract

Contains several useful functions related to string operations. This rich collection is used extensively for code generating, dependency analysis etc.

General String Manipulation Routines

Contains the most useful functions related to string operations.
This rich collection is used extensively for code generating, dependency analysis etc.

```
(* Break the string into parts as specified by Options[ReadList]
    which are {NullWords -> False, TokenWords -> {}, NullRecords -> False,
    RecordLists -> False, WordSeparators -> {" ", "\t"}, RecordSeparators -> "\n".
Clear[tokenize];
tokenize[string_?StringQ, options___]:=
Module[{str, tokens},
    str = StringToStream[string];
    tokens = ReadList[str, Word, FilterOptions[ReadList, options] ];
    Close[str];
    tokens
]
```

```
tokenize["This is a test string."] //InputForm
```

```
{"This", "is", "a", "test", "string."}
```

```
(* Breaks an input string into parts specified by the separatorLists. *)
Clear[split];
split[string_?StringQ, separatorLists_]:=
Module[{str, tokens, sepLists},
    str = StringToStream[string];
    sepLists = Append[separatorLists, {}];
    tokens = Map[Read[str, Word, WordSeparators -> #, RecordSeparators -> {} ]&,
            sepLists];
    Close[str];
    tokens
]
```

```
split["a ; b : e := d = e == d", {{";"}, {":"}, {":="}}] //InputForm
```

```
{"a ", "; b ", ": e ", ":= d = e == d"}
```

```
split["a ; b : e := d = e == d", {{"=", ";"}, { "==", ":"}, {":="}}] //InputForm
```

```
{"a ", "; b ", ": e ", ":= d = e == d"}
```

```
(* Takes the first word as given by separatorList.  *)
Clear[frontToken];
frontToken[string_String, wordSeparators_]:=
    First[ split[string, {wordSeparators}] ]
```

```
frontToken["{pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to pa *)",
        {"(*"}] //InputForm
```

```
"{pa, pb}, {pa, pd}, {pa, pe}, "
```

```
(* Breaks a word into parts repeatedly until the end of word is encountered.
    The break points are specified in separatorLists. *)
Clear[splitRepeated];
splitRepeated[string_?StringQ, separatorLists_]
    /; (Length[separatorLists] > 0) :=
Module[{str, token, n, tokens, i},
    str = StringToStream[string];
    tokens = {};
    i = 0;
    n = Length[separatorLists];
    While[(token = Read[str, Word,
                WordSeparators -> separatorLists[[Mod[i, n] + 1]],
                RecordSeparators -> {} ]) =!= EndOfFile,
            tokens = Append[tokens, token];
            i = i + 1
    ];

    Close[str];
    tokens
];

splitRepeated[args___]:=
    Print["Warning :: splitRepeated[] : called with arguments: [", args, "]"]
```

```
splitRepeated["a=a;a=a:b;b=b=b;c", {{";"}}]
```

{a=a, a=a:b, b=b=b, c}

```
splitRepeated["a=a;a=a:b;b=b=b;c", {{";"}, {"="}}]
```

{a=a, ;a, =a:b, ;b, =b=b, ;c}

```
splitRepeated["OptiCAD/Packages/Graphics", {{"/"}}]
```

{OptiCAD, Packages, Graphics}

```
splitRepeated[EndOfFile, {{";"}, {"="}}]
```

Warning :: splitRepeated[] : called with arguments: [EndOfFile{{;}, {=}}]

NOTE: Because of a bug in Read[], an extra character is prepended and then removed after completing all operations.

```
(* Removes a pattern in the string.
    The pattern is enclosed between the startDelimeter and endDelimeter. *)
Clear[removePattern];
removePattern[inString_String, {startDelimeter_String, endDelimeter_String}] :=
Module[{m, n, splitStrings, reqSubStrings, string},
    string = StringJoin[" ", inString];
    n = StringLength[startDelimeter];
    m = StringLength[endDelimeter];
    splitStrings = splitRepeated[ string, {{startDelimeter}, {endDelimeter}} ];
    reqSubStrings = Select[splitStrings,
        (StringLength[#] < n) || (StringTake[#, n] != startDelimeter)&];
    StringDrop[StringJoin[
            Map[If[(StringLength[#] < m) || (StringTake[#, m] != endDelimeter),
                    #,
                    StringDrop[#, m]]&,
                reqSubStrings
            ]
        ],
        1
    ]
]

removePattern[inString_String, startDelimeter_String, endDelimeter_String] :=
    removePattern[inString, {startDelimeter, endDelimeter}]

removePattern[args___]:=
    Print["Warning :: removePattern[] : called with arguments: [", args, "]"]
```

```
removePattern["(* Apply a of the axes *)operation[angle[ theta_ ] ] ]",
"(*", "*)"] //InputForm
"operation[angle[ theta_ ] ] ]"

removePattern["Clear[operation];
(* Apply a of the axes *)
operation[angle[ theta_ ] ] ]",
"Clear[", "];"] //InputForm
"(* Apply a of the axes *)operation[angle[ theta_ ] ] ]"
```

```
(* Removes patterns from a string.
    The prefixes and suffixes of these patterns are
    given in a list of pairs. *)
Clear[removePatterns];
removePatterns[inString_String, delimeterPairs_?MatrixQ] :=
    Fold[removePattern[#1,#2]&, inString, delimeterPairs]

removePatterns[args___]:=
    Print["Warning :: removePatterns[] : called with arguments: [", args, "]"]
```

```
removePatterns[
    "Clear[operation];(* Apply a of the axes *)operation[angle[ theta_ ] ] ]",
    {{"(*", "*)"}, {"Clear[", "];"}}
]
operation[angle[ theta_ ] ] ]
```

```
(* Removes all quoted strings from a string *)
removeQuotedStrings[string_String]:=
Module[{temp},
    temp = StringJoin[ StringReplace[string, "\\\"" -> ""], "\"anyPattern"];
    StringJoin[ Map[ First, Partition[ splitRepeated[temp, {{"\""}}], 2] ] ]
]

removeQuotedStrings[
    "split[\"a ; b : e := d = e == d\", {{\";\"}, {\":\"}, {\":=\"}}]"
]

split[, {{}, {}, {}}]
```

```
(* Converts a list of objects to one string separated by sep.
    The prefix and suffix
    of the resulting string can be specified as rules.
    Options[] = {StringPrefix, StringSuffix}
    StringPrefix (default = "") is the prefix of the resulting string.
    StringSuffix (default = "") is the suffix of the resulting string.
*)
Clear[listToString];
listToString[strings__, sep_, options___] /; Length[strings] > 0 :=
Module[{stringPrefix, stringSuffix, string},
    separator = ToString[sep];
    stringPrefix = ToString[StringPrefix /. {options, StringPrefix -> ""}];
    stringSuffix = ToString[StringSuffix /. {options, StringSuffix -> ""}];

    string = StringDrop[StringJoin[
                    Map[StringJoin[ToString[#], separator]&, strings]
                ],
            -StringLength[separator]];
    StringJoin[stringPrefix, string, stringSuffix]
]

listToString[{}, _] := ""

listToString[args___]:=
    Print["Warning :: listToStrings[] : called with arguments: [", args, "]"]

listToString[{"aaa", "bbb", "ccc"}, ", "] //InputForm
"aaa, bbb, ccc"
```

```
listToString[{}, ", "] //InputForm
""

listToString[{{1, 2, 3}, "aaa", pqr}, "|",
        StringPrefix -> 999, StringSuffix -> "mm"]
999{1, 2, 3}|aaa|pqrmm

(* Generates a string of c charhs of length n. *)
Clear[nChars];
nChars[n_Integer, c_String] :=
    StringJoin[Table[c, {n}]]

nChars[12, "y"] //InputForm
"yyyyyyyyyyyy"

(* Generates a string of length n containing the <space> character. *)
Clear[nBlanks];
nBlanks[n_Integer] :=
    nChars[n, " "]

nBlanks[12] //InputForm
"            "

(* Checks whether a string has a certain token. This is different from subString
    operation. A token has specific boundaries.
    Options[hasTokenQ] = {SeparatorList} *)
Clear[hasTokenQ];
hasTokenQ[string_String, token_String, options___]:=
Module[{separatorList, augmentedString, positions, f},
    separatorList = SeparatorList /. {options, SeparatorList ->
    {"\001", "\002", "\003", "\004", "\005", "\006", "\007", "\010", "\t", "\n",
     "\013", "\f", "\r", "\016", "\017", "\020", "\021", "\022", "\023", "\024",

     "\025", "\026", "\027", "\030", "\031", "\032", "\033", "\034", "\035",
     "\036", "\037", " ", "!", "\"", "#", "$", "%", "&", "'", "(", ")", "*", "+",
     ",", "-", ".", "/", ":", ";", "<", "=", ">", "?", "@", "[", "\\", "]", "^",
     "_", "`", "{", "|", "}", "~", "\177", " ", "À", "Á", "Â", "Ã", "Ä", "Å",
     "Ç", "È", "É", "Ê", "Ë", "Ì", "Í", "Î", "Ï", "Ð", "Ñ", "Ò", "Ó", "Ô", "Õ",
     "Ö", "Ù", "Ú", "Û", "Ü", "Ý", "Þ", "µ", "×", "÷", "©", "¡", "¢", "\(Pound)",
     "/", "\(Yen)", "ƒ", "\(Section)", "¤", "\(Copyright)", "\(LogicalNegation)",
     "«", "‹", "›", "\(RegisteredTradeMark)", "fi", "®", "\(PlusMinus)", "†", "‡",
     "·", "¦", "\(Paragraph)", "\(CenterDot)", "‚", "„", "”", "»", "…", "‰", "¬",
     "¿", "¹", "`", "´", "ˆ", "¯", "˜", "˘", "˙", "¨", "²", "˚", "¸", "³", "˝",
     "˛", "ˇ", "—", "±", "¼", "½", "¾", "à", "á", "\(Cross)", "ã", "ä", "å", "ç",
     "è", "é", "ê", "ë", "ì", "Æ", "í", "ª", "î", "ï", "ð", "ñ", "Ł", "Ø", "Œ",
     "º", "ò", "ó", "ô", "õ", "ö", "æ", "ù", "ú", "û", "ı", "ü", "\(Divide)", "łł"
     "łø", "łœ", "łß", "łþ", "łÿ", "ł ", "ł "}};
    augmentedString = StringJoin[string, " "];
    positions = StringPosition[string, token, IgnoreCase -> False];
    f = Function[pos,
        MemberQ[separatorList, StringTake[augmentedString, {pos, pos}] ]];
    If[positions == {}, False,
        Or[False, Apply[Sequence, Map[f, Transpose[positions][[2]] + 1] ] ]
    ]
]

hasTokenQ["add1[];Constraints; ", "add"]
False
```

```
(* Generates all characters in the range
    \0 -> \47, \58 -> \64, \91 -> \96, \123 -> \255. *)
Clear[nonAlphaNumericASCIISet];
nonAlphaNumericASCIISet[]:=
    Map[FromCharacterCode, Join[Range[47],
            Range[58, 64], Range[91, 96], Range[123, 255]]]

(* Listable function. It puts quotes around a string. *)
Clear[quote];
quote[listOfStrings__]:=
    Map[quote, listOfStrings]

quote[string_String]:=
    StringJoin["\"", string, "\""]

quote[args___]:=
    Print["Warning :: quote[] : called with arguments: [", args, "]"]

quote["Test"] //InputForm
"\"Test\""

quote[{{"str1", "str2"}, {"str3"}}] //InputForm
{{"\"str1\"", "\"str2\""}, {"\"str3\""}}
```

```
(* Checks whether string ends with suffix.  *)
Clear[suffixQ];
suffixQ[suffix_String, string_String]:=
Module[{posList},
    If[suffix == "", True,
        posList = StringPosition[string, suffix];
        If[posList === {}, False,
            Last[ Last[ posList ] ] == StringLength[string]
        ]
    ]
]

suffixQ[args___]:=
    Print["Warning :: suffixQ[] : called with arguments: [", args, "]"]

{suffixQ[".ma", "file.ma"], suffixQ[".ma", "file.ma.mb"],
 suffixQ["", "anything"], suffixQ["file", ".ma"]}
{True, False, True, False}

suffixQ[{List}]

Warning :: suffixQ[] : called with arguments: [{List}]
```

Chapter 64

General Systems Programming Routines

Chapter Abstract

Implements some useful utilities as functions similar to those found in Unix. These were implemented to reduce dependency on the underlying operating system.

General Systems Programming Routines

Implements some useful functions similar to those found in Unix.
These were implemented to reduce dependency on the underlying operating system.

```
(* Closes all open streams except stdin, stdout and stderr. *)
Clear[CloseAllStreams];
CloseAllStreams[]:=
    Map[Close,
        Complement[Streams[],
            {InputStream["stdin", 0],
             OutputStream["stdout", 1],
             OutputStream["stderr", 2]
            }
        ]
    ]
```

```
(* find recursively descends the directory hierarchy for each
     pathname in the pathname-list, seeking files that match a
     boolean (logical) expression (predicate) and applies action
*)
unixLikeFind[startDirectory_?directoryQ, predicate_, action_, options___]:=
Module[{fileNames, filePairs, directoryPairs, subDirectories, currentDirectory},
    currentDirectory = Directory[];
    SetDirectory[startDirectory];
    fileNames = Map[StringJoin[Directory[], "/", #]&, Reverse[Sort[FileNames[]]]];
    Map[If[predicate[#], action[#]]&, fileNames];
    filePairs = Map[ {FileType[#], #}&, FileNames[] ];
    directoryPairs = Select[ filePairs,
          (#[[1]] == Directory) && (StringTake[#[[2]], 1] != ".")&];
    subDirectories = Map[Last, directoryPairs];
    Map[unixLikeFind[#, predicate, action, options]&, subDirectories];
    SetDirectory[currentDirectory]
]              (* Apply to the starting directory *)

unixLikeFind[args___]:=
    Print["Warning :: unixLikeFind[] : called with arguments: [", args, "]"]
```

```
SetDirectory["OpticAD"];
unixLikeFind[Directory[], True&, Print]
```

```
SetDirectory["OpticAD"];
unixLikeFind[Directory[], True&, {Print[Directory[], " - ", #]}&]
```

```
(* Checks whether a file is a Mathematica notebook *)
Clear[notebookQ];
notebookQ[fileName_String]:=
    (StringLength[fileName] > 3) && (StringTake[fileName, -3] == ".ma") &&
            (FileType[fileName] === File);

notebookQ[_] := False;

SetDirectory["OptiCAD/Packages/Utilities"];
notebookQ["SystemsProgramming.mb"]
False

(* Checks whether the file is a .functions file. *)
functionsFileNameQ[fileName_String]:=
    (StringLength[fileName] > 10) &&
        (StringTake[fileName, -10] == ".functions") &&
            (FileType[fileName] === File);

functionsFileNameQ[_] := False;

SetDirectory["OptiCAD"];
unixLikeFind[Directory[],  notebookQ, createPackage]
/private/tmp/ocg
```

```
(*  basename - strip filename affixes.
    Basename deletes any prefix ending in '/' and the suffix, if
     present in string, from string, and prints the result. *)
Clear[baseName];
baseName[pathName_String, suffix_:""]:=
Module[{lastComponent},
    lastComponent = Last[splitRepeated[pathName, {{"/"}}]];
    If[StringPosition[lastComponent, suffix] != {},
        StringDrop[lastComponent,  Last[StringPosition[lastComponent, suffix]] ],
        lastComponent
    ]
]

baseName["OptiCAD/Packages"]
Packages

baseName["OptiCAD/Packages/Graphics2D", "/Graphics2D"]
Packages

(* returns the base name of a file and its higher directory name. *)
Clear[splitPathName];
splitPathName[pathName_String, suffix_:""]:=
Module[{parts, dir, file},
    parts = splitRepeated[StringJoin["./", pathName], {{"/"}}];
    {dir, file} = Drop[parts, (Length[parts] -2)];
    {dir, baseName[file, suffix]}
]

splitPathName["OptiCAD/Packages/Graphics/Graphics2D"]
{Graphics, Graphics2D}
```

```
splitPathName["OptiCAD/Packages/Graphics/Graphics2D", "cs2D"]
{Graphics, Graphi}
splitPathName["Graphics2D.ma", ".ma"]
{., Graphics2D}
(* Returns the higher level directory name of a path. *)
Clear[dirName];
dirName[filePath_String]:=
Module[{augmentedPath, lastSlashPosition},
    augmentedPath = If[StringTake[filePath, {1, 1}] == "/", filePath,
                        StringJoin["./", filePath]
                    ];
    lastSlashPosition = First[Last[StringPosition[augmentedPath, "/"]]];
    StringDrop[augmentedPath, {lastSlashPosition,StringLength[augmentedPath]}]
]

{Directory[], dirName[Directory[]]}
{OptiCAD/Packages/Utilities, OptiCAD/Packages}
dirName["OptiCAD/Packages/Utilities"]
./OptiCAD/Packages
dirName["OptiCAD"]
.
(* Returns a list of OHDL attributes for a path/file. *)
Clear[OHDLFileType];
OHDLFileType[file_String]:=
Module[{fileExtensionsDatabase, record},
    fileExtensionsDatabase = {
        {".examples.ma",
            "Automatically generated Examples Mathematica Notebook",
            "CouldBeRemoved"},
        {".examples.mb",
            "Automatically generated Examples Mathematica Binary Notebook",
            "CouldBeRemoved"},
        {".ma",
            "Mathematica Notebook",
            "DoNotRemove"},
        {".mb",
            "Automatically generated Mathematica Binary Notebook",
            "DoNotRemove"},
        {".functions",
            "Automatically generated List of Mma functions",
            "CouldBeRemoved"},
        {".m",
            "Automatically generated Mma Package",
            "CouldBeRemoved"},
        {".database",
            "Database File",
            "DoNotRemove"},
        {".OHDL.index",
            "Directory Information File",
            "DoNotRemove"},
        {"", "Unknown file: ", "DoNotRemove"}
    };
    record = First[Select[fileExtensionsDatabase, suffixQ[#[[1]], file]&, 1]];
    record
]
OHDLFileType["Mathematica.ma"]
```

```
{.ma, Mathematica Notebook, DoNotRemove}

(* For a directory, find the OHDLFileStatus and store all in a file .OHDL.index *
Clear[generateDirectoryInfo];
generateDirectoryInfo[directory_String]:=
Module[{currentDirectory, files, types, sizes, ohdlTypes, fileName, outFile},
    currentDirectory = Directory[];
    SetDirectory[directory];   (* Check for failure *)
    files = Sort[FileNames[]];
    types = Map[FileType, files];
    sizes = Map[FileByteCount, files];
    ohdlTypes = Map[OHDLFileType, files];
    fileName = "./.OHDL.index";
    outFile = OpenWrite[fileName, PageWidth -> Infinity, FormatType -> OutputForm
    Map[Write[outFile, #]&, Transpose[{quote[files], types, sizes, quote[ohdlTypes
    Close[outFile];
    SetDirectory[currentDirectory];
]

generateDirectoryInfo["OptiCAD"]

(* Checks whether a path specifies a Directory. *)
Clear[directoryQ];
directoryQ[dirName_String]:=
    FileType[dirName] === Directory

(* Checks whether a path specifies a File. *)
Clear[fileQ];
fileQ[fileName_String]:=
    FileType[fileName] === File


{directoryQ["OptiCAD"], directoryQ["OptiCAD/rc"],
                fileQ["OptiCAD"], fileQ["OptiCAD/rc"]}
{True, False, False, True}

(* Checks whether a file exists. *)
Clear[existFileQ];
existFileQ[fileName_String]:=
    FileType[fileName] =!= None

existFileQ["OptiCAD/Packages"]
False

(* Calls generate directoryInfo on each file/directory in the file hierarchy. *)
Clear[generateTreeIndex];
generateTreeIndex[pathName_?directoryQ]:=
Module[{},
    unixLikeFind[ pathName, directoryQ,
        (Print["Generating Index for ", #];
        generateDirectoryInfo[#])&]
]

generateTreeIndex["OptiCAD"]
OptiCAD
```

```
(* Deletes a file if it exists by first making a copy in backupName. *)
Clear[backupAndDelete];
backupAndDelete[pathName_String, backupName_String]:=
Module[{},
    If[existFileQ[pathName],
        If[existFileQ[backupName], DeleteFile[backupName] ];
        RenameFile[pathName, backupName ]
    ]
];

backupAndDelete[pathName_String]:=
    backupAndDelete[pathName, StringJoin[pathName, ".BACKUP"] ]

(*  Renames the .OHDL.index file to .OHDL.preserved.index *)
Clear[clearDirectoryInfo];
clearDirectoryInfo[dirName_?directoryQ]:=
    backupAndDelete[StringJoin[dirName, "/.OHDL.index"],
          StringJoin[dirName, "/.OHDL.preserved.index"]
    ]

clearDirectoryInfo["OptiCAD"]

(*  Renames all the .OHDL.index files in the hierarchy to .OHDL.preserved.index *)
Clear[clearTreeIndex];
clearTreeIndex[pathName_?directoryQ]:=
        unixLikeFind[ pathName, directoryQ, clearDirectoryInfo]

clearTreeIndex["OptiCAD"]
clearTreeIndex[OptiCAD]
```

```
(* For a specified directory, reads first the .OHDL.index file. Refines or
    modifies the attributes with the .OHDL.override.index file and returns
    these modified attributes *)
Clear[getDirectoryInfo];
getDirectoryInfo[dirName_?directoryQ, options___]:=
Module[{f, index, overrideIndex, combinedIndex, refreshDirectoryInfo},
    refreshDirectoryInfo =
        RefreshDirectoryInfo /. {options, RefreshDirectoryInfo -> True};
    f = Function[record,
            First[Select[combinedIndex, (#[[1]] == record[[1]])&, 1]]];
    If[refreshDirectoryInfo, generateDirectoryInfo[dirName]];
    If[existFileQ[StringJoin[dirName, "/.OHDL.index"]],
        index = Map[ToExpression,
            ReadList[StringJoin[dirName, "/.OHDL.index"], String]],
        index = {};
        Print["Warning :: getDirectoryInfo[] : ReadList failure ", dirName]
    ];
    overrideIndex =
        If[existFileQ[StringJoin[dirName, "/.OHDL.override.index"]],
            Map[ToExpression,
                ReadList[StringJoin[dirName, "/.OHDL.override.index"], String]]
            {}
        ];

    combinedIndex = Join[overrideIndex, index];
    Map[f, index]
]
```

```
getDirectoryInfo["OptiCAD/Packages"]
{{Databases, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Examples, Directory, 512, {, Unknown file: , DoNotExpand}},
  {Geometry, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Graphics, Directory, 512, {, Unknown file: , DoNotRemove}},
  {History, Directory, 512, {, Unknown file: , DoNotExpand}},
  {Init, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Misc, Directory, 512, {, Unknown file: , DoNotExpand}},
  {.OHDL.index, File, 1184, {.OHDL.index, Directory Information File,
    DoNotRemove}}, {.OHDL.override.index, File, 1018,
   {, Unknown file: , DoNotRemove}},
  {OpticalComponents, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Optimization, Directory, 512, {, Unknown file: , DoNotRemove}},
  {README, File, 885, {, Unknown file: , DoNotRemove}},
  {README~, File, 916, {, Unknown file: , DoNotRemove}},
  {Simulator, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Templates, Directory, 512, {, Unknown file: , DoNotExpand}},
  {Translators, Directory, 512, {, Unknown file: , DoNotRemove}},
  {Utilities, Directory, 512, {, Unknown file: , DoNotRemove}}}
```

```
(* From the directory info, get the status of a particular file/directory. *)
Clear[OHDLFileStatus];
OHDLFileStatus[filePath_String]:=
Module[{rec},
    rec = Select[
                getDirectoryInfo[dirName[filePath], RefreshDirectoryInfo -> False]
                (#[[1]] == baseName[filePath])&,
                1
           ];
    If[rec === {}, None, Last[ (First[rec])[[4]] ] ]
]
```

```
OHDLFileStatus["OptiCAD/Packages/Examples"]
DoNotExpand
```

```
OHDLFileStatus["OptiCAD/Packages/Utilities"]
DoNotRemove
```

```
OHDLFileStatus["OptiCAD/Packages/Utilities/SystemsProgramming.mb"]
DoNotRemove
```

```
(* recursively descends the directory hierarchy for each
     pathname in the .OHDL.index, seeking files that match a
     boolean (logical) expression (predicate) and applies action according
    to the status of the file/directory.
 *)
 Clear[operateByIndex];
operateByIndex[startDirectory_?directoryQ, predicate_, action_, options___]:=
Module[{subDirectories, currentDirectory, dir, index, files, dirs, dirsToExpand
    currentDirectory = Directory[];
    dir = SetDirectory[startDirectory];
    If[predicate[dir], action[dir]];
    index = getDirectoryInfo[dir, RefreshDirectoryInfo -> False];
    (* Taking care of simple files *)
    files = Map[First, Select[index,
              ((#[[2]] === File) && (Last[#[[4]] ] =!= "DoNotOperate"))&] ];
    Map[If[predicate[#], action[#]]&, files];
    (* Taking care of directories *)
    dirs = Map[First, Select[index, (#[[2]] === Directory)&]];
    Map[If[predicate[#], action[#]]&, dirs];
    (* Expand allowed directories *)
    dirsToExpand =
        Map[First, Select[index,
            ((#[[2]] === Directory) && (Last[#[[4]] ] =!= "DoNotExpand"))& ]];
    subDirectories = Map[StringJoin[dir, "/", #]&, dirsToExpand];
    Map[operateByIndex[#, predicate, action, options]&, subDirectories];
    SetDirectory[currentDirectory]
]              (* Apply to the starting directory *)

operateByIndex[args___]:=
    Print["Warning :: operateByIndex[] : called with arguments: [", args, "]"]

operateByIndex["OptiCAD/Packages", True&, Print]

operateByIndex["OptiCAD/Packages",
        (fileQ[#] && (OHDLFileStatus[#] == "CouldBeRemoved"))&,
        Print[StringJoin[Directory[], " ; ", #]]&]

operateByIndex["OptiCAD/Packages",
        (fileQ[#] && (OHDLFileStatus[#] == "DoNotOperate"))&,
        Print[StringJoin[Directory[], " ; ", #]]&]
OptiCAD/Packages

operateByIndex["OptiCAD/Packages",
        (directoryQ[#] && (OHDLFileStatus[#] == "DoNotExpand"))&,
        Print[StringJoin[Directory[], " ; ", #]]&]

Warning :: getDirectoryInfo [] : ReadList failureOptiCAD
OptiCAD/PackagesExamples
OptiCAD/PackagesHistory
OptiCAD/PackagesMisc
OptiCAD/PackagesTemplates
OptiCAD/Packages

operateByIndex["OptiCAD/Packages",
        (directoryQ[#] && (OHDLFileStatus[#] != "DoNotExpand"))&,
        Print[StringJoin[Directory[], " ; ", #]]&]

Clear[cleanTree];
cleanTree[tree_?directoryQ]:=
    operateByIndex[tree, (fileQ[#] && (OHDLFileStatus[#] == "CouldBeRemoved"))&,
        Print[StringJoin[Directory[], " ; ", #]]&]

cleanTree["OptiCAD/Packages"]
```

```
(* For a specified path, do the following:
    - Generate the new tree index (.OHDL.index files)
    - Create the .functions files.
    - Generate the dependencies between files.
    - Generate the derived files.
    - regenerate the new tree index.
    *)
Clear[generateTree];
generateTree[tree_?directoryQ]:=
Module[{},

    Print["generateTree[]: About to Generate tree index of Tree ", tree];
    generateTreeIndex[tree];
    Print["generateTree[]: Generated tree index of Tree ", tree];

    Print["generateTree[]: About to Get Functions Files of Tree ", tree];

    operateByIndex[tree, (notebookQ[#] && OHDLFileStatus[#] != "CouldBeRemoved"
              (Print["Creating .functions for ", #]; createFunctionsFile[#])&
    ];
    Print["generateTree[]: Created Functions Files of Tree ", tree];

    Print["generateTree[]: About to Generate dependencies of Tree ", tree];
    obtainDependencies[tree,
        StringJoin[OHDL$PathPrefix,
                 "/Packages/Databases/FunctionDependencies.database"]
    ];
    Print["generateTree[]: Generated dependencies of Tree ", tree];
    Print["generateTree[]: About to GenerateDerivedFiles of Tree ", tree];


    operateByIndex[tree, (notebookQ[#] && OHDLFileStatus[#] != "CouldBeRemoved")&,
                 (Print["Generating for ", #];
                 removeDerivedFiles[StringJoin[Directory[], #]];
                 GenerateDerivedFiles[StringJoin[Directory[], #]])&
    ];
    Print["relax!! generateTree[]: Generated DerivedFiles of Tree ", tree];

    Print["generateTree[]: Generated DerivedFiles of Tree ", tree];
    Print["generateTree[]: About to Generate NEW tree index of Tree ", tree];
    generateTreeIndex[tree]
    Print["generateTree[]: Done ..."];
]

generateTree["OptiCAD/Packages/Utilities"]
```

