

On a Layered Object–Space Based Architecture for Interactive Raster Graphics

door

Fons Kuijk

*geboren op 22 mei 1955
te Haarlem*

October 13, 1995

Promotor: Prof. Dr. L.O. Hertzberger (UvA)
Copromotor: Prof. E.H. Blake (University of Capetown)
Faculteit: Wiskunde en Informatica

October 13, 1995

Contents

Contents	iii
Preface	vii
1: Introduction	1
1.1: Graphics and Human-Computer Communication.....	1
1.2: Background	3
1.3: Towards a New Architecture	5
1.4: The STW Research Project on Interactive Raster Graphics	6
1.5: Outline of the Dissertation	8
2: Graphics Systems and Interaction	9
2.1: Interactive Graphics: Concepts and Nomenclature	9
2.1.1: Scene Specification.....	10
2.1.2: Image Synthesis	10
2.1.3: Graphical Input	12
2.1.4: Graphical Interaction	14
2.2: An Overview on Interaction Tasks	16
2.3: Temporal Aspects of Interaction	19
2.3.1: Animate Response	20
2.3.2: Computer Animation	22
2.3.3: Discussion.....	24
2.4: Dealing with Motion	25
2.4.1: Temporal Anti-aliasing	25
2.4.2: Adaptive Image Generation	25
2.5: Conclusions	29
3: Raster Graphics Hardware	31
3.1: Introduction	31
3.2: An Overview on Hardware Elements	32
3.2.1: Display Device.....	32
3.2.2: Display Controller.....	35
3.2.3: Frame Buffer.....	36
3.2.4: Graphics Processor	40

3.3: Concurrency and Graphics Systems.....	41
3.3.1: Introduction.....	41
3.3.2: Architecture Based Classification.....	42
3.3.3: Distribution Based Classification.....	43
3.3.4: Discussion.....	46
3.4: Conclusions	48
4: A Layered Object-Space Based Architecture	49
4.1: Introduction	49
4.2: Abstract Computational Model	51
4.3: Output Representation.....	54
4.4: A Multilayer Computational Model	56
4.5: Discussion	58
4.5.1: Hidden Surface Removal.....	60
4.5.2: Illumination and Shading.....	61
4.5.3: Display Controller.....	62
4.6: Conclusions	62
5: Dealing with Geometry	65
5.1: Introduction	65
5.2: Representation of Geometry: the Domain	68
5.2.1: The Use of Extents.....	68
5.2.2: Area Representation.....	70
5.3: Efficient Storing of Domains.....	78
5.3.1: Slice-List Structure	79
5.3.2: Grid-List Structure	82
5.3.3: Results.....	85
5.3.4: Conclusions.....	103
5.4: Exact Incremental Hidden Surface Removal	105
5.4.1: Binary Operations	106
5.4.2: Basics of the Algorithm	108
5.4.3: Reducing Complexity	109
5.4.4: Referencing.....	109
5.4.5: Adding a Domain.....	110
5.4.6: Removing a Domain	110
5.4.7: Distributed Hidden Surface Removal.....	112
5.4.8: Results.....	113
5.5: Scan-Conversion	113
5.6: Conclusions	114
6: Illumination and Shading of Polygonal Models	115
6.1: Introduction	115
6.2: Illumination & Shading.....	116

6.2.1: Illumination Model	116
6.2.2: Shading Methods	118
6.2.3: Discussion.....	120
6.3: Interpolation Across Polygons	122
6.3.1: Vector Interpolation.....	122
6.3.2: Angular Interpolation.....	124
6.4: Evaluation of the Intensity Components	131
6.5: Discussion	134
7: An Object-Space Display Controller	137
7.1: Introduction	137
7.2: Unfolding the Scan-Conversion process	138
7.3: Architecture of the Display Controller.....	140
7.4: The X-Processor Array — or Difference Engine	141
7.4.1: Basics of the X-Processor Array	142
7.4.2: The Command Set	143
7.4.3: Implementation of the Difference Engine	145
7.4.4: Structural Simulation of the Difference Engine	149
7.4.5: Features and Estimated Performance	152
7.4.6: Virtual Arrays	155
7.5: Driving the Difference Engine.....	158
7.6: Building a Prototype.....	159
7.6.1: System Components	160
7.6.2: Board Level Components	163
7.6.3: Physical Design.....	167
7.7: Results	169
7.8: Discussion	169
8: Conclusions	171
Acknowledgements	175
References	177
Appendix A: Driving the Difference Engine	201
A.1: A Full Colour System.....	201
A.2: Bus Arbitration.....	205
A.3: The Instruction and Colour Buffer	207
A.4: The Y-Processor.....	211
A.5: Conclusions	211

October 13, 1995

Preface

This dissertation is on the development of an interactive raster graphics system for CAD-type of applications which inherently supports direct manipulation of three-dimensional objects. This feature in particular serves to help the user to understand the three-dimensional geometry of virtual objects as if these objects could be observed and manipulated in the real world.

A graphics system is an essential and conceptually separable part of a workstation. It is a key element for the so-called user interface via which interaction between the application program and the user takes place. The success of the user interface concept and the increased capabilities of workstations stimulated more visually oriented communication. However, each new generation of workstations is expected to be able to handle an increased amount of data and facilitate more demanding visualisation and interaction techniques. This forms the motivation for continuous research on interactive three-dimensional graphics systems for workstations.

The aim of the research described in this dissertation has been to develop a new architecture for an interactive graphics workstation, targeted for displaying three-dimensional polygonal models. Interactivity of a graphics system can be improved by speeding-up the image generation pipeline and/or by reducing the amount of calculation needed to generate successive frames. We concentrated on the latter and found that standard implementations of the image generation pipeline do not provide sufficient functionality for interactive applications.

We concluded that the amount of calculation can be kept low by providing precisely that functionality required for interaction. We state that a layered structure and a consistent raster independent object-space approach are key elements of a more appropriate architecture. The layered structure of the system we propose allows us to make optimal use of incremental algorithms. However, a consequence of this approach is that we have to make use of less common image generation algorithms and non-standard image generation hardware.

For instance, the raster independent object-space paradigm implies that the system requires object-space hidden surface removal. We found that object-space hidden surface removal algorithms can confine update activities needed for incremental changes to just those objects of which the visibility is changed.

Another consequence is that there is no mandate for a frame buffer. At high update rates a frame buffer becomes a bottleneck and does not bring any extra

functionality for interaction. (For a user, the smallest pick primitive is a visible object, not a pixel.) Leaving out a frame buffer implies that the display controller has to produce the pixels of fully shaded primitives — that may even be illuminated by multiple light sources — at a rate of about 80 Mhz, depending on resolution.

We came up with a two-tier modular structured display controller. A high quality illumination model was formulated that maps directly onto this architecture. Worth noting is that the functionality of the custom modules of the display controller is based on an abstraction of the fundamental aspects of image generation. As a result of this, the architecture does not restrict to a specific type of primitive or a specific type of rendering. Even more so, the architecture has proven to be useful for image reconstruction as well [Marais93].

October 13, 1995

Introduction

Synopsis.

Communication between a computer and a user is one of the fundamental issues in the discipline of computing. In this chapter the role that interactive graphics plays in human-computer communication is highlighted. The design of interactive graphics systems should not be focussed on computing resources alone, since this often leads to inefficient brute force systems that lack the flexibility required for high speed interaction. This notion triggered the proposal of a research project named "A new Architecture for Interactive Raster Graphics based on VLSI". The objectives and the context of this project are described. Subsequently the outline of the dissertation is sketched.

1.1. Graphics and Human-Computer Communication

"This area deals with efficient transfer of information between humans and computers and with information structures that reflect human conceptualisation. Fundamental questions include: What are efficient methods of representing objects and automatically creating pictures for viewing? What are effective methods for receiving input or presenting output? How can the risk of misperception and subsequent human error be minimised? How can graphics and other tools be used to understand physical phenomena through information stored in data sets?" [Denning89]

This quote refers to the area of Human-Computer Communication. It is one of the nine subareas of the discipline of computing identified in the *Report of the ACM Task Force on the Core of Computer Science* [Denning89]. The fundamental questions mentioned in this quote illustrate the significance of pictures as an information transfer medium from computers to humans.

The important role visual information exchange plays in Human-Computer Communication is justified by the characteristics of human vision. The human visual system is a most powerful image cognition machine that is able to perceive, analyse, classify and evaluate a great deal of information in real-time [Groß91]. The processing power of today's super computing systems does not even come near

to the processing power of the human visual system. The human visual system has a high input bandwidth, a well developed ability to recognise patterns and dynamic changes, and an adequate three-dimensional reconstruction capability. Thus, communication via pictures is a natural and powerful way to present information to people.

"Interactive computer graphics thus permits extensive, high-bandwidth user-computer interaction. This significantly enhances our ability to understand data, to perceive trends, and to visualise real or imaginary objects —indeed, to create "virtual worlds" that we can explore from arbitrary points of view. By making communication more efficient, graphics makes possible higher-quality and more precise results or products, greater productivity, and lower analysis and design costs." [Foley90]

A graphics system¹⁾ gives access to this high performance computer-to-user communication channel by providing the facilities for a computer application to generate pictures. Hardware facilities for graphics based human-computer communication — a high resolution display and a fair amount of computing resources — are standard features of today's desk-top workstations.

The raster graphics system became a key element of the user interface via which interaction of the application program and the user takes place. The 'desk-top metaphor' is an example of a first generation 2D graphics user interface. It allows the user to manipulate electronic documents and folders via windows on the screen much like the user is used to handle hardcopy documents and folders on his desk. These first generation window systems clearly demonstrated the efficiency of direct manipulation and other types of visualised interactive control of computer applications. This successful debut created a demand that positively influenced the evolution in workstation technology we saw in the last decade. In turn, the increasing performance of workstations triggered a demand for higher level and more demanding interaction. As a result we find that today the imperative need for high speed interactive computer graphics is discerned.

"Computer graphics has always been an expensive proposition and its users a demanding, unsatisfied lot — the picture never got on the screen fast enough, or later, never moved fast enough once it got on the screen, and then, the picture was never sharp enough, never realistic enough." [Fuchs88]

Nowadays, an almost standard requirement is that a workstation should be able to render realistic pictures of user manipulable three-dimensional objects in real-time. Realistic pictures involve high resolution display systems and advanced shading models. Rendering this type of highly complex models (>100K coordinates) in

¹⁾ In this dissertation the concept 'graphics system' is taken to be an abstraction; in this view a graphics system may share resources with the application and the term graphics system relates to both hardware and software elements.

real-time (typically <0.1 seconds) takes considerably more computing resources than offered by present day workstations.

The need for realistic graphics and highly interactive systems, however, is growing in application areas such as design, engineering, scientific visualisation, real-time control systems, simulation, animation and interactive education systems. As a result of experiments with visualisation techniques and an increased understanding of the characteristics of visual perception the added value of dynamically varying pictures has been shown.

Based on these trends it is to be expected that visual communication will continue to play a leading role in human-computer communication. To satisfy the 'demanding, unsatisfied lot', it is not sufficient to rely on developments in hardware technologies alone. In fact these developments put extra strain on graphics systems designers, since increased capabilities of computing systems increase the complexity of applications and consequently increase the demand on performance of graphics systems even more. This justifies the ongoing research effort on interactive graphics systems.

1.2. Background

Within the CWI department of Interactive Systems, research has been focussed on fundamental problems occurring in communication between a computer system and the real world, and on the design and construction of systems that provide the mechanism for this communication.

For the group Computer Graphics —one of the three research groups within the department— communication by means of pictures has been and still is the leading theme. This group has a long tradition in research on structuring of information describing pictures. Throughout the years, this resulted in the design of ILP (Intermediate Language for Pictures) [Hagen80], involvement in the design of GKS (Graphical Kernel System) [ISO85b] and research activities for a STW funded project titled "Raster Graphics Facilities" (NWI14.0130).

Within the department of Interactive Systems, a closely related research theme —human computer interaction— was undertaken to improve basic support for interaction in graphics systems. This research included aspects like control flow, interaction techniques, resource management and graphics support. It resulted in the development of DICE (DIALOGUE CELL system), a user interface management system [Schouten90].

In the course of these projects it became apparent that the characteristics of ergonomically acceptable user interfaces are correlated to the characteristics of the underlying hardware. From a user interface implementor's perspective, existing architectures rarely facilitate the appropriate feedback mechanisms in a natural way. This is due to the fact that hardware architectures of current generation 3D raster graphics systems scarcely differ from architectures of 2D raster graphics systems. Systems may be equipped with Z-buffer hardware or transformation coprocessors, however, that does not imply that the evolution to genuine 3D architectures is

completed. One can state that architectures of today's systems and the algorithms used are too much image (= 2D) oriented. Genuine 3D architectures will have to be different. In a sense the success of existing raster graphics systems seems to turn against itself: it is hard to change directions²⁾. It is also hard to fulfill the diversity of interaction requirements of the numerous types of applications we find today. Most existing 'high end' graphics systems are too specific and lack the flexibility required for high speed interaction. This lack of flexibility is manifest in both the static nature of the application interface and the static semantics of the image generation pipeline.

Even the most successful graphics systems provide a fixed collection of primitives, attributes, and storage schemes. A static application interface leads to tedious specifications for those applications that use primitives which do not fall in a system's fixed collection. A lengthy decomposition process must be done by the application in order to use the primitives available in a graphics package. Not only does this make the application more complex, it also degrades the interactive behaviour of the system in case the decomposition becomes part of the interaction process.

All graphics systems define a pipeline through which a primitive passes when it is rendered. The attribute binding and storage of intermediate representations are fixed by the static semantics of the pipeline. Facilities may be provided to access the pipeline at intermediate stages, but this does not guarantee that the amount of calculation involved in interaction tasks are minimised. Intermediate representations may for instance force recalculation of geometric aspects of the primitive in situations where only attributes have been changed. This suggests a reappraisal of the 3D raster graphics pipeline and of the representation of the primitives that pass the pipeline.

This is the setting from which a project proposal emerged. The objective of the proposal, titled "A new Architecture for Interactive Raster Graphics based on VLSI", was the design and realisation of an experimental interactive 3D graphics system. The research focuses mainly on the semantics of the image generation pipeline. The project would familiarise us with less academic but weighty issues of hardware design and at the same time produce a competitive system which would enable further experiments on interaction techniques.

²⁾ The "2D trap" is being dug deeper and deeper. Adding 3D to a 2D frame buffer was easiest via the Z-buffer. Then came the A-buffer. Finally everything became texture mapping — even the light calculations. The grave has been dug, the coffin is being lowered. It, must be admitted, is a very pretty and realistic coffin and the wisps of mist around the graveyard are very well done. But we want to start out in 3D (i.e., no flat tree bitmaps that are directed towards the viewpoint).

1.3. Towards a New Architecture

The STW³⁾ funded project "A new Architecture for Interactive Raster Graphics based on VLSI" (CWI79.1249) has been assigned to the CWI from September 1986 up to April 1993. The project involved the design and implementation of a prototype highly interactive raster graphics workstation. This dissertation covers parts of the work done for this STW funded project. A research group from the Laboratory for Network Theory of the University of Twente participated in this project, in particular for the design of the VLSI components. In a later stage people from the faculty of Mathematics and Computer Science from the University of Amsterdam succeeded in integrating the VLSI components in a test environment.

Central to the approach of the project is an explicit concern about the underlying data structures used to represent geometric objects. This originates from the conviction that only through careful design of appropriate graphic data structures and algorithms one can profitably map software tasks into hardware, specifically into VLSI. These data structures have to be simple to be useful for implementation in VLSI. The same strategy that produced the RISC revolution should be applied here: first analyse, then strive for simplicity.

In the view of this project, interaction is facilitated by exploiting an optimised architecture of the image generation pipeline, associated by a new type of graphics primitive, introduced in [Hagen86b]. The primitive envisioned, named *pattern*, describes the visual aspects of an area of the image. The representation of a pattern is optimised for the most critical operations that have to be performed on it. New functionality is embedded in the system's architecture to support feedback for interaction. The functionality that is supported by the system will reflect functionality offered by PHIGS [ISO89] and HIRASP [Teunissen88] but needs extensions to be able to facilitate more advanced interaction techniques. Some of the user interface aspects of this functionality originates from the aforementioned STW project "Raster Graphics Facilities".

"By and large, the designs of commercially available products have been motivated primarily by a bottom-up concern with cost-effective hardware technology that meets performance constraints and not by a more general top-down, 'software-first' strategy based on user requirements. A proper top-down strategy would not only include cost-effectiveness and performance, but would also take into account programmability and extensibility. The preoccupation with hardware is understandable in the light of the concern with performance, as measured only by the number of elements processed per refresh cycle for a flicker-free display." [Foley82]

From a user requirements perspective the design of a graphics system should preferably be based on a top-down, software-first strategy. In a proper design cycle chronologically the following activities are undertaken:

³⁾ STW stands for "Stichting voor de Technische Wetenschappen" (that is: Foundation for Technical Sciences).

- characterisation of algorithms, data-structures and requirements;
- construction of a symbolic model;
- simplification and verification of the symbolic model;
- design of a physical system;
- implementation of the physical system;
- testing and maintenance.

Unfortunately, following these steps does not guarantee a 'first-time-right' design. The transition from software (the symbolic model) to hardware (the physical system) is a delicate step. One has to bridge the significant gap between our conception of notations and our conception of devices.

"In software the gap is seen in the comparison of the theoretical complexity of a program and the empirical performance. Calculations based on models at different levels of abstraction can be refined to make estimates of the number of clock cycles required, en route to estimates of run-times in seconds that may be tested. (...)

In hardware the gap is seen in the essential role that timing and performance play in the many notions of specifications and correctness criteria for devices. In a sense, each model of computation for hardware design attempts to bridge, or more accurately, hide this gap." [McEvoy90]

The raw performance of interactive graphics systems is often decisive for the success or failure of a product, for which reason all noteworthy graphics systems are implemented using leading edge technology. Computer graphics has always been strongly influenced by hardware technology. Each step in the evolution of hardware technology made it possible to improve algorithms, improve shading models and increase the complexity of representations.

Due to the everlasting demand for more system performance, a top-down design strategy quickly tends to lead to a design of the 'ultimate' graphics system. Consequently, it is unlikely that for state-of-the-art systems a software-first strategy results in first-time-right abstractions that can be implemented in a cost-effective system, or even worse, that can be implemented at all. Therefore, the realisation of a system (in software as well as in hardware) has to be an iterative process.

1.4. The STW Research Project on Interactive Raster Graphics

The following stages of the project could be distinguished.

The first part of the project adhered to the software-first strategy. Activities included characterisation of requirements, data-structures and algorithms, in particular the algorithms involved in creation, manipulation and display of patterns. It also involved the construction and verification of a model of computation. Based on this model, critical modules that were candidate for VLSI implementation were identified.

October 13, 1995

Next the activities involved specification of modules, influenced by more technical and practical conditions. Given the context of the project it was clear that a physical implementation of the model of computation would go with trade-offs between the initial specification and the technological constraints and limited resources we find in an academic environment. A physical implementation of even a simplified version of the model was considered to be of importance to demonstrate the viability of the system, in particular for those parts of the system that outstandingly differ from the conventional.

With this awareness in mind, the selection of the modules that could actually be implemented in VLSI was made. As a result, this stage was characterised by a bottom-up concern with hardware technology —though conform the skeleton produced by the preceding top-down design phase. In this phase of the project it became clear that, technologically speaking, the lowest level of the architecture —viz., the display controller— was the most demanding and outstanding element of the architecture. This display controller has a two-phase structure: a level that is responsible for producing the pixels and a level that scans visible objects.

For the first level of the display controller —viz., the real-time pixel generator— no adequate components could be found on the market. For the targeted display resolution, fully shaded pixels have to be produced every 12 ns. Some elements of current general purpose processors may operate at such high rates. However, bus access rates are limited to a rate which is insufficient to sustain even a 25 MHz pixel stream needed for medium resolution displays. Therefore we concluded that the pixel-producing part of the display processor had to be implemented in custom VLSI. The actual design of the chip was done at the University of Twente. At the CWI the instruction set was verified by means of a structural simulator. This simulator both validates the functional design and visualises the results of newly designed graphics algorithms. With help of people from the UvA, the chips are recently integrated in a first version prototype to demonstrate the feasibility and the potential image quality of the system.

The second level of the display controller —viz., the object scanners— is not critical from a technological point of view. For future implementation, custom modules for bus arbitration and buffering purposes are specified and partly designed (but not implemented yet). These modules would serve to integrate an array of general purpose processors with the pixel generator.

1.5. Outline of the Dissertation

So far the STW funded research project has been motivated and placed in context. This dissertation is based on several research activities involved in this project. It should as such not be seen as a report on all the research activities that took place.

Chapter 2 is a survey of graphics systems and interaction. An inventory of interaction mechanisms is presented. Next the temporal aspects of interaction and the use of adaptive algorithms as a potential means to satisfy temporal constraints are discussed. Chapter 3 surveys existing graphics system architectures. Different

strategies to implement multi-processor architectures and characteristics are discussed. In Chapter 4 we find the theses of this work. It is argued that the variety of interaction mechanisms that exist are best served by a hierarchical architecture. A layered object-space architecture is proposed. It is illustrated how such a hierarchical architecture excels in support of interaction. The implications of this model of computation on some fundamental aspects are discussed. Chapter 5 is on consequences of the approach on aspects of geometry. It discusses representation of objects, structures for data storage and geometry based operations — i.e., a hidden surface removal algorithm which can handle incremental changes and scan-conversion. The rendering of the visible objects is described in Chapter 6. A reformulated high quality local illumination model is presented that is well suited to be implemented on scan-conversion hardware. Chapter 7 discusses the design and implementation of an object-space display controller — a two-phased architecture which can refresh a display from a structured list of objects. It is shown how this real-time scan-conversion system offers scalability and flexibility. Finally, in Chapter 8 results obtained are evaluated and future work is discussed.

Graphics Systems and Interaction

Synopsis.

This chapter is on aspects of graphics systems in relation with interaction. It begins with a survey of the concepts and terminology of both graphics systems and interaction. Next an inventory of basic interaction tasks is made since we suspect a relation between the response time of an interactive graphics system and how well required input can be mapped on interaction facilities available. Following this, temporal aspects of image generation are addressed. Finally we discuss adaptive methods as the potential means to meet timing requirements.

2.1. Interactive Graphics: Concepts and Nomenclature

"With interactive graphics, we are largely liberated from the tedium and frustration of looking for patterns and trend by scanning many pages of linear text on line printer listings or alphanumeric terminals." [Foley82]

A computer application may find the need to display an abstract, or realistic entity, called a *scene*. For this purpose, an application makes use of a *graphics system* which provides facilities to:

- specify (or model) a scene in terms of a set of logical graphical elements;
- specify parameters according to which the scene is transformed into an *image*;
- handle *graphical input*.

In this work, the graphics system is taken to be the whole of the language through which a programmer can specify pictures and input handling, and the hard- and software modules needed to generate those pictures and provide that input. An application is considered to be *interactive* if the user has ways to influence the behaviour of the application during the execution of the program.

2.1.1. Scene Specification

A scene is the mathematical representation of what has to be displayed. It is constructed by means of logical graphical elements, called *primitives*. Primitives incorporate two components: the *geometry* and an associated set of *attributes* (or *properties*).

The geometry of a primitive defines a set of points (P_{prim}) in a vector space, $P_{\text{prim}} \subseteq \mathbb{R}^n$. A scene specification language may have many types of primitives, where each type defines a certain basic geometrical form (such as a point, line, filled triangle, etc.). The minimal set of primitive types needed to adequately specify a scene is highly dependent on the type of application. International standard graphics systems such as GKS [ISO85b] and PHIGS [ISO89] offer a common set of graphical primitives.

Attributes specify how the geometry appears on the output medium (line type, transparency, colour, etc.). Each type of primitive has a specific set of attributes. Attributes are not necessarily bound to individual primitive instances, they may be bound to a group of primitives or can even be global.

Instances of primitives may be combined to form more interesting graphics objects. Examples of this can be found in graphics systems like ILP [Hagen80], PHIGS and HIRASP [Teunissen88]. These systems provide facilities for specification of hierarchically structured objects. The structuring results in a description of the scene in the form of a directed acyclic graph in which the constituent primitives can be found in the leaves of the graph.

2.1.2. Image Synthesis

A scene is mapped onto the *display-* or *image-space*. This mapping includes both geometric and attribute transformations. The geometric transformation, specified by *viewing* parameters, maps the geometry of the primitives ($\subseteq \mathbb{R}^n$) onto a geometry in the two-dimensional image-space (\mathbb{Z}^2). The attribute transformation results in a function which assigns a colour intensity to each point of this two-dimensional geometry.

Net result of mapping of the scene onto the display space is a set of *display primitives*, which describe an image. The process to construct an image within the limitations of a display device is known as *rendering*. The type of display primitives in which the image is described depends on the type of display technology in play (plotter, laser printer, CRT, etc.). Due to the limitations of the display device, the rendered image can essentially be no more than an approximation of the picture specified by the scene and mapping parameters.

Logical Model

A logical model of the graphics system envisions the mapping of a scene onto the display space as a stepwise process in which primitives travel through a *pipeline* of functional modules [Carlson80]. This model distinguishes between the different logical representation levels of the scene that exist in the image synthesis pipeline and the operations applicable to these representations. In this logical model (shown

in Figure 2.1) each module of the pipeline performs an elementary graphical operation on all passing primitives. Each stage in the pipeline maps one logical representation of the scene onto another. Intermediate representations can be stored and made accessible to improve the response time of the system. The inherent display device dependence of the mapping process as a whole is confined to a mere subset of the modules of the pipeline.

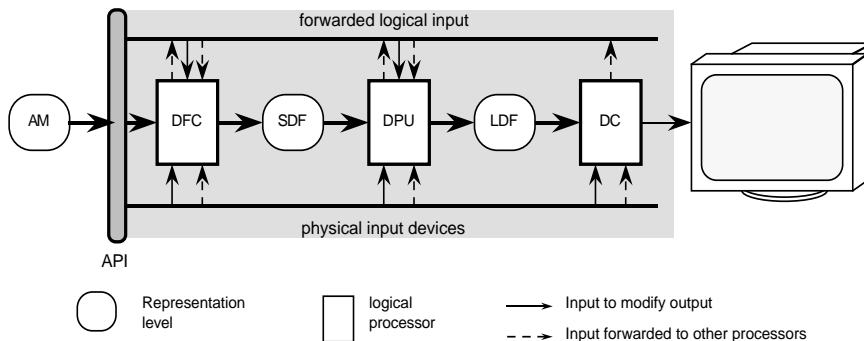


Figure 2.1: Logical model of the image generation pipeline. AM - Application Model, API - Application Programmer's Interface, DFC - Display File Compiler, SDF - Structured Display File, DPU - Display Processing Unit, LDF - Linear Display File, DC - Display Controller.

Logical Representation

Application Model. The application model is the representation of a scene as determined by the application program. This scene is presented as a model from which specific graphical images can be composed. The application program can change the scene by editing its representation. Naming of parts of the representation is allowed and can be used to aid editing.

Structured Display File. The structured display file contains only graphical information of the scene, and is composed as a list of output primitives. The graphics system manages the content of the structured display file. The application program has access to this representation level only via the application interface.

Linear Display File. The linear display file contains the representation of the scene in a form designed for optimal refresh speed. For traditional raster displays the linear display file is the so-called 'frame buffer', a buffer in which an image is represented as a pixel pattern. The display holds the visible image.

Logical Processes

Between each pair of successive representations we find a logical processor. Logical processors accept input parameters by means of which the mapping process can be influenced. The user and/or the application program may indirectly control

these optional parameters.

Display File Compiler. The display file compiler maps the application model onto the structured display file. This mapping process can be parameterised, for instance by surface properties.

Display Processing Unit. The display processing unit maps the structured display file onto the linear display file. This computational expensive mapping involves processes like geometric transformations, hidden surface removal calculations, illumination calculations and clipping. The display processing unit can be parameterised by viewing and illumination settings.

Display Controller. The display controller maps the linear display file onto the screen. Also the display controller can be parameterised. For instance, to accommodate window management functions it may be possible to map parameterised portions of the linear display file onto the screen.

2.1.3. Graphical Input

The image synthesis pipeline converts primitives —in terms of elements of the language an application understands— into display device dependent data. The reverse is true for *graphical input*. Graphical input originates from data that comes in via physical input devices. The raw data that input devices produce have to be converted into a more abstract form, that is, in terms of the language of the application.

Numerous input devices have been developed to facilitate graphical input. Examples of input devices are keyboard, tablet, mouse, lightpen, touchscreen, dial and data-glove. Of course these input devices differ in ergonomical aspects like physical dimensions, resolution and input bandwidth. They also produce data that may differ in aspects like range and dimensionality.