To Generate Packages Load the following:

Init.ma
StringOperations.ma
ReWriteRules.ma
OptionProcessing.ma
NotebookToPackage.ma
Dependencies.ma
FunctionsToContexts.ma

```
tree = StringJoin[OHDL$PathPrefix, "/Packages"]
```

```
Print["generateTree[]: About to Generate tree index of Tree ", tree,
    " ", MemoryInUse[]];
clearDirectoryInfo[tree];
clearTreeIndex[tree];
generateTreeIndex[tree];
generateDirectoryInfo[tree];
Print["generateTree[]: Generated tree index of Tree ", tree,
    " ", MemoryInUse[]];

Print["generateTree[]: About to Get Functions Files of Tree ", tree,
    " ", MemoryInUse[]];

operateByIndex[tree, (notebookQ[#] && OHDLFileStatus[#] != "CouldBeRemoved")&,
                (Print["Creating .functions for ", #, " ", MemoryInUse[]];
                createFunctionsFile[#])&
];
Print["generateTree[]: Created Functions Files of Tree ", tree,
    " ", MemoryInUse[]];

Print["generateTree[]: About to Generate All Functions DataBase of Tree ",
    tree, " ", MemoryInUse[]];

getAllFunctionNames[tree, RegenerateAllFunctions -> True];

Print["generateTree[]: Generated All Functions DataBase of Tree ",
    tree, " ", MemoryInUse[]];


Print["generateTree[]: About to Generate dependencies of Tree ", tree];

backupAndDelete[StringJoin[OHDL$PathPrefix,
                 "/Packages/Databases/FunctionDependencies.database"]];

obtainDependencies[tree,
    StringJoin[OHDL$PathPrefix,
               "/Packages/Databases/FunctionDependencies.database"]
];
CloseAllStreams[];

Print["generateTree[]: Generated dependencies of Tree ",
        tree, " ", MemoryInUse[]];
```

```
generateTree[]: About to Generate dependencies of Tree OptiCAD/Packages
Processing dependencies for Dependencies.ma 857748
Processing dependencies for FunctionsToContexts.ma 1037928
Processing dependencies for Library.ma 1160684
Processing dependencies for Relational.ma 1185184
Processing dependencies for Units.ma 1190636
Processing dependencies for Geometry.ma 1206836
Processing dependencies for Predicates.ma 1623076
Processing dependencies for Graphics3D.ma 1684804
Processing dependencies for Misc.ma 2374504
Processing dependencies for Shapes.ma 2391804
Processing dependencies for Init.ma 2679000
Processing dependencies for draw.ma 2753076
Processing dependencies for Icons.ma 2975568
Processing dependencies for Simulate.ma 3108984
Processing dependencies for Simulator.ma 3119140
Processing dependencies for Misc.ma 3659324
Processing dependencies for NotebookToPackage.ma 3677256
Processing dependencies for TransformMma3D.ma 4136820
Processing dependencies for 3DGraphicsToLines.ma 4232924
Processing dependencies for 3DGraphicsToPoints.ma 4238376
Processing dependencies for ListManipulation.ma 4249524
Processing dependencies for OptionProcessing.ma 4255132
Processing dependencies for RealArithmetic.ma 4295900
Processing dependencies for ReWriteRules.ma 4326864
Processing dependencies for StringOperations.ma 4354524
Processing dependencies for SystemsProgramming.ma 1380108909
generateTree[]: Generated dependencies of Tree /tmp/ocg/Packages 1380769245
```

```
Print["generateTree[]: About to generateContextDependencyDatabase ",
    MemoryInUse[]];

    generateContextDependencyDatabase[];

Print["generateTree[]: generated ContextDependencyDatabase ", MemoryInUse[]];
```

```
generateTree[]: About to generateContextDependencyDatabase 1380769597
generateTree[]: generated ContextDependencyDatabase 1380823573
```

```
    Print["generateTree[]: About to GenerateDerivedFiles of Tree ", tree];
    CloseAllStreams[];
    operateByIndex[tree, (notebookQ[#] && OHDLFileStatus[#] != "CouldBeRemoved")&,
                   (Print["Generating for ", #, "\t", MemoryInUse[]];
                   removeDerivedFiles[StringJoin[Directory[], "/", #]];
                   GenerateDerivedFiles[StringJoin[Directory[], "/", #]])&
    ];
Print["relax!! generateTree[]: Generated DerivedFiles of Tree ",
        tree, " ", MemoryInUse[]];

    Print["generateTree[]: About to Generate NEW tree index of Tree ", tree,
        " ", MemoryInUse[]];
    clearDirectoryInfo[tree];
    clearTreeIndex[tree];
    generateTreeIndex[tree];
    generateDirectoryInfo[tree];
    Print["generateTree[]: Done ... ", tree, " ", MemoryInUse[]];
```

Chapter 65

Generation of the Reference Manual

Chapter Abstract

This notebook processes system related information such as function dependencies, c ontext dependencies and usage messages. It was used to generate the reference manual of OptiCAD which forms Part V of the thesis.

Generation of the Reference Manual

This notebook presents System related information such as Function Dependencies, Context Dependencies, Usage Messages.
It is serving the purpose of a reference manual.

```
OHDL$PathPrefix = "~";
```

List of All User Defined Functions in a Context.

```
functions = Get[
    StringJoin[OHDL$PathPrefix,"/Packages/Databases/AllFunctions.database"]];

functions //TableForm
```

List of All Usage::Function Messages

```
Clear[listToString];
listToString[strings__, sep_, options___] /; Length[strings] > 0 :=
Module[{stringPrefix, stringSuffix, string},
    separator = ToString[sep];
    stringPrefix = ToString[StringPrefix /. {options, StringPrefix -> ""}];
    stringSuffix = ToString[StringSuffix /. {options, StringSuffix -> ""}];

    string = StringDrop[StringJoin[
                    Map[StringJoin[ToString[#], separator]&, strings]
                ],
            -StringLength[separator]];
    StringJoin[stringPrefix, string, stringSuffix]
]

listToString[{}, _] := ""

listToString[args___]:=
    Print["Warning :: listToStrings[] : called with arguments: [", args, "]"]

usageDatabaseFile =
    StringJoin[OHDL$PathPrefix, "/Packages/Databases/FunctionUsage.database"];
rawInput = ReadList[usageDatabaseFile, Word, RecordLists -> True,
            RecordSeparators -> {"\n\n"}, WordSeparators -> {" ", "\t", "\n"}];
OHDL$FunctionUsageDatabase =
            Map[{{#[[1]], #[[2]]}, listToString[Drop[#, 2], " "]}&, rawInput];

OHDL$FunctionUsageDatabase //TableForm
```

Table of All USES Context Dependencies

```
contextDependencies = Get[
    StringJoin[OHDL$PathPrefix,"/Packages/Databases/ContextDependencies.database"
contextDependencies = Map[{First[#], Rest[#]}&, contextDependencies];

contextDependencies // TableForm

<<DiscreteMath`Master`

contextNames = Union[Flatten[contextDependencies]];

adj = Map[Flatten[Rest[#]]&, Sort[contextDependencies]]
      /. MapThread[Rule[#1, #2]&,
                   {contextNames, Range[Length[contextNames]]}]

FromAdjacencyLists[adj]

ShowGraph[adj]
```

List of All Contexts

```
contexts = Map[First, contextDependencies];
```

```
contexts // TableForm

Databases`Dependencies`
Databases`FunctionsToContexts`
Geometry`Geometry`
Geometry`Predicates`
Graphics`Graphics3D`
Graphics`Misc`
Graphics`Shapes`
OpticalComponents`draw`
OpticalComponents`Icons`
Simulator`Framework`
Simulator`Simulator`
Translators`NotebookToPackage`
Translators`TransformMma3D`
Utilities`OptionProcessing`
Utilities`RealArithmetic`
Utilities`StringOperations`
Utilities`SystemsProgramming`
```

```
Clear[itemize];
itemize[{fns__}]:=
Module[{},
    If[fns[[2]] =!= {},
        StringJoin["\n\nThe list of available functions in",
                " this context is as follows:\n",
                "\\begin{itemize}\n",
                Map[StringJoin["\\item ", #, "\n"]&, Union[fns[[2]]]],
                "\\end{itemize}\n"
        ],
        ""
    ]
]
```

```
itemize[{functions[[8]]}]
```

```
The list of available functions in this context is as follows:
\begin{itemize}
\item cone
\item cuboid
\item cylinder
\item disk
\item hologram
\item interferenceFilter
\item lensTwo
\item mirror
\item seedgraphicsobject
\item slicedCylinder
\item slicedTruncatedCone
\item splitter
\item truncatedCone
\item wavePlate
\end{itemize}
```

All Contexts TeX Generation

```
fn = OpenWrite["~/Part5/01.AllContexts/AllContexts.tex",
    FormatType -> OutputForm, PageWidth -> Infinity];
```

```
Write[fn, ColumnForm[
    Map[(
        secName = StringJoin["\section{", #, "}\n"];
        dirName = StringDrop[StringReplace[#, "`"->":"], -1];
        partNo =
            If[existFileQ[StringJoin["~/Part3/", dirName]], "Part3",
                If[existFileQ[StringJoin["~/Part4/", dirName]], "Part4", ""]
            ];
        inputFile = StringJoin["\\input ", partNo, "/", dirName, "/desc.tex\n"];
        cn = #;
        fns = itemize[Select[functions, (#[[1]] == cn)&]];
        If[partNo != "", StringJoin[secName, inputFile, fns], ""]
        )&,
        contexts
    ]
]];
```

```
Close[fn];
```

Table of All USED-IN Context Dependencies

```
f = Function[contextName,
        temp = Select[contextDependencies, MemberQ[#[[2]], contextName]&];
        If[temp === {}, {}, Map[First, temp] ]
    ];
usedInContexts = Map[{#, f[#]}&, contexts];
```

```
usedInContexts // TableForm
```

```
    functionDependenciesFileName =
        StringJoin[OHDL$PathPrefix, "/Packages/Databases/FunctionDependencies.dat
        fullList = ReadList[functionDependenciesFileName, String, RecordLists ->
        fullList = Map[StringReplace[#, {"\n" -> ""}]&, Select[Flatten[fullList],
                    (# != "")&] ];
        OHDL$FullList = Flatten[Map[ToExpression, fullList], 1];
    allFunctions =
        Union[Flatten[OHDL$FullList, 1]];
```

Generating TeX File for Context Dependencies

```
contextTabulate[contextRec_]:=
Module[{preamble, postamble, firstRow, remaining},
    preamble = "";
    postamble = "\\hline\n";
    firstRow = StringJoin[
                    contextRec[[1]],
                    " & ",
                    If[Length[contextRec[[2]]] < 1, "", First[ contextRec[[2]] ]]
                    " \\\\\n"
                ];
    remaining =
     If[Length[contextRec[[2]]] > 1,
        listToString[Map[(#)&, Rest[contextRec[[2]]] ], " \\\\\n& ",
        StringPrefix -> "& ", StringSuffix -> "\\\\\n"],
        ""
    ];

    StringJoin[preamble, firstRow, remaining, postamble]
]
```

```
contextTabulate[usedInContexts[[2]]]
```

```
Databases'FunctionsToContexts' &  \\
\hline
```

```
contextTable[contextDep_, caption_, label_]:=
Module[{preamble, postamble, body},
    preamble =
     StringJoin[
        "\\begin{table}[hbtp]\n\\begin{center}\n\\begin{tabular}{|| 1 | 1 ||}\n",
        "\\hline\n",
        "\\hline\n",
        "{\\bf Context Name} & {\\bf Context Name} \\\\\n",
        "\\hline\n",
        "\\hline\n"
     ];
    postamble =
     StringJoin[
                "\\hline\n\\end{tabular}\n\\caption{", caption,".}\n",
                "\\label{",label,"}\n\\end{center}\n\\end{table}\n"
     ];
    body = listToString[Map[contextTabulate, contextDep], ""];

    StringJoin[preamble, body, postamble]
]

Range[2, 3]

{2, 3}

tabl =
contextTable[contextDependencies[[Range[Ceiling[Length[contextDependencies]/2]]]],
    "Contexts and their Dependencies",
    "uses1"]

\begin{table}[hbtp]
\begin{center}
\begin{tabular}{|| 1 | 1 ||}

\hline
\hline
{\bf Context Name} & {\bf Context Name} \\
\hline
\hline
Databases'Dependencies' & Translators'NotebookToPackage' \\
& Utilities'StringOperations' \\
& Utilities'SystemsProgramming'\\
\hline
Databases'FunctionsToContexts' & Utilities'StringOperations' \\
\hline
Geometry'Geometry' & Geometry'Predicates' \\
& Simulator'Framework' \\
& Utilities'RealArithmetic'\\
\hline
Geometry'Predicates' & Simulator'Framework' \\
& Utilities'RealArithmetic'\\
\hline
Graphics'Graphics3D' & Geometry'Geometry' \\
& Geometry'Predicates' \\
& Simulator'Framework' \\
& Translators'TransformMma3D' \\
& Utilities'RealArithmetic'\\
\hline
Graphics'Misc' & Utilities'OptionProcessing' \\
\hline
Graphics'Shapes' & Simulator'Framework' \\
& Translators'TransformMma3D'\\
\hline
OpticalComponents'draw' & Graphics'Graphics3D' \\
& Graphics'Shapes' \\
& OpticalComponents'Icons' \\
& Simulator'Framework' \\
& Translators'TransformMma3D'\\
\hline
OpticalComponents'Icons' & Graphics'Shapes' \\
& Translators'TransformMma3D'\\
\hline
```

```
\hline
\end{tabular}
\caption{Contexts and their Dependencies.}
\label{uses1}
\end{center}
\end{table}

tab2 =
contextTable[
    contextDependencies[[
            Range[Ceiling[Length[contextDependencies]/2] + 1,
                    Length[contextDependencies]
            ]
    ]],
    "Contexts and their Dependencies (contd ...)",
    "uses2"]

tab3 =
contextTable[usedInContexts[[Range[Ceiling[Length[usedInContexts]/2]]]],
    "Contexts and their Dependents",
    "usedIn1"]

tab4 =
contextTable[usedInContexts[[Range[    Ceiling[Length[usedInContexts]/2] + 1,
                                        Length[usedInContexts]]]],
    "Contexts and their Dependents (contd ...)",
    "usedIn2"]

fn = OpenWrite["~/Part5/02.Dependencies/table1.tex",
    FormatType -> OutputForm, PageWidth -> Infinity];

Write[fn, tab1];

Close[fn];

fn = OpenWrite["~/Part5/02.Dependencies/table2.tex",
    FormatType -> OutputForm, PageWidth -> Infinity];

Write[fn, tab2];

Close[fn];

fn = OpenWrite["~/Part5/02.Dependencies/table3.tex",
    FormatType -> OutputForm, PageWidth -> Infinity];

Write[fn, tab3];

Close[fn];

fn = OpenWrite["~/Part5/02.Dependencies/table4.tex",
    FormatType -> OutputForm, PageWidth -> Infinity];

Write[fn, tab4];

Close[fn];
```

Table of All USES (Function, Context) Dependencies

```
f = Function[fp,
        temp = Select[OHDL$FullList, (#[[1]] == fp)&];
        If[temp === {}, {}, Map[#[[2]]&, temp] ]
    ];
uses = Map[{#, f[#]}&, allFunctions];

uses //TableForm
```

Table of All USED-IN (Function, Context) Dependencies

```
f = Function[fp,
        temp = Select[OHDL$FullList, (#[[2]] == fp)&];
        If[temp === {}, {}, Map[First, temp] ]
    ];
usedIn = Map[{#, f[#]}&, allFunctions];
```

```
usedIn //TableForm
```

```
Clear[functions, functionUsage, contextDependencies, contexts, f, temp,
        usedInContexts, functionDependenciesFileName, fullList,
        OHDL$FullList, allFunctions, uses, usedIn]
```