Logical Model

For current graphics systems, the device dependency of input data is hidden by making use of a *logical input model*. In a logical input model, data input takes place via *logical input devices*. The raw, device dependent input data are converted into a set of *input primitives*. Typical types of input primitives that can be found in all graphics systems are commands and geometrical data. The purpose of this abstraction is portability; mapping of device dependent input data onto input primitives is handled completely within the logical input model. Instances of elementary input primitives may be combined and mapped onto more complex input primitives before they are handed over to the application. The logical input model encapsulates all types of physical input devices and associated user actions. In this way, the application is unaware whether geometrical data is produced by a keyboard or by a pointing device.

Logical Input Device

The logical model of a graphical input device is shown diagrammatically in Figure 2.2. The output of a logical input device is an input primitive. Different classes of logical input devices exist corresponding to different types of input primitives. Each class has a well-defined data type for the measure reported by devices of that class. Current computer graphics standards support the classes: *locator*, *stroke*, *valuator*, *choice*, *pick* and *string* [Rosenthal82].

The *operating mode* of the device determines when an input primitive is delivered to the user of the device. It specifies the behaviour of the device as seen by both the operator and the using agent. Examples of operating modes commonly supported include the conventional *request*, *sample*, and *event* modes. Support for other modes can be defined as long as they operate within the confines of the model.

There are seven types of operations that can change the state of a logical input device and its component processes. These are: *definition*, *initialisation*, *activation*, *utilisation*, *deactivation*, *termination* and *undefinition*.

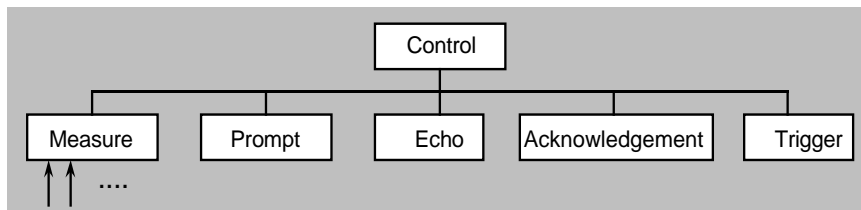


Figure 2.2: Logical model of a graphical input device.

Logical Processes

A logical input device comprises *measure*, *echo*, *prompt*, *trigger*, and *acknowledgement* processes overseen by a single *control* process. The control process is responsible for the overall operation of the logical input device and its associated component processes. Each of the component processes functions independently which, when acting in concert, can result in the production of an input primitive.

When the logical input device is activated, the *control* process activates the *measure*, *echo*, *prompt*, and *trigger* processes. The *measure* process continuously maintains a measure value derived from the input data. This includes all computations necessary to map the input data to the measure value. The *echo* process provides continual feedback of the current value of the measure. This feedback is expressed in terms of the output facilities available at the level of abstraction at which the logical input device is defined. The *prompt* process indicates the availability of its associated logical input device. A *trigger* process continuously monitors one or more sets of conditions. When any one of these sets of conditions is satisfied, the trigger is fired. When the trigger firing report is received by the control process, it produces an appropriate input primitive that

—dependent on the operating mode of the device— may contain the measure which was current at the time of the trigger firing. The input primitive is then reported to the agent that is using the logical input device and the *acknowledgement* process is activated. The acknowledgement process informs whether the input report was accepted or rejected by the using agent.

2.1.4. Graphical Interaction

It is good programming practice to clearly separate interaction facilities from the pure application functionality by means of a *user interface*. The user interface is a logical module which handles all transactions between a program and a user. The term user interface will be used to refer to in- and output devices and the software to control these devices. Interaction implies that (re)actions take place at both sides, however, a distinction can be made depending on which side controls the interaction. In the case of *internal control* the user interface is in charge of the interaction. The user is prompted to enter information. In the case of *external control* the user initiates the interaction. In terms of the logical input model the user interface has to set the appropriate operating mode of the input device(s).

To inter-act takes at least two actors. For graphical interaction, one actor — i.e., the user — makes use of input devices to provide information to an application. In turn, the other actor — i.e., the application — makes use of output devices to inform the user. Interaction requires proper feedback: each input action should immediately be “confirmed” by the system. A logical input device may provide an echo which is handled completely within the user interface. For this, the logical input model uses output functionality provided at the same level of abstraction at which the logical input device is defined¹⁾. On higher levels of abstraction the application program may become responsible for feedback as well. In that case the actions involved in generating graphical feedback include:

- handling of the logical input data; (graphics system)
- handling of the input primitives; (application)
- possible updates of some application data; (application)
- traversing (part of) the graphics pipeline; (graphics system)
- update of the refresh buffer. (graphics system)

This set of actions make up the *graphical feedback-loop* (see Figure 2.3). Graphical feedback implies that, in one way or another, logical input devices have to be coupled to output facilities. Here we find that the traditional logical input model described in the previous sub-section falls short. The model does not contain special functionality for this purpose. As a result, according to this model, the coupling has to take place in the application program.

It is essential that response on user input is acceptably fast. In fact, response on user input is a key determining factor of the quality of a user interface. The time to

¹⁾ Examples of this are on-screen input devices provided by the system such as dials, sliders and buttons.

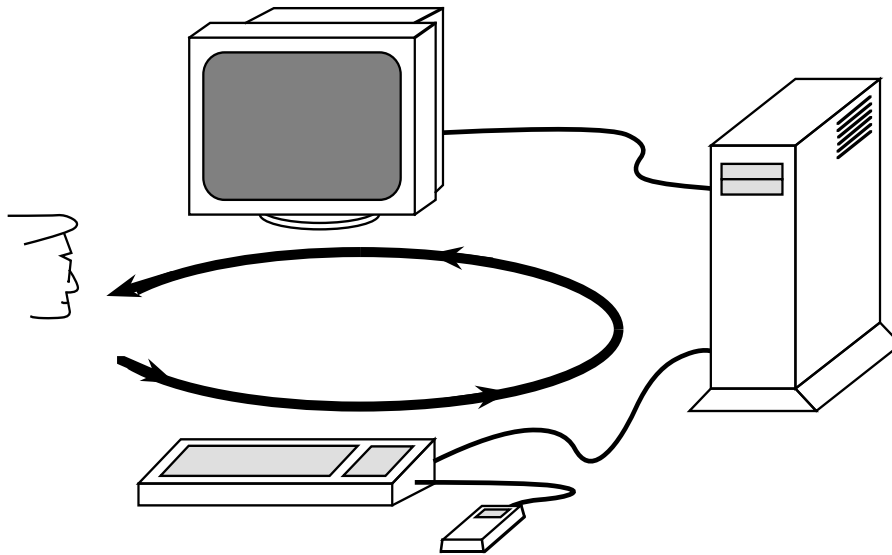


Figure 2.3: Graphical feedback loop. User input is processed and results in a visible reaction.

traverse the entire feedback-loop —i.e., the time elapsed between a user input action and the moment the related feedback appears on the screen— is called *response time*. A system's response time depends on several factors. Obvious factors are the amount of processing involved in the feedback-loop and the raw performance of the computing resources. Other factors are operating system related dependencies such as interrupt handling, context switching and the like. This variety of factors illustrate that concentrating on the throughput rate of the graphics pipeline alone is not enough to ensure fast response. It is essential that the amount of processing involved in the feedback loop is minimised. As will be discussed later, this relates to how effective a user can operate the graphics pipeline, how accessible intermediate representations are, and how these representations relate to the 'real world'.

2.2. An Overview on Interaction Tasks

"...interaction sequences can be decomposed into a series of basic interaction tasks. Each interaction task instance has a set of application requirements, defined by the context of the application in which the task is embedded." [Foley84]

The interaction tasks referred to in this quote are user-oriented, in the sense that they

are the primitive action units performed by a user. These interaction tasks present a different view on interaction as the hardware- and software-oriented view in the context of logical input devices as presented in §2.1.3. The relation between interaction tasks and logical input devices should be clear. Logical input devices are part of the means to accomplish basic interaction tasks.

In [Foley84] Foley et al. classify basic interaction tasks as: *selection*, *position*, *orient*, *path*, *quantify* and *text*. Beside these basic interaction tasks, they distinguish controlling tasks that modify objects being displayed: *stretch*, *sketch*, *manipulate* and *shape*. In their view, manipulate differs from the basic interaction tasks in that the cursor is replaced by an already existing object on the screen. Here the distinction between a cursor and an object is not relevant; the different types of manipulation of an object are assumed to be basic interaction tasks. In this view, the operand of any task may therefore be a cursor as well as a display primitive (or a group of display primitives).

Basic interaction tasks of interest for this dissertation are those tasks that make use of interaction techniques which relate directly to the image generation pipeline, or more specifically: interaction techniques that provide dynamic visual feedback. The important role of these basic interaction tasks in particular should reflect in the design of the architecture. In the next we will confine to these types of basic interaction and controlling tasks.

Selection

A user may select from a collection of displayed entities by means of a pointing device, such as a lightpen or a mouse- trackball- tablet- or joystick-controlled cursor. In principle, the types of entities that can be selected may vary from commands in a menu, symbols and icons to display primitives or groups of display primitives. In this analysis we are interested in selection of display primitives and groups of display primitives.

Selecting or *picking* display primitives essentially involves a search of the display file. Selection typically serves to specify the operand of an operation. It is part of almost any interaction dialogue. Primitives may be embedded in a hierarchical structure. Then, selection may occur at different levels of the hierarchical structure. The hierarchical level at which selection takes place is "defined by the context of the application in which the task is embedded". The application should provide the means for the user to climb up and down the hierarchy.

The graphics system has to make clear which of the primitives will be picked upon triggering based on the current position of the selection device, by for example, high-lighting. Such feedback should ensure non-ambiguity to avoid unpleasant surprises. This may seem an obvious and simple requirement, but in practice it may turn out to be quite complicated. To avoid ambiguity, high-lighting of a partly hidden selection should preferably reveal the structure of the complete selection, i.e., including the obscured parts. However, it may be essential to be able to judge that selection in its three-dimensional context, in which case a simple

overlay of the complete structure on the image is unsatisfactory.

Ideally, the echo process finds all display primitives that (partly) obscure the structure and, for instance, makes these primitives (partly) transparent. In any case, the system has to “know” what part of the current pickable item is visible and what part is invisible.

Position

The user can specify a position by means of a pointing device. The position task may be an open-loop or closed-loop process. For the open-loop technique the visual feedback on the screen serves to show the result of the position task, it does not play a role in specification of the final position. On the other hand, if the position task is a closed-loop process, the user adjusts the position of the operand based on what is shown on the screen until the desired result has been obtained. Therefore the position of the object on the screen will continuously have to be updated²⁾. If the operand is an object rather than a cursor, the task is also known as *dragging*. It may be clear that in this case the requirements on the response of the system are high.

The position task may occur in one, two or three-dimensional space and may be constrained (e.g., along a path or in a plane). Naturally, for proper positioning it is important that feedback ensures non-ambiguity. Consequently, a position task in a three-dimensional space should make it possible to judge the position of the operand in the 3D context. This implies that hidden surfaces have to be removed in real-time, and possibly that display primitives that obscure the operand have to be found and made transparent in real-time as well. In practice we often see that, to avoid such complications, 3D positioning is possible on simplified models only (e.g., wire-frame models).

Panning and zooming are a special kind of position tasks. Add and delete operations can be considered as being position tasks as well. This view is demonstrated for instance in the desk-top metaphor in which deleting of a file is done by dragging the icon representing that file into a trashcan.

Orient

The orient task very much resembles the position task. Yet, there are some differences — such as the user's perception of orientation and the use of input devices — that make it reasonable to classify orient as a separate type of basic interaction task. The orient task may occur in two or three-dimensional space and may be constrained (e.g., rotation around an axis).

The user may specify an orientation by means of an input device like a dial, a locator or a joystick. The orient task may in principle be an open-loop or closed-loop process. In the latter case, the user adjusts the orientation of the operand until the desired result has been obtained. Like for the closed-loop position task, also for a closed-loop orient task hidden surfaces have to be removed in real-time, and it may be necessary that display primitives that obscure the operand have to be found

²⁾ More on open and closed loop interaction techniques in § 2.3.1.

and made transparent. This complication does not occur in wire-frame models. Since for the orient task the operand remains at its local position, finding the display primitives that obscure the operand will in general not be as involving as it can be for dragging.

Path

A user can specify a path by a series of positions and/or orientations. It is seen as a task different than the position and orient task because in this case the order and relative relation of the positions/orientations is of main interest, not the individual positions/orientations that make up the path.

The user may specify a path by means of a pointing device using techniques similar to those for the position and orient task. The path may be distance-based or time-based. The operand may be a cursor or an object. It is important to be able to judge a path in its 3D context, which implies that, similar to what holds for the position task, hidden surfaces have to be removed in real-time and, possibly, that display primitives that obscure the path have to be found and made transparent in real-time.

Quantify

The quantify task serves to input a value, a dimensionless number, that can be bound to any aspect of objects or global entities of the scene. Unlike the interactions tasks listed so far, this does not restrict to geometry related aspects only.

The user may specify a value to quantify a measure by means of, for instance, a valuator or a mouse- or tablet-controlled slider. If the operand is an object by itself, the task is typically a controlling task, rather than a basic interaction task.

Controlling tasks

The basic interaction tasks listed in the above are different types of manipulations of an operand with respect to the rest of the environment. By means of controlling tasks such as stretch and shape, the user can modify (edit) the operand itself. The user typically operates on a particular feature of the operand only, not affecting remaining features. Examples of this type of manipulation may be deformation of the operand by (re)positioning of a vertex or a control point (for curved surfaces) or variation of an appearance control parameter.

The user may edit an operand by means of a selection, position, orient and/or quantify task. In case the task is an open-loop version, the user adjusts the feature directly until the desired result has been obtained. Continuous feedback of the edit task in a three-dimensional space implies that hidden surfaces have to be removed in real-time in case the feature is part of the geometrical description of the operand. In general, the operand will globally speaking remain at its current local position, so that here also finding the display primitives that obscure the operand is not as involving as it is for dragging. It may be that display primitives that obscure the operand have to be found and made transparent. This complication does not occur in wire-frame models.

If the appearance control parameter of the operand is edited, a closed-loop version of the task can be reduced to a continuous redraw of the operand only, based on the current value of the appearance control parameter.

2.3. Temporal Aspects of Interaction

User interfaces are becoming more and more sophisticated due to evolution of both hard- and software. Sophisticated user interfaces should support natural interaction and take into account features of human perception. This leads to *animate response*—that is, response where changes as invoked by the user are visualised by a smooth, natural transition from one state into another. In this sub-section, such temporal aspects of image generation in general, and for animate response in particular are examined.

Motion plays an important role in numerous aspects of human vision. Eyes are in a constant state of movement to generate stimuli needed to see anything at all. Relative motion of objects turns out to be very helpful in our understanding of three-dimensional structures: a change of viewing position generates motion parallax. The human mind is trained to interpret this motion parallax as a measure of distance. The human vision system uses relative motion as a basis of grouping, organising objects that belong together. By exploiting these type of perceptual capabilities of the viewer acquired by years of training in the real 3D world, animate response can improve human computer interaction.

Why should we be concerned about temporal aspects of response, is it not sufficient to let the system simply respond as fast as it can? Systems get faster every day, so if today's workstations are not fast enough yet, systems of the future will be. It is a fact that due to progressing technologies, workstations continue to increase in capacity. As a consequence of this also applications become increasingly more complex. The net result of the increase of capacity may therefore very well be insufficient. More importantly however, responding as fast as possible does not necessarily lead to adequate response.

A system can be too slow, but on the other hand it can also be too fast to be perceived by the human viewer, leaving him in a state of confusion. What is needed is a concern about the temporal aspects of response: the system must be “tuned” to the time frame of the user. This tuning should aim at optimal interaction, taking into account the limited temporal resolution of both the display system and the human vision system. Also, a system may operate within a certain time frame to help the user anticipate to a new situation. The following examples that can be found in some 2D user interfaces may be illustrative.

Low-end systems cannot support dragging of a window with all of its content. Dragging a window with a significant time-lag turns this basically simple task into a tedious exercise: it is next to impossible to place the window at the desired position. A workable solution to get around this problem is to leave the window with its content where it is and let the user drag a simplified representation of the window:

its boundary rectangle. Once the user is satisfied with the new position, the window and its content can be redrawn at the new position. Quality is traded for time.

The visible action upon closing a window, i.e., clearing the screen, can in general be done quite fast. However, it may be better to smoothly lead the user to the new situation by means of a shrinking boundary rectangle moving towards a disk or folder icon. An implicit animation like this gives the user a better idea of what is going on. Here, time is traded for quality.

For animate response in three-dimensional dynamic environments more sophisticated techniques are needed, but basically they have to be of the same nature: adapt to the users time frame.

2.3.1. Animate Response

Animate response is the visual response of a system to changes invoked by the user. It is characterised by a smooth, natural transition from one state into another. It is useful to make a distinction between *implicit animation* and *direct manipulation*. These two categories differ in characteristics of the visual feedback of the transition phase, a difference which reflects in the temporal aspects of the interaction.

Implicit Animation. For implicit animation the current state and the explicit specification of a new state implicitly specifies the transition. The transition is an 'open loop' animation. Although certain parameters may affect the characteristics of the transition process, the transition itself is autonomously generated by the system, that is, not directly controlled by the user. The temporal aspects of the transition can in principle be analysed beforehand. The implicit animation can be a natural transition between two 'key-frames' resulting in for instance an animated sequence of a slowly rotating molecule. One may also think of more abstract transitions in which the relation between two by nature different entities can be visualised such as the relation between a window and an icon as just mentioned.

Implicit animation can be employed by traditional command language type of interaction in which objects are manipulated by means of explicit commands, issued for example by typing or by menu selection. Implicit animation serves to 'guide the eye', to make the transition appear more natural. Based on the type and amount of the change, the system should be able to determine how to best animate a transition. Setting of transition parameters may allow the user to specify speed, image quality and other aspects of the transition. Advanced systems would automatically adapt to skill and preferences.

Direct Manipulation. Direct manipulation allows the user to manipulate a visible representation of a virtual model as if it is a real object which is mechanically connected to an input device like a mouse, a light pen, a trackball or a data-glove. In this type of transition process, the user is in direct control over the transition from one state to an a priori unknown final state. In this case, the visualisation process is an essential part of a closed loop between the system and the user. The user handles the input device based on what he sees on the screen. As a result time constraints are strict.

Direct manipulation has already proven to be successful in a well known type of user interface: the desktop metaphor. This type of user interface proved to be a success mainly because it presents a virtual two dimensional environment which is a metaphor of an environment already familiar to the user: the desk. The interface allows the user to manipulate documents (windows) on a screen just as paper documents can be manipulated on a desk.

A similarly natural and intuitive direct manipulation concept can be used in three-dimensional environments. In such artificial realities —also known as virtual environments or cyberspace— the user is free to manipulate simulated three-dimensional objects in a natural way, is free to walk through and interact with imaginary worlds, in short: can be immersed in a virtual model. Based on the direct manipulation concept we can for instance create remote control systems with which a user is able to control remotely in space (hazardous or far away environments), remotely in time (future) or allow control in a different scale (macro- or microworld) [Shneiderman89].

The most important aspect of *direct* manipulation appears to be *direct* visualisation. If the result of the manipulation is not visualised immediately, the user will not feel immersed in the environment. As stated by Scott Fisher³⁾ "the quality of the graphics appears not to be extremely important, as long as the response is good".

2.3.2. Computer Animation

Why is a film, a long band of static images, often called a 'movie'? Answering that question also gives an answer to why computer animation is at all possible, using video displays which essentially can display static images only. Apparently the characteristics of the human vision system makes it possible to give a viewer an impression of continuous movement by successively presenting displaced static images or *frames*. This ability is based on two distinct features of the human vision system: *persistance of vision* and the so-called *phi phenomenon* [Gregory90].

Persistence of vision is the inability of the retina to follow light flashing at a rate above about 50 Hz. Because of this, a movie projector — that presents frames at the standard rate of 24 Hz — raises the flicker rate to 72 Hz by means of a shutter mechanism. The phi phenomenon tells us that the retina system is tolerant with respect to intermitted images provided the jumps in space or time are not too large. It is likely that this ability to tolerate gaps is developed to maintain continuity as objects are hidden briefly behind obstructions, or retinal images behind blood vessels.

Since the impression of continuous movement is lost below a certain frame rate, it is important to know at what range of rates these frames should be generated. As the time interval between successive stimuli is increased, the observer's perception of apparent movement goes through several stages [Goldstein89]. In an experiment

³⁾ Panel *Virtual Environments and Interactivity: Windows to the Future*, SIGGRAPH'89, Boston.

with two flashing lights at a separation in time of about 30-60 msec, partial movement is perceived. At about 60 msec, the lights appear to move continuously from one position to another. At intervals between about 60-200 msec the phi movement is perceived. Movement appears to occur between the two lights, yet it is difficult to actually perceive an object moving in the space between them. Outside the range 30-200 msec we do not perceive movement.

Also the distance in space between the two lights affects the perception of apparent movement. As the distance increases the time interval or the intensity of the flashes must be increased to maintain the same perception of motion.

Video displays have a constant refresh rate of 50-100 Hz. Therefore, the maximum frame rate is restricted ⁴⁾. This limitation has effect on visualisation of fast moving objects. The displacement of fast moving objects between two individual frames may become quite large. When these objects are displayed just as if they were static, the spatial relation is lost and the impression of watching objects in continuous motion is no longer there. A movie projector however, shows separate frames at a rate as low as 24 frames per second. This rate is sufficient to generate an impression of continuous motion, even of fast moving objects. Why this is so can be found by looking at individual images of fast moving objects on a camera film. The finite exposure time of the film results in blurred images of fast moving objects. Due to *motion blur* the impression of continuous movement is maintained, even at a rate of 24 frames per second. Apparently the human vision system can trade off temporal resolution against spatial resolution.

A number of papers present methods to generate motion blur (also known as temporal anti-aliasing) [Korein83, Potmesil83, Max85, Grant85]. In these papers it is argued that "animation which simulates motion blur feels more natural" or "motion blur smooths out jerkiness". A more theoretical foundation of why and to what extent motion blur is needed can be found in Blake's thesis on computing adaptive detail [Blake89].

Blake's study aims at reducing the computational complexity of computer animation by computing just what is needed to produce convincing pictures. The thesis is centered around two metrics, a spatial (static) priority: "objects further away from the viewpoint are visually less important to the picture than those closer by" and a temporal (dynamic) priority: "objects moving quickly with respect to the observer need to be redrawn more often than those at rest". These intuitive formulations are further developed and extended with the notion that human vision introduces a

⁴⁾ A "frame" should be seen as a virtual picture which can be displayed for several refresh cycles. The *frame rate* is the rate at which successive, different frames are displayed. It should not be confused with the *refresh rate*, which is constant and is dictated by the persistence time of the display medium. (This is analogous to the movie projector that displays frames at a rate of 24 Hz but flickers at a rate of 72 Hz.) Since frames can be swapped only just before the start of a new refresh cycle, the frame rate can never exceed the refresh rate.

trade-off between temporal and spatial resolution. These are the basics for arriving at measures to determine to which extent detail can be left out without being noticeable by a viewer.

Changing images are a function of two space variables and of time. A Fourier transformation of this function results in a spectrum, a function of temporal and spatial frequencies. This function is non-zero in a limited domain only. The effect of motion on this spectrum is that it is sheared in the temporal frequency dimension. The amount of shear is proportional to the velocity of the object components in the two dimensional image. The human vision system is limited in spatial and temporal frequencies by the various transmission systems between our mind and the outside world, resulting in a so-called window of visibility [Watson86]. Since the spectrum of moving objects is sheared, some spatial frequencies which would fall inside the window of visibility for static objects, may fall outside this window when these objects are in motion. Removing the spatial frequencies that fall outside the window of visibility is equivalent with removing details of the moving object that cannot be perceived by the human vision system. These spatial frequencies in combination with the frame rate of the display system appears in aliased form, causing the unnatural or jerky motion mentioned above.

Blake remarked that Fourier analysis is useful for analyzing a problem and obtaining insight into what is happening. In practice, however, computations and algorithms are less likely to use Fourier techniques directly. He argues that optic flow analysis as introduced by Gibson [Gibson79] can be used to obtain a measure of the velocity and/or distortion of object components in the image. This measure can be used to determine the amount of detail needed. It indicates which spatial frequencies have to be left out, or equivalently, to what extent the image should be motion blurred.

Finally, Blake concludes that for planar objects, four orders of optic flow effects result in four orders of frame to frame coherence. In decreasing order of coherence these orders are:

- no displacement;
- translation in the image plane;
- linear transformation in the image plane (shear, rotation, scaling);
- all other, non-linear transformations.

To be able to exploit these types of frame to frame coherence it is necessary to maintain a relation between objects and their projection onto the image plane.

2.3.3. Discussion

The major issue of computer animation is how to convince the viewer, to make him see what he is meant to see. Often images are simply rendered at the highest — and constant — quality level that is possible. For real-time animation systems this requires a major exertion which can hardly be maintained by costly specialised systems — such as flight simulators — and then, for well-defined situations only.

Animation production involves three separable tasks: modeling, animation (i.e., the design of the choreography) and rendering. Animation production is a compute intensive process which may — and, due to the separability, can — take orders of

magnitude more time than the real-time display of the animation itself. This is unlike animate response, where animation and rendering by definition occur in real-time. In spite of this difference in character, computer animation may still teach us how to deal with certain temporal aspects such as limitations enforced by the display system and the human vision system.

In the process of computer animation production, limited computing resources can be traded against production time. This is quite different from what we find in an interactive environment of which the main device necessarily has to be: "the quality of the graphics is not extremely important, as long as the response is good". The main objective for animate response is real-time display of continuous motion, even if one has to deal with complex structures. For animate response, the limit on computing resources necessarily has to be compensated by a reduction of image quality, but then preferably in a reasoned way. This issue will be addressed in the next section.

2.4. Dealing with Motion

In the above, we concluded that for animate response two constraints may complicate the visualisation of continuous motion: the limit on the frame rate as dictated by the display system and the limited capacity of computing resources. The net result of both limits is the same: temporal aliasing may become visible. Depending on its cause, temporal aliasing can be prevented by temporal anti-aliasing or by reduction of the cost of image generation. As has been mentioned before there are several techniques for temporal anti-aliasing. The cost of image generation can be scaled by so-called adaptive image generation techniques. In the next we will examine their usability for animate response.

2.4.1. Temporal Anti-aliasing

Existing temporal anti-aliasing methods [Korein83, Potmesil83, Grant85] are based on supersampling techniques. As a result, the computational cost of one temporal anti-aliased frame is much more than the cost of one aliased frame. Also the more efficient method as presented by Max [Max85] adds computation cost to each frame. This may not seem surprising. However, one should realise that —as mentioned in § 2.3.2— temporal anti-aliased images contain less detail: the high spatial frequencies are filtered out. One could argue that therefore, temporal anti-aliasing should save rather than add computational costs. It is surprising that as of yet there are no methods that are able to make use of this.

In an interactive environment it is likely that computing resources are a limiting factor due to the combination of scene complexity and real-time requirements. We therefore may conclude that existing super-sampling techniques are not suited for animate response.

2.4.2. Adaptive Image Generation

Generating a physically perfect image would by far exceed the processing power of any state of the art supercomputer. Due to this, a whole scale of rendering models emerged, each with a different level of approximation of the 'physical correct' image. Based on requirements of a specific application, one of these rendering models can be selected. If the system is tuned for worst case situations the efficiency will be far from optimal. On the other hand if the system is optimised for 'normal' situations its worst case behaviour may be unacceptable. Adaptive image generation is a means to adjust the image generating process to the possibilities of a particular moment. The quality of the image — and thus the cost — is related to the time available between successive updates. Ideally, this results in the best possible image at any time.

There are two distinct aspects that determine the cost of the image generation process and which can be varied. These are the quality of the rendering process, and the quality of the object representation.

Quality of rendering

Forrest suggested that rendering of primitives should be supported at different quality levels. In [Forrest85] he recognises five quality levels to draw a line, starting with an aliased Bresenham line drawing up to a perfect anti-aliased line of which line endings are drawn according to the specification (e.g., rounded or square). He suggested that such a hierarchy in quality could be exploited in the context of personal workstations. Images should first be rendered at the lowest quality level to get the fastest response and upgraded to higher quality levels if the user does not take immediate action.

Bergman et al. proposed image rendering by adaptive refinement [Bergman86]; that is, improve the quality of a static image as long as there is time to do so. The image successively goes through the following phases: display vertices of polygons, display edges of polygons, display flat shaded polygons, add shadowing, display Gouraud shaded polygons, display Phong shaded polygons, and finally, anti-alias the image where needed. The performance of the method is enhanced by making use of results of previous phases and trimming of the data, i.e., selecting the polygons that should be handled in the next phase.

Both of these methods try to combine interactive response and generation of the best possible image by a stepwise improvement of the image quality. The order of the steps taken are based on rendering costs only, not influenced by application requirements and not related to aspects of human perception. It does not anticipate on the fact that for some applications it might be more important to anti-alias the image to avoid disturbing wrinkles along the edges than to properly shade the object, whereas for other applications the reverse may be true.