Generating the TeX Chapter for Usage Messages.

```
Clear[removeDuplicatesInColumn];
removeDuplicatesInColumn[matrix_, colNo_]:=
Module[{unique, m},
    m = matrix;
    If[Length[Transpose[matrix]] >= colNo,
        unique = Union[Transpose[m][[colNo]]];
        Table[  Flatten[Select[m, (#[[colNo]] == unique[[i]])&, 1]],
                {i, 1, Length[unique]}
        ],
        (* else *)
        m
    ]
]
```

```
removeDuplicatesInColumn[{{}}, 3]
{{}}
```

```
removeDuplicatesInColumn[{{1, 2}, {2, 3}, {4, 5}, {7, 2}, {7, 3}}, 2]
{{1, 2}, {2, 3}, {4, 5}}
```

```
removeDuplicatesInColumn[{{1, 2}, {2, 3}, {4, 5}, {7, 2}, {7, 3}}, 1]
{{1, 2}, {2, 3}, {4, 5}, {7, 2}}
```

```
removeDuplicatesInColumn[
{{"Simulator`Framework`", "boundingBoxEquations"},
    {"Simulator`Simulator`", "AddOrientationConstraint"},
    {"Simulator`Simulator`", "AddOrientationConstraint"},
    {"Simulator`Simulator`", "AddOrientationConstraint"},
    {"Simulator`Simulator`", "boundingBoxEquations"}}, 2]

{{Simulator`Simulator`, AddOrientationConstraint},
  {Simulator`Framework`, boundingBoxEquations}}
```

```
removeDuplicatesInColumn[
  {{"Geometry`Predicates`", "pointOnThePlaneQ"},
   {"Simulator`Framework`", "neQ"},
   {"Simulator`Framework`", "pointOnThePlaneQ"},
   {"Utilities`RealArithmetic`", "neQ"}}, 2]
{{Simulator`Framework`, neQ}, {Geometry`Predicates`, pointOnThePlaneQ}}
```

```
textToTeX[text_]:=
Module[{rules},
    rules = {
        ";" -> "; \\\\\n",
        "{" -> "\{",
        "$" -> "\$",
        "#" -> "\#",
        "&" -> "\&",
        "^" -> "$\uparrow$",
        "_" -> "\_",
        "}" -> "\}",
        "<" -> "$<$",
        ">" -> "$>$",
        "\\" -> "$\\backslash$",

        " " -> "\ "
    };
    StringReplace[text, rules]
]

Clear[formatUsage];
formatUsage[usageMessage_]:=
Module[{rules, metaCharsRemoved, temp1, temp2, insOuts, message},
    metaCharsRemoved = textToTeX[usageMessage];
    temp1 = tokenize[metaCharsRemoved, WordSeparators -> {"In[", "Out["}];
    message = First[temp1];
    If[Length[temp1] > 1,
        temp2 =
            Map[
                ({split[#[[1]], {":="}], split[#[[2]], {"="}]})&,
                Partition[Drop[temp1, 1], 2]

            ];
        insOuts =
            listToString[
                Map[
                    StringJoin[
                        "\\domtog{\n\\In{In[",
                        #[[1,1]],
                        ":= }\n\\Inx{",
                        StringDrop[#[[1, 2]], 2],
                        "}}\n",
                        "{\\Out{Out[",
                        #[[2,1]],
                        "= }\n\\Outx{",
                        StringDrop[#[[2, 2]], 1],
                        "}}\n{caption}\n"]&,
                    temp2
                ],
                "\n"
            ],
        insOuts = ""
    ];
    StringJoin[message, "\n\n", insOuts]
]
```

```
Clear[removeLeadingBlanks];
removeLeadingBlanks[string_String]:=
Module[{f},
    f = (If[
            (StringLength[#] >= 2) && (StringTake[#, 2] == "\\ "),
            StringDrop[#, 2],
            #
    ])&;
    FixedPoint[f, string]
];

removeLeadingBlanks[x___]:= x;

removeLeadingBlanks["\\ \\ \\ Test"] //InputForm
"Test"

removeLeadingBlanks[Hello[]]
Hello[]

Clear[formatUsage];
formatUsage[usageMessage_]:=
Module[{rules, metaCharsRemoved, temp1, temp2, insOuts, message},
    metaCharsRemoved = textToTeX[usageMessage];
    temp1 = tokenize[metaCharsRemoved,
                    RecordSeparators -> {},
                    WordSeparators -> {"In[", "Out["}
    ];
    message = First[temp1];
    If[Length[temp1] > 1,
        temp2 = Map[
                ({split[#[[1]], {":="}], split[#[[2]], {"="}]})&,

                Partition[Drop[temp1, 1], 2]
        ];
        insOuts =
            listToString[
                Map[
                    StringJoin[
                            "\\inOutPairTable{",
                            StringDrop[#[[1,1]], -1],
                            "}\n{",
                            removeLeadingBlanks[StringDrop[#[[1, 2]], 2]],
                            "}\n",
                            "{",
                            StringDrop[#[[2,1]], -1],
                            "}\n{",
                            removeLeadingBlanks[StringDrop[#[[2, 2]], 1]],
                            "}\n"
                    ]&,
                    temp2
                ],
                "\n"
            ],
        insOuts = ""
    ];
    StringJoin[message, "\n\n", insOuts]
]

msg = OHDL$FunctionUsageDatabase[[35, 2]]
Returns the slice of a cylinderical object. In[1]:= slicedCylinder[] Out[1]=\
  -Graphics-
```

```
formatUsage[msg]

Returns\ the\ slice\ of\ a\ cylinderical\ object.\

\inOutPairTable{1}
{slicedCylinder[]\ }
{1}
{-Graphics-}

counter = 1;

uniqueLabel[type_]:=
    StringJoin[type, "Label", ToString[counter++]]

Clear[tabulate];
tabulate[_, _, {}]:="";
tabulate[message_, message2_, {usedRecord_}]:=
Module[{table, lab, usedRec},
    usedRec = usedRecord;
    If[(usedRec[[2]] =!= {}) && (usedRec =!= {}),
        lab = uniqueLabel["Table"];
        StringJoin[
            "\\item[", message,":]\\ \\\\\n",
            "{\\bf ", usedRec[[1,2]], "} ", ToLowerCase[message],
                " the functions shown in Table~\\ref{",lab,"}.\n",
                "\\begin{table}[hbtp]\n\\begin{center}\n",
                "\\begin{tabular}{|| l | l ||}\n",
                "\\hline\n",
                "\\hline\n",
                "{\\bf Context Name} & {\\bf Function Name} \\\\",
                "\\hline\n",
                "\\hline\n",
                listToString[

                    Map[
                        listToString[
                                    Flatten[#],
                                    " & ",
                                    StringSuffix -> " \\\\"
                        ]&,
                        removeDuplicatesInColumn[usedRec[[2]], 2]
                    ],
                    "\n",
                    StringSuffix -> "\n"
                ],
                "\\hline\n",
                "\\hline\n",
                "\\end{tabular}\n\\caption{",
                "\\ \\ The Functions and Their Contexts ", message2,
                " {\\bf ", usedRec[[1,2]],"}.}\n",
                "\\label{",lab,"}\n\\end{center}\n\\end{table}\n"
        ],
        ""
    ]
]
```

```
Clear[formatFunction];
formatFunction[usageRecord_]:=
Module[{sep, secHead, contextName, misc1, misc2, usageMsg, fnUses, fnUsedIn},
    sep = "";
    secHead = StringJoin["\n%%%%%%%%%%%%%\n\\subsection{", textToTeX[usageRecord
    misc1 = "\\begin{description}\n";
    misc2 = "\\end{description}\n";
    contextName = StringJoin["\\item[Context Name:]\\ \\\\\n    ", usageRecord[[:
    usageMsg = StringJoin["\\item[Usage Message:]\\ \\\\\n",  formatUsage[usageR
    fnUses = tabulate["Uses", "Used by", Select[uses, (usageRecord[[1]] == #[[1]:
    fnUsedIn = tabulate["Is Used In", "Making Use of", Select[usedIn, (usageReco

    StringJoin[ secHead, sep,
                misc1, sep,
                contextName, sep,
                usageMsg, sep,
                fnUses  , sep,
                fnUsedIn, sep, misc2]
]

OHDL$FunctionUsageDatabase=
    Map[{Reverse[#[[1]]], #[[2]]}&, Sort[Map[{Reverse[#[[1]]], #[[2]]}&, OHDL$Fu
formatFunction[OHDL$FunctionUsageDatabase[[35]] ]

%%%%%%%%%%%%%
\subsection{displayObject[]}
\begin{description}
\item[Context Name:]\ \\
    Graphics'Misc'
\item[Usage Message:]\ \\
Displays\ in\ 3D\ a\ wire\ graphics\ object\ represented\ by\ an\ edge\ list\\

  Options[displayObject]\ =\ \{Functionality\}\ If\ (Functionality\ -$>$\\

  Identity)\ then\ return\ edgeList.\

\inOutPairTable{1}
{pa\ =\ \{0,\ 0,\ 1\}; \\
\ pb\ =\ \{2,\ 0,\ 1\}; \\
\ pc\ =\ \{2,\ 3,\ 1\}; \\
\ pd\ =\ \{0,\ 3,\ 1\}; \\
\ pe\ =\ \{0,\ 0,\ 0\}; \\
\ pf\ =\ \{2,\ 0,\ 0\}; \\
\ pg\ =\ \{2,\ 3,\ 0\}; \\
\ ph\ =\ \{0,\ 3,\ 0\}; \\
\ object\ =\ \{\ \{pa,\ pb\},\ \{pa,\ pd\},\ \{pa,\ pe\},\ \{pb,\ pc\},\ \{pb,\\

  pf\},\ \{pc,\ pd\},\ \{pc,\ pg\},\ \{pd,\ ph\},\ \{pe,\ pf\},\ \{pe,\ ph\},\\

  \{pf,\ pg\},\ \{pg,\ ph\}\ \}; \\
\ }
{1}
{Null\ }

\inOutPairTable{2}
{displayObject[object]\ }
{2}
{-Graphics-}
\item[Uses:]\ \\
{\bf displayObject} uses the functions shown in Table~\ref{TableLabel1}.
\begin{table}[hbtp]
\begin{center}
\begin{tabular}{|| l | l ||}
\hline
\hline
{\bf Context Name} & {\bf Function Name} \\\hline
\hline
Utilities'OptionProcessing' & FilterOptions \\
\hline
\hline
```

```
\end{tabular}
\caption{\ The Functions and Their Contexts Used by {\bf displayObject}.}
\label{TableLabel1}
\end{center}
\end{table}
\end{description}

sectionFirstChar = "";

insertSectionHead[usageRec_]:=
Module[{firstChar},
    firstChar = StringTake[ToUpperCase[usageRec[[1,2]] ], 1];
    If[firstChar != sectionFirstChar,
        sectionFirstChar = firstChar;
        StringJoin["\\newpage\n\\section{Functions Starting with ", firstChar, "}
        "\\newpage\n"
    ]
]

insertSectionHead[OHDL$FunctionUsageDatabase[[8]] ]

\newpage
\section{Functions Starting with B}

fn = OpenWrite["~/Part5/05.UsageMsg/UsageMsg.tex",
    FormatType -> OutputForm, PageWidth -> 2000];

Do[ Print[t = insertSectionHead[OHDL$FunctionUsageDatabase[[i]] ] ];
    Write[fn, t ];
    Print[i, "..", OHDL$FunctionUsageDatabase[[i, 1]]];
    Write[fn, formatFunction[OHDL$FunctionUsageDatabase[[i]]]],
    {i, 1, Length[OHDL$FunctionUsageDatabase]}]

Close[fn];
```

A System for Prototyping Optical Architectures

Volume V: Reference Manual

BY

Atif Muhammed Memon

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

December 1995

Preface

OptiCAD is a Computer Aided Design (CAD) system for designing optical architectures. It was designed and implemented to facilitate the design and verification of optical architectures.

This thesis is divided into five logical parts. The first part describes the concepts and principles involved in the design of the complete system omitting the implementation details. It is however a complete document describing critical issues, design decisions and algorithms employed.

The second part contains several optical architectures designed and described using *OptiCAD*. These have been carefully chosen so as to illustrate most of the features of *OptiCAD*. This part essentially contains the user's view of the system.

The third part contains the annotated *Mathematica* notebooks that are exclusive to *OptiCAD*. Each notebook is presented as a self contained chapter. These include issues such as component modeling, component libraries, simulator and different kinds of analyses.

A rich collection of tools/utilities is presented in Part IV as *Mathematica* notebooks. These utilities facilitated the development of *OptiCAD* system presented in Part III. In addition, these are also general utilities and hence they are expected to be useful in other domains.

Reading through Parts III and IV is by itself expected to be an illustrative and interesting exercise in appreciating the power of functional programming in general and *Mathematica* in particular. The code is not necessarily efficient. Whenever there was a choice between efficiency and readability/clarity, the latter was preferred.

Part V is a reference manual for *OptiCAD* and the *OHDL* language.

Part V

Reference Manual for OptiCAD

Chapter 66

Introduction to the Reference Manual

Chapter Abstract

The reference manual's purpose, structure and contents are presented.

66.1 Introduction

During the evolution *OptiCAD*, numerous functions were defined and supporting utilities were developed. This Chapter gives an introduction to this reference manual and explains its structure.

66.2 Purpose of the Reference Manual

The many functions developed with the multiplicity of calls make it difficult for the user to follow some of the implementation. Although online help has been provided in the form of **Usage::** messages, it is useful to know the different dependencies among functions. Knowing these simplifies the extension of the system and its implementation.

This reference manual is by no means a complete guide to the still evolving *OptiCAD* system. However, it enumerates and explains using examples the different functions implemented and the dependencies among them.

The manual consists of the three major parts presented as Chapters. The next three Sections give an overview of each Chapter.

66.3 List of All Contexts in *OptiCAD*

It was necessary to divide the implementation into different contexts. Each context is grouped together with its related functions.

66.4 Dependencies Among Contexts

Different functions in one context make use of other functions in other contexts. This creates dependencies which are documented.

66.5 Reference for User Defined Functions

The bulk of the reference manual consists of a detailed enumeration of the implemented functions at this point in time. This includes a reference to the parent context, usage message, and tables of dependencies. Examples have been included to illustrate some of the options of each function.

Chapter 67

List of All Contexts in OptiCAD

Chapter Abstract

All the contexts and their functions are presented.

67.1 Databases‘Dependencies‘

This notebook contains routines to extract dependencies from functions. These dependencies are later used (among other things) to find context dependencies in the generated packages.

The list of available functions in this context is as follows:

- functionPairs
- getAllFunctionNames
- obtainDependencies

- obtainDependenciesForFile
- selfDependencyQ
- uses

67.2 Databases‘FunctionsToContexts‘

This Notebook assumes the availability of a raw functional dependency file. It processes this information to produce a Context Dependency file. This context dependency file is later used to generate Needs[] information in the generated Packages.

The list of available functions in this context is as follows:

- dependencyContexts
- functionNameToUsage
- generateContextDependencyDatabase
- generateUsageDatabase

67.3 Geometry‘Geometry‘

This notebook contains the routines necessary for Geometrical computations. They are extensively used by Graphics3D, Simulation and several other Notebooks.

The list of available functions in this context is as follows:

- equationOfPlane
- normalize

- pointInPlane
- rayAtBoundary
- vectorAtAnAngle

67.4 Geometry`Predicates`

This notebook contains the necessary predicates for geometry. These are not only used by Geometry but by several other notebooks.

The list of available functions in this context is as follows:

- axisQ
- colinearQ
- lineQ
- pointOnThePlaneQ
- xAxisQ
- yAxisQ
- zAxisQ

67.5 Graphics`Graphics3D`

This notebook contains the necessary code for 3D graphics operations.

The list of available functions in this context is as follows:

- operationToMatrix
- orient
- orientBoundingBox
- transform3D

67.6 Graphics‘Misc‘

This notebook contains miscallenous routines for graphics. These are used by the other graphics notebooks.

The list of available functions in this context is as follows:

- displayObject

67.7 OpticalComponents‘draw‘

Each optical component has a **draw**[*self*] definition that it calls to render a 3D icon. This notebook serves as a database that provides these definitions.

The list of available functions in this context is as follows:

- draw

67.8 OpticalComponents‘Icons‘

This notebook contains the necessary code to create icons for optical components. These are created by using the generic graphics shapes provided by other notebooks.

The routines are general and the icons can be customized to a specific size, resolution and orientation.

The list of available functions in this context is as follows:

- ConvexLens
- HalfWavePlate
- laser
- PlanoConvexCylindricalLens
- QuarterWavePlate
- SEED
- SLM
- SSEED
- stand

67.9 Simulator‘Simulator‘

This is an experimental notebook containing code from various other notebooks. This notebook provides the basic framework which was later used to design the simulator template.

The list of available functions in this context is as follows:

- AddObject
- AddOrientationConstraint

- AddOrientationConstraints
- AddPlacementConstraint
- AddPlacementConstraints
- boundingBoxEquations
- ComputeOrientations
- ComputeOutputPosition
- ComputePositions
- coordinateRules
- coordinateRulesToQuadruples
- drawDriver
- GenerateArchitecture
- GetObject
- GetObjects
- handleDefaults
- initialize
- orientationOf
- positionOf
- quadruplesToCoordinateRules

- ShowArchitecture
- showState
- simulate
- ViewArchitecture

67.10 Translators‘NotebookToPackage‘

This notebook in conjuction with many of the utility files, takes a *Mathematica* notebook and generates the corresponding *Mathematica* Package. It removes all the irrelevant information such as example sessions. In a more general setting it can be used to carry out a number of general translations. The notebook also contains various illustrative examples.

The list of available functions in this context is as follows:

- createExamplesNotebook
- createFunctionsFile
- createPackage
- definitionCell
- derivedFileNames
- extractFunctionLines
- extractFunctionNames
- functionsFileNameToContextName

- GenerateDerivedFiles
- nonDefinitionCells
- notebookNameToContextName
- notebookNameToExamplesFileName
- notebookNameTofunctionsFileName
- notebookNameToPackageFileName
- reGenerateDerivedFiles
- removeDerivedFiles
- translateNotebook
- writeTo

67.11 Translators‘TransformMma3D‘

Contains generic routines for carrying out various tranformations on 3D-graphics objects in general and *Mathematica* graphics objects in particular.

The list of available functions in this context is as follows:

- transformMma3D

67.12 Utilities‘OptionProcessing‘

Contains many useful functions related to manipulating *Mathematica* options.

The list of available functions in this context is as follows:

- FilterOptions
- PickFirst
- PickLast

67.13 Utilities‘RealArithmetic‘

Predicates for the relational operators == , ! = , >, < , <=, for the comparison of finite precision real numbers. These are to control complexity of having to deal with finite precision computations.

The list of available functions in this context is as follows:

- eqQ
- geQ
- gtQ
- leQ
- ltQ
- neQ

67.14 Utilities‘StringOperations‘

Contains several useful functions related to string operations. This rich collection is used extensively for code generation, dependency analysis etc.

The list of available functions in this context is as follows:

- frontToken
- hasTokenQ
- listToString
- nBlanks
- nChars
- nonAlphaNumericASCIISet
- quote
- removePattern
- removePatterns
- removeQuotedStrings
- split
- splitRepeated
- suffixQ
- tokenize

67.15 Utilities‘SystemsProgramming‘

Implements some useful utilities as functions similar to those found in Unix. These were implemented to reduce dependency on the underlying operating system.

The list of available functions in this context is as follows:

- backupAndDelete
- cleanTree
- clearDirectoryInfo
- clearTreeIndex
- CloseAllStreams
- directoryQ
- dirName
- existFileQ
- fileQ
- functionsFileNameQ
- generateDirectoryInfo
- generateTree
- generateTreeIndex
- getDirectoryInfo
- notebookQ
- OHDLFileStatus
- OHDLFileType

- **operateByIndex**
- **unixLikeFind**

Chapter 68

Dependencies Among Contexts

Chapter Abstract

All the dependencies among contexts are tabulated.

Context Name	**Context Name**
Databases‘Dependencies‘	Translators‘NotebookToPackage‘ Utilities‘StringOperations‘ Utilities‘SystemsProgramming‘
Databases‘FunctionsToContexts‘	Utilities‘StringOperations‘
Geometry‘Geometry‘	Geometry‘Predicates‘ Simulator‘Framework‘ Utilities‘RealArithmetic‘
Geometry‘Predicates‘	Simulator‘Framework‘ Utilities‘RealArithmetic‘
Graphics‘Graphics3D‘	Geometry‘Geometry‘ Geometry‘Predicates‘ Simulator‘Framework‘ Translators‘TransformMma3D‘ Utilities‘RealArithmetic‘
Graphics‘Misc‘	Utilities‘OptionProcessing‘
Graphics‘Shapes‘	Simulator‘Framework‘ Translators‘TransformMma3D‘
OpticalComponents‘draw‘	Graphics‘Graphics3D‘ Graphics‘Shapes‘ OpticalComponents‘Icons‘ Simulator‘Framework‘ Translators‘TransformMma3D‘
OpticalComponents‘Icons‘	Graphics‘Shapes‘ Translators‘TransformMma3D‘

Table 68.1: Contexts and their Dependencies.

Context Name	Context Name
Simulator'Framework'	Geometry'Geometry' Geometry'Predicates' Graphics'Graphics3D' Graphics'Shapes' Simulator'Simulator' Utilities'OptionProcessing' Utilities'RealArithmetic' Utilities'ReWriteRules' Utilities'StringOperations'
Simulator'Simulator'	Geometry'Geometry' Graphics'Graphics3D' Graphics'Shapes' OpticalComponents'draw' Simulator'Framework' Utilities'OptionProcessing' Utilities'ReWriteRules'
Translators'NotebookToPackage'	Simulator'Framework' Utilities'RealArithmetic' Utilities'StringOperations' Utilities'SystemsProgramming'
Translators'TransformMma3D'	Graphics'Graphics3D'
Utilities'OptionProcessing'	
Utilities'RealArithmetic'	Simulator'Framework'
Utilities'StringOperations'	Utilities'OptionProcessing'
Utilities'SystemsProgramming'	Databases'Dependencies' Simulator'Framework' Translators'NotebookToPackage' Utilities'RealArithmetic' Utilities'StringOperations'

Table 68.2: Contexts and their Dependencies (contd ...).

Context Name	**Context Name**
Databases'Dependencies'	Utilities'SystemsProgramming'
Databases'FunctionsToContexts'	
Geometry'Geometry'	Graphics'Graphics3D' Simulator'Framework' Simulator'Simulator'
Geometry'Predicates'	Geometry'Geometry' Graphics'Graphics3D' Simulator'Framework'
Graphics'Graphics3D'	OpticalComponents'draw' Simulator'Framework' Simulator'Simulator' Translators'TransformMma3D'
Graphics'Misc'	
Graphics'Shapes'	OpticalComponents'draw' OpticalComponents'Icons' Simulator'Framework' Simulator'Simulator'
OpticalComponents'draw'	Simulator'Simulator'
OpticalComponents'Icons'	OpticalComponents'draw'

Table 68.3: Contexts and their Dependents.