Both methods make use of otherwise idle cycles in a personal computer to improve the image quality. The rendering is supposed to start at the lowest quality level. When displaying continuous motion idle cycles are rare, so that these methods are likely to produce images of the lowest quality level only.

Quality of representation

Alternatively, rendering costs can be tuned by selecting the level of detail of a scene. To be able to render objects at different levels of detail, Clark made use of hierarchical data structures in which sub-hierarchies contain objects modelled in greater and greater detail [Clark76]. Such hierarchical structures allow for a simple incremental approach: as long as there is time, objects can be progressively refined. A metric which takes into account the viewing distance and speed of the objects, can be used to indicate which objects are best candidates for further refinement [Blake89]. Again, if the rendering is supposed to start at the lowest level it likely that when displaying continuous motion, images of the lowest refinement level will be produced only.

Hierarchical data structures may offer the means to manage the complexity, and thus the image generation costs. They allow definition of a so-called graphical working set [Clark76], which is that fraction of the structure that at a certain time [Hegron87] is potentially of interest. This notion has been put into practise in flight simulator applications for a long time. There, the most appropriate representation of an object is selected from a hierarchical data structure at display time. Such data structures are carefully optimised for the application. Also automatic generation of the hierarchy, and preferably support for temporal anti-aliasing and texturing should be considered.

Not all object representations have an inherent hierarchical structure. In such a situation we need a hierarchy that can automatically be generated from the representation at hand. Recursive spatial subdivision seems an obvious way to automatically generate an oct-tree like hierarchical structure. Although the result is a simple uniform representation, rendering of such a hierarchy is not very efficient and the representation is not compact. Furthermore, transformation of objects will require restructuring of the hierarchy. From this we conclude that such hierarchies are not suited for animate response.

Rubin & Whitted stated: "creation of a hierarchical database is a non-trivial operation" [Rubin80]. They presented a homogeneous representation from which nodes of the hierarchy are procedurally generated. Their representation is a graph structured hierarchy of nothing but bounding volumes. Each bounding volume contains subspaces, which are the bounding volumes of the next level in the hierarchy. Their representation allows sharing of subspaces. A step to the next level in the hierarchy of bounding volumes implies that a volume, being the difference between the bounding volume and its subspaces, has to be subtracted. However, subtraction of such a volume reveals previously obscured parts of the scene. This would imply a redo of (a part of) the visibility calculation. It would be more efficient to have a hierarchy that incrementally adds volumes, leaving the rest as it was. The proposed sharing of subspaces greatly reduces the amount of data storage but introduces combinatorial problems when creating the hierarchy. Since an optimal result requires careful consideration, Rubin & Whitted concluded that it would benefit to off-load the structuring of the hierarchy to the model creation stage. Rubin [Rubin82] expanded the representation to allow for geometrical

transformations between nodes. Such an expansion may be useful in an animate environment.

How well can object descriptions support temporal anti-aliasing? As we saw in § 2.3.2, temporal anti-aliasing goes with removing high spatial frequencies above a certain cut-off frequency. This cut-off frequency is inversely proportional to the speed of an object. An object can be described by a sum of band limited terms of increasing frequencies, a representation which can for instance be obtained by Fourier analyses. The effect of temporal anti-aliasing on such representation would be that the faster this object moves, the less terms would remain. This behaviour may seem ideal since then the rendering costs would be inversely proportional to the speed of the object. Unfortunately however, an object description by a sum of band limited terms is in general not the most efficient. For one thing, it is not a natural representation of objects, so that Fourier transformation is needed to obtain a representation like that. Also the rendering of such a representation would be costly. In a static or near to static situation, a description of even the simplest object would at least need terms up to a frequency equivalent with the display resolution, otherwise visible detail would be lost. As a result, whether a description by band limited terms pays off is dependent on the extent to which Fourier transformation simplifies temporal anti-aliasing and complicates rendering. It is likely to pay off only in situations where any other representation would be of similar complexity, such as may be the case for textures.

Norton et al. [Norton82] described a method of limiting texture detail for textures expressed as a sum of band limited terms by means of "clamping" those terms that exceed a certain frequency. Since the authors just considered spatial anti-aliasing, their clamping frequency is fixed as it is determined by the display resolution only. A similar method could in principle be used for temporal anti-aliasing if the clamping frequency is related to the speed of the object. Other spatial anti-aliasing methods for textures exploit hierarchies of various resolution texture maps [Crow84, Glassner86]. Temporal anti-aliasing of such textures can be done by selecting the appropriate resolution map, again related to the speed of the object. However, for these texture representations the costs of texture mapping is independent of the texture density. Consequently, reducing texture resolution in that case will not reduce the cost of the image generation.

Quality Related to Cost

In order to optimally balance image quality and temporal aspects of the animation, selection of the level of detail used for rendering should not only be based on the visual aspects, but also on the actual costs of rendering of the different levels of detail. For implicit animation in particular, it would be desirable to have an absolute quantification of the rendering cost as function of the level of detail (and rendering algorithms used). With this, we could determine the maximum level of detail for which the animation can be sustained. To obtain an absolute quantification of the costs however, would require an extensive simulation of the rendering hardware that

will be used. Another complication that has to be dealt with is that the complexity of the scene is likely to be view dependent.

In [Funkhouser93] Funkhouser and Séquin describe an adaptive display algorithm that adjusts the image quality to maintain a uniform frame rate. They “perform a constrained optimization to choose a level of detail and rendering algorithm for each potentially visible object in order to generate the best image possible within the target frame rate”. Their cost heuristics is based on measurements of rendering several levels of detail on the graphics system of interest. Their benefit heuristics is a somewhat ad hoc estimate of the “contribution to model perception”.

2.5. Conclusions

In this chapter we discussed the issues involved in graphics systems in relation with interaction. In this we emphasised on animate response. We noted that the frame rate needed to display continuous motion can be related to the speed of the object components. A movie projector —the display engine of a mechanical “graphics system” — is operated completely time independent of the camera. It is clear that both mechanical devices are operated at a constant frame rate to avoid complex synchronisation issues.

In computer graphics applications, however, we do not deal with mechanical devices and in general there is a rather tight coupling between the image generation system (the “camera”) and the image display system (the “projector”). In this case, a more complex synchronisation scheme is feasible. A more flexible synchronisation scheme would give us the possibility to vary the frame rate, related to the speed of the object components in the image. Making optimal use of frame coherence is a promising strategy to optimise interactive environments. In its simplest form frame coherence can be exploited by partially updating the image. The image has to be redrawn only in the area where the image is known to be changed. In the extreme, each object might have its individual frame rate.

The success of adaptive image generation methods based on incremental calculations is highly dependent on to what extent increments make use of previously obtained results. If rendering of one level does not make sufficient use of the results obtained by previous levels, the total cost of generation of the final image may even exceed the cost of non-incremental methods, which renders these type of adaptive algorithms useless for dynamic environments.

In this sense adaptive techniques that directly aim for a certain quality level, based on pre-estimated cost are likely to be more successful. Tuning the level of image quality should adhere to priorities prescribed by the characteristics of human perception, not by the cost of the quality improvement. To be able to do so, we need metrics which take into account the measure of quality —in terms of level of detail and rendering quality— and the associated costs in relation with the computing resources. Providing such metrics is beyond the scope of this dissertation, but we

have learned that the architecture should provide flexibility to be able to adjust the quality in a reasoned way. Again, in the extreme it should be possible to differentiate between individual objects.

October 13, 1995

Raster Graphics Hardware

Synopsis.

This chapter is on hardware aspects of raster graphics systems. Techniques involved in graphics systems targeted for displaying 3D polygonal models are examined. Image generation involves computing tasks of vastly differing order of granularity and complexity. The multitudinous equivalent calculations involved make graphics systems obvious candidates for parallelisation. Different task subdivision strategies and their applicability for graphics systems are discussed.

3.1. Introduction

Whereas the previous chapter focussed on software aspects of graphics systems, in this chapter we will concentrate on hardware elements necessary for the display of raster graphics. In downstream order these elements are:

- graphics processor,
- frame buffer,
- display controller,
- display device.

Several techniques exist to improve the performance of a graphics system. A systems designer can choose for virtually any performance level, but of course there is a price to pay. As a result we find on the market systems that range from cheap low-performance systems in which a single general purpose CPU also takes care of all graphics tasks, up to expensive high-performance systems in which graphics tasks are offloaded to powerful special purpose graphics subsystems.

Graphics systems architecture is a specialised branch of computer architecture. It is driven, therefore, by the same advances in semiconductor technology that have driven general purpose computer architecture over the last several decades. Many of the same speed-up techniques can be used, including pipelining, parallelism, and tradeoffs between memory and computation [Molnar90].

The design of special purpose hardware tends to be based on the characteristics of a

particular class of popular applications. Because of this, we may find that a system based on general purpose hardware, with mediocre performance for standard types of applications, turns out to be competitive for applications for which the characteristics do not adhere to the expected standard behaviour. Throughout the years we have seen several 'waves' of moving functionality in the direction of dedicated graphics processors and the reverse, moving functionality to the general purpose host CPU. These tidal waves in the ocean of computer graphics systems design are caused by changes in hardware technology (e.g., the RISC versus CISC competition) as well as changes in software technology and de facto standards (graphics languages versus pixel based window systems). The current trend is that a substantial amount of functionality is performed by the host CPU. The CPU is then assisted by a rather simple rasteriser (for instance the REX of Silicon Graphics' Indigo [SiliconGraphics93]). More high-end versions may still have dedicated hardware for geometry based processing such as transformation and triangularisation [SiliconGraphics92].

In high-end systems we usually find that concurrent processing is taken to be the answer to improve the performance of a graphics system. The main stream of research on graphics hardware involves strategies for concurrent processing. This chapter will address several issues of concurrent graphics systems

3.2. An Overview on Hardware Elements

In the next subsections the hardware elements necessary for the display of raster graphics will be discussed in upstream order. This discussion serves to indicate important characteristics of these hardware elements only. More complete surveys on graphics hardware can be found in the literature (e.g., [Glassner85, Molnar90]).

3.2.1. Display Device

For the display of computer graphics images, different types of display technologies exist, each with its specific characteristics. Based on the elementary display primitive we can make a distinction between vector devices (e.g., pen plotters, vector displays etc.) and raster devices (e.g., matrix printers, LCD's etc.), since these are two conceptually differing technologies.

Where speed is concerned, graphics systems typically make use of cathode-ray tubes (CRT's) for display of images. A CRT has a phosphor-based coating on the inside surface, which glows when it is hit by an electron beam. The persistence time of the phosphor-based coating is typically in the order of 10-60 milliseconds. Because of this, images displayed on CRT's are transient, so that they have to be cyclically redisplayed. CRT's come in two flavours: we can make a distinction between vector displays and raster displays.

Vector display. Vector displays are calligraphic systems for which the elementary display primitive is a line segment. The position of a line segment is specified by the location of a begin- and end-point only. For CRT's, line segments are actually

drawn, that is, the electron beam is directed along each of the line segments. The beam is switched off when it is directed towards a next line segment. The inherent limited resolution of the digital representation of begin- and end-point causes quantisation. However, the lines on the screen themselves are continuous as shown in Figure 3.1. Because of this, the display space of vector displays is said to be continuous.

The refresh buffer for vector displays contains the data of all the elements that have to be displayed. Therefore, the size of its content is dependent on the image complexity.

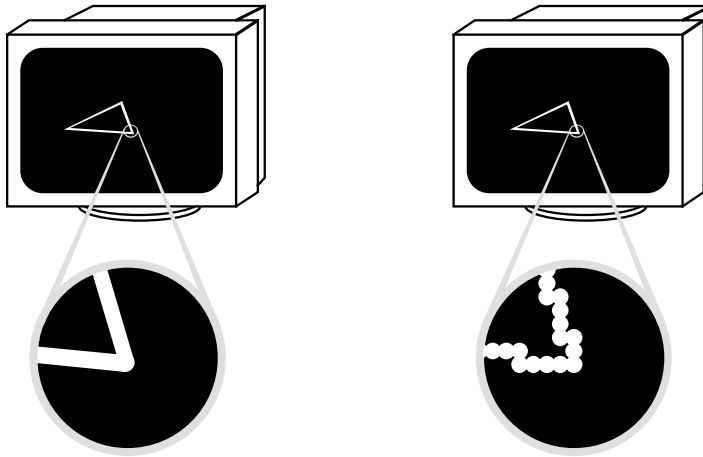


Figure 3.1: Rendering of an object for a vector display results in a set of continuous lines (left). Rendering the same object for a raster display results in a two dimensional array of intensity values (right).

Raster display. For raster displays the image is represented by a multitudinous two-dimensional array (raster) of picture elements — or *pixels*. The electron beam follows a fixed path that covers the entire display area, independent of what is actually displayed (see Figure 3.2). The image is drawn by setting the electron beam intensity according to the pixel value of the pixel that is related to the electron beam position. This results in a series of discrete dots on the screen as shown in Figure 3.1. Consequently, the display space of raster displays is said to be discrete. Each pixel of the array carries intensity information of a point of the *rasterised* image. Since the total number of pixels that cover the display area is fixed, the size of the refresh buffer is fixed — i.e., independent of the image complexity.

Nowadays, raster displays are the most common types of CRT display devices used. They became successful because:

- their technology is based on simple standard television technology which makes them cheap;

- representation of intricately coloured solid areas is facilitated so that highly realistic images can be rendered;
- their refresh process is not influenced by the complexity of the scene or the extensiveness of the rendering process.

These characteristics turned out to be of more importance than the high resolution and more dynamic characteristics of vector displays. In the remainder of this work, the individual term 'display' is assumed to stand for 'raster display'.

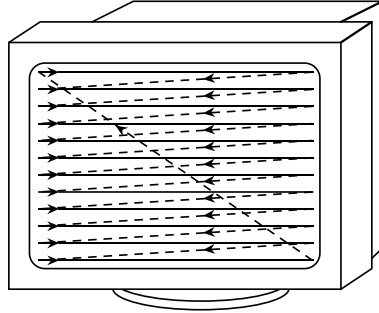


Figure 3.2: The path of the electron beam when scanning the display area.

The electron beam that is fired onto the coating of the CRT can be deflected by means of either electrostatic or magnetic fields. The beam intensity can be varied, which changes the brightness of the spot where the beam hits the coating. Beam deflection and beam intensity are controlled by electronic circuitry. This circuitry is integrated with the CRT in a so-called *monitor*.

The display area is scanned by the electron beam in a number of horizontal sweeps going from left to right (see Figure 3.2). When reaching the end of the horizontal sweep, the intensity is set to zero and the beam will 'fly back' to the start of the next line just below the previous line. This is known as *horizontal retrace*. This continues until the bottom of the display area is reached. Then again the intensity is set to zero and the beam will be redirected to the top of the display area, ready for the next refresh cycle. Bringing the beam back to the top of the display area is known as *vertical retrace*.

The electronic circuitry of the monitor that handles the scanning of the display area is controlled via the video input signal. The voltage level of the video input signal directly controls the beam intensity¹⁾. The proper relation between the video signal and the actual position of the electron beam is secured by means of synchronisation pulses. These pulses can be communicated via a separate *sync* signal. Also we find systems in which these synchronisation pulses are

¹⁾ Colour displays actually have three electron beams and three video input signals to be able to control the red, green and blue components of an image individually (see for instance [Foley90]).

superimposed onto the video signal (for colour systems composited with the green video signal only). This does not interfere with the beam intensity control, because synchronisation pulses show up in the vertical and the horizontal retrace time as illustrated in Figure 3.3.

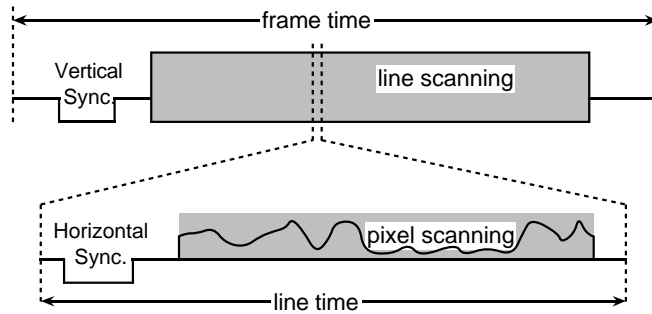


Figure 3.3: Composite sync. The video signal carries pixel brightness information as well as synchronisation pulses. The synchronisation pulses relate the brightness information with the position of the electron beam on the screen.

3.2.2. Display Controller

Images displayed on CRT's are transient, so that they have to be cyclically redisplayed. This cyclic redisplay — also known as the *refresh process* — is performed by the *display controller* at a sufficiently fast repetition rate (typically 50-100 Hz). The display controller reads the digital pixel data from the frame buffer. The display controller as a rule incorporates a lookup table which translates pixel values in intensity values that are input to the digital-to-analogue converters (DAC's)²⁾. A lookup table may be used for gamma correction or to expand 8-bit pixel values into 3×8 -bit colour intensities.

Normally, successive pixels are stored in a frame buffer on successive memory locations. Shires [Shires86] surveys the Intel 82786, a display processor that supports windowing in hardware. The CPU provides the description of the parts of various windows that are to be shown on the screen. The display processor then fetches the implied pixels.

The display controller also generates the horizontal and vertical synchronisation pulses, which ensures that the scanning of the display area by the electron beam controlled by the monitor circuitry is in lockstep with the scanning of the frame buffer performed by the display controller. As a result, the data representing a particular pixel defines the brightness of the exact same spot on the display area in each refresh cycle.

²⁾ The display controller of a colour system incorporates three DAC's. Each of these controls the red, the green, and the blue electron beam individually.

The output of the DAC('s) and the two types of synchronisation pulses superimposed on that forms the analogue video signal to control the monitor.

The function of a display controller may be simple, but the high speed of the circuitry involved makes the implementation non-trivial. A high resolution display has more than 1M pixels so that the maximum pixel rate is well over 100MHz. The BT485/125MHz RAMDAC [Brooktree88] is a popular chip for high resolution systems that takes care of the most critical issues. It not only incorporates a lookup table and a DAC, but also features five parallel pixel input ports. The pixels input via the multiple ports are serialised on-chip. As a result of this, the pixel data can be supplied at a five times lower rate. The importance of this will be addressed in the next subsection.

3.2.3. Frame Buffer

To assure a flicker-free display of an image on a CRT, the refresh process is time-bound. Traversal of the image generating pipeline is costly. Moreover, the execution time of some of the functions in the pipeline depends on the momentary complexity of the scene. This addresses the function of a *refresh buffer*. A refresh buffer contains the final product of the rendering process — that is, the set of display primitives that make up the image. Due to this, the refresh buffer decouples the refresh process from the rendering process so that the image representation stored in the refresh buffer is guaranteed to be continuously redisplayed within the time constraints imposed by the display medium. For raster graphics systems the refresh buffer is commonly known as the *frame buffer*. Due to the synchronisation mechanism described in the above, there is a one to one relation between the location of the data in the frame buffer and the position of the pixel on the display. As a result, a pixel's position is not stored in the frame buffer, only the value of its intensity. The number of bits of information stored per pixel (also, the number of *bitplanes*) varies, dependent on the type of system: a monochrome system can do with one bit per pixel. A professional full-colour system may have 36 bitplanes to store colour information (12 bits for each of the red, green and blue video signals) and, in addition to that, may have several bitplanes to store overlay, Z-buffer, alpha channel and even object identification data.

We noted before that for a high resolution system the sustained pixel rate may be well over 100MHz. Therefore, the display controller will have to be able to read all pixels at that rate. It would be ideal if the entire content of the buffer can also be updated in one refresh cycle. This implies that pixels have to be written into the frame buffer at a similar rate simultaneously. As a result, the frame buffer for a high-resolution system should have a total access rate of well over 200MHz³⁾.

³⁾ In fact, in case the system makes use of the Z-buffer algorithm for hidden surface removal, pixels are likely to be addressed several times per frame, so that even an access rate of 200MHz is insufficient to be able to replace the content of the entire buffer in one refresh

However, the memory access rate of a cheap conventional DRAM memory is in the order of 10MHz, which is clearly insufficient. This access problem is known as the 'frame buffer bottleneck'. In the following, several answers to reduce this memory access problem will be reviewed.

Separate bus

In a simple architecture, the frame buffer may share the memory address space with the rest of the system memory. The pixel values are stored in and retrieved from the frame buffer memory via the system bus (see Figure 3.4). In this way the CPU and the display controller have to compete for memory access. It may be clear that this architecture can be used for low performance low resolution systems only.

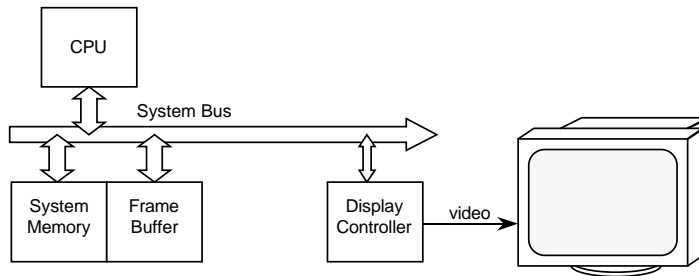


Figure 3.4: A simple graphics system in which the frame buffer access for both read and write takes place via the system bus.

A better access to the frame buffer can be obtained if the frame buffer is implemented as a dual ported memory. Then, pixels are retrieved from the frame buffer via a separate port. The display controller is connected to this port via the *pixel bus*. In this way the communication needed for the refresh process does not affect the system memory bus (see Figure 3.5). In the dual ported memory, display controller access via the pixel bus has priority over access via the system bus.

One step further in this direction is the complete separation of the frame buffer from the system memory. Frame buffer access is handled by a co-processor — the *graphics processor* — via a private bus, the *graphics processor bus* (see Figure 3.6). The frame buffer may again be implemented as a dual ported memory so that the communication needed for the refresh process does not affect the graphics processor bus. Of course, also in this architecture, display controller access via the pixel bus has priority over access via the graphics processor bus.

The result of separation of the system bus, the graphics processor bus and the pixel bus is that the effective number of cycles during which the frame buffer can be accessed increases.

cycle.

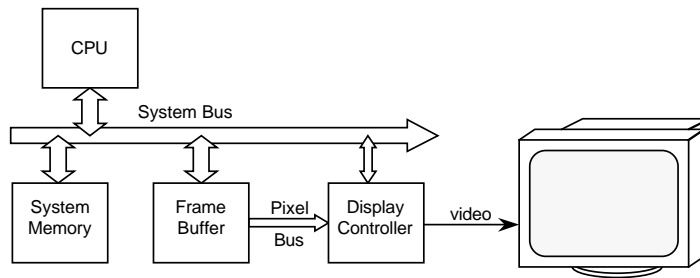


Figure 3.5: A graphics system in which the frame buffer is implemented as dual-ported memory, so that frame buffer access by the display controller does not affect the system bus data transport.

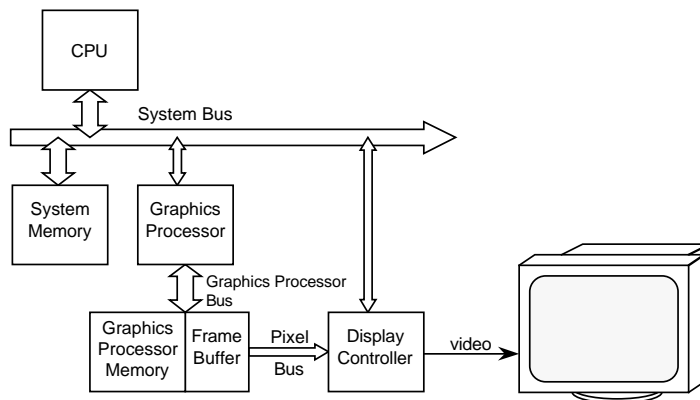


Figure 3.6: Graphics system centered around a graphics processor which off-loads the CPU. Also the frame buffer is completely separated from the system memory, which increases the effective frame buffer access rate.

Double buffering

We already pointed out that the display controller has to have access priority. This means that for high resolution systems the frame buffer is effectively blocked for updates, except during retrace times which can be a mere 10-20% of the total time — depending on system parameters like resolution and such. Limited access for frame buffer updates can be avoided by means of *double buffering*. Then, one buffer can be used for image display, while at the same time, the other is accessible for writing pixels for the next frame continuously. Whenever this next frame is ready for display, the function of the two buffers is swapped. Swapping of the two buffers can be synchronised with the vertical retrace. Then it is guaranteed that the individual frames displayed on the screen are consistent. If the calculation and update of individual frames alone takes more than one refresh cycle — which to a

certain extent could be acceptable even for animation— double buffering is mandatory. It may be clear that this solution is expensive in terms of hardware.

Video RAM

A “poor man’s” alternative to considerably widen the relatively small access window is offered by the *video RAM* (VRAM), a memory chip designed especially for frame buffer implementations. Just like a conventional memory chip, the VRAM allows for random access to any of its memory locations. What comes extra is a secondary data port. Via this port a serial register can be shifted out (and in). This serial register is as wide as the memory array and can be loaded with the data of one complete row of the memory in just one clock cycle. The data that resides in the serial register can be shifted out while at the same time the memory is available for random access read/write operations. Due to the parallel loading of the serial shift register, the number of cycles the frame buffer is inaccessible for writing because of the refresh process has effectively reduced with a factor equal to the width of the shift register (typically 256 or 512). The maximum rate of the serial port of most VRAM’s is typically less than 30MHz. Note that this rate is insufficient for high resolution systems. Also note that double buffering may still be needed in case the update of individual frames takes more than one refresh cycle.

VRAM’s are also used because it reduces the requirements on the address generator needed for pixel output.

Multiple banks

The access time of available memory chips (being any type of RAM or VRAM) is insufficient to accommodate the refresh of high resolution systems⁴⁾. To increase the bandwidth, the frame buffer can be configured as multiple banks that supply pixels in parallel (e.g., 5 banks for a 1280×1024 display). These pixels are then loaded in parallel into a high-speed shift register which is shifted out at the required rate. Such serialisation circuitry is often integrated in the DAC chip (see also subsection 3.2.2. Display Controller).

3.2.4. Graphics Processor

In one way or another, graphics systems usually incorporate a *graphics processor*—a dedicated processor that handles (some of the) tasks of the image synthesis pipeline (see §2.1.2.). Among graphics systems, we will find a great variety in which functionality is handled by such a graphics processor. As has been mentioned in the introduction of this chapter, throughout time, we have witnessed a constant change in which of the functions that make up the image synthesis pipeline are handled by the graphics processor and which functions are handled by the host CPU. Graphics standards such as GKS [ISO85b] and PHIGS [ISO89], made it possible to

⁴⁾ Note that for high resolution systems we need to store > 1 Mbyte, and we would need to have an access time < 10 ns, a low power consumption and preferably a low price at the same time.

pinpoint functionality that could be implemented in hardware. If we restrict ourselves to raster graphics systems, we have seen graphics processors that just handle scan-conversion up to processors that handle the complete GKS functionality [Günther88].

The 'wheel of reincarnation', first formulated by Ivan Sutherland in the 70's, is a philosophical tendency in graphics hardware design. Sutherland noticed that there is an inclination among system designers to off-load graphical computation from the CPU to a graphics processor, and then repeat this process (i.e., off-load the computation again from the graphics processor to a graphics sub-processor and so on). The process is repetitive, since when designers add more and more registers and functions to a display processor, the processor becomes a full fledged CPU again. At that stage though, the processor has become slow and inefficient, so that need arises for a small, fast processor, starting the cycle anew.

In [Gutttag86] Gutttag et al. discuss issues that must be taken into account in the design of a VLSI 32-bit microprocessor specialised for graphics applications. They particularly stress the requirement that such a processor should be general enough to perform any graphics operation. Examples of single chip graphics processors that adhere to this concept are Texas Instruments' TMS34020 and Intell's i860 (described in [Molnar90]). Both processors are perfect examples of the wheel of reincarnation philosophy; they are sufficiently general purpose to be used as a stand alone CPU as well.

On the other extreme we find for instance the Raster Engine (REX) that is incorporated in the IRIS Indigo [SiliconGraphics93]. The REX — implemented as a single high-density ASIC (Application Specific Integrated Circuit) — is a small, fast processor that primarily takes care of span interpolation for flat and linear shading with optional dithering. All other operations such as the viewing transformation, the decomposition into triangles and the calculation of the slopes of the triangles which finally leads to the generation of point-sampled spans are performed by the host CPU, a RISC processor that is sufficiently powerful to do so.

3.3. Concurrency and Graphics Systems

3.3.1. Introduction

Effective interactive computer graphics applications require considerable computing resources to guarantee a sufficiently fast response. In the last decade, we have witnessed a remarkable improvement of computing power due to improved processor technologies. This evolution can be expected to continue. However, to satisfy the need for higher image quality, scene complexity and interactivity, experts in the field estimate that four to five orders of magnitude more processing power than available in present day processors is necessary. Such a gap can only be bridged by making use of highly parallel multiprocessor systems.