Context Name	Context Name
Simulator'Framework'	Geometry'Geometry' Geometry'Predicates' Graphics'Graphics3D' Graphics'Shapes' OpticalComponents'draw' Simulator'Simulator' Translators'NotebookToPackage' Utilities'RealArithmetic' Utilities'SystemsProgramming'
Simulator'Simulator'	Simulator'Framework'
Translators'NotebookToPackage'	Databases'Dependencies' Utilities'SystemsProgramming'
Translators'TransformMma3D'	Graphics'Graphics3D' Graphics'Shapes' OpticalComponents'draw' OpticalComponents'Icons'
Utilities'OptionProcessing'	Graphics'Misc' Simulator'Framework' Simulator'Simulator' Utilities'StringOperations'
Utilities'RealArithmetic'	Geometry'Geometry' Geometry'Predicates' Graphics'Graphics3D' Simulator'Framework' Translators'NotebookToPackage' Utilities'SystemsProgramming'
Utilities'StringOperations'	Databases'Dependencies' Databases'FunctionsToContexts' Simulator'Framework' Translators'NotebookToPackage' Utilities'SystemsProgramming'
Utilities'SystemsProgramming'	Databases'Dependencies' Translators'NotebookToPackage'

Table 68.4: Contexts and their Dependents (contd ...).

Chapter 69

Reference for User Defined Functions

Chapter Abstract

List of all the defined functions, examples and a brief explanation of what a function does is presented.

69.1 Functions Starting with A

69.1.1 AddObject[]

Context Name:

Simulator'Simulator'

Usage Message:

User called function. Adds a component into the global component database.

Uses default attributes unless specified by user.

Is Used In:

AddObject is used in the functions shown in Table 69.1.

Context Name	Function Name
Simulator'Simulator'	GetObject

Table 69.1: The Functions and Their Contexts Making Use of **AddObject**.

69.1.2 AddOrientationConstraint[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function. Adds an orientation constraint to a global database of constraints. The various constraints supported are:

1. parallelTo[firstObject_, secondObject_],
2. perpendicularTo[firstObject_, secondObject_],
3. atAnAngle[firstObject_, angle[theta_], secondObject_],
4. relativeAngles[firstObject_, secondObject_, angles__],
5. principalAxesAtAnglesOf[firstObject_, secondObject_, angles__],
6. atAnAngleWithXAxis[object_, angle[thetaX_]],
7. atAnAngleWithYAxis[object_, angle[thetaY_]],
8. atAnAngleWithZAxis[object_, angle[thetaZ_]],
9. orientationOf[object_, angles__].

Uses:

AddOrientationConstraint uses the functions shown in Table 69.2.

Is Used In:

AddOrientationConstraint is used in the functions shown in Table 69.3.

Context Name	Function Name
Simulator'Simulator'	AddOrientationConstraint
Simulator'Simulator'	orientationOf

Table 69.2: The Functions and Their Contexts Used by **AddOrientationConstraint.**

Context Name	Function Name
Simulator'Simulator'	AddOrientationConstraint
Simulator'Simulator'	AddOrientationConstraints

Table 69.3: The Functions and Their Contexts Making Use of **AddOrientationConstraint.**

69.1.3 AddOrientationConstraints[]

Context Name:

Simulator'Simulator'

Usage Message:

Adds multiple constraints calling AddOrientationConstraint[] on each one.

Uses:

AddOrientationConstraints uses the functions shown in Table 69.4.

Context Name	Function Name
Simulator'Simulator'	AddOrientationConstraint

Table 69.4: The Functions and Their Contexts Used by **AddOrientationConstraints.**

69.1.4 AddPlacementConstraint[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function. Adds a placement constraint to a global database of constraints. The various constraints supported are (1) relativeTo[firstObject_, secondObject_, offset_] (2) absolute[object_, position_] (3) default[object_].

Uses:

AddPlacementConstraint uses the functions shown in Table 69.5.

Context Name	Function Name
Simulator`Simulator`	AddPlacementConstraint

Table 69.5: The Functions and Their Contexts Used by **AddPlacementConstraint.**

Is Used In:

AddPlacementConstraint is used in the functions shown in Table 69.6.

Context Name	Function Name
Simulator`Simulator`	AddPlacementConstraint
Simulator`Simulator`	AddPlacementConstraints

Table 69.6: The Functions and Their Contexts Making Use of **AddPlacementConstraint.**

69.1.5 AddPlacementConstraints[]

Context Name:

Simulator'Simulator'

Usage Message:

Adds multiple constraints calling AddPlacementConstraint[] on each one.

Uses:

AddPlacementConstraints uses the functions shown in Table 69.7.

Context Name	Function Name
Simulator'Simulator'	AddPlacementConstraint

Table 69.7: The Functions and Their Contexts Used by **AddPlacementConstraints.**

69.1.6 attributeOf[]

Context Name:

Utilities‘ReWriteRules‘

Usage Message:

Returns the replacement pattern of attributeName as given in listOfRules.

```
In[1]:= attributeOf[{SoftwareName -> "Mathematica", NumberOfDisks ->
        4}, NumberOfDisks]

Out[1]= 4
```

Is Used In:

attributeOf is used in the functions shown in Table 69.8.

Context Name	Function Name
Simulator‘Framework‘	boundingBoxEquations

Table 69.8: The Functions and Their Contexts Making Use of **attributeOf**.

69.1.7 axisQ[]

Context Name:

Geometry'Predicates'

Usage Message:

Checks whether the given line is the same as one of the principal axes.

```
In[1]:= axisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]

Out[1]= False

In[2]:= axisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]

Out[2]= True
```

Is Used In:

axisQ is used in the functions shown in Table 69.9.

Context Name	Function Name
Graphics'Graphics3D'	operationToMatrix

Table 69.9: The Functions and Their Contexts Making Use of **axisQ**.

69.2 Functions Starting with B

69.2.1 backupAndDelete[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Deletes a file if it exists by first making a copy in backupName.

Uses:

backupAndDelete uses the functions shown in Table 69.10.

Context Name	Function Name
Utilities'SystemsProgramming'	backupAndDelete
Utilities'SystemsProgramming'	existFileQ
Simulator'Framework'	leQ

Table 69.10: The Functions and Their Contexts Used by **backupAndDelete.**

Is Used In:

backupAndDelete is used in the functions shown in Table 69.11.

Context Name	Function Name
Utilities'SystemsProgramming'	backupAndDelete
Utilities'SystemsProgramming'	clearDirectoryInfo

Table 69.11: The Functions and Their Contexts Making Use of **backupAndDelete.**

69.2.2 baseName[]

Context Name:

Utilities`SystemsProgramming`

Usage Message:

Basename deletes any prefix ending in '/' and the suffix (if present) from the input string, and prints the result.

```
In[1]:= baseName["OptiCAD/ocg"]

Out[1]= ocg

In[2]:= baseName["OptiCAD/Graphics.Icons", ".Icons"]

Out[2]= Graphics
```

69.2.3 boundingBoxEquations[]

Context Name:

Simulator'Simulator'

Usage Message:

Returns a set of six equations each representing one side of the bounding box of a specific object.

Uses:

boundingBoxEquations uses the functions shown in Table 69.12.

Context Name	Function Name
Simulator'Framework'	attributeOf
Graphics'Shapes'	cuboid
Geometry'Geometry'	equationOfPlane
Graphics'Graphics3D'	orient
Simulator'Simulator'	orientationOf
Simulator'Simulator'	positionOf

Table 69.12: The Functions and Their Contexts Used by **boundingBoxEquations.**

Is Used In:

boundingBoxEquations is used in the functions shown in Table 69.13.

Context Name	Function Name
Simulator'Framework'	ComputeOutputPosition

Table 69.13: The Functions and Their Contexts Making Use of **boundingBoxEquations.**

69.3 Functions Starting with C

69.3.1 cleanTree[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Removes any derived files from the directory tree.

Uses:

cleanTree uses the functions shown in Table 69.14.

Context Name	Function Name
Utilities‘SystemsProgramming‘	fileQ
Simulator‘Framework‘	leQ
Utilities‘SystemsProgramming‘	OHDLFileStatus
Utilities‘SystemsProgramming‘	operateByIndex
Utilities‘SystemsProgramming‘	operateByIndex

Table 69.14: The Functions and Their Contexts Used by **cleanTree**.

69.3.2 clearDirectoryInfo[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Renames the .OHDL.index file to .OHDL.preserved.index

Uses:

clearDirectoryInfo uses the functions shown in Table 69.15.

Context Name	Function Name
Utilities'SystemsProgramming'	backupAndDelete
Utilities'SystemsProgramming'	dirName

Table 69.15: The Functions and Their Contexts Used by **clearDirectoryInfo**.

Is Used In:

clearDirectoryInfo is used in the functions shown in Table 69.16.

Context Name	Function Name
Utilities'SystemsProgramming'	clearTreeIndex

Table 69.16: The Functions and Their Contexts Making Use of **clearDirectoryInfo**.

69.3.3 clearTreeIndex[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Renames all the .OHDL.index files in the hierarchy to .OHDL.preserved.index.

Uses:

clearTreeIndex uses the functions shown in Table 69.17.

Context Name	Function Name
Utilities‘SystemsProgramming‘	clearDirectoryInfo
Utilities‘SystemsProgramming‘	directoryQ
Utilities‘SystemsProgramming‘	unixLikeFind

Table 69.17: The Functions and Their Contexts Used by **clearTreeIndex**.

69.3.4 CloseAllStreams[]

Context Name:

Utilities`SystemsProgramming`

Usage Message:

Closes all open streams except stdin, stdout and stderr.

69.3.5 colinearQ[]

Context Name:

Geometry`Predicates`

Usage Message:

Checks whether a set of points is colinear. NOTE: Not yet implemented. Current status = (Always returns False).

69.3.6 ComputeOrientations[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function. From the given orientation constraints, computes the orientation of each component in the database.

69.3.7 ComputeOutputPosition[]

Context Name:

Simulator`Simulator`

Usage Message:

Given an object and a message, it computes the coordinates of the output message.

Uses:

ComputeOutputPosition uses the functions shown in Table 69.18.

Context Name	Function Name
Simulator`Simulator`	boundingBoxEquations

Table 69.18: The Functions and Their Contexts Used by **ComputeOutputPosition.**

Is Used In:

ComputeOutputPosition is used in the functions shown in Table 69.19.

Context Name	Function Name
Simulator`Framework`	performAction
Simulator`Simulator`	simulate

Table 69.19: The Functions and Their Contexts Making Use of **ComputeOutput-Position.**

69.3.8 ComputePositions[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function. From the given position constraints, computes the position of each component in the database.

Uses:

ComputePositions uses the functions shown in Table 69.20.

Context Name	Function Name
Simulator`Simulator`	handleDefaults

Table 69.20: The Functions and Their Contexts Used by **ComputePositions**.

69.3.9 cone[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns a cone using Mathematica Graphics primitives.

```
In[1]:= cone[]

Out[1]= -Graphics-
```

Uses:

cone uses the functions shown in Table 69.21.

Context Name	Function Name
Graphics'Shapes'	truncatedCone

Table 69.21: The Functions and Their Contexts Used by **cone**.

69.3.10 ConvexLens[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a ConvexLens constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[ConvexLens[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

ConvexLens uses the functions shown in Table 69.22.

Context Name	Function Name
Graphics'Shapes'	lensTwo

Table 69.22: The Functions and Their Contexts Used by **ConvexLens.**

Is Used In:

ConvexLens is used in the functions shown in Table 69.23.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.23: The Functions and Their Contexts Making Use of **ConvexLens.**

69.3.11 coordinateRules[]

Context Name:

Simulator`Simulator`

Usage Message:

Returns the coordinate rules of an axis.

69.3.12 coordinateRulesToQuadruples[]

Context Name:

Simulator`Simulator`

Usage Message:

Transforms the coordinate rules to anther representation of quarruples.

Is Used In:

coordinateRulesToQuadruples is used in the functions shown in Table 69.24.

Context Name	Function Name
Simulator`Simulator`	handleDefaults

Table 69.24: The Functions and Their Contexts Making Use of **coordinateRulesToQuadruples**.

69.3.13 createExamplesNotebook[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

For a notebook.ma file, extracts all the example cells i.e., input, output pairs and stores them in a notebook.examples.ma file.

Uses:

createExamplesNotebook uses the functions shown in Table 69.25.

Context Name	Function Name
Translators‘NotebookToPackage‘	nonDefinitionCells
Translators‘NotebookToPackage‘	notebookNameToExamplesFileName
Translators‘NotebookToPackage‘	notebookNameToPackageFileName
Translators‘NotebookToPackage‘	translateNotebook
Translators‘NotebookToPackage‘	writeTo

Table 69.25: The Functions and Their Contexts Used by **createExamplesNotebook.**

Is Used In:

createExamplesNotebook is used in the functions shown in Table 69.26.

Context Name	Function Name
Translators‘NotebookToPackage‘	GenerateDerivedFiles

Table 69.26: The Functions and Their Contexts Making Use of **createExamplesNotebook.**

69.3.14 createFunctionsFile[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

For a notebook.ma file, extracts all defined functions and stores them in a notebook.functions file.

Uses:

createFunctionsFile uses the functions shown in Table 69.27.

Context Name	Function Name
Translators‘NotebookToPackage‘	extractFunctionNames
Translators‘NotebookToPackage‘	notebookNameTofunctionsFileName
Translators‘NotebookToPackage‘	translateNotebook

Table 69.27: The Functions and Their Contexts Used by **createFunctionsFile**.

Is Used In:

createFunctionsFile is used in the functions shown in Table 69.28.

Context Name	Function Name
Utilities‘SystemsProgramming‘	generateTree

Table 69.28: The Functions and Their Contexts Making Use of **createFunctionsFile**.

69.3.15 createPackage[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

From a notebook file, creates a Mathematica Package file. The *.functions file is created as a side effect.

Uses:

createPackage uses the functions shown in Table 69.29.

Context Name	Function Name
Translators'NotebookToPackage'	createPackage
Translators'NotebookToPackage'	notebookNameToContextName
Translators'NotebookToPackage'	notebookNameToPackageFileName
Translators'NotebookToPackage'	writeTo

Table 69.29: The Functions and Their Contexts Used by **createPackage**.

Is Used In:

createPackage is used in the functions shown in Table 69.30.

Context Name	Function Name
Translators'NotebookToPackage'	createPackage
Translators'NotebookToPackage'	GenerateDerivedFiles

Table 69.30: The Functions and Their Contexts Making Use of **createPackage**.

69.3.16 cuboid[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns a cuboid using Mathematica Graphics primitives.

```
In[1]:= cuboid[]

Out[1]= -Graphics-
```

Uses:

cuboid uses the functions shown in Table 69.31.

Context Name	Function Name
Graphics‘Shapes‘	cuboid

Table 69.31: The Functions and Their Contexts Used by **cuboid**.

Is Used In:

cuboid is used in the functions shown in Table 69.32.

Context Name	Function Name
Simulator‘Framework‘	boundingBoxEquations
Graphics‘Shapes‘	cuboid
OpticalComponents‘draw‘	draw
Graphics‘Shapes‘	interferenceFilter
Graphics‘Shapes‘	seedgraphicsobject

Table 69.32: The Functions and Their Contexts Making Use of **cuboid**.

69.3.17 cuboidtolines[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns a wire-frame for the given cubiod.

69.3.18 cylinder[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns a cylinder using Mathematica Graphics primitives.

In[1]:=
```
cylinder[]
```

Out[1]=
```
-Graphics-
```

Uses:

cylinder uses the functions shown in Table 69.33.

Context Name	Function Name
Graphics‘Shapes‘	truncatedCone

Table 69.33: The Functions and Their Contexts Used by **cylinder**.

Is Used In:

cylinder is used in the functions shown in Table 69.34.

Context Name	Function Name
OpticalComponents‘Icons‘	laser
OpticalComponents‘Icons‘	stand

Table 69.34: The Functions and Their Contexts Making Use of **cylinder**.

69.4 Functions Starting with D

69.4.1 definitionCell[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Returns cellContents if the cell contains a definition.

Uses:

definitionCell uses the functions shown in Table 69.35.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell
Utilities'StringOperations'	removeQuotedStrings

Table 69.35: The Functions and Their Contexts Used by **definitionCell.**

Is Used In:

definitionCell is used in the functions shown in Table 69.36.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell
Translators'NotebookToPackage'	extractFunctionLines
Databases'Dependencies'	inputCellToPairs
Translators'NotebookToPackage'	nonDefinitionCells

Table 69.36: The Functions and Their Contexts Making Use of **definitionCell.**

69.4.2 dependencyContexts[]

Context Name:

Databases'FunctionsToContexts'

Usage Message:

Returns a list of all uses contexts for a specifed context. If the global variable OHDL$ContextDependencies is empty, this function will regenerate it.

In[1]:=
```
dependencyContexts["OHDLUtilities'SystemsProgramming'"]
```

Out[1]=
{*OHDLUtilities'SystemsProgramming'*, *OHDL-Databases'Dependencies'*, *OHDLTranslators'NotebookToPackage'*, *OHD-LUtilities'RealArithmetic'*, *OHDLUtilities'StringOperations'*}

In[2]:=
```
dependencyContexts["OHDLDatabases'Units'"]
```

Out[2]=
Warning::dependencyContexts[]: OHDLDatabases'Units' is Null. Returning Identity. {*OHDLDatabases'Units'*}

Uses:

dependencyContexts uses the functions shown in Table 69.37.

Context Name	Function Name
Databases'FunctionsToContexts'	generateContextDependencyDatabase

Table 69.37: The Functions and Their Contexts Used by **dependencyContexts**.

69.4.3 derivedFileNames[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Returns all derived files (package, .functions, examples) names for a notebook file.

In[1]:=

```
derivedFileNames["OptiCAD/Utilities/StringOperations.ma"]
```

Out[1]=

```
{OptiCAD/Utilities/StringOperations.examples.ma,
OptiCAD/Utilities/StringOperations.m,
OptiCAD/Utilities/StringOperations.functions}
```

Uses:

derivedFileNames uses the functions shown in Table 69.38.

Context Name	Function Name
Translators'NotebookToPackage'	notebookNameToExamplesFileName
Translators'NotebookToPackage'	notebookNameTofunctionsFileName
Translators'NotebookToPackage'	notebookNameToPackageFileName

Table 69.38: The Functions and Their Contexts Used by **derivedFileNames**.

Is Used In:

derivedFileNames is used in the functions shown in Table 69.39.

Context Name	Function Name
Translators'NotebookToPackage'	removeDerivedFiles

Table 69.39: The Functions and Their Contexts Making Use of **derivedFileNames**.

69.4.4 directoryQ[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Checks whether a path specifies a Directory.

Uses:

directoryQ uses the functions shown in Table 69.40.

Context Name	Function Name
Utilities'SystemsProgramming'	dirName

Table 69.40: The Functions and Their Contexts Used by **directoryQ**.

Is Used In:

directoryQ is used in the functions shown in Table 69.41.

Context Name	Function Name
Utilities'SystemsProgramming'	clearTreeIndex
Utilities'SystemsProgramming'	generateTreeIndex

Table 69.41: The Functions and Their Contexts Making Use of **directoryQ**.

69.4.5 dirName[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Returns the higher level directory name of a path.

In[1]:=
```
{Directory[], dirName[Directory[]]}
```

Out[1]=
```
{OptiCAD/Packages, OptiCAD}
```

In[2]:=
```
dirName["OptiCAD/Packages/Utilities"]
```

Out[2]=
```
./OptiCAD/Packages
```

In[3]:=
```
dirName["ocg"]
```

Out[3]=
```
.
```

Is Used In:

dirName is used in the functions shown in Table 69.42.

Context Name	Function Name
Utilities‘SystemsProgramming‘	clearDirectoryInfo
Utilities‘SystemsProgramming‘	directoryQ
Utilities‘SystemsProgramming‘	getDirectoryInfo
Utilities‘SystemsProgramming‘	OHDLFileStatus

Table 69.42: The Functions and Their Contexts Making Use of **dirName**.

69.4.6 disk[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns a disk using Mathematica Graphics primitives.

In[1]:=
```
disk[]
```

Out[1]=
```
-Graphics-
```

Is Used In:

disk is used in the functions shown in Table 69.43.

Context Name	Function Name
Graphics'Shapes'	truncatedCone
Graphics'Shapes'	wavePlate

Table 69.43: The Functions and Their Contexts Making Use of **disk**.

69.4.7 displayObject[]

Context Name:

Graphics'Misc'

Usage Message:

Displays in 3D a wire graphics object represented by an edge list.

Options[displayObject] = {Functionality} If (Functionality -> Identity) then return edgeList.

In[1]:=
```
pa = {0, 0, 1};
pb = {2, 0, 1};
pc = {2, 3, 1};
pd = {0, 3, 1};
pe = {0, 0, 0};
pf = {2, 0, 0};
pg = {2, 3, 0};
ph = {0, 3, 0};
object = { {pa, pb}, {pa, pd}, {pa, pe}, {pb, pc}, {pb, pf},
{pc, pd}, {pc, pg}, {pd, ph}, {pe, pf}, {pe, ph}, {pf, pg},
{pg, ph} };
```

Out[1]=
```
Null
```

In[2]:=
```
displayObject[object]
```

Out[2]=
```
-Graphics-
```

Uses:

displayObject uses the functions shown in Table 69.44.

Context Name	Function Name
Utilities'OptionProcessing'	FilterOptions

Table 69.44: The Functions and Their Contexts Used by **displayObject**.

69.4.8 draw[]

Context Name:

OpticalComponents‘draw‘

Usage Message:

Draw Self Definitions for Library Components.

Uses:

draw uses the functions shown in Table 69.45.

Context Name	Function Name
OpticalComponents‘Icons‘	ConvexLens
Graphics‘Shapes‘	cuboid
OpticalComponents‘draw‘	draw
OpticalComponents‘Icons‘	HalfWavePlate
Graphics‘Shapes‘	hologram
Graphics‘Shapes‘	interferenceFilter
OpticalComponents‘Icons‘	laser
Graphics‘Shapes‘	mirror
Graphics‘Graphics3D‘	orient
OpticalComponents‘Icons‘	PlanoConvexCylindricalLens
OpticalComponents‘Icons‘	SEED
OpticalComponents‘Icons‘	SLM
Graphics‘Shapes‘	splitter
OpticalComponents‘Icons‘	SSEED
Translators‘TransformMma3D‘	transformMma3D

Table 69.45: The Functions and Their Contexts Used by **draw**.

Is Used In:

draw is used in the functions shown in Table 69.46.

Context Name	Function Name
OpticalComponents‘draw‘	draw
Simulator‘Simulator‘	drawDriver

Table 69.46: The Functions and Their Contexts Making Use of **draw**.

69.4.9 drawDriver[]

Context Name:

Simulator‘Simulator‘

Usage Message:

The main driver that calls component specific draw[self].

Uses:

drawDriver uses the functions shown in Table 69.47.