The inherent nature of graphics algorithms makes that they are well suited to be implemented on multiprocessor systems. In increasing levels of complexity, the computing tasks can be organised based on pixels, vertices, polygons, patches,

objects or frames. Furthermore, the image generation pipeline consists of a number of clearly separable tasks. As a result graphics algorithms can be mapped onto numerous multiprocessor configuration alternatives. For isolated graphics operations — such as geometric transformations — a proper optimal projection onto parallel hardware is not hard to find (e.g., [Clark82]). However, if one considers the graphics pipeline as a whole, the situation is much more complex.

The characteristics of a particular application running on a multiprocessor configuration are a determining factor of the overall efficiency of the system. The numerous multiprocessor systems, academic as well as commercial, that can be found today have specific characteristics and are often well suited for a particular type of images and a particular type of rendering only. However, in today's computer graphics applications, we must be able to deal with more than just one type of images and more than just one type of rendering. We find a great variety in the internal characteristics of the image synthesis process, caused by the variety of shading techniques [Hall88], hidden surface removal algorithms [Sutherland74] and graphics primitives [Hopgood83, Dam88]. When we consider the fact that the characteristics of applications vary considerably, and in an interactive environment, even momentarily, it will be clear that it is impossible to speak about "the" general optimal multiprocessor architecture.

Concurrency can be viewed upon with different interests. For instance, one may focus on hardware configurations, i.e., the architecture of the system. From another point of view one may be concerned with the algorithmic aspects of concurrency, i.e., how a particular problem can be distributed. In the next we will summarise a classification based on these two different views, that is, based on architectures and based on distribution. In these classifications we restrict to those types of concurrency that can be found in existing graphics architectures.

3.3.2. Architecture Based Classification

In computer graphics systems, we recognise the following classes of parallel architectures: *SIMD*, *MIMD* and *systolic array*. These classes may come in different levels of granularity.

Classes

SIMD. A single instruction, multiple data system (also known as vector processor) is an arrangement of synchronously operating processors: all processing elements execute the same instruction in lock-step. The centralised control makes a SIMD system relatively cheap and well suited for massive parallelism. However, the efficiency of a SIMD system may reduce dramatically when conditional instructions are executed. SIMD systems are well suited for specialised tasks on multiple data streams.

MIMD. A multiple instruction, multiple data system is a configuration of asynchronously operating processors. Synchronisation is handled by message passing. Job scheduling and communication are the most critical aspects of a

MIMD system. Scalability and configurability made this type of system popular. On the other hand, this type of architecture is costly, since each processor must have its own instruction buffer and controller. Besides, debugging of a MIMD system can be quite tedious. Due to the asynchronous operation of the processing elements, a MIMD system can (at least potentially) handle data dependent operations quite efficiently. MIMD systems are the most flexible type of multiprocessor systems. In computer graphics systems we may find MIMD systems configured such that the processing elements asynchronously execute the same set of instructions on different portions of data (multi-stream), as a pure pipeline architecture (single-stream) or as a combination of these two extremes.

Systolic array. A systolic array processor can be considered as being a mixture of a pipeline architecture and a vector processor. Individual elements of the array handle instruction level tasks only. The dimensionality of the array and the interconnection of the processing elements — which may allow multiple data streams — tends to be problem specific. As such, a systolic array should be seen as a dedicated multi-stream processor rather than as a flexible programmable system. Information — which can be data as well as instructions — flows between processing elements in a pipelined fashion, as regular as a heartbeat (hence the name *systolic* array). The simple and regular communication and control structure makes the systolic array rather simple to design and implement.

Granularity

Beside the above classification of architectures, we can distinguish several levels of granularity. Especially for MIMD-type architectures, we may find concurrent systems that involve processing elements that range from *mainframes*, *workstations*, *loosely coupled* multi-processors, *tightly coupled* multi-processors, down to *instruction set* level multi-processors.

Mainframes. Processing can be off-loaded to a mainframe, a vector-processor, a supercomputer or even a cluster of supercomputers. These types of number crunching facilities can usually be accessed via a local-area network (LAN) or even a world-spanning wide-area network (WAN). Because of the mode in which these compute servers tend to operate and the characteristics of the network involved, this approach is typical for non-interactive highly compute intensive preprocessing tasks, which are outside the scope of this overview.

Workstations. A processing task can be distributed among a cluster of workstations, connected via a LAN. Such a cluster may very well be heterogeneous. Normal operating system facilities can be used to start remote procedures. Communication costs are relatively high. Although network technologies are improving, the response time of the facilities cannot be guaranteed to be sufficient for closed-loop interactions.

Loosely coupled multi-processors. In loosely coupled systems, the processors are connected to neighbouring processors via point-to-point communication links. Each processor has a private local memory. Data exchange and synchronisation is handled by sending messages. The communication cost is relatively low, that is, compared to communication via a LAN. In spite of this, communication costs can still become a dominating factor.

Tightly coupled multi-processors. Processors of a multi-processor system may share (a part of) their main memory. In that case they are said to be tightly coupled. The communication costs of tightly coupled multi-processor systems are low because bulky data can be shared and synchronisation can be obtained via shared status information. On the other hand, memory contention puts an upper limit on the number of processors that can efficiently be coupled in this way.

Instruction set level processors. Finally we have the category of instruction set level processors in which we find dataflow processors, neural networks and the like. These architectures execute machine instructions in parallel. A serious inconvenience of these type of architectures is that programming is non-trivial. The systolic array is an example of an instruction set level processor.

3.3.3. Distribution Based Classification

From an algorithmic point of view the following task distribution strategies for the image synthesis process can be recognised: *functional*, *image-space* and *object-space* subdivision.

Classes

Functional. A simple strategy to distribute the processing for any type of application is functional subdivision. For computer graphics systems this implies that processors are allocated for one of the tasks (or even sub-tasks) found in the image synthesis pipeline (geometric transformation, clipping, shading etc.). Because of the structure of the image synthesis pipeline (see § 2,1,2), this approach results in a serially pipelined architecture; data is passed from one processor to the next, each processor performing a specific task. The individual processing elements may be highly optimised for the specific task assigned to them. Vector transformations for instance requires floating point arithmetic, whereas rasterisation can do with integer operations.

Functional subdivision may seem to be an obvious strategy for graphics systems, however, one should realise that load balancing and capability of handling large data streams are essential for optimal efficiency. One of the characteristic features of the image synthesis process is that throughout the image synthesis pipeline the amount of data increases considerably. Although the processing elements involved may accommodate for this, it still is a potential problem for load balancing. The amount in which the data increases as well as the complexity of the processing tends to be data dependent. Also some of the tasks (e.g., clipping) may cause interrupts in the data-flow. These effects negatively influence the efficiency of a functionally

subdivided system.

The modules of a functionally subdivided system tend to be highly optimised, which implies that they are targeted for a particular type of primitive only. If more types of primitives are to be handled, the pipeline may have to be split up in different parallel path.

Image-space. Another task subdivision approach often used for distributed image synthesis is image-space subdivision; each processor performs all operations for one or more pixels of the image. The image synthesis process is basically organised within the following outer loop:

```
for each pixel do
  begin /* body of rendering algorithm */
    ...
    for each object do
      begin
        ...
      end
    ...
  end
```

The pixel-wise organisation of the image-space subdivided rendering process more or less reflects a major concern for the set of pixels that have to be produced. The rendering algorithm is raster dependent.

The maximum level of concurrency that can be obtained is determined by the image resolution, which is a constant, hence limited, factor. The efficiency of the image-space subdivision approach may be low in case the image complexity is not homogeneous. In that case, adaptive subdivision methods are favourable. However, adaptive subdivision methods considerably complicate the implementation.

An implication of image-space task subdivision is that multiple copies of geometric data have to be distributed among the processors. This goes with considerable communication costs, especially during start-up. For dynamic applications, image-space subdivision implies that multiple copies of the same data that exist throughout the system have to be updated, or may have to be retransmitted from one part of the system to another.

Object-space. A complementary approach, also known as object-space subdivision, is based on the principle that processors are allocated to process one or more graphical objects (e.g., lines, areas, characters). Here the image synthesis process is organised within the following outer loop:

```
for each object do
  begin /* body of rendering algorithm */
    ...
    for each pixel do
      begin
        ...
      end
    ...
  end
end
```

This object-based organisation reflects a possible interest for the objects that have to be processed. In this case, parts of the rendering algorithm can be raster independent. The measure of raster independency relates to the extend of the part in the inner loop.

The maximum level of concurrency that can be obtained is determined by the number of objects in the scene. Hence, the more complex the scene, the higher the maximum level of concurrency. Task scheduling for object-space subdivision can be complex, since the amount of processing required per object may differ —e.g., depending on the size of the object. Furthermore, in an interactive environment the number of objects in a scene is a dynamic quantity. These effects complicate load-balancing, so that it may be difficult to optimise the overall efficiency of the system.

Granularity

Also for concurrent algorithms we distinguish different levels of granularity. These are known as the classes *coarse*, *medium* and *fine* grain. This classification is based on the number of instructions executed between synchronisation events. The following definitions do not imply that there is a general consensus on the exact quantification of these levels.

Coarse grain. Tasks are executed with hardly any information exchange. Typically the number of instructions is over a thousand. The cost of communication is negligible in relation with the processing costs.

Medium grain. Tasks executed are in the order of tens to hundreds of instructions. The cost of communication is an important factor.

Fine grain. Tasks are up to several instructions long, the communication cost is a determining factor.

3.3.4. Discussion

Many solutions have been proposed to distribute the work involved in fast generation of high quality images across multiprocessor systems [Molnar90]. We have seen that all of the “pure” strategies mentioned above have their limitations.

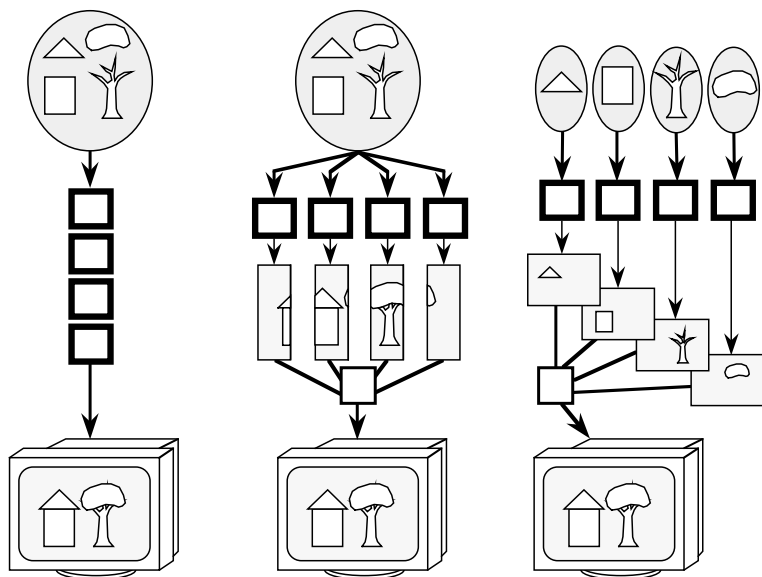


Figure 3.7: “Pure” subdivision strategies. On the left functional subdivision, in the middle image space subdivision and on the right object space subdivision. The image generation process itself is a pipeline of several sequential processes. Therefore it should come as no surprise that this type of multiprocessing can be found in a rudimentary form in even the first generation of graphics systems. However, subdivision of the image generation pipeline in smaller tasks can be done up to a limited number of steps only, so that the maximum degree of multiprocessing by means of pipelining only is limited. We noted before that, especially for a purely pipelined architecture, data dependent operations make it hard to balance the load. The longer the pipeline, the less efficient some modules will be used.

The ultimate pure image-space partitioned system would imply a processor per pixel. This many processors alone is impractical, but even a compromise — i.e., a processor for parts of the image — implies that all primitives have to be processed by all processors (i.e., the system is object-serial) Hence the throughput is limited by the processor speed.

The ultimate pure object-space partitioned system (a processor per object) results in multiple pixel streams that have to be combined (i.e., the system is pixel-serial). A simple solution implies synchronisation which reduces efficiency. An asynchronous solution requires extra memory and composition hardware.

Practical solutions in general are mixtures of several of these pure strategies. To obtain a sufficiently high degree of multiprocessing, systems designers have to combine pipelining and parallelism. Parallelism can then materialise in the form of parallel modules or even parallel pipelines. This leads to hybrid architectures such as the hypothetical architecture shown in Figure 3.8.

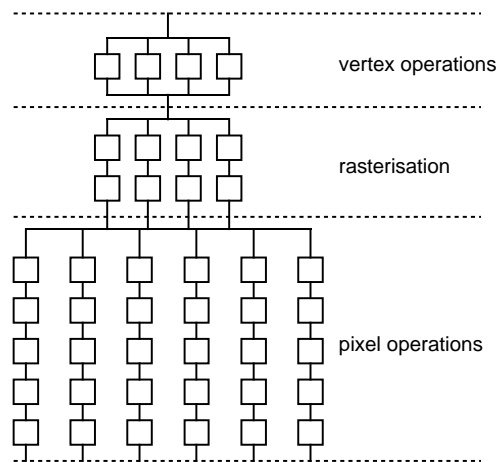


Figure 3.8: Example of a generic hybrid multiprocessor graphics system. Parts of the graphics pipeline are parallelised to obtain a sufficient amount of processing power.

When parallelising distinct image generation tasks we can choose between image-space subdivision and object-space subdivision for each of these tasks individually. Of these two, image-space subdivision seems to be more appropriate for the rasterisation phase, the phase that generates and operates on pixels. Object-space subdivision on the other hand seems to be more appropriate for the pre-rasterisation phase, the phase which deals with graphics primitives and their vertices.

3.4. Conclusions

In most of today's graphics systems we may recognise a geometry processor, a rasteriser and a frame buffer. Each of these modules are candidate for subdivision in parallel units to improve the speed of the system.

The demand for higher image quality implies that the number of frame buffer accesses per pixel will increase. Also the amount of information per pixel will increase. Traditional memory architectures cannot accommodate for that. What is needed is a high bandwidth between rasteriser and pixel memory — that is, much higher than the bandwidth offered by standard memory chips. In existing systems this is obtained by organising the frame buffer memory in parallel banks so that

multiple pixels can be accessed per memory cycle. For a high bandwidth the rasteriser and the pixel memory should be integrated preferably into one system module.

A Layered Object-Space Based Architecture

Synopsis.

The responsiveness of interactive applications is an important measure of the quality of the underlying graphics system. In this chapter we propose to improve responsiveness by means of reduction of the effort needed to perform interaction tasks. We describe the abstract computational model of an interactive graphics system and discuss requirements on output representation. It is argued how this leads to a layered object-space architecture. Some consequences of this approach on fundamental aspects of the graphics system are reviewed.

4.1. Introduction

An essential requirement for any graphics system is easy and versatile interaction. Raster graphics systems, as it stands today, are still not able to fulfill interaction requirements fully. The underlying technical reason for this is the discrepancy between objects that are defined by position and shape and their pixel representations viewed and pointed to by the user. The process of changing from object definition to raster representation destroys whatever high-level structure one has in the scene. Since one generally operates on a semantic level of interaction, the pixel representation per se is not a representation eligible for manipulation purposes. However, it is the pixel representation that in a great extent determines the algorithms and architectures of existing graphics systems.

Presently there exist two mainstream approaches to make raster graphics systems more viable for interaction:

- Perform each step involved in the graphical feedback loop (see § 2.1.4.) as quickly as possible by pushing the hardware limits to the maximum, viz., running many processors per given task in the image synthesis pipeline.

- Restructure the functional model (§ 2.1.2.) to reduce the computational complexity of the graphical feedback loop, viz., looking at image synthesis from a fresher perspective.

The principle of the first approach is that in cases where a proper representation level is not available for direct manipulation, it can at least be generated as quickly as possible. Obviously there is nothing wrong with this approach, as long as it stays cost effective and exhibits extensible (read 'programmable') behaviour. A common characteristic of the approach is then to identify and isolate a simple (subset of) operation(s) and map it, frequently in a conceptually simplistic manner, to hardware. Due to the availability of advanced VLSI design tools, there is a manifest incentive to do this. We, however, observe a major shortcoming with this approach. There are ultimate limits to what can be done by brute-force hardware speed-ups.

The computational complexity community has long ago come to know that the laws of parallel computation are qualitatively different from that of the sequential computation; algorithms do not always smoothly translate from uniprocessor to multiprocessor architectures. We believe that without clarifying the algorithmic improvements, brute-force mappings of existing graphical algorithms into hardware will introduce only temporary speed-ups and these improvements will be nullified in time by growing user demands.

One should be careful not to assign the problems of image synthesis to hardware designers solely. By concentrating on hardware design alone we may end up in a dead-end street from which it is hard to back-out again. Software issues should also be thoroughly studied. The real solution to the hard problems of computer graphics will come, in our view, from a direction which considers the intrinsic difficulty of user-driven problems from a computational standpoint. This brings us to the second approach.

The second approach is no different from the first one in terms of its goal, i.e., produce a responsive raster graphics system. Yet, the methodology is quite different. In this case, one analyses interaction tasks, and then tries to develop original data structures and algorithms and devise new architectural organisations that guarantee that for all interaction tasks representations of the proper level are at hand. Only then one maps expedient tasks into hardware as much as this is justified.

We subscribe to the second approach: we will first examine the structure of the image synthesis pipeline in relation with interaction requirements, and only then try to push the hardware limits to the maximum where this is needed. We will base a possible restructuring of the image synthesis pipeline on the following:

MAXIM 1: Reduction of the effort needed to perform interaction tasks in terms of computation and in terms of data storage and retrieval will lead to a more responsive graphics system.

In other words one should try to come to a 'lean production' model. Not lean in the sense of 'of poor quality', but lean in the sense of 'restrict to what is absolutely necessary' ¹⁾. This may seem to be a rather obvious strategy. If lean production is supposed to be appropriate for graphics systems, one should wonder why this architectural line is so little investigated.

To a certain extent this can be explained by the fact that in spite of the high level of assistance offered by modern VLSI design tools, hardware design is not simple. The tendency of system design is to find a simplified way to perform an isolated task that — often merely because of the simplification — can be implemented in VLSI. In contrast with this, when looking for a system-wide solution we often need more flexibility which in general leads to more complexity. As a result, we may even find that solutions preferred from a system-wide point-of-view are not suited to be implemented in VLSI.

Simplification is tempting, it seems to pay off rather well (as is unmistakably demonstrated by the success of RISC-technology). We argue that by looking at the conglomerate of tasks from a proper perspective, we may have to invest in details in order to gain in the end. Yet, it is not always clear how to estimate whether investments made are worth while. For a better understanding, we will first examine the abstract computational model of image synthesis and interaction tasks.

4.2. Abstract Computational Model

The image synthesis process can be defined in terms of abstract levels (see also § 2.1.2.). Upon traversing the image synthesis pipeline from the application towards the physical display we first encounter the highest level and at the end we find the lowest level of abstraction. Between these two we find several intermediate levels. Levels are interconnected by a bidirectional data stream (see Figure 4.1). The output data stream is the stream that flows in the direction of the display, the input data stream is the stream that flows in the opposite direction, i.e., towards the application interface. In each abstraction level we may distinguish (at least conceptually) between an input and an output environment. Both the input and the output environment have a representation store and processes that have access to that representation store. In Figure 4.1 these processes are represented by lines. An arrow directed towards the representation store indicates that the representation may be edited by the process. An arrow directed from the representation store indicates that the representation is used by that process.

¹⁾ Lean production is a term also known in the automobile industry [Womack91]. The essentials of lean car production are: just-in-time, flexibility, and quality circles. Of course in car production different measures play a role in the economics of the process, but still, the analogy between lean car production and lean image production is of interest.

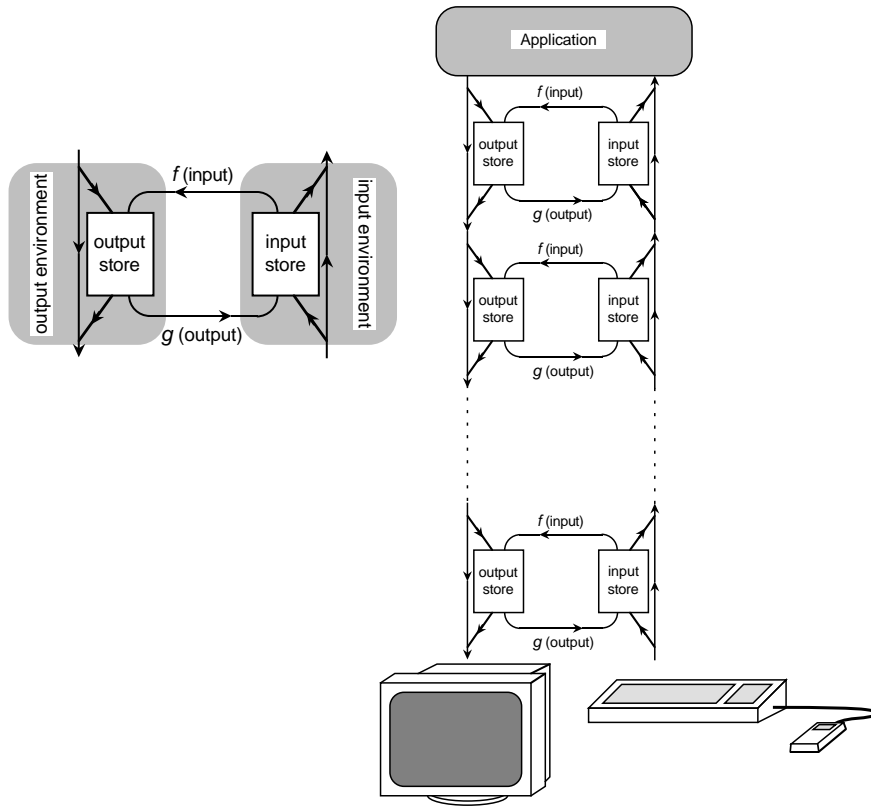


Figure 4.1: Computational model of one abstract level (left). The model is symmetrical with respect to the input and the output environment. The complete image synthesis pipeline is made up by several of these abstract levels (right).

Figure 4.1 should make clear that the model is symmetrical with respect to input and output. The logical input device in the input environment can pass the input through to a higher level, or invoke a function $f(\text{input})$ which causes changes in the output environment. Similarly, the output environment can pass the output through to a lower level, or invoke the function $g(\text{output})$ which causes a change of state in the input environment.

Via the function $f(\text{input})$, the output environment acts as the agent using a logical input device at the same level of abstraction (§ 2.1.3). In this way, a logical input device may directly control components of the state of the output environment at the same level of abstraction. An example of this is a logical input device which provides a transformation matrix used directly by the output environment (e.g. for dragging objects).

Similarly, via the function $g(\text{output})$, the computational model allows the output environment to be either a part of the state of the component processes of a logical

input device, or input to a process. An example of this is a pick logical input device for which the input is a location in the coordinate space and the current collection of graphical information at that level of abstraction.

Note that for graphical interaction, the model allows a logical input device to control components of the state of the output environment both directly, or via a path involving the transfer of input primitives to a higher level of abstraction, followed by an invocation of an output function which causes appropriate changes at the lower level. However, interpretation of Maxim 1 with respect to this computational model leads to:

MAXIM 2: Changes of the output representation as a direct result of an interaction task should be initiated from, and handled at, the lowest possible level of abstraction.

In other words, the path of a feedback loop should be as short as possible ²⁾. As a direct consequence of this maxim we come to the seemingly trivial:

MAXIM 3: Each level of abstraction, from which interaction tasks may initiate changes of the output representation, should be immediately available to the processes involved in interaction tasks.

The consequence of Maxim 3 is that the intended representation levels should not only be available conceptually, but should actually be stored in memory rather than being reproduced when necessary. Note that, in accordance with Maxim 1, this does not rule out the possibility to share data, e.g., between different representation levels, as long as this does not impede access.

Note that Maxim 3 does not suggest to store all intermediate results “just to be on the safe side”. Storage is costly; all intermediate representation levels that are actually stored should be accounted for.

²⁾ If we again compare the image synthesis pipeline with a car production line, then Maxim 2 states that upon a customers request for a car with a particular type of engine, we should allocate a semi-manufactured car that just arrived at the point of the production line where engines are built in, rather than to allocate a car-production slot that is about to enter the production line. In this way we are able to serve the customer as quickly and efficiently as possible.

4.3. Output Representation

With the logical model that is presented in § 2.1.2 in mind, one can reason that the logical processes to traverse and/or edit output representations —also known as *display files*— can be accommodated by appropriate structuring. This notion is represented by:

MAXIM 4: The content of a display file should reflect a basic concern with performance and structuring.

Performance. Performance results from fast execution of each primitive in the display file. The representation of primitives in combination with the algorithms that implement each logical process must emphasise the execution speed. The performance of the system will be determined by the slowest algorithm. Graphics algorithms can be implemented either in software or in silicon.

Structuring. The primitives are structured into groups to accommodate editing and traversing of the display file. The structure of a display file should support editing of 'group' information, for instance, to facilitate dragging of compound structures. At the same time, elementary information should also be directly accessible for editing, for instance, to support pick operations. An appropriate structuring of a display file will aid the logical process of generating the display file of a lower level based on a confined change of a display file of a higher level. It will be this generation speed which determines how display files should be structured.

Interpretation of Maxim 1 in this context leads to two basic notions related to editing of display files. These are the notion of incremental changes and the notion of local changes.

MAXIM 5: Algorithms that operate on the output representation should be able to make changes incrementally.

By incremental changes we mean changes that are relative to an existing state. In this way, an editing operation can be represented by updating an entity in a structured display file. The change takes effect by first letting the change propagate through the display file at hand and then generate from this updated display files on each of the lower levels of abstraction.

MAXIM 6: Algorithms that operate on the output representation should be able to maintain the notion of locality.

The notion of locality means that a change will affect a relatively small portion of

the display file. If the corresponding part in the display file can be easily identified, it is possible to minimise both the update efforts in the display file at hand, as well as the generation of the corresponding updated display files on each of the lower levels of abstraction.

We conclude that structuring of the output representation at the different levels of abstraction should be particularly helpful in supporting these types of incremental and local changes. Typical editing operations on the structured and logical display files that must be supported include the following.

Geometrical Transformations. The geometry of an object is represented in the structured display file by a group of primitives. An editing operation of particular interest is the transformation of the object as a whole as an essential element of the position task (see § 2.2). The structured display file must have a level where a complete group of primitives can be addressed and manipulated. With the appropriate output representation this editing operation can then reduce to a change of only one value. Naturally this change will have to propagate through to the lower levels of abstraction.

Colour Control. Little attention has been given to provide firmware for dynamic control of colours. Fast regeneration should include colour function evaluation. This can be done if, in the process of mapping the higher level display file onto the lower level, the area information and the colour function per area can be preserved. Then the effect of shading, reflection and transparency can be realised by colour function compositions.

Appearance Control. The attributes that are traditionally used for creating dynamic effects, such as highlighting, are primarily used for fast low level feedback. They affect either groups or individual output primitives. In both cases the corresponding elements must be easily tractable in the display file. Appearance control introduces no further requirements beyond those already encountered for transformations.

Insert and Delete. Insertion and deletion of objects in the display file require local rearrangements for proper hidden surface removal and lighting effects. Such rearrangements must be calculated quickly, followed by a reexecution of the hidden surface removal algorithm.

Picking Manipulations. Picking an output primitive is an operation which involves a search through the display file. Usually, the graphics system will provide some form of feedback to identify the picked primitive by, for example, high-lighting. Picking and feedback will be best served by functionality that identifies the relevant elements in real-time. Feedback will assist during picking. Higher level feedback will use upward references realised either by an explicit administration or by fast searching traversal.

4.4. A Multilayer Computational Model

Based on the set of maxims presented in this chapter, we now construct a computational model for the image generation pipeline. The basic principles of this model are: proper representation levels should be available for interaction tasks, and the aspects of editing display files as invoked by interaction tasks should be basis of the structuring of the display files.

A specification of the proper levels of abstraction is implicitly indicated by Maxim 3. Based on the inventory of interaction tasks presented in § 2.2., we conclude that — apart from the application model — three geometry based levels of image representation in particular must be accessible for interaction purposes. In [Hagen87, Akman88] these representation levels are named *high*, *medium* and *low*.

DEFINITION: The high level representation contains objects of representation types generated by the application model, in the (3D) coordinate system of that application model (known as *world coordinates* in graphics standards like GKS and PHIGS). In principle there is no limit on the type of primitives that can be present ³⁾. At this level the representation is view independent.

DEFINITION: The medium level representation contains graphical objects that result from a perspective or projective transformation, viz., it is view dependent. All objects of any type in the the high level are mapped to objects represented by means of just one type of primitive, the *pattern*. A pattern is a “2½D” *domain* ⁴⁾ associated with a *colour function* that assigns a colour for each point in the domain (see also § 5.2). Each pattern of the medium level representation maintains a reference to the primitive of the high level representation from which it originated. Patterns are associated with instructions that operate on domains and colour functions ⁵⁾. Besides patterns and associated instructions, the display list also contains structuring elements which determine an acyclic directed graph that specifies the order of execution of the display list elements.