Context Name	Function Name
OpticalComponents‘draw‘	draw

Table 69.47: The Functions and Their Contexts Used by **drawDriver**.

Is Used In:

drawDriver is used in the functions shown in Table 69.48.

Context Name	Function Name
Simulator‘Simulator‘	GenerateArchitecture

Table 69.48: The Functions and Their Contexts Making Use of **drawDriver**.

69.5 Functions Starting with E

69.5.1 edgesToVertices[]

Context Name:

Translators‘Misc‘

Usage Message:

Takes an object represented as edges and extracts the vertices.

69.5.2 ellipsoid[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns the shape of an ellipsoid using Mathematica Graphics primitives.

In[1]:=
```
ellipsoid[]
```

Out[1]=
```
-Graphics-
```

69.5.3 equationOfPlane[]

Context Name:

Geometry'Geometry'

Usage Message:

Given a set of points (at least three), it returns an equation of the plane on which those points lie.

```
In[1]:= equationOfPlane[{{0, 1, 0}, {1, 0, 0}, {9, 9, 9}}]

Out[1]= 1. - 1. x - 1. y + 1.88889 z == 0.
```

Uses:

equationOfPlane uses the functions shown in Table 69.49.

Context Name	Function Name
Simulator'Framework'	neQ
Geometry'Predicates'	pointOnThePlaneQ

Table 69.49: The Functions and Their Contexts Used by **equationOfPlane**.

Is Used In:

equationOfPlane is used in the functions shown in Table 69.50.

Context Name	Function Name
Simulator'Framework'	boundingBoxEquations
Simulator'Framework'	pointOfIntersection

Table 69.50: The Functions and Their Contexts Making Use of **equationOfPlane**.

69.5.4 existFileQ[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Checks whether a file exists in the current directory.

Is Used In:

existFileQ is used in the functions shown in Table 69.51.

Context Name	Function Name
Utilities'SystemsProgramming'	backupAndDelete
Utilities'SystemsProgramming'	getDirectoryInfo
Translators'NotebookToPackage'	removeDerivedFiles

Table 69.51: The Functions and Their Contexts Making Use of **existFileQ**.

69.5.5 extractFunctionLines[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Takes an input cell and returns as a string all the function names defined in that cell.

Uses:

extractFunctionLines uses the functions shown in Table 69.52.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell

Table 69.52: The Functions and Their Contexts Used by **extractFunctionLines.**

Is Used In:

extractFunctionLines is used in the functions shown in Table 69.53.

Context Name	Function Name
Translators'NotebookToPackage'	extractFunctionNames

Table 69.53: The Functions and Their Contexts Making Use of **extractFunction-Lines.**

69.5.6 extractFunctionNames[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

From an input cell, returns all function names after removing all comments, Clears[] and white spaces.

In[1]:=
```
extractFunctionNames["Clear[operation];
\n (*Comment1*)\n operation[]:=\n Print[](*Comment2*)"]
```

Out[1]=
```
operation[]
```

Uses:

extractFunctionNames uses the functions shown in Table 69.54.

Context Name	Function Name
Translators'NotebookToPackage'	extractFunctionLines
Utilities'StringOperations'	nBlanks
Utilities'StringOperations'	removePattern

Table 69.54: The Functions and Their Contexts Used by **extractFunctionNames.**

Is Used In:

extractFunctionNames is used in the functions shown in Table 69.55.

Context Name	Function Name
Translators'NotebookToPackage'	createFunctionsFile

Table 69.55: The Functions and Their Contexts Making Use of **extractFunction-Names.**

69.6 Functions Starting with F

69.6.1 fileQ[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Checks whether a path specifies a valid file.

Is Used In:

fileQ is used in the functions shown in Table 69.56.

Context Name	Function Name
Utilities‘SystemsProgramming‘	cleanTree

Table 69.56: The Functions and Their Contexts Making Use of **fileQ**.

69.6.2 FilterOptions[]

Context Name:

Utilities'OptionProcessing'

Usage Message:

From ProgrammingExamples'FilterOptions' of Maeder.

```
In[1]:= FilterOptions[Graphics3D, PlotRange -> {2, 2, 2}, AnyOther ->
        Value]

Out[1]= Sequence[PlotRange -> {2, 2, 2}]
```

Is Used In:

FilterOptions is used in the functions shown in Table 69.57.

Context Name	Function Name
Graphics'Misc'	displayObject
Utilities'StringOperations'	tokenize
Simulator'Simulator'	ViewArchitecture

Table 69.57: The Functions and Their Contexts Making Use of **FilterOptions**.

69.6.3 frontToken[]

Context Name:

Utilities'StringOperations'

Usage Message:

Takes the first word as given by the separatorList.

```
In[1]:= frontToken["{pa, pb}, {pa, pd}, {pa, pe}, (* Those adjacent to
pa *)", {"(*"}]

Out[1]= "{pa, pb}, {pa, pd}, {pa, pe}, "
```

Uses:

frontToken uses the functions shown in Table 69.58.

Context Name	Function Name
Utilities'StringOperations'	split

Table 69.58: The Functions and Their Contexts Used by **frontToken**.

Is Used In:

frontToken is used in the functions shown in Table 69.59.

Context Name	Function Name
Databases'FunctionsToContexts'	functionNameToUsage
Databases'Dependencies'	functionPairs
Databases'Dependencies'	selfDependencyQ

Table 69.59: The Functions and Their Contexts Making Use of **frontToken**.

69.6.4 functionNameToUsage[]

Context Name:

Databases‘FunctionsToContexts‘

Usage Message:

For a given contextName and functionName, returns a Usage message from the database. If the usage message is not in the database, the reverse of the function name is returned.

In[1]:=
```
functionNameToUsage["unixLikeFind[]",
"Utilities‘SystemsProgramming‘"]
```

Out[1]= *Usage Message for unixLikeFind*

In[2]:=
```
functionNameToUsage["find[args]", "Utilities‘SystemsProgramming‘"]
```

Out[2]= *dnif*

Uses:

functionNameToUsage uses the functions shown in Table 69.60.

Context Name	Function Name
Utilities‘StringOperations‘	frontToken
Databases‘FunctionsToContexts‘	generateUsageDatabase

Table 69.60: The Functions and Their Contexts Used by **functionNameToUsage**.

69.6.5 functionPairs[]

Context Name:

Databases'Dependencies'

Usage Message:

Reads a specified .functions file and returns a structure of the form contextName, listOfFunctionsInFile.

In[1]:=
```
functionPairs["OptiCAD/Packages/Databases/Dependencies.functions"]
```

Out[1]=
```
{Databases'Dependencies', {functionPairs, getAllFunctionNames, inputCellToPairs, obtainDependencies, obtainDependenciesForFile, selfDependencyQ, uses}}
```

In[2]:=
```
functionPairs["OptiCAD/Packages/Simulator/Simulator.functions"]
```

Out[2]=
```
{Simulator'Simulator', {AddObject, AddOrientationConstraint, AddOrientationConstraints, AddPlacementConstraint, AddPlacementConstraints, boundingBoxEquations, ComputeOrientations, ComputeOutputPosition, ComputePositions, coordinateRules, coordinateRulesToQuadruples, drawDriver, GenerateArchitecture, GetObject, GetObjects, handleDefaults, initialize, orient, orientationOf, orientBoundingBox, positionOf, quadruplesToCoordinateRules, ShowArchitecture, showState, simulate, ViewArchitecture}}
```

Uses:

functionPairs uses the functions shown in Table 69.61.

Context Name	Function Name
Utilities'StringOperations'	frontToken
Translators'NotebookToPackage'	functionsFileNameToContextName

Table 69.61: The Functions and Their Contexts Used by **functionPairs**.

69.6.6 functionsFileNameQ[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Checks whether the file is a .functions file.

Is Used In:

functionsFileNameQ is used in the functions shown in Table 69.62.

Context Name	Function Name
Databases‘Dependencies‘	getAllFunctionNames

Table 69.62: The Functions and Their Contexts Making Use of **functionsFileNameQ.**

69.6.7 functionsFileNameToContextName[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Given a full pathName for *.functions file name, returns the Context name.

In[1]:=
```
functionsFileNameToContextName[
"OptiCAD/Utilities/SystemsProgramming.functions" ]
```

Out[1]=
```
OHDLUtilities'SystemsProgramming'
```

Is Used In:

functionsFileNameToContextName is used in the functions shown in Table 69.63.

Context Name	Function Name
Databases'Dependencies'	functionPairs

Table 69.63: The Functions and Their Contexts Making Use of **functionsFileNameToContextName**.

69.7 Functions Starting with G

69.7.1 GenerateArchitecture[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function that reads the objects' orientations and positions and generates the whole layout as a 3D graphics.

Uses:

GenerateArchitecture uses the functions shown in Table 69.64.

Context Name	Function Name
Simulator`Simulator`	drawDriver

Table 69.64: The Functions and Their Contexts Used by **GenerateArchitecture.**

Is Used In:

GenerateArchitecture is used in the functions shown in Table 69.65.

Context Name	Function Name
Simulator`Simulator`	ShowArchitecture

Table 69.65: The Functions and Their Contexts Making Use of **GenerateArchitecture.**

69.7.2 generateContextDependencyDatabase[]

Context Name:

Databases`FunctionsToContexts`

Usage Message:

Reads the already generated FunctionDependencies.database file and produces a flat list of currentContexts, allUsedContexts. It saves this information in a global variable OHDL$ContextDependencies.

Options[] = RegenerateContextDependencyDatabase -> True/False.

True (default): forces generation of new file.

False : returns the global variable OHDL$ContextDependencies.

Uses:

generateContextDependencyDatabase makes use of the functions shown in Table 69.66.

Context Name	Function Name
Databases`FunctionsToContexts`	generateContextDependencyDatabase

Table 69.66: The Functions and Their Contexts Used by **generateContextDependencyDatabase.**

Is Used In:

generateContextDependencyDatabase is used in the functions shown in Table 69.67.

Context Name	Function Name
Databases‘FunctionsToContexts‘	dependencyContexts
Databases‘FunctionsToContexts‘	generateContextDependencyDatabase

Table 69.67: The Functions and Their Contexts Making Use of **generateContextDependencyDatabase**.

69.7.3 GenerateDerivedFiles[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

For a notebook file, generates all the derived files i.e. package, examples and *.functions files

Uses:

GenerateDerivedFiles uses the functions shown in Table 69.68.

Context Name	Function Name
Translators‘NotebookToPackage‘	createExamplesNotebook
Translators‘NotebookToPackage‘	createPackage

Table 69.68: The Functions and Their Contexts Used by **GenerateDerivedFiles**.

Is Used In:

GenerateDerivedFiles is used in the functions shown in Table 69.69.

Context Name	Function Name
Utilities‘SystemsProgramming‘	generateTree
Translators‘NotebookToPackage‘	reGenerateDerivedFiles

Table 69.69: The Functions and Their Contexts Making Use of **GenerateDerivedFiles.**

69.7.4 generateDirectoryInfo[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

For a directory, find the OHDLFileStatus and store all in a file .OHDL.index.

Uses:

generateDirectoryInfo uses the functions shown in Table 69.70.

Context Name	Function Name
Utilities‘SystemsProgramming‘	OHDLFileType
Utilities‘StringOperations‘	quote

Table 69.70: The Functions and Their Contexts Used by **generateDirectoryInfo.**

Is Used In:

generateDirectoryInfo is used in the functions shown in Table 69.71.

Context Name	**Function Name**
Utilities'SystemsProgramming'	generateTreeIndex
Utilities'SystemsProgramming'	getDirectoryInfo

Table 69.71: The Functions and Their Contexts Making Use of **generateDirectory-Info.**

69.7.5 generateTree[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

For a specified path, do the following:

1. Generate the new tree index (.OHDL.index files),
2. Create the .functions files,
3. Generate the dependencies between files,
4. Generate the derived files, and
5. regenerate the new tree index.

Uses:

generateTree uses the functions shown in Table 69.72.

Context Name	Function Name
Translators'NotebookToPackage'	createFunctionsFile
Translators'NotebookToPackage'	GenerateDerivedFiles
Utilities'SystemsProgramming'	generateTreeIndex
Utilities'SystemsProgramming'	notebookQ
Databases'Dependencies'	obtainDependencies
Utilities'SystemsProgramming'	OHDLFileStatus
Utilities'SystemsProgramming'	operateByIndex
Utilities'SystemsProgramming'	operateByIndex
Translators'NotebookToPackage'	removeDerivedFiles

Table 69.72: The Functions and Their Contexts Used by **generateTree.**

69.7.6 generateTreeIndex[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Calls generate directoryInfo on each file/directory in the file hierarchy.

Uses:

generateTreeIndex uses the functions shown in Table 69.73.

Context Name	Function Name
Utilities'SystemsProgramming'	directoryQ
Utilities'SystemsProgramming'	generateDirectoryInfo
Utilities'SystemsProgramming'	unixLikeFind

Table 69.73: The Functions and Their Contexts Used by **generateTreeIndex.**

Is Used In:

generateTreeIndex is used in the functions shown in Table 69.74.

Context Name	Function Name
Utilities'SystemsProgramming'	generateTree

Table 69.74: The Functions and Their Contexts Making Use of **generateTreeIndex**.

69.7.7 generateUsageDatabase[]

Context Name:

Databases'FunctionsToContexts'

Usage Message:

Reads the function usage database and returns a database containing records of the form {{contextName, functionName}, "Usage Message"}.

Options[generateUsageDatabase] = {RegenerateFunctionUsageDatabase -> True or False}. True (default): Forces reading of the file. False: Returns Identity.

Uses:

generateUsageDatabase uses the functions shown in Table 69.75.

Context Name	Function Name
Utilities'StringOperations'	listToString

Table 69.75: The Functions and Their Contexts Used by **generateUsageDatabase**.

Is Used In:

generateUsageDatabase is used in the functions shown in Table 69.76.

Context Name	Function Name
Databases‘FunctionsToContexts‘	functionNameToUsage

Table 69.76: The Functions and Their Contexts Making Use of **generateUsage-Database**.

69.7.8 geQ[]

Context Name:

Utilities‘RealArithmetic‘

Usage Message:

One of the predicates for the relational operators == , != , > , < , <= , for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

Uses:

geQ uses the functions shown in Table 69.77.

Context Name	Function Name
Simulator‘Framework‘	eqQ
Simulator‘Framework‘	gtQ

Table 69.77: The Functions and Their Contexts Used by **geQ**.

69.7.9 getAllFunctionNames[]

Context Name:

Databases'Dependencies'

Usage Message:

Starting from the given path, it returns the list of structures returned by functionPairs[]. All subdirectories are also explored. Options[getAllFunctionNames] = {RegenerateAllFunctions -> True/False}. True (default) forces regeneration of the resulting file. False returns a previously generated file.

In[1]:=
```
getAllFunctionNames["OptiCAD/Packages/Graphics"]
```

Out[1]=
```
{{Graphics'Shapes', {AffineShape, BeginPackage, cone, Cone, cuboid,
cuboidtolines, cylinder, Cylinder,
disk, DoubleHelix, ellipsoid, Helix, hologram, interferenceFilter, lensTwo,
mirror, RotateShape, seedgraphicsobject, slicedCylinder, slicedTruncated-
Cone, Sphere, splitter, Torus, TranslateShape, truncatedCone, wavePlate,
WireFrame}}, {Graphics'Misc', {displayOdject}}, {Graphics'Graphics3D',
{operationToMatrix, transform3D}}}
```

In[2]:=
```
getAllFunctionNames["OptiCAD/Packages/Geometry"]
```

Out[2]=
```
{{Geometry'Predicates', {isTheLineAnAxis, isTheLineX-
Axis, isTheLineYAxis, isTheLineZAxis, validLine}}, {Geometry'Geometry',
{colinearQ, equationOfPlane, normalize, pointInPlane, pointOnThePlaneQ,
rayAtBoundary, vectorAtAnAngle}}}
```

Uses:

getAllFunctionNames uses the functions shown in Table 69.78.

Is Used In:

getAllFunctionNames is used in the functions shown in Table 69.79.

Context Name	Function Name
Utilities'SystemsProgramming'	functionsFileNameQ
Utilities'SystemsProgramming'	unixLikeFind

Table 69.78: The Functions and Their Contexts Used by **getAllFunctionNames**.

Context Name	Function Name
Databases'Dependencies'	obtainDependencies

Table 69.79: The Functions and Their Contexts Making Use of **getAllFunctionNames**.

69.7.10 getDirectoryInfo[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

For a specified directory, reads first the .OHDL.index file. Refines or modifies the attributes with the .OHDL.override.index file and returns these modified attributes.

Uses:

getDirectoryInfo uses the functions shown in Table 69.80.

Is Used In:

getDirectoryInfo is used in the functions shown in Table 69.81.

Context Name	Function Name
Utilities‘SystemsProgramming‘	dirName
Utilities‘SystemsProgramming‘	existFileQ
Utilities‘SystemsProgramming‘	generateDirectoryInfo
Simulator‘Framework‘	leQ

Table 69.80: The Functions and Their Contexts Used by **getDirectoryInfo.**

Context Name	Function Name
Utilities‘SystemsProgramming‘	OHDLFileStatus
Utilities‘SystemsProgramming‘	operateByIndex

Table 69.81: The Functions and Their Contexts Making Use of **getDirectoryInfo.**

69.7.11 GetObject[]

Context Name:

Simulator‘Simulator‘

Usage Message:

User called function. Returns a specific components of a specific type. Attributes may be provided by user.

Uses:

GetObject uses the functions shown in Table 69.82.

Context Name	Function Name
Simulator‘Simulator‘	AddObject
Utilities‘OptionProcessing‘	PickFirst

Table 69.82: The Functions and Their Contexts Used by **GetObject.**

Is Used In:

GetObject is used in the functions shown in Table 69.83.

Context Name	**Function Name**
Simulator`Simulator`	GetObjects

Table 69.83: The Functions and Their Contexts Making Use of **GetObject**.

69.7.12 GetObjects[]

Context Name:

Simulator`Simulator`

Usage Message:

User called function. Returns a list of specific components of a specific type. Attributes may be provided by user.

Uses:

GetObjects uses the functions shown in Table 69.84.

Context Name	Function Name
Simulator`Simulator`	GetObject

Table 69.84: The Functions and Their Contexts Used by **GetObjects**.

69.7.13 gtQ[]

Context Name:

Utilities‘RealArithmetic‘

Usage Message:

One of the predicates for the relational operators == , != , > , < , <= , for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

Is Used In:

gtQ is used in the functions shown in Table 69.85.

Context Name	Function Name
Simulator‘Framework‘	geQ

Table 69.85: The Functions and Their Contexts Making Use of **gtQ**.

69.8 Functions Starting with H

69.8.1 HalfWavePlate[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a HalfWavePlate constructed from Mathematica Graphics primitives.

```
In[1]:= Show[ Graphics3D[HalfWavePlate[] ] ]

Out[1]= -Graphics-
```

Uses:

HalfWavePlate uses the functions shown in Table 69.86.

Context Name	Function Name
Graphics'Shapes'	wavePlate

Table 69.86: The Functions and Their Contexts Used by **HalfWavePlate**.

Is Used In:

HalfWavePlate is used in the functions shown in Table 69.87.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.87: The Functions and Their Contexts Making Use of **HalfWavePlate**.

69.8.2 handleDefaults[]

Context Name:

Simulator'Simulator'

Usage Message:

Translates a set of coordinates so that all lie in the first quadrant.

Uses:

handleDefaults uses the functions shown in Table 69.88.

Context Name	Function Name
Simulator'Simulator'	coordinateRulesToQuadruples
Simulator'Simulator'	quadruplesToCoordinateRules

Table 69.88: The Functions and Their Contexts Used by **handleDefaults**.

Is Used In:

handleDefaults is used in the functions shown in Table 69.89.

Context Name	Function Name
Simulator'Simulator'	ComputePositions

Table 69.89: The Functions and Their Contexts Making Use of **handleDefaults**.

69.8.3 hasTokenQ[]

Context Name:

Utilities`StringOperations`

Usage Message:

Checks whether a string has a certain token. This is different from subString operation. A token has specific boundaries. Options[hasTokenQ] = {SeparatorList}.

```
In[1]:= hasTokenQ["add1[];
Constraints;
", "add"]

Out[1]= False
```

Is Used In:

hasTokenQ is used in the functions shown in Table 69.90.

Context Name	Function Name
Databases`Dependencies`	uses

Table 69.90: The Functions and Their Contexts Making Use of **hasTokenQ**.

69.8.4 hologram[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns an icon for a hologram using Mathematica Graphics primitives.

```
In[1]:= hologram[]

Out[1]= -Graphics-
```

Is Used In:

hologram is used in the functions shown in Table 69.91.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.91: The Functions and Their Contexts Making Use of **hologram**.

69.9 Functions Starting with I

69.9.1 initialize[]

Context Name:

Simulator`Simulator`

Usage Message:

Resets the users Global variables.

69.9.2 inputCellToPairs[]

Context Name:

Databases`Dependencies`

Usage Message:

Breaks an input definition cell into {functionName, Body} Pairs. All comments and quoted strings are removed first.

Options[inputCellToPairs] = {HeadBodySeparator, FunctionSeparator, FunctionsInFullForm}.

HeadBodySeparator (default = ":="): string between the head and body of the definition.

FunctionSeparator (default = "\n\n"): separator between functions.

FunctionsInFullForm (default = False): not supported yet.

If True return only the symbol of the function header (no args, conditions).

```
In[1]:= inputCellToPairs["a := \n\n b\n\nc:=d"]

Out[1]= {{"a", " b"}, {"c", "d"}}
```

Uses:

inputCellToPairs uses the functions shown in Table 69.92.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell
Utilities'StringOperations'	nBlanks
Utilities'StringOperations'	nChars
Utilities'StringOperations'	removePattern
Utilities'StringOperations'	removeQuotedStrings

Table 69.92: The Functions and Their Contexts Used by **inputCellToPairs**.

69.9.3 interferenceFilter[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns an icon for an interferenceFilter using Mathematica Graphics primitives.

```
In[1]:= interferenceFilter[]

Out[1]= -Graphics-
```

Uses:

interferenceFilter uses the functions shown in Table 69.93.

Context Name	Function Name
Graphics'Shapes'	cuboid

Table 69.93: The Functions and Their Contexts Used by **interferenceFilter**.

Is Used In:

interferenceFilter is used in the functions shown in Table 69.94.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.94: The Functions and Their Contexts Making Use of **interferenceFilter**.

69.10 Functions Starting with L

69.10.1 laser[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a laser constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[laser[] ] ]
```

Out[1]= *-Graphics-*

Uses:

laser uses the functions shown in Table 69.95.

Context Name	Function Name
Graphics'Shapes'	cylinder
OpticalComponents'Icons'	laser
OpticalComponents'Icons'	stand
Translators'TransformMma3D'	transformMma3D

Table 69.95: The Functions and Their Contexts Used by **laser**.

Is Used In:

laser is used in the functions shown in Table 69.96.

Context Name	Function Name
OpticalComponents'draw'	draw
OpticalComponents'Icons'	laser

Table 69.96: The Functions and Their Contexts Making Use of **laser**.

69.10.2 lensTwo[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns an icon for a lensTwo using Mathematica Graphics primitives.

In[1]:=
```
lensTwo[]
```

Out[1]=
```
-Graphics-
```

Is Used In:

lensTwo is used in the functions shown in Table 69.97.

Context Name	Function Name
OpticalComponents'Icons'	ConvexLens

Table 69.97: The Functions and Their Contexts Making Use of **lensTwo**.

69.10.3 leQ[]

Context Name:

Utilities‘RealArithmetic‘

Usage Message:

One of the predicates for the relational operators == , != , > , < , <= , for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

Is Used In:

leQ is used in the functions shown in Table 69.98.

Context Name	Function Name
Utilities‘SystemsProgramming‘	backupAndDelete
Utilities‘SystemsProgramming‘	cleanTree
Utilities‘SystemsProgramming‘	getDirectoryInfo
Translators‘NotebookToPackage‘	removeDerivedFiles

Table 69.98: The Functions and Their Contexts Making Use of **leQ**.

69.10.4 lineQ[]

Context Name:

Geometry‘Predicates‘

Usage Message:

Checks whether a specific line is valid, i.e., it is not a point.

In[1]:=
```
lineQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
```

Out[1]=
```
True
```

In[2]:=
```
lineQ[line[point[{0, 0, 0}], point[{0, 0, 0}]]]
```

Out[2]=
```
False
```

Is Used In:

lineQ is used in the functions shown in Table 69.99.

Context Name	Function Name
Graphics‘Graphics3D‘	operationToMatrix

Table 69.99: The Functions and Their Contexts Making Use of **lineQ**.

69.10.5 listToString[]

Context Name:

Utilities'StringOperations'

Usage Message:

Converts a list of objects to one string separated by "sep". The prefix and suffix of the resulting string can be specified as rules. Options[] = {StringPrefix, StringSuffix}. StringPrefix (default = "") is the prefix of the resulting string. StringSuffix (default = "") is the suffix of the resulting string.

```
In[1]:= listToString[{"aaa", "bbb", "ccc"}, ", "]

Out[1]= "aaa, bbb, ccc"

In[2]:= listToString[{}, ", "]

Out[2]= ""

In[3]:= listToString[{{1, 2, 3}, "aaa", pqr}, "|", StringPrefix -> 999,
        StringSuffix -> "mm"]

Out[3]= 999{1, 2, 3}—aaa—pqrmm
```

Is Used In:

listToString is used in the functions shown in Table 69.100.

Context Name	Function Name
Databases‘FunctionsToContexts‘	generateUsageDatabase
Databases‘Dependencies‘	obtainDependenciesForFile
Simulator‘Framework‘	simulateDriver
Translators‘NotebookToPackage‘	writeTo

Table 69.100: The Functions and Their Contexts Making Use of **listToString**.

69.10.6 ltQ[]

Context Name:

Utilities‘RealArithmetic‘

Usage Message:

One of the predicates for the relational operators == , != , > , < , <= , for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

69.11 Functions Starting with M

69.11.1 mirror[]

Context Name:

Graphics`Shapes`

Usage Message:

Returns an icon for a mirror using Mathematica Graphics primitives.

In[1]:=
```
mirror[]
```

Out[1]=
```
-Graphics-
```

Is Used In:

mirror is used in the functions shown in Table 69.101.

Context Name	Function Name
OpticalComponents`draw`	draw

Table 69.101: The Functions and Their Contexts Making Use of **mirror**.

69.12 Functions Starting with N

69.12.1 nBlanks[]

Context Name:

Utilities'StringOperations'

Usage Message:

Generates a string of length n containing the <space> character.

In[1]:=
```
nBlanks[12]
```

Out[1]=
```
"            "
```

Uses:

nBlanks uses the functions shown in Table 69.102.

Context Name	Function Name
Utilities'StringOperations'	nChars

Table 69.102: The Functions and Their Contexts Used by **nBlanks**.

Is Used In:

nBlanks is used in the functions shown in Table 69.103.

Context Name	Function Name
Translators'NotebookToPackage'	extractFunctionNames
Databases'Dependencies'	inputCellToPairs

Table 69.103: The Functions and Their Contexts Making Use of **nBlanks**.

69.12.2 nChars[]

Context Name:

Utilities'StringOperations'

Usage Message:

Generates a string of c characters of length n.

In[1]:=
```
nChars[12, "y"]
```

Out[1]=
```
"yyyyyyyyyyyy"
```

Is Used In:

nChars is used in the functions shown in Table 69.104.

Context Name	Function Name
Databases'Dependencies'	inputCellToPairs
Utilities'StringOperations'	nBlanks

Table 69.104: The Functions and Their Contexts Making Use of **nChars**.

69.12.3 neQ[]

Context Name:

Utilities‘RealArithmetic‘

Usage Message:

One of the predicates for the relational operators == , != , > , < , <= , for the comparison of finite precision real numbers. To have a controlled complexity over the thorny problem of having to deal with finite precision computations.

Is Used In:

neQ is used in the functions shown in Table 69.105.

Context Name	Function Name
Geometry‘Geometry‘	equationOfPlane
Geometry‘Geometry‘	normalize
Graphics‘Graphics3D‘	operationToMatrix
Geometry‘Predicates‘	pointOnThePlaneQ

Table 69.105: The Functions and Their Contexts Making Use of **neQ**.

69.12.4 nonAlphaNumericASCIISet[]

Context Name:

Utilities‘StringOperations‘

Usage Message:

Generates all characters in the range "\0" -> "\47", "\58" -> "\64", "\91" -> "\96", "\123" -> "\255".

69.12.5 nonDefinitionCells[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Returns contents of a cell that does not contain any function definition.

Uses:

nonDefinitionCells uses the functions shown in Table 69.106.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell

Table 69.106: The Functions and Their Contexts Used by **nonDefinitionCells**.

Is Used In:

nonDefinitionCells is used in the functions shown in Table 69.107.

Context Name	Function Name
Translators'NotebookToPackage'	createExamplesNotebook

Table 69.107: The Functions and Their Contexts Making Use of **nonDefinitionCells**.

69.12.6 normalize[]

Context Name:

Geometry'Geometry'

Usage Message:

Given a vector, it returns its normalized vactor.

In[1]:=
```
normalize[{2, 3, 4}]
```

Out[1]=
```
{0.371391, 0.557086, 0.742781}
```

In[2]:=
```
normalize[{0, 0, 0}]
```

Out[2]=
```
{1, 0, 0}
```

Uses:

normalize uses the functions shown in Table 69.108.

Context Name	Function Name
Simulator'Framework'	neQ

Table 69.108: The Functions and Their Contexts Used by **normalize**.

69.12.7 notebookNameToContextName[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Given a full pathName for notebook name, returns the Context name.

```
In[1]:= notebookNameToContextName[
        "OptiCAD/Utilities/SystemsProgramming.ma" ]

Out[1]= OHDLUtilities'SystemsProgramming'
```

Is Used In:

notebookNameToContextName is used in the functions shown in Table 69.109.

Context Name	Function Name
Translators'NotebookToPackage'	createPackage
Databases'Dependencies'	obtainDependenciesForFile

Table 69.109: The Functions and Their Contexts Making Use of **notebookNameToContextName**.

69.12.8 notebookNameToExamplesFileName[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

For a notebook.ma file, returns the notebook.examples.ma file name

```
In[1]:= notebookNameToExamplesFileName["string.ma"]

Out[1]= string.examples.ma
```

Is Used In:

notebookNameToExamplesFileName is used in the functions shown in Table 69.110.

Context Name	Function Name
Translators‘NotebookToPackage‘	createExamplesNotebook
Translators‘NotebookToPackage‘	derivedFileNames

Table 69.110: The Functions and Their Contexts Making Use of **notebookNameToExamplesFileName.**

69.12.9 notebookNameTofunctionsFileName[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Returns a *.functions file name from a given notebook name.

```
In[1]:= notebookNameTofunctionsFileName["string.ma"]

Out[1]= string.functions
```

Is Used In:

notebookNameTofunctionsFileName is used in the functions shown in Table 69.111.

Context Name	Function Name
Translators'NotebookToPackage'	createFunctionsFile
Translators'NotebookToPackage'	derivedFileNames

Table 69.111: The Functions and Their Contexts Making Use of **notebookNameTofunctionsFileName**.

69.12.10 notebookNameToPackageFileName[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

Returns a package file name from a given notebook name.

```
In[1]:= notebookNameToPackageFileName["string.ma"]

Out[1]= string.m
```

Is Used In:

notebookNameToPackageFileName is used in the functions shown in Table 69.112.

Context Name	Function Name
Translators‘NotebookToPackage‘	createExamplesNotebook
Translators‘NotebookToPackage‘	createPackage
Translators‘NotebookToPackage‘	derivedFileNames

Table 69.112: The Functions and Their Contexts Making Use of **notebookNameToPackageFileName.**

69.12.11 notebookQ[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Checks whether a file is a Mathematica notebook.

In[1]:=
```
notebookQ["SystemsProgramming.mb"]
```

Out[1]=
```
False
```

Is Used In:

notebookQ is used in the functions shown in Table 69.113.

Context Name	Function Name
Utilities'SystemsProgramming'	generateTree
Databases'Dependencies'	obtainDependencies

Table 69.113: The Functions and Their Contexts Making Use of **notebookQ**.

69.13 Functions Starting with O

69.13.1 obtainDependencies[]

Context Name:

Databases‘Dependencies‘

Usage Message:

For a specified path, obtains all function dependencies within and across files.

Stores the dependency information in a specified file.

Uses:

obtainDependencies uses the functions shown in Table 69.114.

Context Name	Function Name
Databases‘Dependencies‘	getAllFunctionNames
Utilities‘SystemsProgramming‘	notebookQ
Databases‘Dependencies‘	obtainDependenciesForFile
Utilities‘SystemsProgramming‘	operateByIndex
Utilities‘SystemsProgramming‘	operateByIndex

Table 69.114: The Functions and Their Contexts Used by **obtainDependencies.**

Is Used In:

obtainDependencies is used in the functions shown in Table 69.115.

Context Name	Function Name
Utilities'SystemsProgramming'	generateTree

Table 69.115: The Functions and Their Contexts Making Use of **obtainDependencies.**

69.13.2 obtainDependenciesForFile[]

Context Name:

Databases'Dependencies'

Usage Message:

Obtains all the dependencies between functions of a specific file.

Uses:

obtainDependenciesForFile uses the functions shown in Table 69.116.

Context Name	Function Name
Utilities'StringOperations'	listToString
Translators'NotebookToPackage'	notebookNameToContextName
Translators'NotebookToPackage'	translateNotebook
Databases'Dependencies'	uses

Table 69.116: The Functions and Their Contexts Used by **obtainDependenciesForFile.**

Is Used In:

obtainDependenciesForFile is used in the functions shown in Table 69.117.

Context Name	Function Name
Databases'Dependencies'	obtainDependencies

Table 69.117: The Functions and Their Contexts Making Use of **obtainDependenciesForFile.**

69.13.3 OHDLFileStatus[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

From the directory info, get the status of a particular file/directory.

In[1]:=
```
OHDLFileStatus["OptiCAD/Packages/Examples"]
```

Out[1]=
```
DoNotExpand
```

In[2]:=
```
OHDLFileStatus["OptiCAD/Packages/Utilities"]
```

Out[2]=
```
DoNotRemove
```

Uses:

OHDLFileStatus uses the functions shown in Table 69.118.

Context Name	Function Name
Utilities'SystemsProgramming'	dirName
Utilities'SystemsProgramming'	getDirectoryInfo

Table 69.118: The Functions and Their Contexts Used by **OHDLFileStatus.**

Is Used In:

OHDLFileStatus is used in the functions shown in Table 69.119.

Context Name	Function Name
Utilities'SystemsProgramming'	cleanTree
Utilities'SystemsProgramming'	generateTree

Table 69.119: The Functions and Their Contexts Making Use of **OHDLFileStatus**.

69.13.4 OHDLFileType[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Returns a list of OHDL attributes for a path/file.

```
In[1]:= OHDLFileType["Mathematica.ma"]

Out[1]= {.ma, Mathematica Notebook, DoNotRemove}
```

Is Used In:

OHDLFileType is used in the functions shown in Table 69.120.

Context Name	Function Name
Utilities'SystemsProgramming'	generateDirectoryInfo

Table 69.120: The Functions and Their Contexts Making Use of **OHDLFileType**.

69.13.5 OHDL$initialize[]

Context Name:

Init‘Init‘

Usage Message:

The main function that initializes the whole system. All system global variables are set here. This routine should only be called to reset the whole system.

69.13.6 operateByIndex[]

Context Name:

Utilities‘SystemsProgramming‘

Usage Message:

Recursively descends the directory hierarchy for each pathname in the .OHDL.index, seeking files that match a boolean (logical) expression (predicate) and applies action according to the status of the file/directory.

Uses:

operateByIndex uses the functions shown in Table 69.121.

Context Name	Function Name
Utilities‘SystemsProgramming‘	getDirectoryInfo
Utilities‘SystemsProgramming‘	operateByIndex

Table 69.121: The Functions and Their Contexts Used by **operateByIndex.**

Is Used In:

operateByIndex is used in the functions shown in Table 69.122.

Context Name	Function Name
Utilities‘SystemsProgramming‘	cleanTree
Utilities‘SystemsProgramming‘	generateTree
Databases‘Dependencies‘	obtainDependencies
Utilities‘SystemsProgramming‘	operateByIndex

Table 69.122: The Functions and Their Contexts Making Use of **operateByIndex.**

69.13.7 operationToMatrix[]

Context Name:

Graphics'Graphics3D'

Usage Message:

returns the transformation matrix corresponding to the operation specified. Different operations are (1) reflectThrough[plane[a, b, c, d]], (2) rotateAround[line[point[{x1, y1, z1}], point[{x2, y2, z2}]], angle[theta]], (3) translateBy[{l, m, n}], (4) reflectThroughXY[ ], (5) reflectThroughYZ[ ], (6) reflectThroughXZ[ ], (7) rotateAroundXBy[angle[theta]], (8) rotateAroundYBy[angle[theta]], (9) rotateAroundZBy[angle[theta]], (10) shear[{b, c, d, f, g, i}], (11) scaleOverall[scaleFactor], (12) scaleLocal[{xScale, yScale, zScale}].

Uses:

operationToMatrix uses the functions shown in Table 69.123.

Context Name	Function Name
Geometry'Predicates'	axisQ
Geometry'Predicates'	lineQ
Simulator'Framework'	neQ
Graphics'Graphics3D'	operationToMatrix
Geometry'Geometry'	pointInPlane
Geometry'Predicates'	xAxisQ
Geometry'Predicates'	yAxisQ
Geometry'Predicates'	zAxisQ

Table 69.123: The Functions and Their Contexts Used by **operationToMatrix.**

Is Used In:

operationToMatrix is used in the functions shown in Table 69.124.

Context Name	Function Name
Graphics'Graphics3D'	operationToMatrix
Graphics'Graphics3D'	transform3D

Table 69.124: The Functions and Their Contexts Making Use of **operationToMatrix.**

69.13.8 orient[]

Context Name:

Simulator'Graphics3D'

Usage Message:

Every object has a bounding box. Initially, the bounding box is placed in the first octant with edges aligned to the coordinate axes. Every bounding box has a reference point (every other measure of the component is relative to this point) and a principal axis (A unit vector in the x-direction of the local coordinate system) whose tail is the reference point. The orientation is represented by {thetaX, thetaXY, thetaP}. The final orientation of the object is determined by the following steps. (1) The object is assumed to be in the first octant with its reference point at the origin. (2) Rotate the object around z-axis by an angle of thetaX. (0 < = thetaX < 2 Pi). (3) Let L be a line passing through the origin in the xz-plane and perpendicular to the rotated principal axis. (4) Rotate the object around L by thetaXY (0 <= thetaXY < 2 Pi). Hence thetaXY is the angle between the principal axis in its final orientation and the xy-plane. (5) Rotate the object around the principal axis by thetaP (0 <= thetaP < 2 Pi). The function orient[] will return the oriented object for a given object.

69.13.9 orientationOf[]

Context Name:

Simulator'Simulator'

Usage Message:

Returns the position of an object as stored in the global orientations database.

Is Used In:

orientationOf is used in the functions shown in Table 69.125.

Context Name	Function Name
Simulator'Simulator'	AddOrientationConstraint
Simulator'Framework'	boundingBoxEquations

Table 69.125: The Functions and Their Contexts Making Use of **orientationOf**.

69.13.10 orientBoundingBox[]

Context Name:

Simulator‘Graphics3D‘

Usage Message:

Initially, the bounding box is placed in the first octant with edges aligned to the coordinate axes. Every bounding box has a reference point (every other measure of the component is relative to this point) and a principal axis (A unit vector in the x-direction of the local coordinate system) whose tail is the reference point. The orientation is represented by {thetaX, thetaXY, thetaP}. The final orientation of the object is determined by the following steps. (1) The object is assumed to be in the first octant with its reference point at the origin. (2) Rotate the object around z-axis by an angle of thetaX. (0 < = thetaX < 2 Pi). (3) Let L be a line passing through the origin in the xz-plane and perpendicular to the rotated principal axis. (4) Rotate the object around L by thetaXY (0 <= thetaXY < 2 Pi). Hence thetaXY is the angle between the principal axis in its final orientation and the xy-plane. (5) Rotate the object around the principal axis by thetaP (0 <= thetaP < 2 Pi). The function orientBoundingBox[] will return the oriented bounding Box.

69.14 Functions Starting with P

69.14.1 PickFirst[]

Context Name:

Utilities'OptionProcessing'

Usage Message:

Pick Unique First Rules.

In[1]:=
```
PickFirst[{{A -> a, B -> b, X -> x}, {A -> aa, C -> cc, D
-> dd}, {A -> aaa}}]
```

Out[1]=
```
{A -> a, B -> b, C -> cc, D -> dd, X -> x}
```

Is Used In:

PickFirst is used in the functions shown in Table 69.126.

Context Name	Function Name
Simulator'Simulator'	GetObject
Simulator'Framework'	performAction
Utilities'OptionProcessing'	PickLast
Simulator'Framework'	processMessage
Simulator'Simulator'	simulate
Simulator'Framework'	simulateComponent

Table 69.126: The Functions and Their Contexts Making Use of **PickFirst**.

69.14.2 PickLast[]

Context Name:

Utilities'OptionProcessing'

Usage Message:

Pick the last occurance of each rule.

In[2]:=
```
PickLast[{{A -> a, B -> b, X -> x}, {A -> aa, C -> cc, D ->
dd}, {A -> aaa}}]
```

Out[2]=
```
{A -> aaa, B -> b, C -> cc, D -> dd, X -> x}
```

Uses:

PickLast uses the functions shown in Table 69.127.

Context Name	Function Name
Utilities'OptionProcessing'	PickFirst

Table 69.127: The Functions and Their Contexts Used by **PickLast**.

69.14.3 PlanoConvexCylindricalLens[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a PlanoConvexCylindricalLens constructed from Mathematica Graphics primitives.

In[1]:=

```
Show[ Graphics3D[PlanoConvexCylindricalLens[] ] ]
```

Out[1]=

```
-Graphics-
```

Uses:

PlanoConvexCylindricalLens uses the functions shown in Table 69.128.

Context Name	Function Name
Graphics'Shapes'	slicedCylinder

Table 69.128: The Functions and Their Contexts Used by **PlanoConvexCylindricalLens.**

Is Used In:

PlanoConvexCylindricalLens is used in the functions shown in Table 69.129.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.129: The Functions and Their Contexts Making Use of **PlanoConvexCylindricalLens.**

69.14.4 pointInPlane[]

Context Name:

Geometry‘Geometry‘

Usage Message:

Returns a point on the given plane.

In[1]:=
```
N[pointInPlane[plane[2, 4, 5, 7]]]
```

Out[1]=
```
{-3.5, 0, 0}
```

Is Used In:

pointInPlane is used in the functions shown in Table 69.130.

Context Name	Function Name
Graphics‘Graphics3D‘	operationToMatrix

Table 69.130: The Functions and Their Contexts Making Use of **pointInPlane**.

69.14.5 pointOnThePlaneQ[]

Context Name:

Geometry'Predicates'

Usage Message:

Checks whether the given point lies on the given plane.

In[1]:=
```
pointOnThePlaneQ[{0, 0, 0}, {1, 0, 0, 0}]
```
Out[1]=
```
True
```
In[2]:=
```
pointOnThePlaneQ[{0, 0, 0}, {1, 1, 1, 1}]
```
Out[2]=
```
False
```

Uses:

pointOnThePlaneQ uses the functions shown in Table 69.131.

Context Name	Function Name
Simulator'Framework'	eqQ
Simulator'Framework'	neQ
Geometry'Predicates'	pointOnThePlaneQ

Table 69.131: The Functions and Their Contexts Used by **pointOnThePlaneQ**.

Is Used In:

pointOnThePlaneQ is used in the functions shown in Table 69.132.

Context Name	Function Name
Geometry‘Geometry‘	equationOfPlane
Geometry‘Predicates‘	pointOnThePlaneQ

Table 69.132: The Functions and Their Contexts Making Use of **pointOnThePlaneQ**.

69.14.6 positionOf[]

Context Name:

Simulator‘Simulator‘

Usage Message:

Returns the position of an object as stored in the global positions database.

Is Used In:

positionOf is used in the functions shown in Table 69.133.

Context Name	Function Name
Simulator‘Framework‘	boundingBoxEquations

Table 69.133: The Functions and Their Contexts Making Use of **positionOf**.

69.15 Functions Starting with Q

69.15.1 quadruplesToCoordinateRules[]

Context Name:

Simulator'Simulator'

Usage Message:

Transforms the quadruple representation to rules representation.

Is Used In:

quadruplesToCoordinateRules is used in the functions shown in Table 69.134.

Context Name	Function Name
Simulator'Simulator'	handleDefaults

Table 69.134: The Functions and Their Contexts Making Use of **quadruplesToCoordinateRules.**

69.15.2 QuarterWavePlate[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a QuarterWavePlate constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[QuarterWavePlate[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

QuarterWavePlate uses the functions shown in Table 69.135.

Context Name	Function Name
Graphics'Shapes'	wavePlate

Table 69.135: The Functions and Their Contexts Used by **QuarterWavePlate**.

69.15.3 quote[]

Context Name:

Utilities'StringOperations'

Usage Message:

Listable function. It puts quotes around a string.