³⁾ For practical reasons we will restrict to polygons. The system can be extended by providing a module to map a specific representation into the generic representation used in the medium level. For “standard” mappings such modules will be provided.

⁴⁾ On this level and below, z-coordinates are relevant only for visibility calculation, which involves determination of the *relative* depth ordering. This ‘minor’ role of the z-coordinate is reflected in the representation of a domain (see Chapter 5).

⁵⁾ Examples of instructions associated with patterns of the medium and the low level representation are geometric transformations, colour transformations and dynamic attributes. These type of instructions are part of the display files and can be associated with individual patterns or with a group of patterns. We also make use of *global instructions* which have to be accessible for all mapping processes, and are therefore not part of the display files of a particular level. By means of global instructions we specify for instance the lighting

DEFINITION: The low level representation is a view dependent list of non-overlapping visible areas that result from hidden surface removal of the patterns of the medium level representation. The areas are represented by means of patterns only. Each pattern maintains a reference to the pattern of the medium level representation from which it originated. The patterns are associated with instructions that operate on domains and colour functions⁵⁾. The structure of the display list is based on geometrical position, rather than on logical coherence. In § 5.3 this data structure will be discussed in detail.

With these definitions of three intermediate representation levels, we come to the layered architecture shown in Figure 4.2. Note that all the representation levels of this layered architecture are in object-space, that is, each of the levels has distinct identifiable graphical objects rather than a collection of anonymous picture elements.

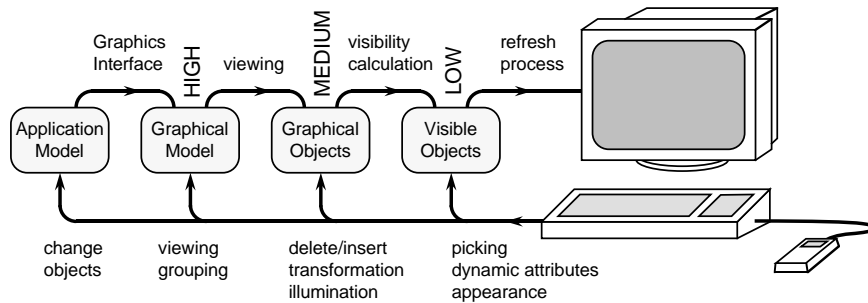


Figure 4.2: A multilayer architecture. Interaction tasks can be initiated from intermediate representation levels.

It is required that we maintain consistency among the different representation levels. For example, after a change, hidden elements may have to be recovered which necessitates going back to higher levels. Therefore, each element of the low level maintains a reference to the corresponding element in the medium level representation, and similarly, each element of the medium level maintains a reference to the corresponding element in the high level representation.

We will consider the representations as objects in the sense of object-oriented languages. Each object at any given level has a set of associated "methods" that it responds to. When an action is performed on an object, the object itself may initiate a series of actions to keep the data structures intact. Some actions require permanent changes in the objects at the high level (e.g., deleting an object). Some actions are just temporary (e.g., making an object transparent to reveal what is behind). These actions may be effectuated by providing each object with the

conditions.

appropriate methods. The user need not and should not know how these administrative tasks are carried out. All the user wants is to see the immediate changes that he wants to carry out, and is to be guaranteed that all through the interaction tasks the geometry is kept correct.

We claim that this layered architecture provides support for interaction tasks because of the following features:

- Selection of visible primitives (picking), as most essential interaction task, is supported by the low level representation directly. The representation of patterns as well as the geometry-based data structure of the low level representation, are optimised for position-based identification (see Chapter 5). Referencing of primitives from one level to the primitives of the next higher level makes that the identification process can effortlessly propagate to higher levels.
- Each of the three intermediate representation levels can be accessed immediately, so that changes for interaction tasks can be initiated from each of these levels. As a result, the path of a feedback loop can be minimised.
- The structure of the display files as discussed in § 4.3 in combination with the stored intermediate representation levels, makes it possible to apply incremental changes so that the propagation of changes can be restricted and affect a mere subset of the image information. In this way, the effort needed to update the output representation, also of each of the lower levels, can be minimised.

4.5. Discussion

Based on the viewpoint presented here, pixels can be considered as being artifacts of the raster technology. Interaction support as such does not account for a pixel representation of the image: none of the interaction tasks mentioned in § 2.2. operate on pixels directly. A pixel representation is unstructured and does not support the notion of locality and incremental changes (see Maxim's 4, 5 and 6). In the multilayer architecture proposed here the refresh process can therefore at least conceptually be initiated from the low level, a level that contains a list of visible areas rather than an array of pixels ⁶⁾.

This point of view is in contrast with what we often find in other architectures where pixels play a dominant role in both the rendering algorithms and the structure of the architecture. For instance in [Fournier88] Fournier and Fussel present a formal study in which they present arguments to make use of a rasterised representation level. Their paper:

⁶⁾ Note that strictly speaking this does not rule out the possibility to introduce an extra representation level, for instance when forced by technical issues.

(...) constitutes a first attempt at a disciplined analysis of the power of a frame buffer seen as a computational engine for use in graphics algorithms. We (that is, Fournier and Fussel) show the inherent power of frame buffers to perform a number of graphics algorithms in terms of the number of data fields (registers) required per pixel, the types of operations allowed on these registers, and the input data.

In their point of view the frame buffer is seen as a major memory resource. Their abstract model of a frame buffer is an array of automata each with a number of registers. The study involves the time and space complexity of several algorithms. According to them, the simple time-space trade-off they show for some of the problems, and the drop in cost of memory caused by technological changes, provide new motivation for the use of frame buffers with the capability of doing more than simply storing a digital representation of a scene to be displayed on a raster monitor.

There is nothing wrong with reconsidering the trade-off between memory usage and the speed of a computation. However, one should be careful. Technical issues rather than a drop in cost of memory should be considered when arguing for making use of frame buffers for doing more than simply storing a digital representation of a scene. As has been explained in § 3.2.3, systems designers already have to face the frame buffer bottleneck. By concentrating on the trade-off between memory usage and the speed of a computation alone we are tempted to add more and more information per pixel just to try to speed up algorithms. Yet, more information per pixel implies more communication — hence, an increase of the frame buffer access problem. Also, with a limited number of registers per pixel it is not possible to give an exact answer to the visible surface problem or to solve the exact area sampling problem [Fournier88]. It is clear that in practice, the amount of information (i.e., the number of registers) per pixel has to be limited.

Pixels are there, and people (ab)use them. In our opinion, the problems of pixel-based systems indicate that pixels should not play a dominant role for graphics systems in both the rendering algorithms and the structure of the system's architecture. We consider pixels merely as being artifacts of the raster technology that indeed will have to be produced somewhere along the line.

We observed that all three intermediate levels of our multilayer architecture are in object-space. This has certainly consequences for the hidden surface removal and shading methods. Also, in this view the refresh process is initiated from the low level (i.e., a list of areas rather than an array of pixels), which has consequences for the display controller. While this is unusual, we believe that it is well worth investigating. In the next we will have a closer look at these consequences.

4.5.1. Hidden Surface Removal

In the proposed multilayer architecture, the mapping of the medium level onto the low level representation involves hidden surface removal. This implies that our hidden surface removal algorithm is an object-space algorithm since both the medium and the low level representation are object-space representations.

Apart from some high-end flight simulators, object-space hidden surface removal algorithms are not what we normally find in graphics systems. The reason for that is primarily because object-space algorithms are considered to be difficult to implement, expensive and, consequently, slow. In contrast, the most frequently used image-space hidden surface algorithm, the so-called Z-buffer algorithm, is considered to be simple, cheap and fast. So the question does arise: "can we afford an object-space hidden surface removal algorithm in a supposedly interactive environment?" When we compare these alternative approaches, we should consider the following motivations:

- An object-space hidden surface removal algorithm takes place before rasterisation of the graphics primitives. This may reduce the data stream in the last section of the pipeline (dependent on the average size of the objects). An object-space algorithm avoids rasterising primitives that in the end turn out to be invisible.
- Image-space methods do not produce a visibility map. As a result a redraw of the complete scene is necessary to find out which object is visible at a certain image location ⁷⁾.
- Object-space algorithms are independent of the resolution of the display device. Consequently, certain operations (such as zooming) do not require a reexecution of the object-space hidden surface removal algorithm at all, as would be the case for image-space methods.
- Image-space algorithms necessitate a redraw of the entire image whenever one primitive is moved or deleted, whereas an object-space approach may provide facilities for incremental updates.
- Image-space hidden surface removal implies point sampling. Point sampling inherently leads to artifacts, and thus to a lower image quality. For instance, objects smaller than the distance between two sample points may or may not contribute to the final image, dependent on the position of these objects relative to the sample grid.
- Object-space algorithms allow for exact area sampling, a technique that produces a weight factor for the intensity per pixel. Exact area sampling can be seen as the ultimate supersampling technique ⁸⁾ which increases the apparent resolution and thereby reduces aliasing effects.
- The parts of a scene that are illuminated by a light source are exactly those parts that would be visible when viewed from the position of the light source. This illustrates how an object-space hidden surface removal algorithm can also be used for calculating shadows.

⁷⁾ This redraw may be an optimised process, it is not necessary to actually produce pixels.

⁸⁾ Supersampling is a well known anti-aliasing technique. It is based on increasing the number of samples per pixel. The resulting pixel value is taken to be the average of all sampled values.

We bring forward that under certain circumstances an object-space hidden surface removal algorithm may even be competitive. The cost of the object space hidden surface removal algorithm is compensated in particular by not rasterising primitives that are not visible. These savings depend on the shading techniques used. This illustrates the importance of looking at problems from a proper perspective as mentioned in § 4.1. As shading techniques improve in quality and consequently become more expensive, it essentially becomes more expensive to opt for image-space hidden surface removal. Then we should rather ask ourselves: “can we afford not to use an object-space hidden surface removal algorithm in a supposedly interactive environment?”

In Chapter 5 we will present an object-space hidden surface removal algorithm and analyse the cost effectiveness in more detail.

4.5.2. Illumination and Shading

The definitions of the medium and the low level representation state that in both levels we find one type of primitive only, i.e., the pattern. Object-oriented methods associate objects and the methods these objects respond to. The architecture presented prescribes abstraction levels based on geometry, not based on the evaluation of shading methods. Also, global instructions — for instance the once that describe the lighting conditions — are accessible for mapping processes at each of the levels of abstraction. This provides flexibility with respect to the level of abstraction at which certain parts of the colour evaluation have to take place. In this way, different shading methods and evaluation strategies can be incorporated in the same architecture. This flexibility holds in particular for colour evaluation that can take place at the medium and low level of the architecture. The separation of domain and colour function as characteristic feature of the pattern representation in combination with object-oriented methods makes it possible to implement optimal evaluation strategies.

In general, lazy evaluation will be the preferred strategy. Then, colour function evaluation takes place at the low level rather than at the medium level representation. In this way, we avoid that evaluation of the colour function is performed for primitives that are invisible. On the other hand, there are situations for which it would be most welcome if the evaluation of the colour function for primitives that are invisible has already taken place. Take for example a primitives that is about to become visible, for instance due to movement of an object that momentarily obscures that primitive. Then colour function evaluation would not have to take place at the very moment that that primitive becomes visible, which may be during a real-time feedback loop.

We will briefly address this issue in § 6.2.3. Here we restrict ourselves to say that the layered object-space architecture provides sufficient flexibility to be able to support different colour function evaluation strategies.

4.5.3. Display Controller

In § 4.5 we stated that the refresh process can conceptually be initiated from the low level representation. For the display of continuous motion in general a frame rate of 24 Hz — the standard movie projector frame rate — is sufficient⁹⁾. In § 3.2.2 we learned that the refresh rate for CRT displays is 50-100 Hz. If we compare this with the required frame rate, we observe that there is a difference of a mere factor of 3. As a result we find ourselves confronted with a trade-off between computation and storage. It is tempting to relax the system requirements (i.e., rasterise frames at a rate of 24 Hz) and introduce an intermediate frame store (note, this does not necessarily have to be a pixel based representation).

However, since for interaction purposes there is no mandate for such a frame store, we will refresh the display directly from the low level representation. Given the real-time requirements of the refresh process, it is clear that a display controller that actually operates from the low level representation is quite a challenge. Such a system would bring the fastest possible response on interactions. In Chapter 7 we will present such an object-space display controller architecture.

4.6. Conclusions

We argued that reducing the effort needed to perform interaction tasks will lead to a more responsive system. An architecture that strictly adheres to the concepts as presented in this chapter has far-reaching consequences for the type of hidden surface removal algorithm that can be used. In turn this has consequences for the rasterisation process (the possibility to do exact area sampling) and shading technique (lazy shading). From this we conclude that the feasibility of this approach has to be demonstrated by the design and implementation of a practical object-space hidden surface removal algorithm. This and other issues of dealing with geometry will be discussed in Chapter 5.

We also concluded that the low level representation of the architecture — that contains information in the form of non-overlapping areas — can be used to directly supply information for the refresh process. We decided to design and implement a display controller that can actually refresh the display based on this type of information. In doing so, we obtain a raster graphics system of which the interactivity is comparable to that of the vector graphics systems that were quite common in the early days of computer graphics. In order to be successful, such an object based display controller should still be able to display realistically shaded area's. For this we needed a high quality shading method that is optimised for fast rasterisation. Such a method will be presented in Chapter 6, whereas the object based display controller itself will be discussed in Chapter 7.

⁹⁾ Note that frame rate is not to be confused with the refresh rate (see also § 2.3.2).

Dealing with Geometry

Synopsis.

The main theme of this chapter is how to deal with geometry and geometrical operations in a display resolution independent way. We will attend only that part of the layered object-space architecture that comes after the viewing transformation. This chapter covers representation of objects, storage structures, and geometry based processes like hidden surface removal and scan-conversion. An exact incremental hidden surface removal algorithm is presented which is based on binary operations on surface elements. By means of these operations, individual objects can be added, deleted and made transparent, and incremental changes can be confined to those objects of which the visibility is changed. The scan-conversion process involves exact area sampling as anti-aliasing method.

5.1. Introduction

Throughout the years, the common interest of applications that pushed the development of computer graphics systems has been the ability to interact with what is displayed on a screen. Although raster graphics systems offered more functionality and improved the image quality by being able to display solid shaded areas, they could replace the vector graphics systems in interactive environments only after the introduction of (hardware implemented) techniques that allowed fast incremental operations on the image, the so-called bitmap operations (or rasterops). These types of operations for 2D graphics interaction could be implemented successfully primarily because raster graphics systems provide direct access to the complete 2D image information on which the bitmap operations operate.

Increased performance of processors and development of graphics algorithms made it possible to introduce 3D graphics applications, even for low end computer systems. Also for interactive 3D applications support of incremental changes on the image is obligatory. The implementation of operations that perform incremental changes for 3D applications — analogous to bitmap operations that perform incremental changes for 2D applications — should be based on operations on a 3D representation. Here we face an intrinsic problem: even for so-called 3D computer

graphics systems, the image representation produced by the system is two-dimensional only ¹⁾.

A fundamental step in generating images of 3D scenes is clipping and hidden surface removal: the resulting image consists of (parts of) surfaces that are visible from a certain position in space. Several types of algorithms exist to tackle this classic computer graphics problem [Sutherland74]. These algorithms come in two fundamentally different types, i.e., *image-space* and *object-space* hidden surface removal.

The 3D graphics systems on the market today make use of hardware implemented image-space hidden surface removal. Image-space hidden surface removal implies that for each pixel in the image the system infers which object is visible. This reflects how these systems evolved from 2D raster graphics systems. It is relatively simple to add hardware to a 2D raster graphics system and obtain an efficient implementation of the Z-buffer algorithm — one of the simplest image-space hidden surface removal algorithms [Foley90]. However, a consequence of applying the Z-buffer algorithm is that information is lost: the result of the hidden surface removal cannot be cross-referenced to the objects in the higher levels of abstraction. As a result, the system has no direct overview of which objects are visible for the user so that support for incremental changes is hampered. Therefore we conclude that this technology driven evolution from 2D to 3D graphics systems has inherent difficulties to meet the basic requirements for interactive 3D applications (see Chapter 4).

An exact (i.e., object-space) hidden surface removal algorithm on the other hand, operates in 3D space. It computes which parts of the input objects are visible, and so produces 3D objects in a form independent of the resolution of the display device. These visible elements can refer directly to objects in the higher level models (up to the application model). Via these references, the object-space approach allows for incremental operations. Temporary changes can be handled efficiently. In our view object-space based hidden surface removal algorithms are a sound basis for 3D graphics interaction.

In § 4.5.1 we listed motivations that should also be considered when comparing the two alternative classes of hidden surface removal algorithms. In summary: by applying image-space hidden surface removal essential geometrical information is lost, hence it is impossible to combine the Z-buffer algorithm with exact anti-aliasing. Also, the image-space algorithms produce results that are valid for a particular resolution only and cannot be scaled arbitrarily. Image-space algorithms operate at the very end of the image generation pipeline. This implies that all objects are actually drawn — i.e., all pixel values of each object are calculated —

¹⁾ Systems that incorporate a so-called Z-buffer maintain information of the maximum z-value only, all other z-values are lost. Therefore it cannot be considered as being 3D information.

before the visibility is checked²⁾. On the other hand, object-space hidden surface removal is computationally complex and the implementation is dependent on the representation of the objects.

We bring forward that, in spite of the computational complexity, under certain circumstances an object-space hidden surface removal algorithm can be competitive. Namely, the cost of object-space hidden surface removal is compensated in particular by not rasterising primitives that are not visible. These costs can be substantial, depending on the depth complexity of the scene, on the complexity of the shading method used, and on the number of light sources. The cost of the Z-buffer algorithm is measured in the total number of pixels covered by all the input primitives. As we will see, the cost of the object-space hidden surface algorithm is quasi output sensitive —that is, the cost relates to the number of primitives that turn out to be visible, not to the number of pixels covered. Since the costs of the two methods are based on different measures, it is not possible to compare these methods directly without specification of the characteristics of a scene and the rendering method used.

As we will see in § 5.4.6, given a reasonable depth complexity, not rasterising invisible (parts of) Gouraud shaded primitives does not make up for the costs of the object-space algorithm. Yet, for more advanced techniques the object-space algorithm may result in a reduction of the total cost of rendering that in cases can be substantial. The current trend towards more realistic graphics, even for interactive applications, will lead to more complex shading that includes texturing and shadows. Object-space hidden surface removal may then save on costs. The fundamental questions for hidden surface removal and calculation of shadows —being: “which objects are visible?” and “which objects are illuminated?” — can be answered by the same object-space algorithm. Applications that allow direct manipulation in 3D need collision detection, a closely related object-space problem. By making use of these shared interests the overall efficiency of object-space algorithms will increase.

It should be clear that for an efficient object-space hidden surface removal algorithm, carefully designed data structures are essential. These structures also have to be suited for dynamic modifications. Therefore, we will first concentrate on data representations before we will discuss the hidden surface removal algorithm and scan-conversion process.

²⁾ In principle shading calculations can be deferred until after the image-space hidden surface removal calculation at the cost of storage of substantially more information per pixel [Deering88, Lastra95]. However, this is costly and aggravates the frame buffer access problem.

5.2. Representation of Geometry : the Domain

In this context, the graphics primitive used after the viewing transformation is called a *pattern*. Patterns describe the visual aspects of an area of the image [Hagen85, Hagen86b]. Patterns are the main elements of the medium and the low level representation of the layered architecture (§ 4.4). The geometry of a pattern is described by its *domain*, whereas a *colour function* assigns a colour for each point in that domain.

A domain is a 2½D raster independent representation of the geometry that is optimised for the geometrical operations that have to be performed on it — such as hidden surface removal and scan-conversion³⁾. For this, the domain description includes an extent, the representation is guaranteed to be free of pathological cases and the geometrical data are sorted. These features of the domain representation will be discussed in more detail in the following sub-sections.

5.2.1. The Use of Extents

In algorithms involved in image generation we frequently find the very basic “point-in-primitive” test⁴⁾. The point in primitive test is also the kernel of the selection task, which is essential for interaction (see also § 4.4).

The cost of a point-in-primitive test is dependent on the geometrical and topological complexity of the primitive. In its bare form, it involves calculation of line intersections, each of which takes several additions, subtractions and multiplications. It is not efficient to spend that much computational effort on primitives that are relatively far away from a given point P, as most of them will be. A more efficient approach is to first get a rough indication of the relative positions by making use of a simple 2D enclosure which fully contains the primitive projected on the x-y plane. By means of enclosures, the relative expensive point-in-primitive test can be traded for a cheaper point-in-enclosure test. If P is found to be on the outside of the enclosure, it will be on the outside of the primitive as well. Only when P is found to be within the enclosure, the exact and more expensive point-in-primitive test will tell us whether P is an interior or exterior point.

The efficiency of the use of enclosures depends on the average complexity of the primitives and on the type of enclosure. That is, how complicated is the point in enclosure test and how well does the enclosure ‘fit’. If the enclosure fits tightly, the number of point-in-primitive tests will be optimally reduced. On the other hand, tight fitting enclosures may be costly to produce and may have a more costly point-

³⁾ Domains are said to be 2½D objects, because after the viewing transformation the position of the domain in the image is determined by the x and y coordinates alone. Projection of domains on the 2D image plane is done by parallel projection (i.e., simply by nullifying the z-coordinate).

⁴⁾ The point-in-primitive test referred to here is a test of a point in the projected primitive, i.e., a test in the 2D x-y plane.

in-enclosure test. In deciding on a suitable type of enclosure we have to consider these effects.

Through the use of enclosures, the costs of the point-in-primitive test for a scene that contains N primitives is $N \times C_E + E \times C_P$, where E is the number of enclosures inside which P lies, C_E is the cost of the point-in-enclosure test, and C_P is the average cost of the point-in-primitive test. The method will pay off when $C_E < C_P \times \frac{(N-E)}{N}$. Note that this does not include the costs of producing the enclosure.

Efficient enclosures are circles and axes-aligned rectangles [Foley90]. We opted for the axes-aligned rectangle also known as *extent* (E) or *bounding box*⁵⁾ because it is extremely simple to produce and the cost of the point-in-extent test is at maximum 4 compares:

```
if (Px < El || Px > Er || Py < Eb || Py > Et)
    return (FALSE);
```

In which the subscripts l , r , t and b stand for left, right, top and bottom.

To illustrate the efficiency of the method, assume that $C_E < \frac{1}{10} C_P$ (this will be true for even a simple triangular area if the enclosure is an axes-aligned rectangle). Then the method already pays off when P is outside the enclosure of at least 10% of the primitives.

The test to determine if two primitives overlap is even more complex, and subsequently more costly, than the point-in-primitive test. It is worth noting that the cost of the point-in-enclosure test C_E for axes-aligned rectangles is equal to the cost to test for overlap of two extents. The test for overlapping extents is:

```
if (E1r < E2l || E1l > E2r || E1t < E2b || E1b > E2t)
    return (FALSE);
```

Consequently it will pay off even more to test for overlap of extents before testing for overlap of the primitives themselves.

⁵⁾ The term "bounding box" is somewhat confusing, because in general a box is associated with a 3D object, whereas here it is supposed to refer to a 2D entity. To avoid possible confusion only the term "extent" will be used.

5.2.2. Area Representation

For computer graphics applications, areas — being 2D or 3D primitives — are usually represented as polygons; a list of vertices which — by the order of the vertices — describes the outline of an area (see Figure 5.1). This type of area representation is popular mainly because of its ostensible simplicity. However, when analysing the computational complexity the representation is not simple at all. All of the vertices have to be taken into account to be able to determine whether a point is an interior or exterior point. Due to this, polygon-based geometrical operations — such as area-filling and determination of the union or intersection of areas — are complex. This was the motivation for the design of the pattern representation. Patterns are planar primitives. As a result the geometry of a 3D pattern (i.e., its domain) can be described by the coefficients of a plane equation and the projection of the domain onto the x - y plane; a 2D area. If necessary, the third dimension of each point in the domain can be reproduced by means of the coefficients of the plane equation.

2D areas can be described in several ways. In [Hagen85] ten Hagen suggested a raster independent representation based on scanlines and scanpoints. From this representation other, more practical, representations were derived. Each of these representations can represent flat regions in continuous 3D space without limitations on complexity of the topology — that is connected, disconnected, with holes, convex or concave.

What follows is a presentation of three area representations, followed by a discussion on the characteristics. In this discussion, the representation of the third dimension is conveniently left out.

Scanline-Scanpoint representation

The Scanline-Scanpoint representation, first presented in [Hagen85], is based on horizontal scanlines and points on these lines. The scanlines are sorted on their y -value and can be found on arbitrary positions, however, their number is minimal. In other words, the representation does not include scanlines which can unambiguously be reconstructed on the bases of information of other scanlines. Consequently, we find scanlines only at positions where the outline changes (see Figure 5.1). Each scanline has a number of scanpoints that are specified by their position on the scanline and by their "type". Scanpoints are sorted on their x -value. The type information is necessary to make the representation unambiguous.

An important aspect of the Scanline-Scanpoint representation is that — unlike the polygon representation — it is not necessary to scan the complete area description to determine the topology of the pattern around a certain point. The geometrical values in the representation are sorted, so that merely two scanlines, the one above and the one below the point, contain all necessary information on the topology around the point.

When put in practice, operations on the Scanline-Scanpoint representation turned out to be quite complex, mainly because for every operation the scanpoints of two successive scanlines had to be related with each other.

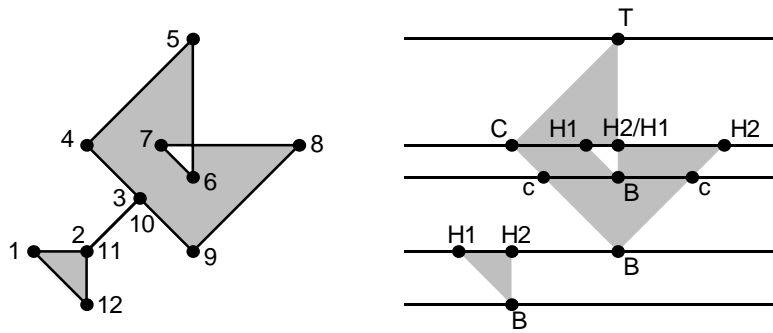


Figure 5.1: Polygon representation (left) versus Scanline-Scanpoint representation (right). The former representation is merely an ordered list of vertex coordinates. In the latter representation, the topology of a 2D area is unambiguously described by the position of the scanlines and the position and type of the scanpoints.

Scanline-Edge representation

To avoid the necessity to relate scanpoints of two successive scanlines, we constructed the Scanline-Edge representation. Like the Scanline-Scanpoint representation, the Scanline-Edge representation has scanlines only at those positions of the domain where the outline changes. Each scanline has a number of edges that are specified by their position on the scanline and by their slope [Kuijk88]. Scanlines and edges are sorted on their relative position. Note that each scanline necessarily has an even number of edges and that the topmost delimiting scanline has no edges at all (see Figure 5.2). Horizontal edges are discarded. The information that relates to one scanline is self-contained. That is, the position of one single scanline and the edges that relate to that scanline is sufficient to unambiguously describe the topology of the area on and above that scanline up to the next.

The following algorithm — which scans from bottom to top— converts a traditional polygon representation into a Scanline-Edge representation:

```
make edge_list                                /* sorted on:
                                             y.bot else on
                                             x.bot else on
                                             slope */
initialise y on lowest y.bot encountered during sort
```

(next page ...)

```

while (edges in edge_list) do
  begin
    if (y.bot == y)
      insert into active_list  /* sorted on:
                                x.val else on
                                slope else
                                remove redundancy */
    generate scanline (y)      /* based on active_list */
    if (y.top == y)
      remove edge from active_list
    y.next = minimum (  y.bot /* of edges in edge_list */,
                       y.top /* of edges in active_list */)
    calculate x.val of edges in active_list at y.next
    while (disordered x.val in active_list) do
      begin
        calculate y.val of intersections
        y.int = minimum (y.val of intersections)
        generate scanline (y.int)
        exchange edges crossing at y.int
      end
    y = y.next
  end
end

```

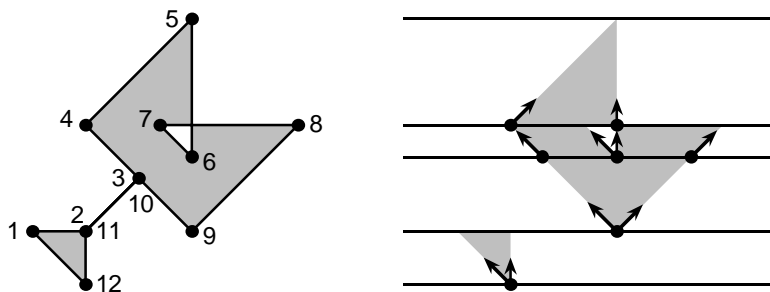


Figure 5.2: Polygon representation versus Scanline-Edge representation. The topology of a 2D area is unambiguously described by the position of the scanlines and the position and slope of the edges.