```
In[1]:= quote["Test"]

Out[1]= "\"Test\""

In[2]:= quote[{{"str1", "str2"}, {"str3"}}]

Out[2]= {{"\"str1\"", "\"str2\""}, {"\"str3\""}}
```

Uses:

quote uses the functions shown in Table 69.136.

Context Name	Function Name
Utilities'StringOperations'	quote

Table 69.136: The Functions and Their Contexts Used by **quote**.

Is Used In:

quote is used in the functions shown in Table 69.137.

Context Name	Function Name
Utilities‘SystemsProgramming‘	generateDirectoryInfo
Utilities‘StringOperations‘	quote
Databases‘Dependencies‘	uses

Table 69.137: The Functions and Their Contexts Making Use of **quote**.

69.16 Functions Starting with R

69.16.1 rayAtBoundary[]

Context Name:

Geometry‘Geometry‘

Usage Message:

The inputs to this function include {{x0,y0}, {a,b}, n} – an input vector starting from {x0,y0} with direction of the unit vector {a,b}, and surface – a list of constraints & refractive indices represented by a data-structure {f(x,y) == 0, xl <= x <= xh &&, yl <= y <= yh, ni, no} where ni is the refractive index of the first medium and no is the refractive index of the second medium. The function returns {{xi,yi}, {c,d}, no} – a ray going through {xi, yi}, i.e., the point of intersection of input ray and surface, parallel to the unit vector {c,d}, and Reflection/Refraction – a boolean quantity indicating whether the outgoing ray is a reflected one or refracted one.

In[1]:=

```
ni = 1;
no = 3;
x0 = -1;
y0 = -3;
{a, b} = normalize[{1,3}];
f = ((#1-3)↑2 + #2↑2 - 9) &;
bc = ( leQ[0,#1] && leQ[#1,3] && leQ[-3,#2] && leQ[#2,3] ) &;
icr = {{-1,-3},normalize[{1,3}],ni};
s = {f, bc, ni, no};
rayAtBoundary[icr,s]
```

Out[1]=

```
{{{0., -4.44089 10↑-16 }, {0.948683, 0.316228}, 3}, Refraction}
```

69.16.2 reGenerateDerivedFiles[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

Recreates all derived files (package, *.functions, examples) names for a notebook file.

Uses:

reGenerateDerivedFiles uses the functions shown in Table 69.138.

Context Name	Function Name
Translators'NotebookToPackage'	GenerateDerivedFiles
Translators'NotebookToPackage'	removeDerivedFiles

Table 69.138: The Functions and Their Contexts Used by **reGenerateDerivedFiles**.

69.16.3 removeDerivedFiles[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

Removes all derived files (package, *.functions, examples) names for a notebook file.

Uses:

removeDerivedFiles uses the functions shown in Table 69.139.

Context Name	Function Name
Translators‘NotebookToPackage‘	derivedFileNames
Utilities‘SystemsProgramming‘	existFileQ
Simulator‘Framework‘	leQ

Table 69.139: The Functions and Their Contexts Used by **removeDerivedFiles.**

Is Used In:

removeDerivedFiles is used in the functions shown in Table 69.140.

Context Name	Function Name
Utilities‘SystemsProgramming‘	generateTree
Translators‘NotebookToPackage‘	reGenerateDerivedFiles

Table 69.140: The Functions and Their Contexts Making Use of **removeDerived-Files.**

69.16.4 removeHead[]

Context Name:

Utilities‘ReWriteRules‘

Usage Message:

Removes the head of the input object.

In[1]:=
```
removeHead[List[1, 2, 3]]
```

Out[1]=
```
Sequence[1, 2, 3]
```

In[2]:=
```
removeHead[Graphics3D[{}]]
```

Out[2]=
```
{}
```

In[3]:=
```
removeHead[Abcd -> abCD]
```

Out[3]=
```
Sequence[Abcd, abCD]
```

Is Used In:

removeHead is used in the functions shown in Table 69.141.

Context Name	Function Name
Simulator‘Framework‘	performAction
Simulator‘Framework‘	simulateComponent

Table 69.141: The Functions and Their Contexts Making Use of **removeHead**.

69.16.5 removePattern[]

Context Name:

Utilities‘StringOperations‘

Usage Message:

Removes a pattern in the string. The pattern is enclosed between the "startDelimeter" and "endDelimeter".

In[1]:=
```
removePattern["(* Apply a of the axes *)operation[angle[ theta_ ]
] ]", "(*", "*)"]
```

Out[1]=
```
"operation[angle[ theta_ ] ] ]"
```

In[2]:=
```
removePattern["Clear[operation];
(* Apply a of the axes *)operation[angle[ theta_ ] ] ]", "Clear[",
"];
"]
```

Out[2]=
```
"(* Apply a of the axes *)operation[angle[ theta_ ] ] ]"
```

Uses:

removePattern uses the functions shown in Table 69.142.

Context Name	Function Name
Utilities‘StringOperations‘	removePattern
Utilities‘StringOperations‘	splitRepeated

Table 69.142: The Functions and Their Contexts Used by **removePattern**.

Is Used In:

removePattern is used in the functions shown in Table 69.143.

Context Name	Function Name
Translators'NotebookToPackage'	extractFunctionNames
Databases'Dependencies'	inputCellToPairs
Utilities'StringOperations'	removePattern
Utilities'StringOperations'	removePatterns

Table 69.143: The Functions and Their Contexts Making Use of **removePattern**.

69.16.6 removePatterns[]

Context Name:

Utilities'StringOperations'

Usage Message:

Removes patterns from a string. The prefixes and suffixes of these patterns are given in a list of pairs.

```
In[1]:= removePatterns["Clear[operation];
(* Apply a of the axes *)operation[angle[ theta_ ] ] ]", {{"(*",
"*)"}, {"Clear[", "];
"}}]

Out[1]= operation[angle[ theta_ ] ] ]
```

Uses:

removePatterns uses the functions shown in Table 69.144.

Context Name	Function Name
Utilities'StringOperations'	removePattern

Table 69.144: The Functions and Their Contexts Used by **removePatterns**.

69.16.7 removeQuotedStrings[]

Context Name:

Utilities'StringOperations'

Usage Message:

Removes all quoted strings from a string.

In[1]:=
```
removeQuotedStrings["split[\"a ;
b :  e := d = e == d\", {{\";
\"}, {\":\"}, {\":=\"}}]"]
```

Out[1]=
```
split[, {{}, {}, {}}]
```

Is Used In:

removeQuotedStrings is used in the functions shown in Table 69.145.

Context Name	Function Name
Translators'NotebookToPackage'	definitionCell
Databases'Dependencies'	inputCellToPairs

Table 69.145: The Functions and Their Contexts Making Use of **removeQuoted-Strings**.

69.17 Functions Starting with S

69.17.1 SEED[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a SEED constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[SEED[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

SEED uses the functions shown in Table 69.146.

Context Name	Function Name
Graphics'Shapes'	seedgraphicsobject

Table 69.146: The Functions and Their Contexts Used by **SEED**.

Is Used In:

SEED is used in the functions shown in Table 69.147.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.147: The Functions and Their Contexts Making Use of **SEED**.

69.17.2 seedgraphicsobject[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns an icon for a seedgraphicsobject using Mathematica Graphics primitives.

In[1]:=
```
seedgraphicsobject[]
```

Out[1]=
```
-Graphics-
```

Uses:

seedgraphicsobject uses the functions shown in Table 69.148.

Context Name	Function Name
Graphics'Shapes'	cuboid

Table 69.148: The Functions and Their Contexts Used by **seedgraphicsobject.**

Is Used In:

seedgraphicsobject is used in the functions shown in Table 69.149.

Context Name	Function Name
OpticalComponents'Icons'	SEED
OpticalComponents'Icons'	SLM
OpticalComponents'Icons'	SSEED

Table 69.149: The Functions and Their Contexts Making Use of **seedgraphicsobject.**

69.17.3 selfDependencyQ[]

Context Name:

Databases'Dependencies'

Usage Message:

Takes a pair of the form

{{contextOfFunctionName, functionName},

pairsOfAllContextFunctions}

and returns True if functionName belongs to pairsOfAllContextFunctions.

In[1]:=
```
dependency = {{"Context", "ComputeOrientations[ ]"} ,
{"OHDLSimulator'Simulator'", "ComputeOrientations"}};
selfDependencyQ[dependency]
```

Out[1]=
```
True
```

In[2]:=
```
dependency = {{"Context", "ComputeOrientations[ ]"} ,
{"OHDLSimulator'Simulator'", "selfDependencyQ"}};
selfDependencyQ[dependency]
```

Out[2]=
```
False
```

Uses:

selfDependencyQ uses the functions shown in Table 69.150.

Context Name	Function Name
Utilities'StringOperations'	frontToken

Table 69.150: The Functions and Their Contexts Used by **selfDependencyQ**.

Is Used In:

selfDependencyQ is used in the functions shown in Table 69.151.

Context Name	Function Name
Databases‘Dependencies‘	uses

Table 69.151: The Functions and Their Contexts Making Use of **selfDependencyQ**.

69.17.4 ShowArchitecture[]

Context Name:

Simulator‘Simulator‘

Usage Message:

Calls GenerateArchitecture[] if no graphics has been generated. Then calls ViewArchitecture[].

Uses:

ShowArchitecture uses the functions shown in Table 69.152.

Context Name	Function Name
Simulator‘Simulator‘	GenerateArchitecture
Simulator‘Simulator‘	ViewArchitecture

Table 69.152: The Functions and Their Contexts Used by **ShowArchitecture**.

69.17.5 showState[]

Context Name:

Simulator`Simulator`

Usage Message:

Provides values for all user's global variables. Options[showState] = {FullInfo -> True/False} False (default) does not show the less important variables. True otherwise.

69.17.6 simulate[]

Context Name:

Simulator‘Simulator‘

Usage Message:

The main module that takes an object and a message. It extracts the type of message and applies the corresponding simulate[self] function.

Uses:

simulate uses the functions shown in Table 69.153.

Context Name	Function Name
Simulator‘Simulator‘	ComputeOutputPosition
Utilities‘OptionProcessing‘	PickFirst

Table 69.153: The Functions and Their Contexts Used by **simulate**.

69.17.7 slicedCylinder[]

Context Name:

Graphics`Shapes`

Usage Message:

Returns the slice of a cylinderical object.

```
In[1]:= slicedCylinder[]

Out[1]= -Graphics-
```

Uses:

slicedCylinder uses the functions shown in Table 69.154.

Context Name	Function Name
Graphics`Shapes`	slicedTruncatedCone

Table 69.154: The Functions and Their Contexts Used by **slicedCylinder**.

Is Used In:

slicedCylinder is used in the functions shown in Table 69.155.

Context Name	Function Name
OpticalComponents`Icons`	PlanoConvexCylindricalLens

Table 69.155: The Functions and Their Contexts Making Use of **slicedCylinder**.

69.17.8 slicedTruncatedCone[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns a slice of a segment of a conical object using Mathematica Graphics primitives.

In[1]:=
```
slicedTruncatedCone[]
```

Out[1]=
```
-Graphics-
```

Is Used In:

slicedTruncatedCone is used in the functions shown in Table 69.156.

Context Name	Function Name
Graphics‘Shapes‘	slicedCylinder

Table 69.156: The Functions and Their Contexts Making Use of **slicedTruncated-Cone.**

69.17.9 SLM[]

Context Name:

OpticalComponents‘Icons‘

Usage Message:

Returns an icon of a SLM constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[SLM[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

SLM uses the functions shown in Table 69.157.

Context Name	Function Name
Graphics‘Shapes‘	seedgraphicsobject

Table 69.157: The Functions and Their Contexts Used by **SLM**.

Is Used In:

SLM is used in the functions shown in Table 69.158.

Context Name	Function Name
OpticalComponents‘draw‘	draw

Table 69.158: The Functions and Their Contexts Making Use of **SLM**.

69.17.10 split[]

Context Name:

Utilities'StringOperations'

Usage Message:

Breaks an input string into parts specified by the separatorLists.

In[1]:=
```
split["a ;
b :  e := d = e == d", {{";
"}, {":"}, {":="}}]
```

Out[1]=
```
{"a ", ";
b ", ": e ", ":= d = e == d"}
```

In[2]:=
```
split["a ;
b :  e := d = e == d", {{"=", ";
"}, { "==", ":"}, {":="}}]
```

Out[2]=
```
{"a ", ";
b ", ": e ", ":= d = e == d"}
```

Is Used In:

split is used in the functions shown in Table 69.159.

Context Name	Function Name
Utilities'StringOperations'	frontToken

Table 69.159: The Functions and Their Contexts Making Use of **split**.

69.17.11 splitPathName[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Returns the base name of a file and its higher directory name.

```
In[1]:= splitPathName["OptiCAD/ocg.old"]

Out[1]= {OptiCAD, ocg.old}

In[2]:= splitPathName["OptiCAD/ocg.old", ".old"]

Out[2]= {OptiCAD, ocg}

In[3]:= splitPathName["ocg.old", ".old"]

Out[3]= {., ocg}
```

69.17.12 splitRepeated[]

Context Name:

Utilities‘StringOperations‘

Usage Message:

Breaks a word into parts repeatedly until the end of word is encountered. The break points are specified in separatorLists.

In[1]:=
```
splitRepeated["a=a;
a=a:b;
b=b=b;
c", {{";
"}}]
```

Out[1]=
```
{a=a, a=a:b, b=b=b, c}
```

In[2]:=
```
splitRepeated["a=a;
a=a:b;
b=b=b;
c", {{";
"}, {"="}}]
```

Out[2]=
```
{a=a, ;
a, =a:b, ;
b, =b=b, ;
c}
```

In[3]:=
```
splitRepeated["/a/newton/export/home/ocg.old", {{"/"}}]
```

Out[3]=
```
{a, newton, export, home, ocg.old}
```

Is Used In:

splitRepeated is used in the functions shown in Table 69.160.

Context Name	Function Name
Utilities'StringOperations'	removePattern

Table 69.160: The Functions and Their Contexts Making Use of **splitRepeated**.

69.17.13 splitter[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns an icon for a splitter using Mathematica Graphics primitives.

In[1]:=
```
splitter[]
```

Out[1]=
```
-Graphics-
```

Is Used In:

splitter is used in the functions shown in Table 69.161.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.161: The Functions and Their Contexts Making Use of **splitter**.

69.17.14 SSEED[]

Context Name:

OpticalComponents'Icons'

Usage Message:

Returns an icon of a SSEED constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[SSEED[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

SSEED uses the functions shown in Table 69.162.

Context Name	Function Name
Graphics'Shapes'	seedgraphicsobject

Table 69.162: The Functions and Their Contexts Used by **SSEED**.

Is Used In:

SSEED is used in the functions shown in Table 69.163.

Context Name	Function Name
OpticalComponents'draw'	draw

Table 69.163: The Functions and Their Contexts Making Use of **SSEED**.

69.17.15 stand[]

Context Name:

OpticalComponents‘Icons‘

Usage Message:

Returns an icon of a stand constructed from Mathematica Graphics primitives.

In[1]:=
```
Show[ Graphics3D[stand[] ] ]
```

Out[1]=
```
-Graphics-
```

Uses:

stand uses the functions shown in Table 69.164.

Context Name	Function Name
Graphics‘Shapes‘	cylinder
Translators‘TransformMma3D‘	transformMma3D
Graphics‘Shapes‘	truncatedCone

Table 69.164: The Functions and Their Contexts Used by **stand**.

69.17.16 suffixQ[]

Context Name:

Utilities'StringOperations'

Usage Message:

Checks whether string ends with a specified suffix.

```
In[1]:= {suffixQ[".ma", "file.ma"], suffixQ[".ma", "file.ma.mb"],
suffixQ["", "anything"], suffixQ["file", ".ma"]}

Out[1]= {True, False, True, False}
```

Uses:

suffixQ uses the functions shown in Table 69.165.

Context Name	Function Name
Utilities'StringOperations'	suffixQ

Table 69.165: The Functions and Their Contexts Used by **suffixQ**.

Is Used In:

suffixQ is used in the functions shown in Table 69.166.

Context Name	Function Name
Utilities'StringOperations'	suffixQ

Table 69.166: The Functions and Their Contexts Making Use of **suffixQ**.

69.18 Functions Starting with T

69.18.1 tokenize[]

Context Name:

Utilities'StringOperations'

Usage Message:

Break the string into parts as specified by Options[ReadList] which are {NullWords -> False, TokenWords -> {}, NullRecords -> False, RecordLists -> False, WordSeparators -> {" ", "\t"}, RecordSeparators -> "\n"}.

```
In[1]:= tokenize["This is a test string."]

Out[1]= {"This", "is", "a", "test", "string."}
```

Uses:

tokenize uses the functions shown in Table 69.167.

Context Name	Function Name
Utilities'OptionProcessing'	FilterOptions

Table 69.167: The Functions and Their Contexts Used by **tokenize**.

69.18.2 transformMma3D[]

Context Name:

Translators'TransformMma3D'

Usage Message:

Translator function. Takes a Mathematica graphics object and translates it to a form suitable for transform3D[]. It takes the result of the transformation and translates it back to a Mathematica primitive object represenatation. Can work with Point, Line, Cuboid, Text, Graphics3D, and other higher level list representations of all the above.

Uses:

transformMma3D uses the functions shown in Table 69.168.

Context Name	Function Name
Translators'TransformMma3D'	transformMma3D
Graphics'Graphics3D'	transform3D

Table 69.168: The Functions and Their Contexts Used by **transformMma3D**.

Is Used In:

transformMma3D is used in the functions shown in Table 69.169.

Context Name	Function Name
OpticalComponents'draw'	draw
OpticalComponents'Icons'	laser
Graphics'Graphics3D'	orient
OpticalComponents'Icons'	stand
Translators'TransformMma3D'	transformMma3D
Graphics'Shapes'	truncatedCone
Graphics'Shapes'	wavePlate

Table 69.169: The Functions and Their Contexts Making Use of **transformMma3D**.

69.18.3 transform3D[]

Context Name:

Graphics'Graphics3D'

Usage Message:

Apply the operation in sequence to an edge list of the object. Options[transform3D] = {ShowSteps -> True/False}. False (default) intermediate versions of the object are not returned. True otherwise.

Uses:

transform3D uses the functions shown in Table 69.170.

Context Name	Function Name
Graphics'Graphics3D'	operationToMatrix

Table 69.170: The Functions and Their Contexts Used by **transform3D**.

Is Used In:

transform3D is used in the functions shown in Table 69.171.

Context Name	Function Name
Graphics'Graphics3D'	orient
Graphics'Graphics3D'	orientBoundingBox
Translators'TransformMma3D'	transformMma3D

Table 69.171: The Functions and Their Contexts Making Use of **transform3D**.

69.18.4 translateNotebook[]

Context Name:

Translators'NotebookToPackage'

Usage Message:

This is an overloaded function. For one definition translateNotebook[notebookFile_String, translations_?MatrixQ, options___], it takes a notebook file and a table of "translations" of the form {outputFile, cellType, translationFunction}, applies translationFunction to all cells of "cellType" and stores results in the file "outputFile". For the definition translateNotebook[notebookFile_String, outputFile_String, cellType_String, translationFunction_, options___], it applies "translationFunction" to all cells of type "cellType" and stores results in "outputFile". For the definition translateNotebook[notebookFile_String, outputFile_String, translations _?MatrixQ, options___], it takes a notebook file and a table of translations of the form {cellType, translationFunction}, applies translationFunction to all cells of "cellType" and stores all results in one file called "outputFile". Options[translateNotebook] = {CellSeparator, PreserveCellHeaders, OutputPageWidth, PreserveFontInfo, Echo, AppendToOutputFile}. CellSeparator (default = ""), Specify separator between cells in output file. Preserve-

CellHeaders (default = If one cell then False else True), Have separate cells in output file or all merged into one. OutputPageWidth (default = Infinity), The word wrap width of output file. PreserveFontInfo (default = True), Preserve italics, bold, colors etc. Echo (default = False), echo results to screen too. AppendToOutputFile (default = False), do not delete output file if it exists.

In[1]:=
```
(* Extract all input and output cells from 3Doperations.ma file,
converting all characters to upper case in both kinds of cells.
*) translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
"/tmp/ma2m.m", {{"input",ToUpperCase}, {"output", ToUpperCase}},
Echo -> False, CellSeparator -> "\n"]
```

Out[1]=
```
/tmp/ma2m.m
```

In[2]:=
```
(* Extract all subsection cells from 3Doperations.ma file,
printing characters in the reverse direction.  *)
translateNotebook["OptiCAD/Packages/Graphics/3Doperations.ma",
"/tmp/ma2m.m", "subsection", StringReverse, Echo -> True,
CellSeparator -> "\n"]
```

Out[2]=
```
/tmp/ma2m.m
```

Uses:

translateNotebook uses the functions shown in Table 69.172.

Context Name	Function Name
Translators'NotebookToPackage'	translateNotebook

Table 69.172: The Functions and Their Contexts Used by **translateNotebook.**

Is Used In:

translateNotebook is used in the functions shown in Table 69.173.

Context Name	Function Name
Translators'NotebookToPackage'	createExamplesNotebook
Translators'NotebookToPackage'	createFunctionsFile
Databases'Dependencies'	obtainDependenciesForFile
Translators'NotebookToPackage'	translateNotebook

Table 69.173: The Functions and Their Contexts Making Use of **translateNotebook**.

69.18.5 truncatedCone[]

Context Name:

Graphics'Shapes'

Usage Message:

Returns a segment of a conical object using Mathematica Graphics primitives.

```
In[1]:= truncatedCone[]

Out[1]= -Graphics-
```

Uses:

truncatedCone uses the functions shown in Table 69.174.

Context Name	Function Name
Graphics'Shapes'	disk
Translators'TransformMma3D'	transformMma3D

Table 69.174: The Functions and Their Contexts Used by **truncatedCone**.

Is Used In:

truncatedCone is used in the functions shown in Table 69.175.

Context Name	Function Name
Graphics'Shapes'	cone
Graphics'Shapes'	cylinder
OpticalComponents'Icons'	stand

Table 69.175: The Functions and Their Contexts Making Use of **truncatedCone**.

69.19 Functions Starting with U

69.19.1 unixLikeFind[]

Context Name:

Utilities'SystemsProgramming'

Usage Message:

Recursively descends the directory hierarchy for each pathname in the pathname-list, seeking files that match a boolean (logical) expression (predicate) and applies action

In[1]:=
```
unixLikeFind[Directory[], True&, Print]
```

Out[1]=
```
OptiCAD/Packages
```

Uses:

unixLikeFind uses the functions shown in Table 69.176.

Context Name	Function Name
Utilities'SystemsProgramming'	unixLikeFind

Table 69.176: The Functions and Their Contexts Used by **unixLikeFind**.

Is Used In:

unixLikeFind is used in the functions shown in Table 69.177.

Context Name	**Function Name**
Utilities'SystemsProgramming'	clearTreeIndex
Utilities'SystemsProgramming'	generateTreeIndex
Databases'Dependencies'	getAllFunctionNames
Utilities'SystemsProgramming'	unixLikeFind

Table 69.177: The Functions and Their Contexts Making Use of **unixLikeFind**.

69.19.2 uses[]

Context Name:

Databases`Dependencies`

Usage Message:

Returns from functionLists all elements used in functionName of contextName.

Options[uses] = {RecursionInfo -> True/False}. True (default) forces check for recursion dependency;

False otherwise.

In[1]:=
```
uses["myContext", {"f[]", "a = p[q];
b = m[1];
c = s[p];
d = f[b]"}, {{"cp", {"p"}}, {"cms", {"m", "s"}}, {"cxy", {"x",
"y"}}, {"myContext", {"f"}}}]
```

Out[1]=
```
{{{"myContext", "f[]"}, {"cp", "p"}}, {{"myContext", "f[]"}, {"cms",
"m"}}, {{"myContext", "f[]"}, {"cms", "s"}}, {{"myContext", "f[]"},
{"myContext", "f"}}}
```

In[2]:=
```
uses["myContext", {"f[]", "a = p[q];
b = m[1];
c = s[p];
d = f[b]"}, {{"cp", {"p"}}, {"cms", {"m", "s"}}, {"cxy", {"x",
"y"}}}, RecursionInfo -> False]
```

Out[2]=
```
{{{"myContext", "f[]"}, {"cp", "p"}}, {{"myContext", "f[]"}, {"cms",
"m"}}, {{"myContext", "f[]"}, {"cms", "s"}}}
```

Uses:

uses uses the functions shown in Table 69.178.

Is Used In:

uses is used in the functions shown in Table 69.179.

Context Name	Function Name
Utilities'StringOperations'	hasTokenQ
Utilities'StringOperations'	quote
Databases'Dependencies'	selfDependencyQ

Table 69.178: The Functions and Their Contexts Used by **uses**.

Context Name	Function Name
Databases'Dependencies'	obtainDependenciesForFile

Table 69.179: The Functions and Their Contexts Making Use of **uses**.

69.20 Functions Starting with V

69.20.1 vectorAtAnAngle[]

Context Name:

Geometry'Geometry'

Usage Message:

This function takes as inputs a ray, and an angle (theta), and produces as output the unit rays that make the angle (theta) with the given vector. The input vector is represented by {{x0,y0}, {a,b}} starting from the point {x0,y0} in direction of the unit vector {a,b}.

```
In[1]:= N[ vectorAtAnAngle[{{1, 1}, {1, 1}}, 90 Degree] ]

Out[1]= {{{1., 1.}, {-0.707107, 0.707107}}, {{1., 1.}, {0.707107, -0.707107}}}
```

69.20.2 ViewArchitecture[]

Context Name:

Simulator‘Simulator‘

Usage Message:

Shows the generated architecture using user specified options.

Options[ViewArchitecture] = {AmbientLight -> GrayLevel[0.], AspectRatio -> Automatic, Axes -> False, AxesEdge -> Automatic, AxesLabel -> None, AxesStyle -> Automatic, Background -> Automatic, Boxed -> True, BoxRatios -> Automatic, BoxStyle -> Automatic, ColorOutput -> Automatic, DefaultColor -> Automatic, Epilog -> {}, FaceGrids -> None, Lighting -> True, LightSources -> {{{1., 0., 1.}, RGBColor[1, 0, 0]}, {{1., 1., 1.}, RGBColor[0, 1, 0]}, {{0., 1., 1.}, RGBColor[0, 0, 1]}}, PlotLabel -> None, PlotRange -> Automatic, PlotRegion -> Automatic, Plot3Matrix -> Automatic, PolygonIntersections -> True, Prolog -> {}, RenderAll -> True, Shading -> True, SphericalRegion -> False, Ticks -> Automatic, ViewCenter -> Automatic, ViewPoint -> {1.3, -2.4, 2.}, ViewVertical -> {0., 0., 1.}, DefaultFont :> $DefaultFont, DisplayFunction :> $DisplayFunction}

Uses:

ViewArchitecture uses the functions shown in Table 69.180.

Context Name	Function Name
Utilities‘OptionProcessing‘	FilterOptions

Table 69.180: The Functions and Their Contexts Used by **ViewArchitecture**.

Is Used In:

ViewArchitecture is used in the functions shown in Table 69.181.

Context Name	**Function Name**
Simulator'Simulator'	ShowArchitecture

Table 69.181: The Functions and Their Contexts Making Use of **ViewArchitecture**.

69.21 Functions Starting with W

69.21.1 wavePlate[]

Context Name:

Graphics‘Shapes‘

Usage Message:

Returns an icon for a wavePlate using Mathematica Graphics primitives.

```
In[1]:= wavePlate[]

Out[1]= -Graphics-
```

Uses:

wavePlate uses the functions shown in Table 69.182.

Context Name	Function Name
Graphics‘Shapes‘	disk
Translators‘TransformMma3D‘	transformMma3D

Table 69.182: The Functions and Their Contexts Used by **wavePlate**.

Is Used In:

wavePlate is used in the functions shown in Table 69.183.

Context Name	Function Name
OpticalComponents‘Icons‘	HalfWavePlate
OpticalComponents‘Icons‘	QuarterWavePlate

Table 69.183: The Functions and Their Contexts Making Use of **wavePlate**.

69.21.2 writeTo[]

Context Name:

Translators‘NotebookToPackage‘

Usage Message:

Translates the input to another form given by a translation table. The specific translation depends on the Contents -> contentsType specified as an option.

Uses:

writeTo uses the functions shown in Table 69.184.

Context Name	Function Name
Utilities‘StringOperations‘	listToString

Table 69.184: The Functions and Their Contexts Used by **writeTo**.

Is Used In:

writeTo is used in the functions shown in Table 69.185.

Context Name	Function Name
Translators‘NotebookToPackage‘	createExamplesNotebook
Translators‘NotebookToPackage‘	createPackage

Table 69.185: The Functions and Their Contexts Making Use of **writeTo**.

69.22 Functions Starting with X

69.22.1 xAxisQ[]

Context Name:

Geometry'Predicates'

Usage Message:

Checks whether the given line is the same as the x-axis.

```
In[1]:= xAxisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]

Out[1]= False

In[2]:= xAxisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]

Out[2]= True
```

Is Used In:

xAxisQ is used in the functions shown in Table 69.186.

Context Name	Function Name
Graphics'Graphics3D'	operationToMatrix

Table 69.186: The Functions and Their Contexts Making Use of **xAxisQ**.

69.23 Functions Starting with Y

69.23.1 yAxisQ[]

Context Name:

Geometry'Predicates'

Usage Message:

Checks whether the given line is the same as the y-axis.

In[1]:=
```
yAxisQ[line[point[{1, 0, 0}], point[{0, 0, -1}]]]
```

Out[1]=
```
False
```

In[2]:=
```
yAxisQ[line[point[{0, 0, 0}], point[{0, 1, 0}]]]
```

Out[2]=
```
True
```

Is Used In:

yAxisQ is used in the functions shown in Table 69.187.

Context Name	Function Name
Graphics'Graphics3D'	operationToMatrix

Table 69.187: The Functions and Their Contexts Making Use of **yAxisQ**.

69.24 Functions Starting with Z

69.24.1 zAxisQ[]

Context Name:

Geometry`Predicates`

Usage Message:

Checks whether the given line is the same as the z-axis.

```
In[1]:= zAxisQ[line[point[{0, 0, 0}], point[{0, 0, -1}]]]

Out[1]= True

In[2]:= zAxisQ[line[point[{0, 0, 0}], point[{1, 0, 0}]]]

Out[2]= False
```

Is Used In:

zAxisQ is used in the functions shown in Table 69.188.

Context Name	Function Name
Graphics`Graphics3D`	operationToMatrix

Table 69.188: The Functions and Their Contexts Making Use of **zAxisQ**.

Bibliography

[1] Ghafoor A., Guizani M., and Sheikh S. "All-Optical Circuit-switched Multistage Interconnection Networks". *IEEE, Journal on Selected Areas of Communication*, 9(8):1218–1226, October 1991.

[2] Alferd V. Aho, Ravi Sethi, and Jeffery D. Ullman. *"Compilers: Principles, Techniques, and Tools"*. Addison-Wesley, 1986.

[3] Nabeel Al-Mosli. "Design & Implementation of a 3D-Graphics System: An Interactive Hierarchical Modelling Approach". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, June 1995.

[4] John Backus. "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs". *Communications of the ACM*, 21(8):613–641, August 1978.

[5] Roger Bailey. *"Functional Programming with HOPE"*. Ellis Horwood Series in Computers and Their Applications. Ellis Horwood ltd., 1990.

[6] Henri E. Bal and Dick Grune. *"Programming Language Essentials"*. Addison-Wesley, 1994.

[7] Partha P. Banerjee and Ting-Chung Poon. *"Principles of Applied Optics"*. Asken Associates Incorporated Publishers, 1991.

[8] D. Barnhard. "Lens Lab 3D". Technical report, 1993.

[9] Alain Bergeron, Henri H. Arsenault, and Denis Gingras. "Single-Rail Translation-Invariant Optical Associative Memory". *Applied Optics*, 34(2):352–357, January 1995.

[10] S. Bian, K. Xu, and J. Hong. "Near Neighbouring Neurons Interconnected Neural Network". *Optics Communications*, 76(3,4):199–202, May 1990.

[11] K. Brenner and A. Huang. "Optical Implementation of the Perfect Shuffle Interconnection". *Applied Optics*, 27(1):135–137, January 1988.

[12] L. Z. Cai and T. H. Chao. "Optical Image Subtraction using a LCTV SLM and White Light Imaging Grating Interferometer". *Journal of Modern Optics*, 37(6):1127–1138, 1990.

[13] H. J. Caulfield. "Improved Relaxation Processor for Parallel Solution of Linear Algebric Equations". *Applied Optics*, 29(20):2978–2981, July 1990.

[14] H. J. Caulfield, J. Shamir, J. E. Ludman, and P. Greguss. "Reversibility and Energetics in Optical Computing". *Optics Letters*, 15(16):912–914, August 1990.

[15] T. J. Cloonan et al. "Architectural Issues Related to the Optical Implementation of an EGS Network based on Embedded Control". *Optical Quantum Electronics*, 24:S415–S442, 1992.

[16] T. J. Cloonan and M. J. Herron. "Optical Implementation and Performance of One-Dimensional and Two-Dimensional Trimmed Augmented Data-Manipulator Networks for Multi-Processor Computer Systems". *Optical Engineering*, 28(4):305–314, 1989.

[17] McAulay Alastair D. *"Optical Computer Architectures: The Application of Optical Concepts to Next Generation Computers"*. John Wiley and Sons Inc., 1991.

[18] S. Dasgupta. "The Structure of Design Processes". *Advances in Computers*, 28:1–67, 1989.

[19] A. K. Datta, A. Basuray, and S. Mukhopadhyay. "Arithmetic Operations in Optical Computations Using a Modified Trinary Number System". *Optics Letters*, 14(9):426–428, May 1989.

[20] E. R. Davies. *"Machine Vision: Theory, Algorithms, Practicalities"*. Academic Press, 1990.

[21] L. Dron. "Multiscale Veto Model: A Two-Stage Analog Network for Edge Detection and Image Reconstruction". *International Journal of Computer Vision*, 11(1):45–61, August 1993.

[22] P. Egbert and W. Kubitz. "Application Graphics Modeling Support Through Object Orientation". *COMPUTER*, pages 84–90, Oct 1992.

[23] M. T. Fatehi. "Optical Flip-Flops and Sequential Logic Circuits Using LCLV". *Applied Optics*, 23(13):2163–2171, 1 July 1984.

[24] C. Ferrera and C. Vazquez. "Anamorphic Multiple Matched Filter for Character Recognition, Performance with Signals of Equal Size". *Journal of Modern Optics*, 37(8):1343–1354, 1990.

[25] M. A. Flavin and J. L. Horner. "Average Amplitude Matched Filter". *Optical Engineering*, 29(1):31–37, January 1990.

[26] D. Foley and A. van Dam. *"Fundamentals of Interactive Computer Graphics"*. Addison-Wesley, 1990.

[27] A. Ghosh and P. Paparao. "Matrix Preconditioning: A Robust Operation For Optical Linear algebra Operations". *Applied Optics*, 26(14):2734–2737, July 1987.

[28] G. R. Gindi, A. F. Gmitro, and K. Parthasarathy. "Hopfield Model Associative Memory with Nonzero Diagonal Terms in Memory Matrix". *Applied Optics*, 27(1):129–134, January 1988.

[29] K. Hara, K. Kojima, K. Mitsunga, and K. Kyuma. "Optical Flip-Flop Based on Parallel Connected Algaas/gaas PNPN Structure". *Optics Letters*, 15(13):749–751, July 1990.

[30] P. Henderson. "Functional Programming, Formal Specification, and Rapid Prototyping". *IEEE Transactions on Software Engineering*, SE-10(5):241–250, 1986.

[31] H. S. Hinton et al. "Free-Space Digital Optical Systems". *Proceedings of the IEEE*, 82(11):1632–1649, November 1994.

[32] Ellis Horowitz and Sartaj Sahni. *"Fundamentals of Computer Algorithms"*. Computer Science Press, 1978.

[33] H. Huang, L. Lilu, and Z. Wang. "Parallel Multiple Matrix Multiplication Using an Orthogonal Shadow Casting and Imaging System". *Optics Letters*, 15(19):1085–1087, October 1990.

[34] M. N. Islam. "All Optical Cascadable NOR Gate With Gain". *Optics Letters*, 15(8):417–419, April 1990.

[35] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[36] J. Jahns. "Optical Implementation of the Banyan Network". *Optics Communications*, 76(5,6):321–324, May 1990.

[37] J. Jahns and B. A. Brumback. "Integrated Optical Shift and Split Model Based on Planar Optics". *Optics Communications*, 76(5,6):318–320, May 1990.

[38] J. Jahns and M. Murdocca. "Crossover Networks and their Optical Implementations". *Applied Optics*, 27(15):3155–3160, August 1988.

[39] J. Jahns and S. J. Walker. "Imaging with Planar Optical Systems". *Optics Communications*, 76(5,6):313–317, May 1990.

[40] H. F. Jordan. *"Digital Optical Computers at Boulder, Center for Optoelectronic Computing Systems"*. University of Colorado, Boulder, 1991.

[41] A. Kemper and M. Wallrath. "An Analysis of Geometric Modeling in Database Systems". *ACM Computing Surveys*, 19(1):47–91, March 1987.

[42] E. Kerbis, T. J. Cloonan, and F. B. McCormick. "An All-Optical Realization of a 2×1 Free-Space Switching Node". *IEEE Photon Technology Letters*, 2(8):600–602, August 1990.

[43] B. Kliewer. "HOOPS: Powerful Portable 3D-Graphics". *BYTE*, pages 193–194, July 1989.

[44] Kubota Pacific Computer Inc. *"Dore Reference Manual"*, 1991.

[45] A.O. Lafi. "The Design and Implementation of a Structured Programming Language for the Description of Optical Architectures". Master's thesis, King Fahd University of Petroleum and Minerals, Information and Computer Science Department, Dhahran 31261, Saudi Arabia, January 1992.

[46] P. Lalanee, J. Taboury, and P. Chavel. "A Proposed Generalization of Hopfields Algorithm". *Optics Communications*, 63(1):21–25, July 1987.

[47] A. L. Lentine et al. "Symmetric Self-Electrooptic Effect Device: Optical Set-Reset Latch, Differential Logic Gate and Differential Modulator/Detector". *IEEE Journal of Quantum Electonics*, 25(8):1928–1936, August 1989.

[48] John R. Levine, Tony Mason, and Doug Brown. *"Lex & Yacc"*. OReilly & Associates, Inc., 1992.

[49] S. Lin, L. Liu, and Z. Wang. "Optical Implementation of the 2D-Hopfield Model for a 2D-Associative Memory". *Optics Communications*, 70(2):87–91, Feburary 1989.

[50] A. W. Lohmann, W. Stroke, and G. Stucke. "Optical Perfect Shuffle". *Applied Optics*, 25:1530, 1986.

[51] Guizani M. "Picosecond Multistage Interconnection Network Architectures for Optical Computing". *Applied Optics*, 33(8):1587–1599, March 1994.

[52] M. A. Memon, S. Ghanta, and S. Guizani. "An Optical Architecture for Edge Detection". In *Seventh IASTED International Conference on Parallel and Distributed Computing and Systems.* Georgetown University, Washington D.C., 1995.

[53] F. L. Miller, J. Maeda, and H. Kubo. "Template Based Method of Edge Linking using a Weighted Decision". In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1808–1815, 1993.

[54] M. Mirsalehi and T. K. Gaylord. "Analytic Expressions for the Sizes of Logically Minimized Truth Tables for Binary Addition and Subtraction". *Applied Optics*, 29(23):3339–3344, August 1990.

[55] R. L. Morrison. "Symmetries that Simplify the Design of Spot Array Phase Gratings". *Journal of Optical Society of America A*, 1992.

[56] S. Mukhopadhyay. "An Optical Conversion System: From Binary to Decimal and decimal to Binary". *Optics Communications*, 76(5,6):309–312, May 1990.

[57] M. Murdocca. *"A Digital Design Methodology for Optical Computing"*. The MIT Press, 1990.

[58] V. K. Murty. "Exact Parallel Matrix Inversion Using para-Hensel Codes with Systolic Processors". *Applied Optics*, 27(10):2022–2024, May 1988.

[59] J. A. Neff. "Major Initiatives for Optical Computing". *Optical Engineering*, 26(1):002–009, January 1987.

[60] M. Oita, J. Ohta, S. Tai, and K. Kyuma. "Optical Implementation of Large Scale Neural Networks Using a Time Division Multiplexing Technique". *Optics Letters*, 15(4):227–229, Feburary 1990.

[61] D. V. Pentelic. "Optical Computation of Determinants". *Optics Communications*, 64(5):421–424, 1987.

[62] D. Peri. "Optical Implementation of a Phase Retrieval Algorithm". *Applied Optics*, 26(9):1782–1785, May 1987.

[63] J.P. Pratt. *"HATCH Users Manual"*. Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[64] J.P. Pratt and V.P. Heuring. "A Methodology for the Design of Continuous-Dataflow Synchronous System". Technical report, Center for Optoelectronic Computing Systems, University of Colorado, Boulder, 1989.

[65] M. E. Prise et al. "Design of an Optical Digital Computer". In W. Firth, N. Peyhambarian, and A. Tallet, editors, *Optical Bistability IV*, pages C2–15–C2–18. Les Editions de Physique, 1988.

[66] A. Requicha. "Representation for Rigid Solids: Theory, Methods and Systems". *ACM Computing Surveys*, 12(4):437–463, Dec 1980.

[67] Bahaa E. A. Saleh and Malvin Carl Teich. *"Fundamentals of Photonics"*. John Wiley & Sons Inc., 1991.

[68] J. M. Senior and S. D. Cusworth. "Wavelength Division Multiplexing in Optical Fiber Sensor Systems and Networks: A Review". *Optics and Laser Technology*, 22(2):113–126, 1990.

[69] Ravi Sethi. *"Programming Languages: Concepts and Constructs"*. Addison-Wesley, 1989.

[70] Ben A. Sijtsma. "Requirements for a Functional Programming Enviornment". In Rogardt Heldal, Carsten Kehler, and Philip Wadler, editors, *Functional Programming, Glasgow 1991*, pages 339–346. Springer-Verlag, 1992.

[71] H. A. Simon. *"Science of the Artificial"*. The MIT Press, second edition, 1981.

[72] WRI Staff. "*MathLink* External Communication in *Mathematica*". Technical report, WRI, 1991.

[73] WRI Staff. "The 3-Script File Format". Technical report, WRI, 1991.

[74] WRI Staff. "The *Mathematica* Compiler". Technical report, WRI, 1991.

[75] WRI Staff. "Guide to Standard *Mathematica* Packages". Technical report, WRI, 1993.

[76] N. Streibl. "Beam Shaping with Optical Array Generators". *Journal of Modern Optics*, 36(12):1559–1573, 1989.

[77] Boleslaw K. Szymanski, editor. *"Parallel Functional Languages and Compilers"*. ACM Press Frontier Series. Addison-Wesley, 1991.

[78] David A. Turner, editor. *"Research Topics in Functional Programming"*. The UT Year of Programming series. Addison-Wesley, 1990.

[79] U.S. Air Force. *"VHDL Users Manual"*, 1985.

[80] P. Wegner. "Dimensions of Objected-Oriented Modeling". *COMPUTER*, pages 12–21, Oct 1992.

[81] B. S. Wherret, S. Desmond Smith, F. A. P. Tooley, and A. C. Walker. "Optical Components for Digital Optical Circuits". *Future Generation Computer Systems*, 3:253–259, 1987.

[82] D. R. J. White, K. Atkinson, and J. D. M. Osburn. "Taming EMI in Microprocessor Systems". *IEEE Spectrum*, 22(12):30–37, December 1990.

[83] P. Wisskirchen. *"Object-Oriented Graphics"*. Springer-Verlag, 1990.

[84] S. Wolfram. *"Mathematica: A System for Doing Mathematics by Computer"*. Addison-Wesley, second edition, 1991.

[85] Amnon Yariv. *"Optical Electronics"*. Saunders College Publishing: HBJ College Publishers, 1991.

[86] E. Yee and J. Ho. "Neural Network Recognition and Classification of Aerosol Particle Distributions Measured with a Two-Spot Laser Velocimeter". *Optics Communications*, 74(5):295–300, January 1990.

[87] L. Zhang and L. Liu. "Incoherent Optical Implementation of 2D-Complex Discrete Fourier Transform and Equivalent 4-F System". *Optics Communication*, 74(5):295–300, January 1990.

Vita

Name	:	Atif M. Memon
Date of Birth	:	26 August 1970
Place of Birth	:	Hyderabad, Sindh, Pakistan.

I received the Bachelor of Science degree in Computer Science from the Institute of Computer Science, Foundation for Advancement of Science and Technology, University of Karachi, Karachi, Pakistan in January 1992.

Immediately after BS, I taught for one year at the Institute of Computer Science, Foundation for Advancement of Science and Technology.

I expect to receive Master of Science degree in Computer Science in 1995 from the King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.

Will be joining the Department of Information and Computer Science, KFUPM as a lecturer.

My hobbies include reading, playing with computers, fiddling with electronic gadgets, walking and eating ice-cream.