Note that two successive vertices of the polygon's vertex list make up the edges for edge_list. The coordinates of the endpoints of an edge are (x.bot, y.bot) and (x.top, y.top). The algorithm is based on a sweepline technique just as the scanline algorithm described in [Foley90] used for rasterisation of polygons except that it:

- operates in continuous coordinate space,

- generates scanlines at particular positions only (vertices and intersections),
- removes redundant information (double, aligned and horizontal edges).

Several binary operations that operate on the Scanline-Edge representation have been successfully implemented which resulted in a working object-space hidden surface removal algorithm (see § 5.4). The implementation of these operations was quite complicated, primarily because of the fact that the slices between two scanlines are of different height. Furthermore, the representation has redundant information: for an edge that spans several scanlines the position and slope is specified at each of these scanlines. This led us to experiment with the following representation.

Sorted-Edge representation

Unlike the two representations above, the Sorted-Edge representation is not scanline-based. It is a compact representation which comprises a sorted list of edges. When generating this representation from a polygon representation, horizontal edges and double edges are discarded, connected edges in direct line are joined together, and possible intersecting edges are eliminated by subdividing one of the edges of each pair that would otherwise intersect. In this way pathological cases are avoided and redundant information is removed.

Edges that describe a closed area can be sorted in different ways. For algorithms based on sweepline techniques for instance, we may be interested in the order in which edges become active or inactive. This would suggest an ordering on y_{bot} or y_{top} (see Figure 5.3). However, for stabbing queries for a particular y value⁶⁾ it would be useful if the ordering of edges would be such that edges are reported monotonically in x by simply following the list of edges, that is, if the edges are sorted on their relative x position. The example in Figure 5.3 illustrates that these options in general lead to different orderings.

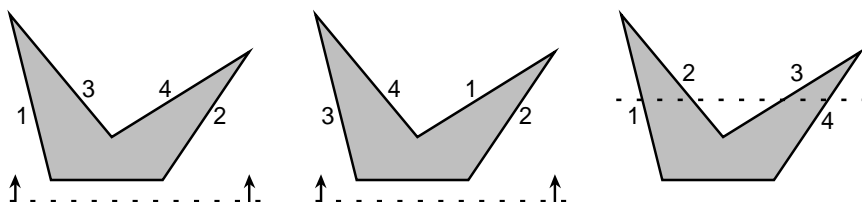


Figure 5.3: The order in which edges become active (left) or become inactive (middle) when sweeping in y -direction and ordering of edges for stabbing queries (right).

Since any set of numbers $\subseteq \mathbb{R}$ can be ordered, ordering on y_{bot} or y_{top} is always possible. Whether edges can be sorted on their relative x -position for all

⁶⁾ In other words, queries that report the set of edges that are 'active' on a particular y -value — i.e., edges that intersect the line $F(x, y) = y$.

possible y -values is less obvious. It would not be useful if the sorting would depend on the y -value, in other words, if there are situations for which the order of the sorted edges are different for different y -values. This indicates the relevance of the following:

THEOREM: There exists an ordering of nonintersecting nonhorizontal edges of a domain, for which for all possible y -values stabbing queries that follow that order report edges monotonically in x .

PROOF: Let $E_1 \cdots E_n$ be an ordering of edges for y_1 so that a stabbing query following that order reports S_1 , a set of monotonically ordered edges. Let S_2 be the set of edges reported for y_2 ($y_2 \neq y_1$) by following the same order $E_1 \cdots E_n$. If two successive edges $E_i, E_j \in S_2$ are incorrectly ordered, then one of the following statements has to be true.

- 1) E_i and E_j share a point at y_1 .
- 2) E_i and/or $E_j \notin S_1$.
- 3) E_i and E_j intersect between y_1 and y_2 .

If 1) is true E_i and E_j can be interchanged and S_1 will still be monotonically ordered. If 2) is true E_i and E_j can be interchanged without consequences for the monotonic ordering of S_1 . Statement 3) cannot be true, since it is in conflict with the assumption that edges do not intersect.

This implies that reordering $E_1 \cdots E_n$ to obtain a correctly ordered set for y_2 can be done without consequences for the monotonic ordering of S_1 . Based on the same reasoning, incorrectly ordered edges can be interchanged for any y -value. Hence it is possible to come to a set of edges correctly ordered for all y -values. \square

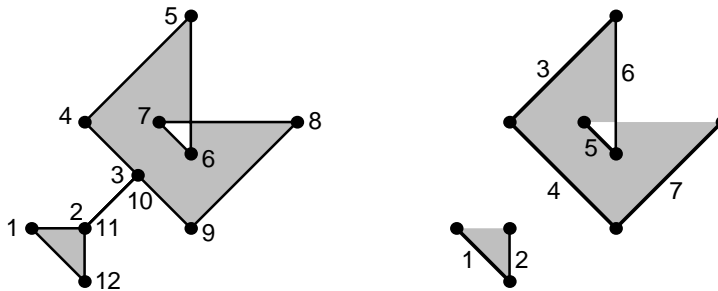


Figure 5.4: Polygon representation versus Sorted-Edge representation. In the Sorted-Edge representation redundant edges such as horizontal and double edges are removed, and aligned edges are joined together. Intersecting edges are subdivided (Note: only nonhorizontal edges).

As it turns out, an ordering of edges based on the relative x -position is more useful than ordering on y_{bot} or y_{top} . This can be explained by the following. When scan-converting a domain, each edge becomes active and inactive just once,

whereas the relative order in x-direction is of interest for every pixel row. Also in other geometry-based algorithms the order of edges in x-direction is used quite often. Therefore a representation in which the edges are ordered based on the relative x-position (Figure 5.4) makes the implementation of these algorithms simpler and more efficient at the same time.

A domain is a closed area so that two edges share one vertex. To reduce the amount of memory needed for storage, each edge in the Sorted-Edge representation is described by pointers (indices) to the two vertices that make up the endpoints of the edge rather than by the vertex coordinates. Vertex coordinates are stored in an array (see Figure 5.5) in principle in arbitrary order. The sorted edge list comes in the form of an array of sorted index pairs. Since the number of vertices of a domain will be relatively small, the number of bits needed for one index (8) is small compared to the number of bits needed for the coordinates of one vertex (128). To accommodate algorithms operating on the representation, the vector coordinates — that in principle can be stored in arbitrary order — are sorted in y.

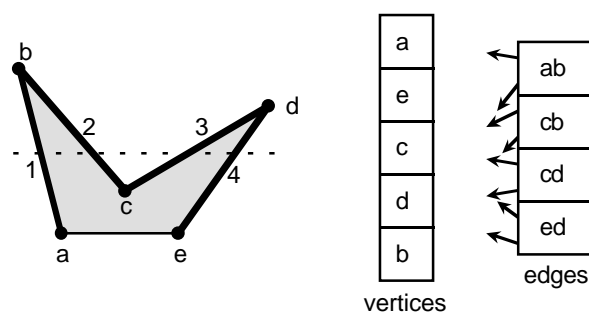


Figure 5.5: The Sorted-Edge representation is an array of vertices and an array of sorted edges which are pointers (indices) to these vertices.

The Sorted-Edge representation is generated from a polygon representation using the following algorithm:

```

make edge_list                                /* sorted on:
                                              y.bot else on
                                              x.bot else on
                                              slope */
initialise y on lowest y.bot encountered during sort
while (edges in edge_list) do
  begin
    if (y.bot == y) {
      insert into sorted-edge_list /* sorted on:
                                  x.val else on
                                  slope else
                                  remove redundancy */
      mark active                  /* in sorted-edge_list */
    }
    if (y.top == y)
      mark inactive;
    y.next = minimum (  y.bot /* of edges in edge_list */,
                      y.top /* of active edges */)
    calculate x.val of active edges at y.next
    while (disordered x.val of active edges) do
      begin
        calculate y.val of intersections
        y.int = minimum (y.val of intersections)
        cut off one edge of pairs crossing at y.int
        mark cut-off edge inactive
        insert remainder into sorted-edge_list
                                /* sorted on:
                                x.val else on
                                slope else
                                remove redundancy */
        mark remainder active    /* in sorted-edge_list */
      end
    y = y.next
  end
end

```

Discussion

How do the above representations relate to each other, and, what is the effect on efficiency? In this discussion the characteristics of the polygon, the Sorted-Edge and the Scanline-Edge representation will be compared. The Scanline-Scanpoint representation is left out of this discussion, because it is considered to be impractical.

October 26, 1995

The Scanline-Edge representation basically subdivides domains in a number of slices, each of which can be processed individually. Because of this, fine grained distributed processing based on individual slices is possible for operations that involve one domain only. Operations that involve two or more input domains are more complicated. In general slices of different domains are not aligned. In practice this makes it impossible to distribute processing on a slice basis and binary operations such as union or intersection of domains may lead to redundant scanlines that have to be removed.

The Sorted-Edge representation cannot be subdivided without cost. Consequently, the Sorted-Edge representation by nature should be processed as a whole.

The cost to store several simple objects using a polygon representation, the Scanline-Edge representation and the Sorted-Edge representation is listed in Table 5.1. From this we see that the Scanline-Edge representation is the least efficient in terms of storage. What we get in return for this is the possibility to treat slices as separate entities.





	polygon	scanline	sortedge
	6C (384)	11C (704)	6C + 6P (432)
	8C (512)	16C (1024)	8C + 8P (576)
	8C (512)	6C (384)	8C + 4P (544)
	8C (512)	21C (1344)	10C + 10P (720)

Table 5.1: Costs to store areas in different representations, expressed in coordinates (C) and indices (P). Between parentheses the corresponding number of bits needed are shown (assuming that C takes 64 and P takes 8 bits).

Due to the sorting of data of both the Scanline-Edge and Sorted-Edge representations several time-critical operations on domains, such as:

- a point-in-domain test,
- clipping,
- union and intersection of domains, and
- scan-conversion

can be done efficiently. All these operations profit from the one-time investment of sorting of the geometrical data. The overall gain depends on which operations occur most frequently, on what possible restrictions are imposed on the geometrical data (e.g., grid-aligned), and on the characteristics of the targeted processor architecture. For simple domains, such as triangles and quadrilaterals the Sorted-Edge representation seems to be favourable. For more complex domains, the Scanline-Edge representation may be more efficient. A further study on these representations in particular with respect to multi-processing is planned.

5.3. Efficient Storing of Domains

For hidden surface removal, all domains that overlap have to be found. As will be shown in § 5.4, an incremental hidden surface removal algorithm involves finding the subset of domains overlapping one query domain. An efficient first step is to find among all domains those domains of which the extent overlaps the extent of the query domain as explained in § 5.2.1. There we have learned that the test for overlapping extents is relatively cheap (see expression 5.2). However, for a serious dynamic application the total number of domains can be quite large so that it is necessary to reduce the search space in order to avoid costly memory access.

Reduction of the search space implies a proper organisation of the data store. This can be done by subdivision of the space in partitions. Partitions can be found in the form of a grid [Franklin80, Slater88], horizontal strips [Sechrest82], quad-trees [Warnock69] or more object based organisations such as BSP-trees [Fuchs80]. In one way or another, each of these space partitions maintains a list of objects associated with that partition. In case an object falls within several partitions, each of these partitions maintains a reference to that one object. As a result, a coarse subdivision implies long lists of objects per partition, a fine subdivision will increase the average number of references to the objects. Therefore, Franklin et al. [Franklin86], proposed to adapt the space subdivision for optimal tuning on the characteristics of a scene.

Theoretically optimal results can be obtained by balanced tree structures such as segment trees [Bentley80, Wood85, Mairson88, Berg92, Krefeld92]. These data structures lead to theoretically optimal algorithms, yet in practice these are difficult to implement [Snippe92]. Also, especially for dynamic applications, the cost of setting up and maintaining a data structure should be related to the savings on extent overlap tests resulting from reduction of the search space. The asymptotic behaviour of an overlap reporting algorithm may be important, but should be judged in the proper perspective. After all, the number of visible domains will be limited, simply because of the limited resolution of the display device. It is not worthwhile investing in complicated, theoretically optimal well balanced tree structures with high setup costs.

We sought a suitable structure in which each object is referenced just once. This to make it simple to make changes. The structure should lead to near optimal reduction of the search space and yet should be easy to implement because of possible implementation in VLSI.

The most simple structures are based on regular space subdivision. A disadvantage of regular subdivisions is that the distribution of objects in space may not be homogeneous, which reduces the effectiveness of such subdivisions. Linear-sorted lists are not optimal. However, if the lists are short, the simplicity obtained may compensate the theoretical inefficiency. In the next two sub-sections it is shown that a simple combination of these two types of arrangements leads to useful results.

We started by saying that the first step on a query will involve a check based on the extent. Because of this, the position of a domain in the data structures we present is determined by its extent, i.e., by (one or more of) the values `domain.top`, `domain.bot`, `domain.left`, `domain.right`.

5.3.1. Slice-List Structure

A simple arrangement of objects first presented in [Kuijk88] is the Slice-List structure. For this structure the visible space is subdivided in y-direction, leading to a number of horizontal slices of equal width. A linear list of domains is associated with each slice (see Figure 5.6). The position of domains stored in the Slice-List structure is based on the following two criteria:

- 1) The list in which a domain is stored is the list associated to the segment within which `domain.bot` falls.
- 2) In the lists, domains are sorted on decreasing `domain.top`.

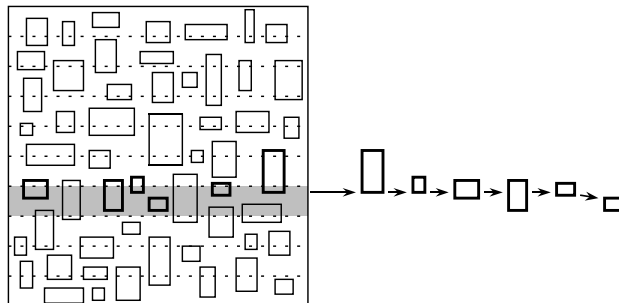


Figure 5.6: The Slice-List data structure. Each slice has a list of domains of which the bottommost point of the extent falls in that region. In the lists, the domains are ordered based on the y-value of the topmost point of the extent.

Note that this arrangement — that will be used several times more — shows just a few nonoverlapping extents. A more realistic picture would be cluttered with lots of extents of which the majority overlap.

The intent of this ordering is that a query for overlapping extents primarily addresses those domains that overlap a query extent in y-direction (see Figure 5.7). Domains in the lists associated with slices above the query extent (the dark grey area) and domains in the lists that come after a domain of which `domain.top` is below the query extent (the light grey area) cannot overlap. If overlap in y-direction is found, in the sequel, the test for overlap in x-direction will have to take place.

The preference for ordering in y-direction is related to the requirements for scan-conversion. In that process all domains that contribute to a specific scanline have to be found in an efficient way. (More on scan-conversion in § 5.5.)

The following algorithm will report all domains of which the extent overlaps the query extent `Q`:

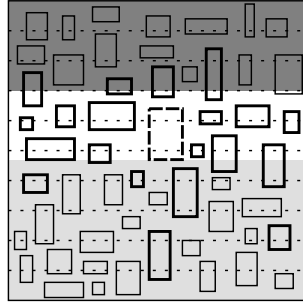


Figure 5.7: The Slice-List data structure reduces addressing domains outside an interval of y-values when searching for extents overlapping a query extent (shown dashed). The highlighted domains will be addressed only.

```

for (i = 1; slice[i-1].bot < Q.top; i = i + 1) do
  begin
    domain = first-domain-in-slice;
    while (domain.top > Q.bot) do
      begin
        if (!( domain.right < Q.left ||
                domain.left > Q.right ||
                domain.bot > Q.top) 7))
          report (domain);
          domain = domain.next;
          if (domain == NULL)
            break;
      end
    end
  end
end

```

It will be clear that the Slice-List structure needs $O(N)$ space, since all domains are stored (i.e., referenced) just once. The time complexity of the query is $O(N)$: in the worst case all N domains have to be checked for overlap. More of interest is the asymptotical behaviour in a well behaved expected case.

Let the range in y-direction (Y) be covered by m slices. Then the average length of the lists in the slices will be N/m . Assume that the domains are evenly distributed and of average height h . Then the number of domains that will be addressed for a query extent with $Q.top$ in the k^{th} segment and height h_Q , is on average:

⁷⁾ The last slice for which $slice.bot < Q.top$ may contain domains for which $domain.bot > Q.top$. To avoid reporting of those nonoverlapping extents the test $domain.bot > Q.top$ cannot be spared.

$$\frac{N}{Y} \cdot (h + h_Q) + \left(\frac{k}{m} - \frac{Q.top}{Y} \right) \cdot \frac{N}{m} + k \quad (5.1)$$

The first term represents the average number of domains that actually overlap in y -direction (N/Y is the average density per unit length, $h + h_Q$ is the interval to be considered). The other terms relate to domains that are addressed, but do not overlap in y -direction. More specifically, the second term represents the average “round-off error” introduced by the finite resolution of the space subdivision. In other words, it is the average number of domains in slice k for which $domain.bot > Q.top$. In this term $k/m - Q.top/Y$ is the corresponding proportion of the slice and N/m is the average slice density. The third term, k , reflects the fact that in each slice that is addressed (i.e., $1 \dots k$), the first domain with $domain.top > Q.bot$ has to be found in order to stop the loop. Note that k is directly proportional to m . Also note that because of this term the cost of the query depends on y .

This dependency on y can be reduced if h_{max} , the maximum height of all domains stored, is known. Then the `for`-loop in the search algorithm can start with $i = k - \left\lceil (h_Q + h_{max}) \cdot m / Y \right\rceil$ so that the average cost of a query is:

$$\frac{N}{Y} \cdot (h + h_Q) + \left(\frac{k}{m} - \frac{Q.top}{Y} \right) \cdot \frac{N}{m} + \min \left[\left\lceil \frac{(h_Q + h_{max}) \cdot m}{Y} \right\rceil, k \right] \quad (5.2)$$

which is independent of the y value (except when $k < \left\lceil (h_Q + h_{max}) \cdot m / Y \right\rceil$). The extra reduction in search-space is illustrated in Figure 5.8. The lists associated with the dark grey shaded slices are not visited at all.

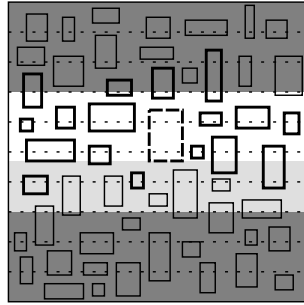


Figure 5.8: Reduction of the number of domains that have to be checked if the maximum height of all domains stored is known. Only the highlighted domains will be addressed.

The second and the third terms of expression 5.2 depend on the number of slices m . In order to reduce the value of the second term, the number of slices should be increased. This not only reduces the average number of domains in a slice N/m , it also reduces the maximum distance between $Q.top$ and the maximum value of $domain.bot$ in slice k . However, increasing the number of slices will increase the cost as represented by the last term. Note that this last term is independent of N , so

that the optimal number of slices is dependent on the domain arrangement at hand. This more or less suggests an adaptive approach, analogous to the adaptive grid solution as described in [Franklin86].

In general there will be a relation between N and h_{\max} . Domains in this structure do not overlap. Consequently, the more domains there are, the smaller h_{\max} is likely to be. When optimising the number of slices, also the cost to build and modify the structure should be considered. These costs relate directly to N/m , the average length of the lists in the slices. We will continue on cost and less well behaved distributions in § 5.3.3.

5.3.2. Grid-List Structure

The Slice-List data structure resulted in a reduction of the search space in y -direction only. We wanted to extend this idea and obtain a similar reduction of the search space in x -direction. For this, the one dimensional slice structure is replaced by a two dimensional grid structure. A linear list of domains is associated with each grid-cell (see Figure 5.9). Unlike the grid structures proposed in [Franklin80] and [Slater88] that store a list of all objects that intersect a grid-cell, we store domains just once. The position of domains stored in the Grid-List structure is based on the following two criteria:

- 1) The list in which a domain is stored is the list associated to the cell within which the point (domain.left, domain.bot) falls.
- 2) In the lists, domains are sorted on decreasing (domain.right + domain.top).

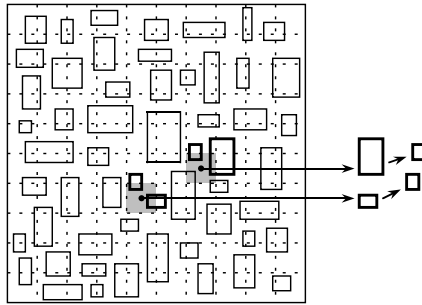


Figure 5.9: The Grid-List data structure. Each grid-cell has a list of domains of which the lower-left point of the extent falls in that region. In the lists, the domains are ordered based on the sum of the x and y coordinates of the upper-right point of the extent.

The particular ordering in the list makes that a query for overlapping extents primarily addresses domains that are confined within a triangular space determined by the query extent (see Figure 5.10). Domains in the lists associated with cells above and on the right of the query extent (the dark grey area) cannot overlap. Also, domains in the lists that come after a domain of which $\text{domain.right} + \text{domain.top}$ is below the line $x + y = Q.\text{left} + Q.\text{bottom}$ (the light grey area) cannot overlap.

The following algorithm will report all domains of which the extent overlaps the query extent Q :

```

for (j=1; cell[i-1][j-1].bot<Q.top; j=j+1) do
  begin
    for (i=1; cell[i-1][j-1].left<Q.right; i=i+1) do
      begin
        domain = first-domain-in-cell;
        while (domain.(right+top) > Q.(left+bot)) do
          begin
            if (!( domain.right < Q.left ||
                    domain.top   < Q.bot   ||
                    domain.left  > Q.right ||
                    domain.bot   > Q.top) 8))
              report (domain);
            domain = domain.next;
            if (domain == NULL)
              break;
          end
        end
      end
    end
  end
end

```

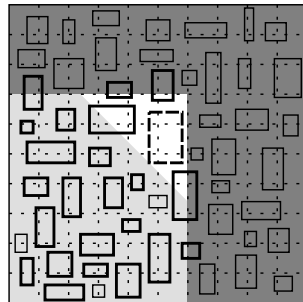


Figure 5.10: The Grid-List data structure reduces addressing domains outside a triangular area defined by a query extent (shown dashed) when searching for overlapping extents. The highlighted domains will be addressed only.

The Grid-List structure needs $O(N)$ space, since all domains are stored (i.e., referenced) just once. The time complexity of the query is $O(N)$: in the worst case all N domains have to be checked for overlap. More of interest is the asymptotical behaviour in a well behaved expected case. Like for the Slice-List structure, the behaviour of the algorithm can be improved by reducing the number of cycles of the `for`-loops if the maximum height h_{\max} and the maximum width w_{\max} of all domains stored so far are known. Since in this case, there are two nested loops, the reduction

is even more significant. The extra reduction in search-space is illustrated in Figure 5.11. The lists associated with the dark grey shaded cells are not visited at all. Also, this reduction makes the cost of the query independent of the position of the query domain⁹⁾.

Let the visible space ($X \times Y$) be covered by a grid of $m \times m$ cells. Then the average length of the lists in the cells will be N/m^2 . Assume that the domains are evenly distributed, of average width w and of average height h . Let the maximum width of all domains in the structure be w_{\max} and the maximum height be h_{\max} . Then the number of domains that will be addressed for a query extent with $Q.\text{top}$ in row j , $Q.\text{right}$ in column j , height h_Q and width w_Q , is on average:

$$\begin{aligned} & \frac{N}{X \cdot Y} \cdot \frac{(w + w_Q + h + h_Q)^2}{2} \\ & + \left(j \cdot \left(\frac{i}{m} - \frac{Q.\text{right}}{X} \right) + i \cdot \left(\frac{j}{m} - \frac{Q.\text{top}}{Y} \right) - \left(\frac{i}{m} - \frac{Q.\text{right}}{X} \right) \cdot \left(\frac{j}{m} - \frac{Q.\text{top}}{Y} \right) \right) \cdot \frac{N}{m^2} \\ & + \min \left[\left\lceil \frac{(w_Q + w_{\max}) \cdot m}{Y} \right\rceil, i \right] \cdot \min \left[\left\lceil \frac{(h_Q + h_{\max}) \cdot m}{X} \right\rceil, j \right] \end{aligned} \quad (5.3)$$

The first term represents the average number of domains that overlap a right-angled isosceles triangle, the white area shown in Figure 5.10. $N/(X \cdot Y)$ is the average density per unit area and $(w + w_Q + h + h_Q)^2/2$ is the area of the triangle. The other terms relate to domains that are addressed, but do not overlap the triangle. Much like what we have seen in expression 5.2, the second term represents the average "round-off error" introduced by the finite resolution of the space subdivision and the last term reflects the fact that in each cell that is addressed the first domain beyond the region of interest has to be found to stop the `for`-loops.

We learn from 5.3 that the total cost for a particular domain arrangement is related to the number of grid-cells. Increasing the number of cells will reduce the cost as defined by the second term, yet increase the cost indicated by the last term. When optimising the number of cells, also the cost to build and modify the structure should be considered. This cost relates directly to N/m^2 , the average length of the lists in the cells. We will continue on cost and less well behaved distributions in the next subsection.

⁸⁾ Similar to the algorithm for the Slice-List structure the tests `domain.left > Q.right` and `domain.bot > Q.top` cannot be spared.

⁹⁾ With exceptions for small i and j , similar to the exception for small k for the Slice-List structure.

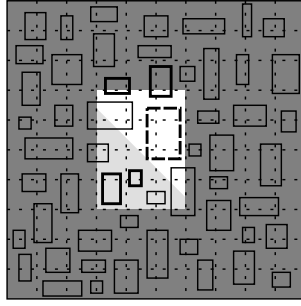


Figure 5.11: Reduction of the number of domains that have to be checked if the maximum height and width of all domains stored is known. Only the highlighted domains will be addressed.

5.3.3. Results

If we evaluate the Slice-List and the Grid-List data structures based on the reduction in search space we have to relate the results expressed in 5.2 and 5.3 to the average number of domains that actually overlap:

$$\frac{N}{X \cdot Y} \cdot (w + w_Q) \cdot (h + h_Q) \quad (5.4)$$

If we compare this with the first terms of 5.2 and 5.3 and assume that $w_Q \ll X$ and $h_Q \ll Y$ it is clear that the Grid-List structure potentially results in a better reduction in search space. A comparison of these two structures should also include the total number of grid-cells and slices and the investment in memory that goes with it. Both slices as well as cells contain just pointers to the first domain of the sorted list. Therefore the amount of memory involved is relatively small. It should be noted that the cost to build and modify the structure in both cases relates to the number of items in the linear list, which is inversely proportional to the number of slices or cells and thus inversely proportional to the investment in memory.

To test the behaviour more quantitatively, also in less optimal situations, several test scenes with different characteristics have been used (see Figure 5.12). These test scenes — that have been selected to represent the characteristics of various applications — are the following:

Random. This scene consists of homogeneously distributed domains. The size of the extents in x- and y-direction is 0.0075-0.0125 times the window size. The total number of domains in the scene is 100K, the average size is 10^{-4} , so that the average coverage is 10.

Landscape. The landscape distribution is a distribution in which the number of domains near the horizon (on 2/3 rd of the image height) increases while the size of

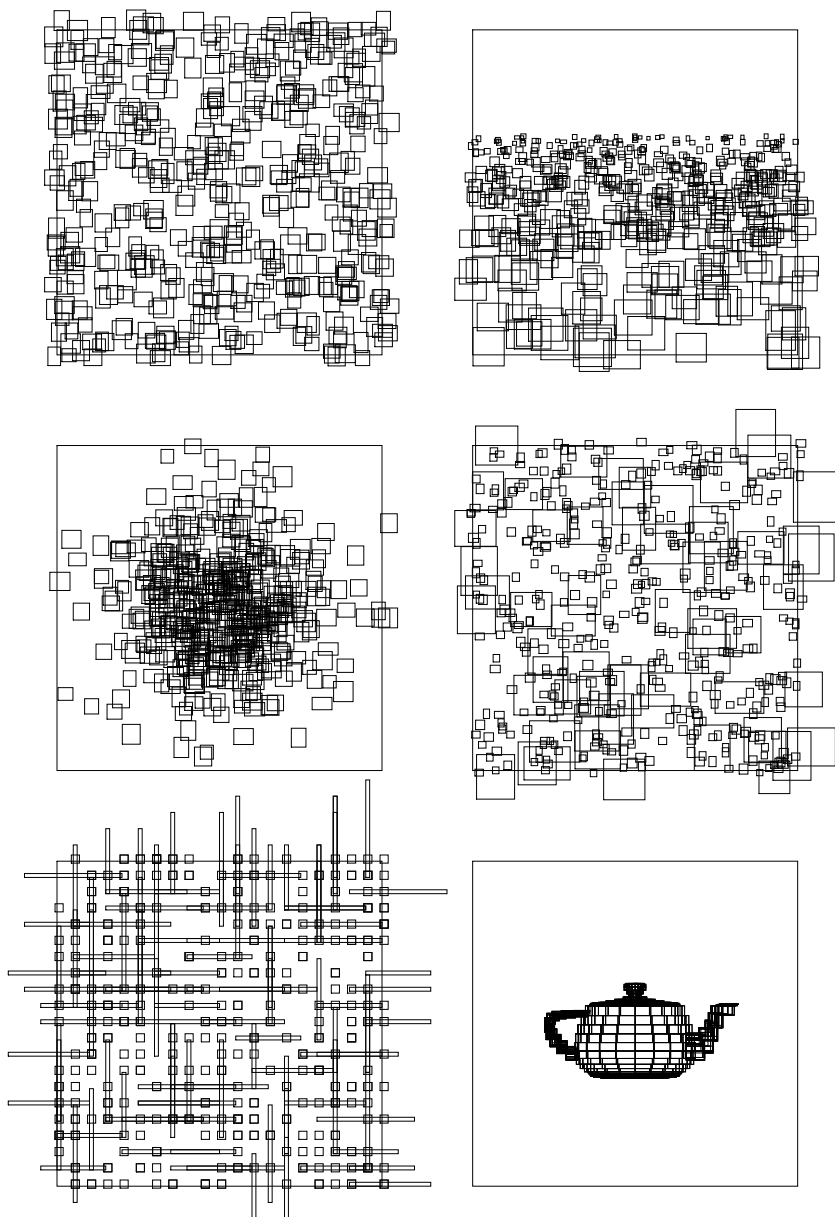


Figure 5.12: Scenes used for testing the behaviour of the data storage structure. They are (top row from left to right): Random and Landscape, on the middle row: Cluster and Two-size, and on the bottom row: Vlsi and Teapot. Note that to avoid cluttering, both the number and size of the extents actually used differ from what is shown here.

October 26, 1995

the domains decreases in such a way that the average coverage remains constant (that is, for the part below the horizon). The total number of domains and the coverage, averaged over the entire image, is the same as those of the random distribution.

Cluster. The cluster distribution is a gaussian-like distribution in which the number of domains near the center of the image increases. The size of the domains, the total number of domains and the coverage, averaged over the entire image, is the same as those of the random distribution.

Two-size. The two-size distribution is a homogeneous distribution of domains. The domains come in two size classes, that is, domains of 0.5 times and 2.5 times the size of the domains of the random distribution. The total number of domains and the coverage, averaged over the entire image, is the same as those of the random distribution.

Vlsi. The Vlsi distribution is a distribution of domains that are centered on a grid structure. The part of the grid that overlays the image is 100×100 cells. The domains are either small squares of 0.005 times the window size, or they are horizontal or vertical strips of half the width of the small squares and of length 0.04-0.06 times the window size. The total number of domains and the coverage, averaged over the entire image, is the same as those of the random distribution.

Teapot. The teapot distribution is a projection of the domains that result from triangulation of the utah-teapot model. The total number of domains is about 40K. The average coverage of the part of the image that contains the teapot is about the same as the coverage of the random distribution.

As can be seen in Figure 5.12, the distribution of domains across the image of these six test scenes differs, and for some of these scenes also the size of the domain extents varies.

Based on these test scenes we will analyse how domains are distributed among the slices and cells for different slice and grid "resolutions". Then we will examine how well the data structures succeed in reducing the number of domains to be checked for overlap with a particular query extent for these different resolutions.

Distribution among Slices

In each of the Figures 5.13-5.15 we see how the domains are distributed among the slices for the six scenes. The different figures show the distribution for space subdivisions in 10, 100 and 1000 slices. We observe that for all these resolutions the scenes Random and Two-size have a homogeneous distribution of their domains among the slices: the maximum of the number of domains per slice is inversely proportional to the number of slices. Also the scene Vlsi has such a homogeneous distribution for subdivision in 10 and 100 slices (Figures 5.13 and 5.14). However, the distribution of domains of the same scene for 1000 slices (Figure 5.15) clearly

demonstrates the effect of grid alignment of the domains. Due to this alignment, a further subdivision of the visible space in more than 100 slices does not result in an equivalent reduction of the maximum of the number of domains per slice.

The scenes Landscape, Cluster and Teapot exhibit an inhomogeneous distribution of domains. Still we find that the maximum of the number of domains per slice for the scenes Landscape and Cluster is practically inversely proportional to the number of slices. The scene Teapot does not behave that nicely. This scene is characterised by 'hot-spots', i.e., locations which contain a large number of very small domains. As a result, subdivision of the slices does not necessarily subdivide these local clusters proportionally.

Distribution among Cells

In Figure 5.16 we see how, for each of the six scenes, the domains are distributed among the cells of a grid of 40×40 cells. Figures 5.17 and 5.18 show the maximum number of domains per cell of each horizontal row of cells for space subdivision in 100×100 and 1000×1000 cells.

We observe also that the scenes Random, Two-size and Vlsi show a homogeneous distribution for a subdivision in 100×100 cells (Figure 5.17): the maximum of the number of domains per cell is inversely proportional to the number of cells. The effect of grid alignment on the distribution of the domains of the Vlsi scene shows up in Figure 5.18. The scenes Landscape, Cluster and Teapot are inhomogeneously distributed, but again for the scenes Landscape and Cluster the maximum of the number of domains per slice is practically inversely proportional to the number of cells.

We note that in terms of number of lists in the structure, the Grid-List structure does not reduce the maximum length of the linear lists substantially better than the Slice-List structure, except when the number of slices exceed the scene resolution such as for grid-aligned scenes. Hot-spots of the scene Teapot still spring out. In going from 1K slices to 1M cells we reduced the maximum of the number of domains in the list with just a factor of 15 (compare the height of the peaks in Teapot of Figures 5.15 and 5.18).

A possible advantage of grid-cell subdivision has to be demonstrated by analysing the reduction in search space.

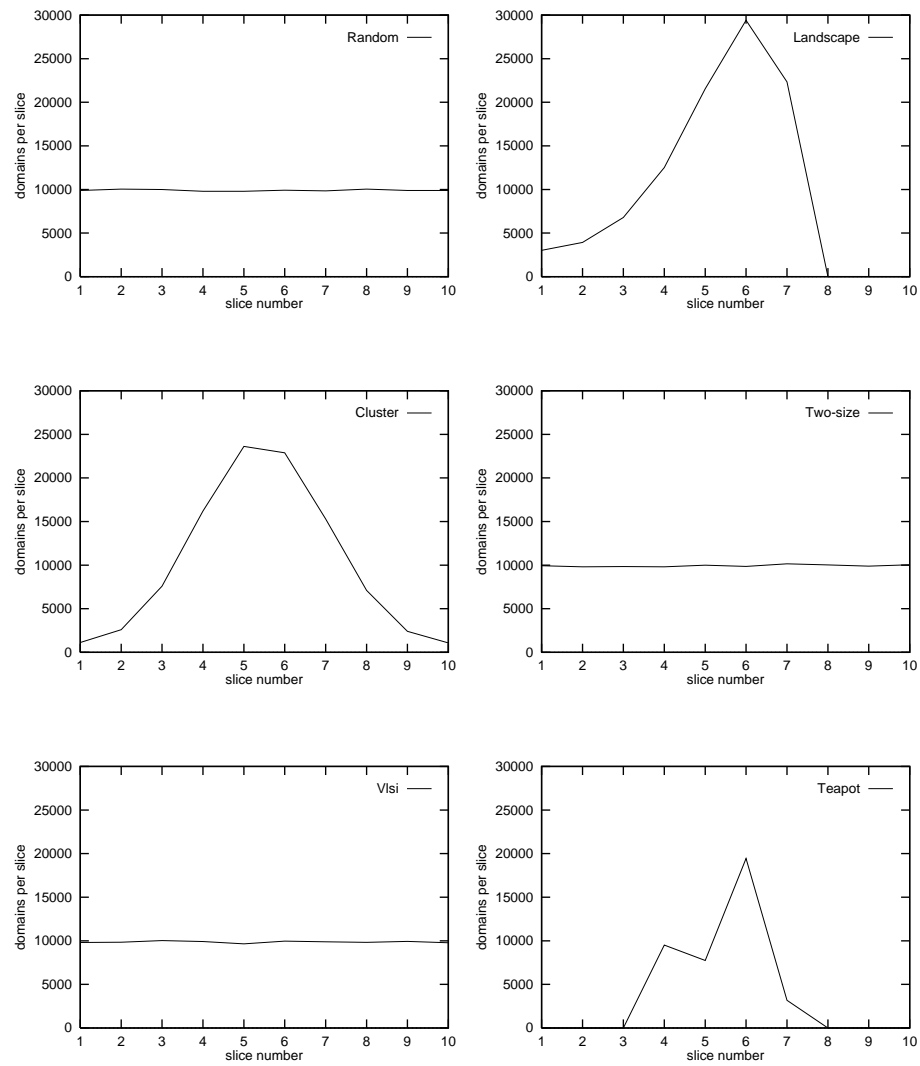


Figure 5.13: The distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over 10 horizontal slices.

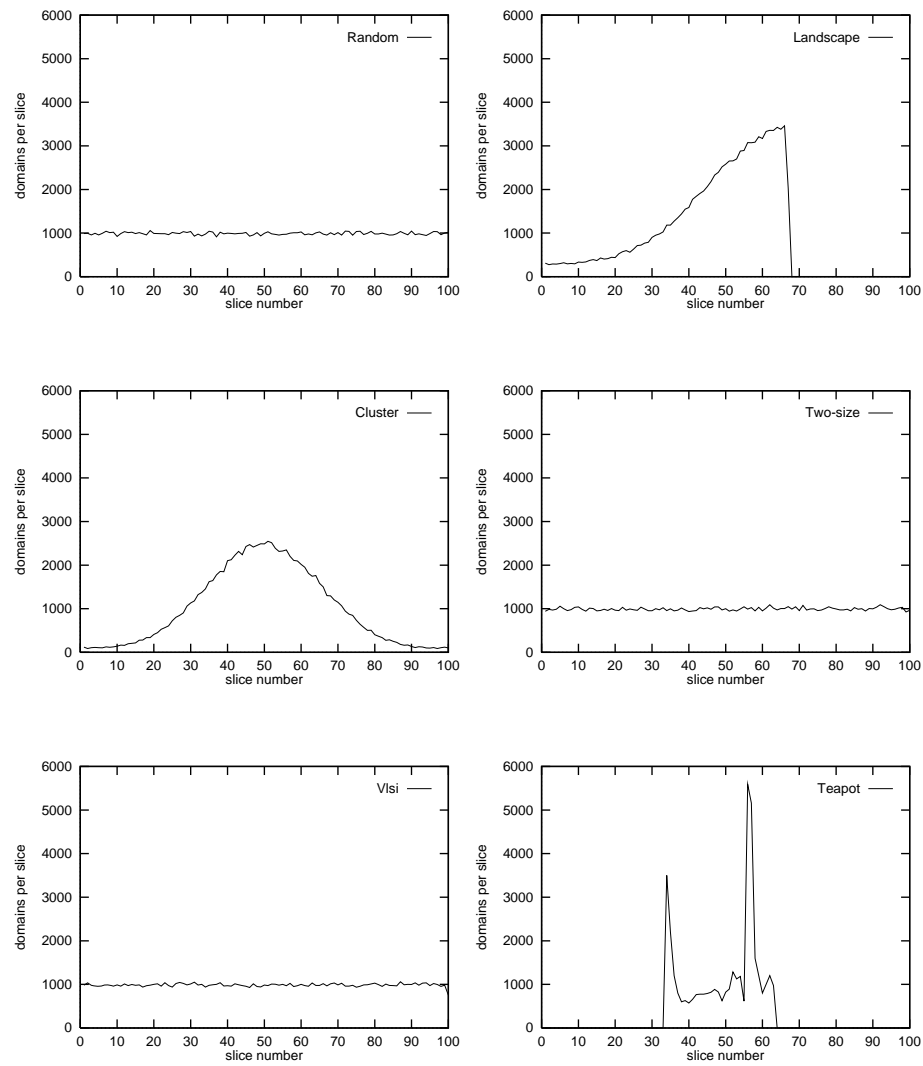


Figure 5.14: The distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over 100 horizontal slices.

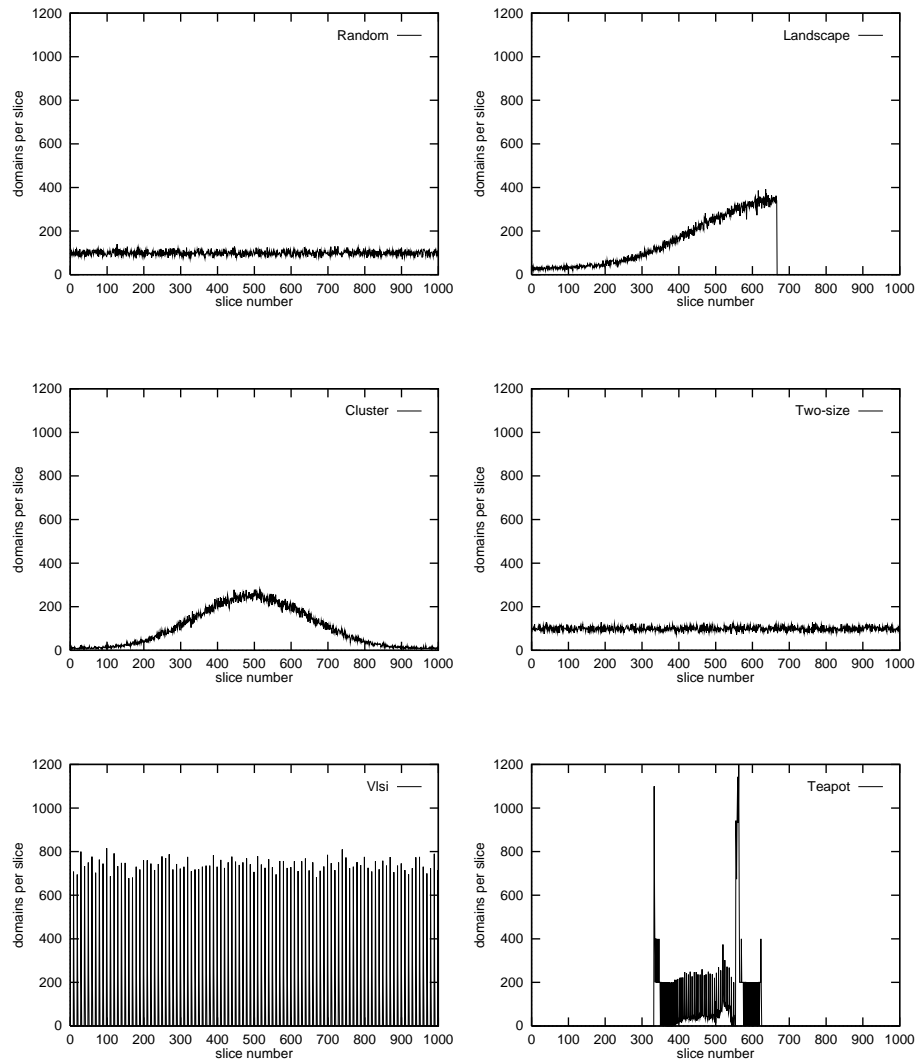


Figure 5.15: The distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over 1000 horizontal slices.

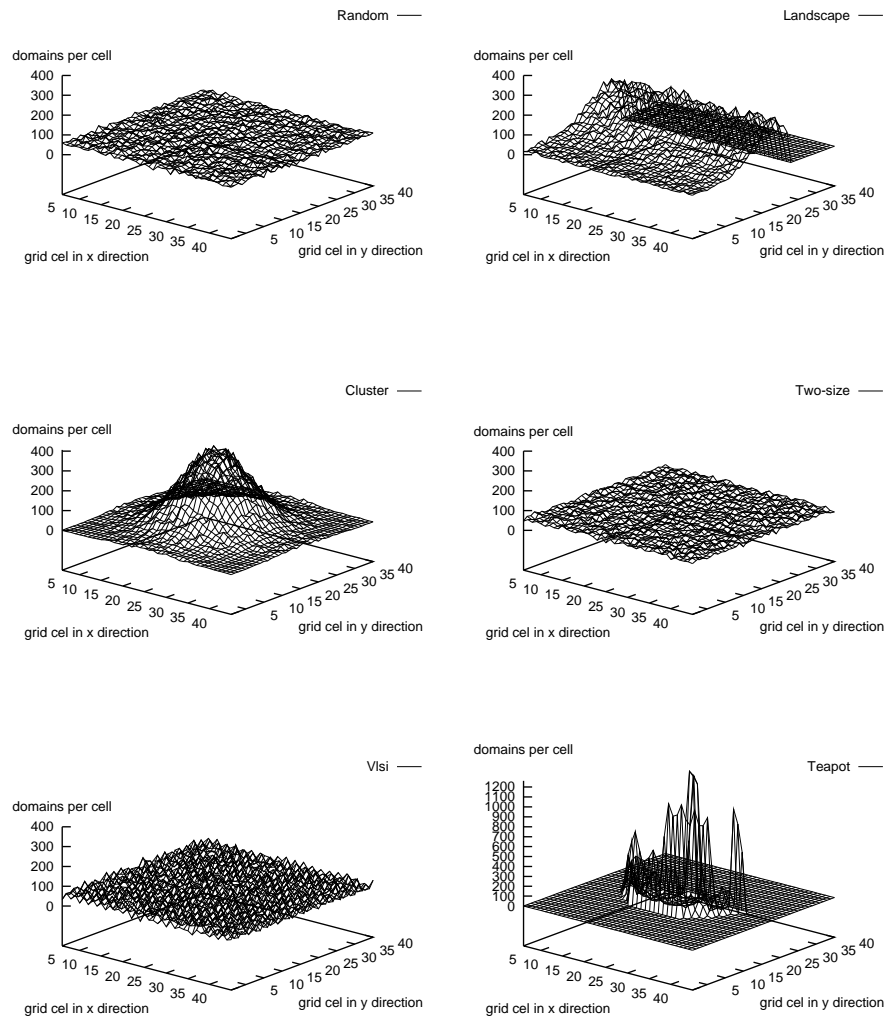


Figure 5.16: The distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over a grid of 40×40 cells.

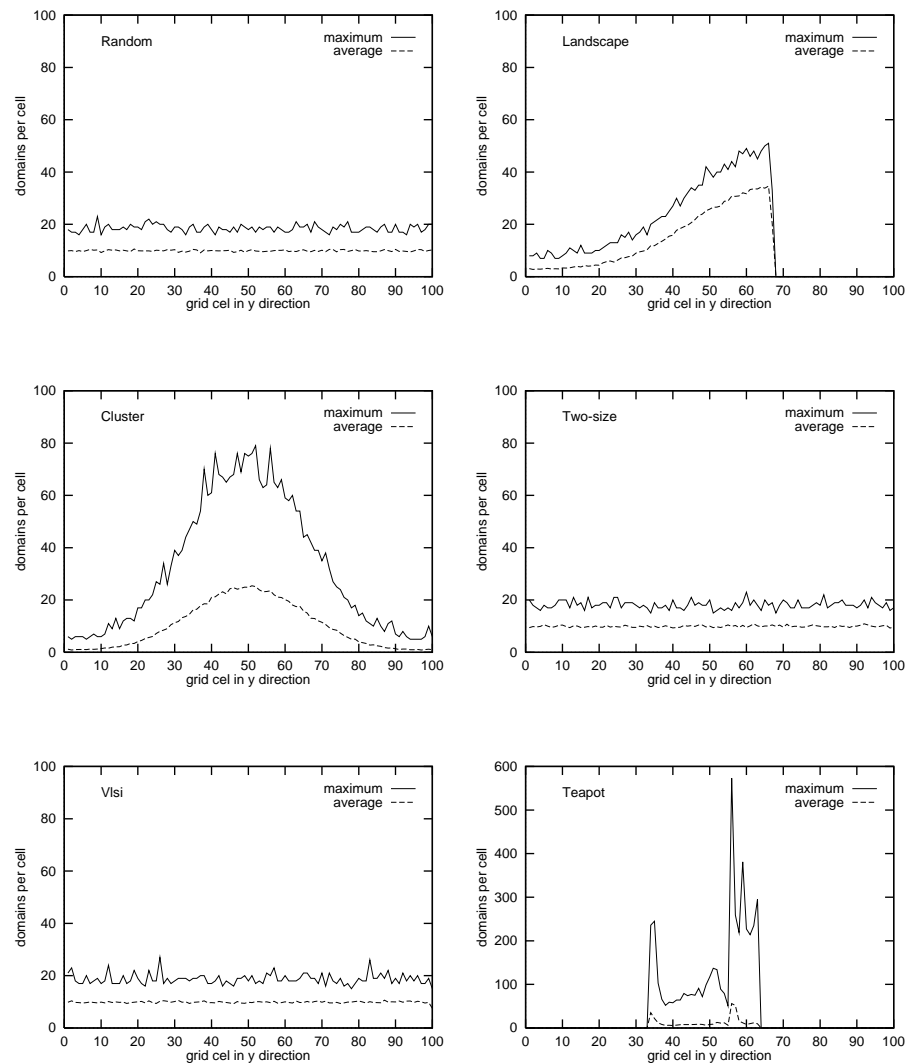


Figure 5.17: The maximum number and the average number of domains per cell for each horizontal row of cells for the distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over a grid of 100×100 cells.

Note that $100 \times$ the average number of domains per cell-row is equivalent with the number of domains per slice for the distribution shown in Figure 5.14.

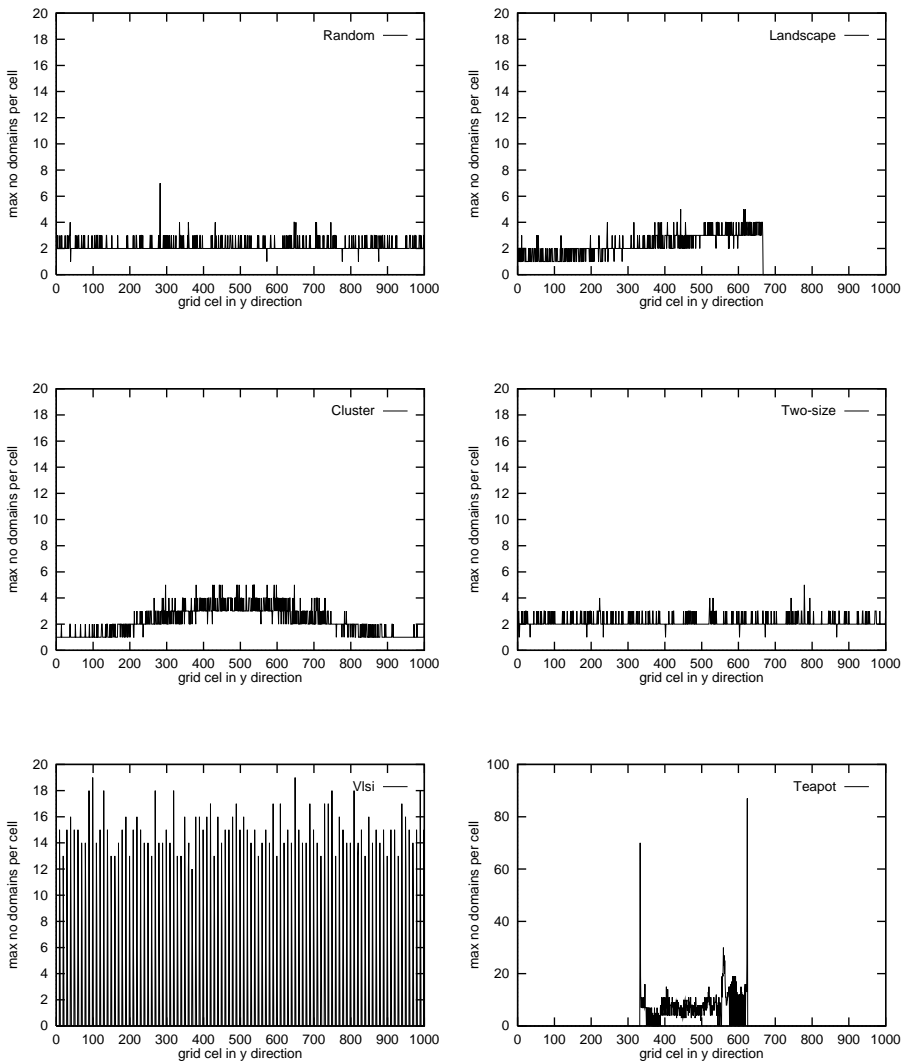


Figure 5.18: The maximum number of domains per cell for each horizontal row of cells for the distribution of the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot over a grid of 1000×1000 cells.

Effective Reduction of the Search-Space

How well do the Slice-List and the Grid-List structure reduce the search space? Expression 5.2 and 5.3 indicate what can be expected for homogeneously distributed domains. In the next we examine the number of domains that are addressed when looking for domains that overlap a query domain for each of the six test scenes ¹⁰⁾. The query domain used is 0.01 times the window size. To be able to analyse the behaviour of the algorithm at every part of the image, we scanned the entire image using 1M different positions.

In Figures 5.19-5.21 we present the results obtained for the Slice-List structure for different resolutions. The graphs show both the average number of domains addressed and the average number of domains that actually overlap as a function of y . In Figure 5.19 we find the number of domains addressed for a space subdivision in 10 slices. The results found are conform expression 5.2. We clearly see the dramatic effect of the limited resolution in what we called the round-off error that is described by the second term of expression 5.2 (see § 5.3.1). This effect drops off when going to higher resolutions (Figures 5.20 and 5.21). We find that when going to resolutions of more than 100 slices we face a limit in the reduction of the number of domains addressed.

By using the Slice-List structure the number of domains addressed have been reduced substantially: for a sufficiently fine grained resolution, we address 2% of the 100K domains of the Random scene on average. This is equal to the average number of domains of which the extent overlaps in y -direction only, so this result may seem reasonable. Yet, the extent of a mere 2% of the domains addressed actually turn out to overlap the extent of the query domain. For some locations of some of the other scenes the result is even worse.

In Figures 5.22-5.24 we present the results obtained for the Grid-List structure for different resolutions. These graphs show the average number of domains addressed as a function of x and y . The results found are conform expression 5.3. Also here we see the effect of the limited resolution that drops off when going to higher resolutions (Figures 5.23 and 5.24).

In Figure 5.25 we show the average number of domains of which the extent actually overlaps the extent of the query domain as a function of x and y . Of the three different resolutions shown in Figures 5.22-5.24, the graphs suggest that a grid of 100×100 cells seems to be the most optimal resolution. Note that in that case the size of the grid-cells is in the same order as the average size of the domains. Also note that the reduction in search space for the sets Vlsi and Two-size turns out to be less effective. The somewhat larger domains

¹⁰⁾ We count the number of memory accesses, so addressing empty slices and cells is counted as well — this will show up in the graphs that relate to the scenes Landscape, Cluster and Teapot.

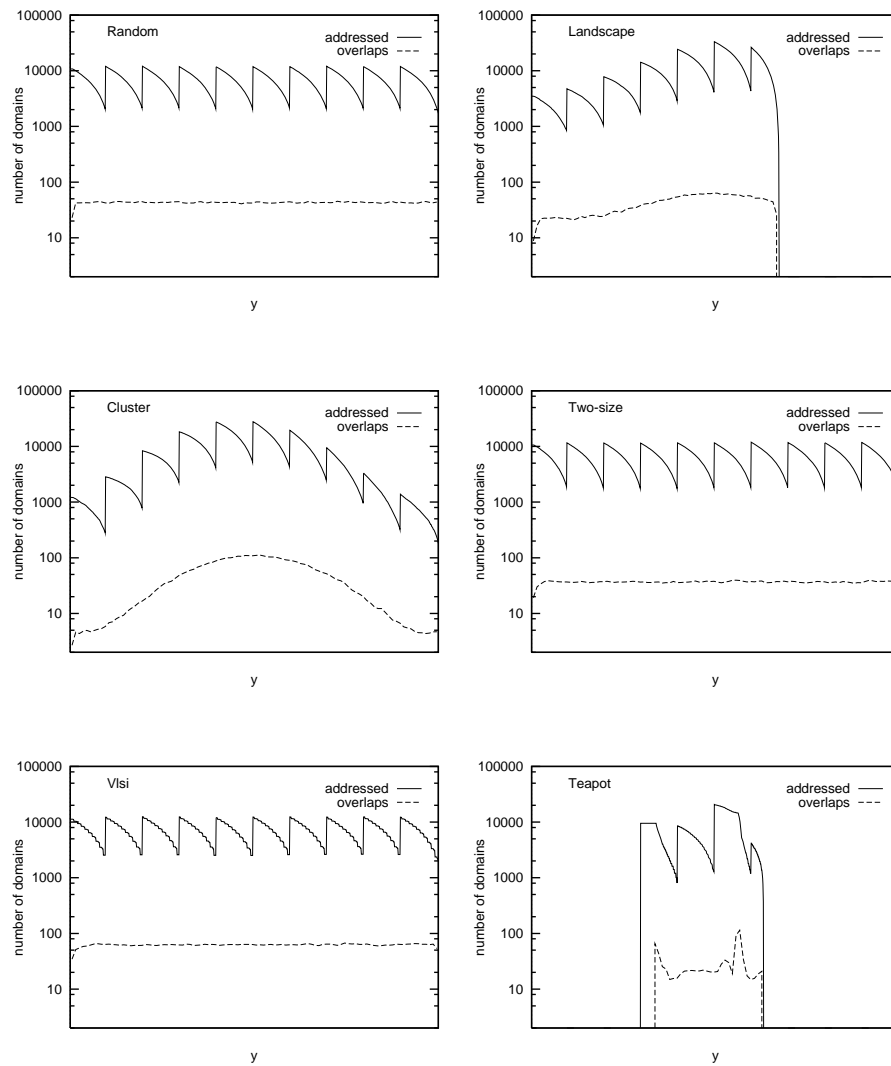


Figure 5.19: Search space reduction for 10 slices. These graphs show the average number of domains addressed and the average number of domains of which the extent overlap the extent of a query domain as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

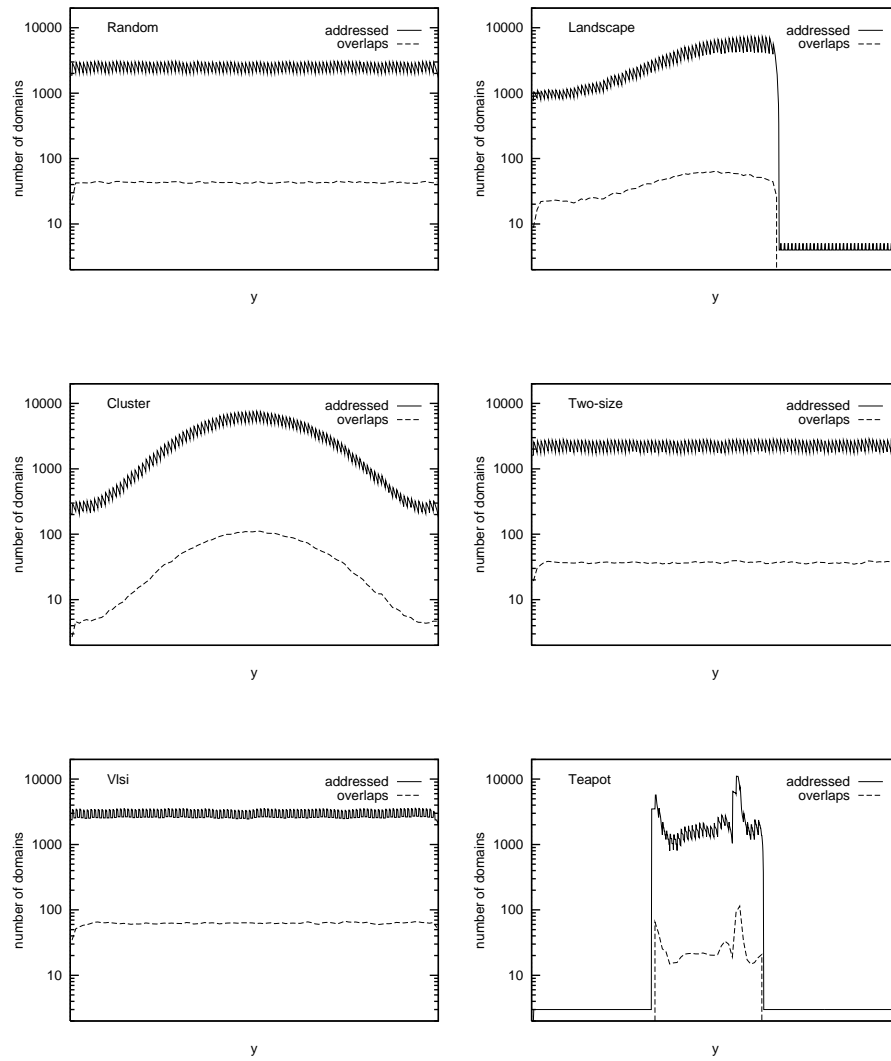


Figure 5.20: Search space reduction for 100 slices. These graphs show the average number of domains addressed and the average number of domains of which the extent overlap the extent of a query domain as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

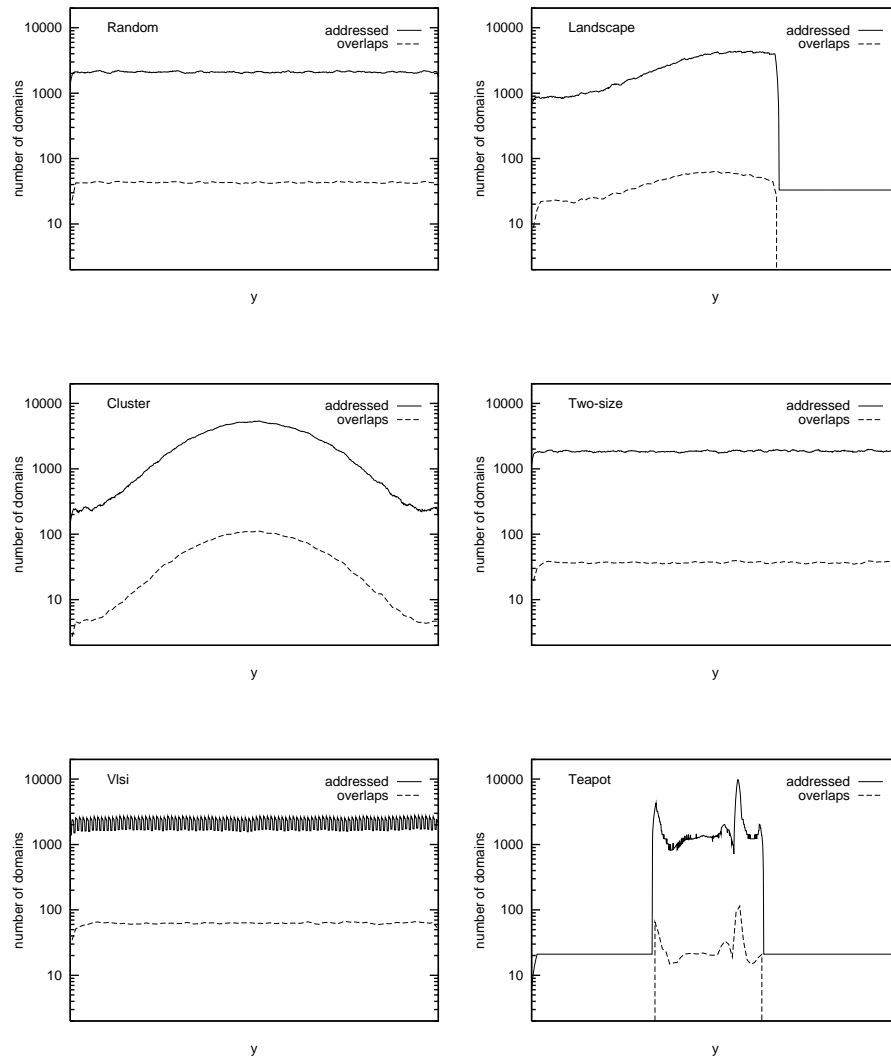


Figure 5.21: Search space reduction for 1000 slices. These graphs show the average number of domains addressed and the average number of domains of which the extent overlap the extent of a query domain as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

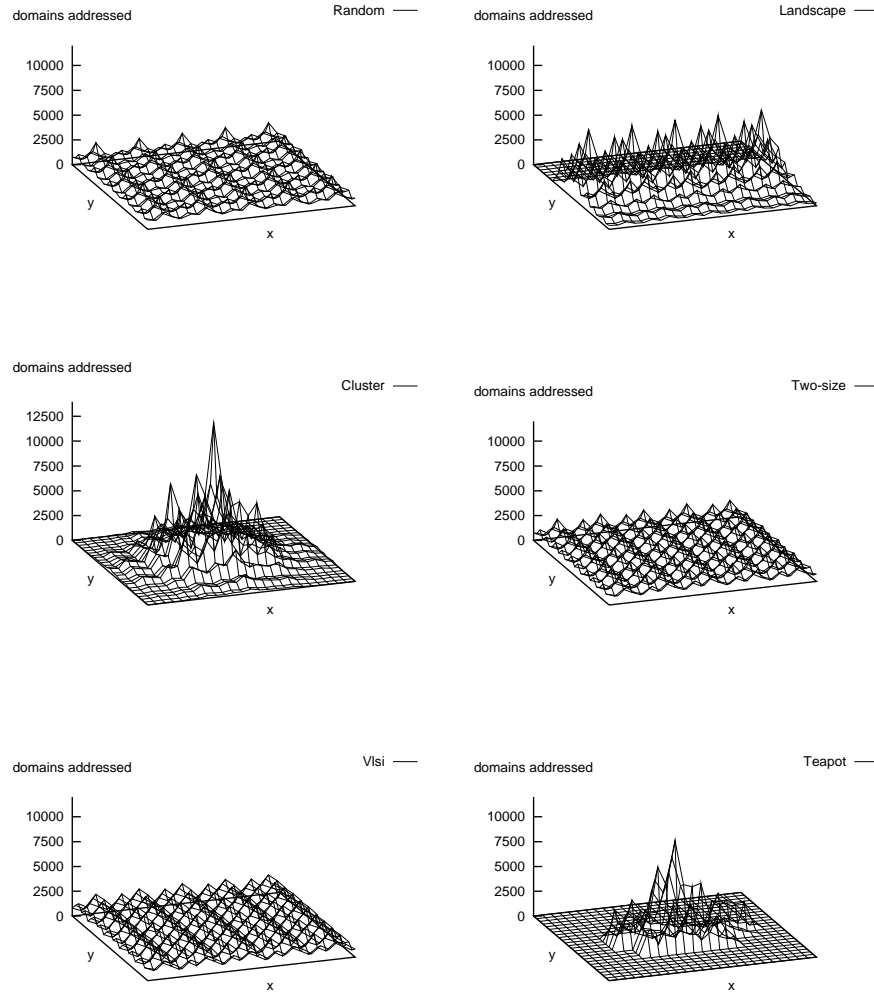


Figure 5.22: Search space reduction for a grid of 10×10 cells. These graphs show the average number of domains addressed as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

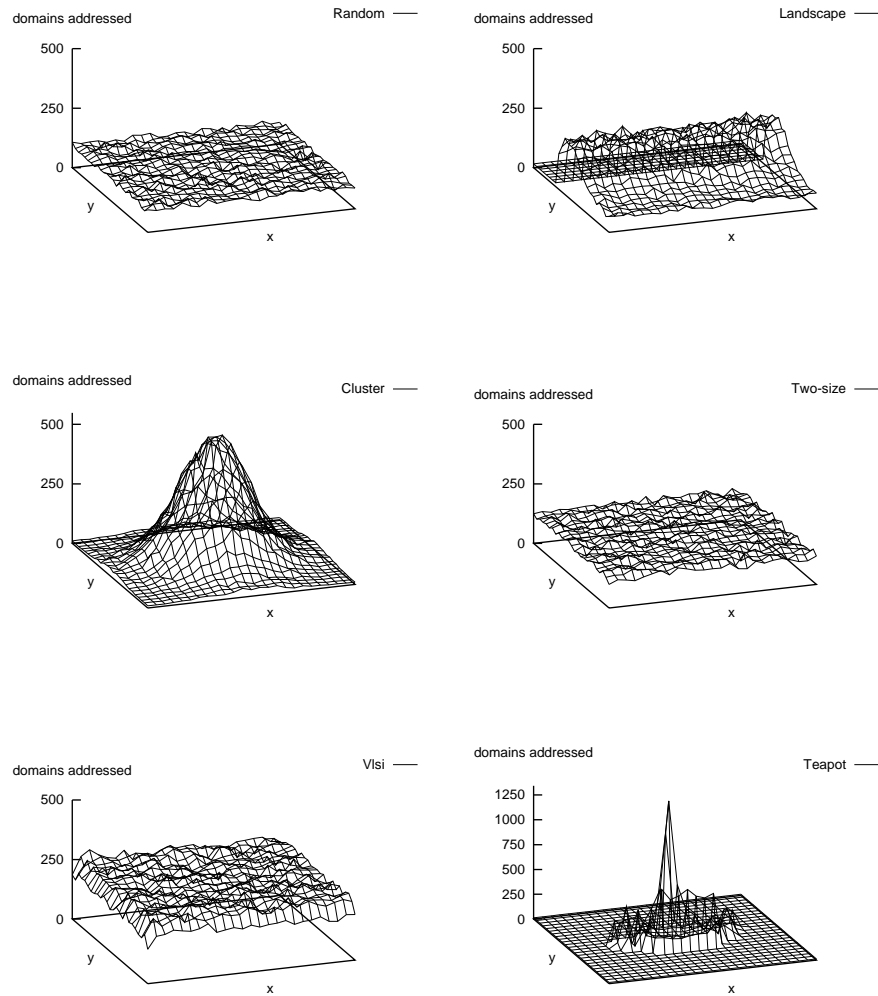


Figure 5.23: Search space reduction for a grid of 100×100 cells. These graphs show the average number of domains addressed as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

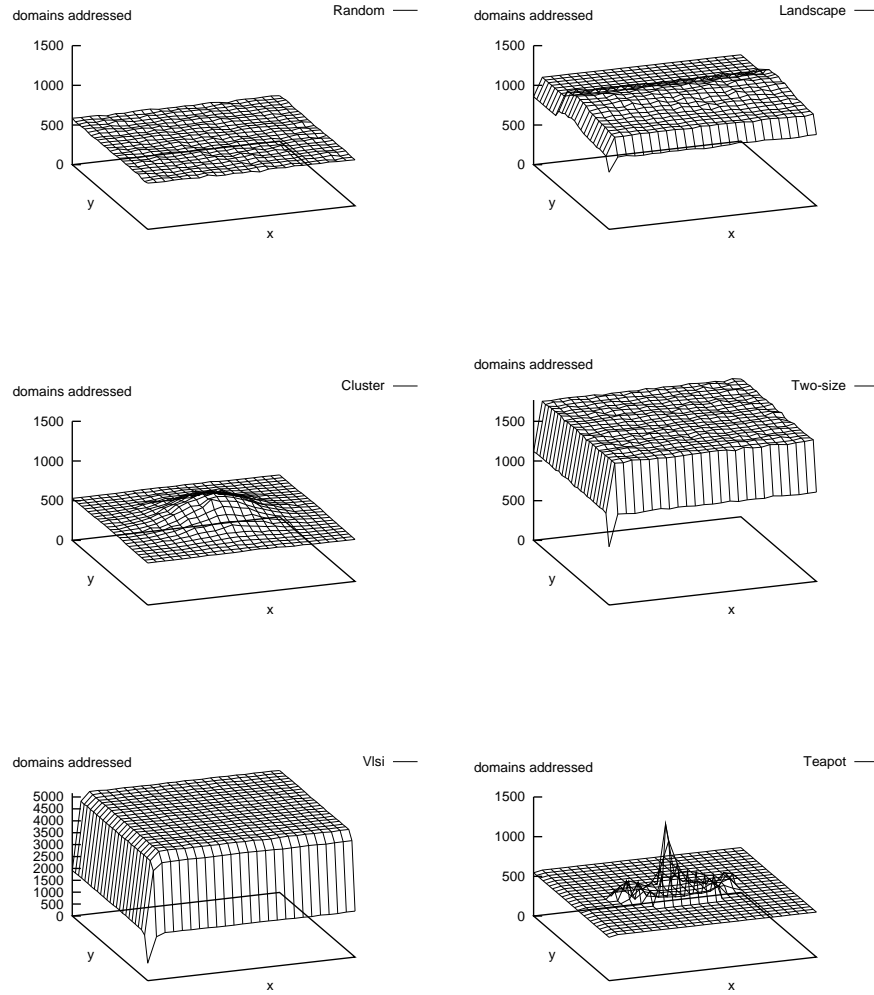


Figure 5.24: Search space reduction for a grid of 1000×1000 cells. These graphs show the average number of domains addressed as a function of the position of the query domain. The full sets are the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

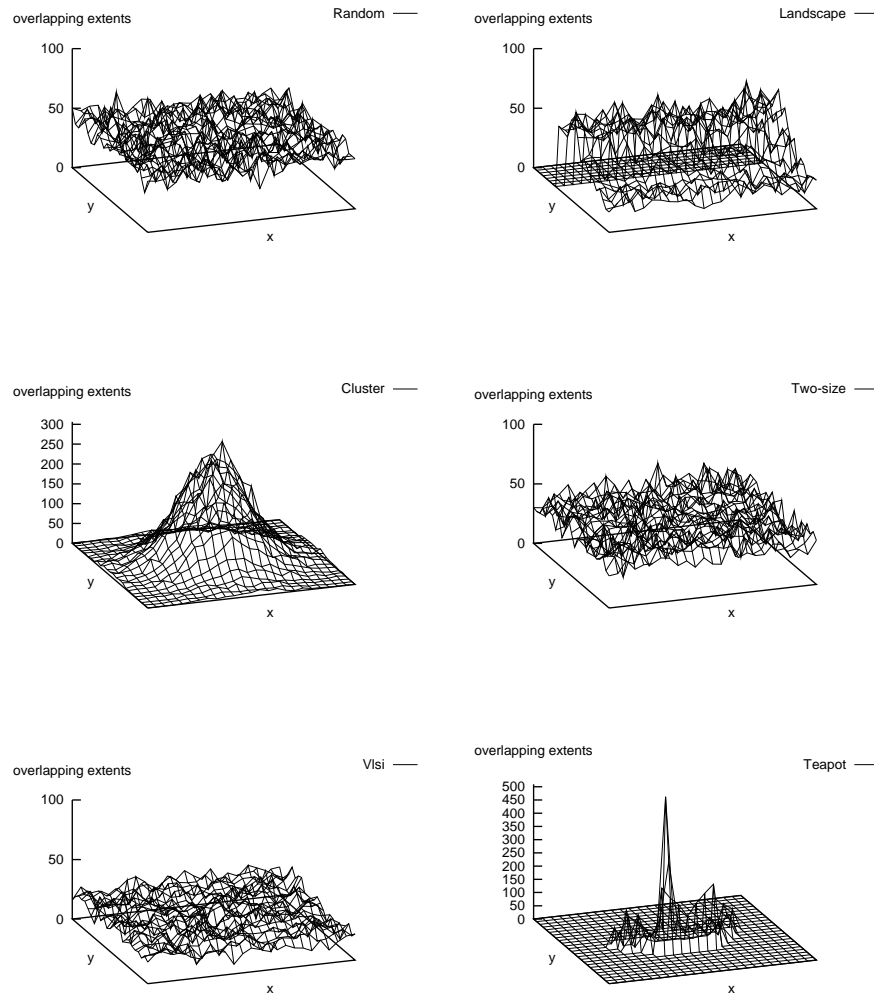


Figure 5.25: These graphs show the domains of which the extent overlap the extent of a query domain as a function of the position of the query domain for the 100K domains of the five test scenes Random, Landscape, Cluster, Two-size and Vlsi and the 40K domains of the test scene Teapot.

October 26, 1995

we find in these sets make that more grid-cells are addressed. The cost of the addressing of grid-cells is most apparent for a grid of 1000×1000 cells (Figure 5.24).

Table 5.2 and Figure 5.26 present another representation of the result. They show the ratio of number of domains addressed per domain of which the extent overlaps the extent of a query domain for resolutions ranging from 10×10 up to 1000×1000 cells. We find that for a grid of 100×100 cells, we address on average less than 0.1% of the 100K domains of the Random scene. This is less than $2\times$ the number of domains of which the extent overlaps the extent of the query domain. It will be clear that this result is much more satisfying.

	Random	Landscape	Cluster	Two-size	Vlsi	Teapot
subdivision	query extent size 10					
10	16.6	25.4	18.7	22.7	15.2	43.3
15	10.2	13.6	11.5	12.0	8.1	16.6
30	4.3	5.4	4.9	5.0	4.1	7.0
60	2.4	3.2	3.0	2.8	2.5	3.9
100	1.9	2.4	3.1	2.5	3.1	2.9
150	1.9	3.1	4.1	3.5	4.1	3.7
300	2.3	4.7	8.8	6.3	9.4	4.2
600	5.8	13.3	31.7	18.5	30.1	8.5
1000	13.7	33.2	83.2	46.9	78.5	18.2
subdivision	query extent size 100					
10	2.7	11.3	3.5	3.2	6.1	7.3
15	2.3	10.1	2.6	2.8	5.0	7.0
30	1.9	8.7	2.1	2.4	5.1	2.7
60	1.8	8.5	2.1	2.4	3.3	2.2
100	1.8	9.4	2.4	2.6	4.0	2.0
150	1.9	11.4	3.3	3.1	5.2	2.2
300	2.8	22.4	8.2	6.2	11.9	4.2
600	6.6	66.7	27.4	18.4	38.4	12.5
1000	15.7	169.1	72.9	47.3	100.8	32.1

Table 5.2: Efficiency of the data structure (see also Figure 5.25).

5.3.4. Conclusions

From the above we conclude that the Grid-List structure leads to a near optimal reduction of the search space. The structure makes that for the optimum resolution for the given test scenes we address in the order of 2-10 times the number of domains of which the extent overlaps the extent of a query domain. We claim that using this simple structure and combining that with a final check for extent overlap is in practice much more efficient than a more complicated structure that reports exactly those domains of which the extent actually overlap the extent of a query domain.

By looking at the results at more detail we observe that the hot-spots of the scene Teapot do not noticeably affect the efficiency. However, we do find a dependency of the result on the maximum size of the domains — as is most apparent for the scenes Landscape and Vlsi. If the set would contain just one domain that

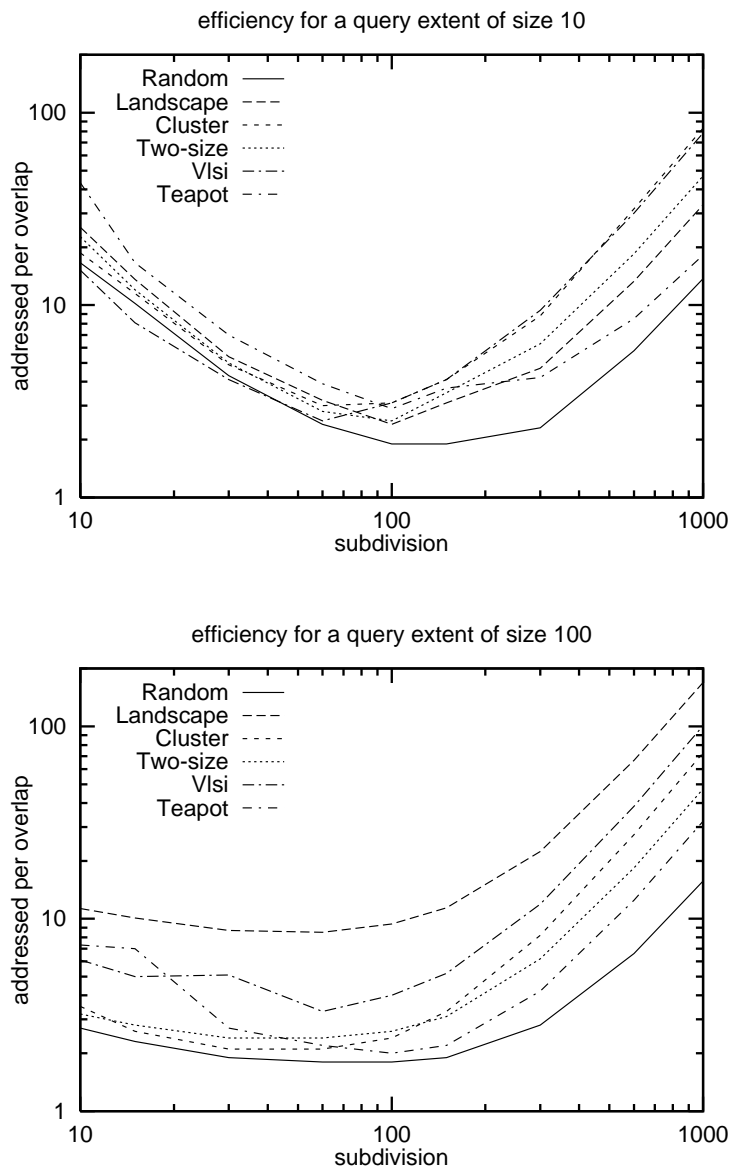


Figure 5.26: Efficiency of the data structure (see also Table 5.3). The topmost graph shows the number of domains that are addressed per overlapping extend reported for a query extent of size 10 for grids of 10×10 up to 1000×1000 . The graph at the bottom shows the result for a query extent of size 100.

spans the entire image we may have to address all the cells as indicated in Figure 5.10, leading to a much reduced efficiency.

Dependency on the maximum size of the domains of a set can be avoided by introducing a *multi-scale Grid-List* structure. Such a multi-scale structure would combine for instance a 2×2 a 20×20 and a 200×200 grid. A domain is then stored at the level in which the size of the grid-cells is in the same order as the size of the domain. A multi-scale structure would even simplify the search algorithm, because then for each of the levels the maximum size of domains is fixed. In this dissertation we will not go into further details of such a basically simple extension.

5.4. Exact Incremental Hidden Surface Removal

The exact incremental hidden surface removal (HSR) algorithm discussed in this section is designed especially to be embedded in the architecture presented in Chapter 4. The algorithm is optimised in relation with required functionality and context (in particular with the scan-conversion process) rather than optimised for asymptotic computational complexity.

Support for graphical interactions such as direct manipulation as discussed in previous chapters implies that the HSR algorithm should be able to handle deletion or repositioning of objects efficiently, and it should be able to deal with less trivial situations like transparent and interpenetrating objects. Temporary changes, by their nature, require a different approach than permanent changes. All of this can be supported by an adequate object representation and the binary operations that form the gist of the algorithm.

Both input and output primitives of the HSR algorithm are types — in the sense of ‘abstract data types’ — of a specially designed representation: being the flat polygon-like primitives we call patterns as described in § 5.2. The perspective transformation has already been applied upon the input primitives, so that the projection onto the 2D image-plane that follows the HSR reduces to a parallel projection — which amounts to simply leaving out the z-coordinates. Therefore the 3D geometry of a pattern is represented by a so-called domain, being a projection of the surface area onto the x-y plane and the coefficients of the plane equation.

Exact — or object space — hidden surface removal implies that for each input primitive the algorithm has to produce the visible part. In the perspective transformed space in which patterns are defined, this can be done by binary operations on the geometry that is projected onto the x-y plane. The relative depth ordering of the primitives that are projected determine what operations are to be performed and on which primitives. In other words: hidden surface removal can be done by an adequate set of binary operations on 2D domains, taking into account the depth values as represented by the coefficients of the plane equation of the domains involved.

5.4.1. Binary Operations

The binary operations on domains needed for the hidden surface removal algorithm have two input domains. Since we allow for interpenetrating domains, a combination of two domains may lead to four differently covered areas, as is illustrated in Figure 5.27¹¹⁾.

Given two input domains A and B we define the following binary operations (see also Figure 5.28):

DEFINITION: $A - B$ is that part of A that is not obscured by B.

DEFINITION: A_{-B} is that part of A that is not covered by B.

DEFINITION: $A \cap B$ is that part of the area that is covered by both A and B where A is not obscured by B.

DEFINITION: $A \cup B$ is the area that is covered by A or B.

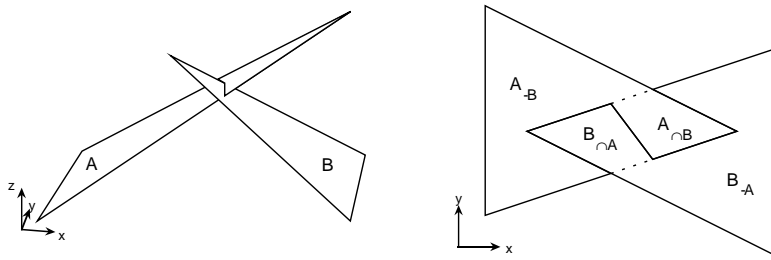


Figure 5.27: Example of a combination of two interpenetrating triangles (left) that leads to four differently covered areas.

Given these definitions we come to the following:

COROLLARY: $A - B = A \cap B \cup A_{-B}$.

COROLLARY: A_{-B} , $A \cap B$, B_{-A} and $B \cap A$ are disjunct and completely cover $A \cup B$.

The option to produce either $A - B$ or the two separate areas $A \cap B$ and A_{-B} is needed in particular to be able to handle transparent objects. Also note that we did not define $A \cap B$, because this operation would not tell us which of the two input domains is "on top".

¹¹⁾ Note that a domain can have holes and does not have to be one connected area. Of interest here is by which of its input domains the area is covered and, if covered by both, in which order. Therefore we find four different combinations at most.

We implemented a function $\text{separate}(A, B)$ that, depending on the topology of the two input domains and the transparency as is determined by the colour functions of the input domains, produces one of the following combinations of disjunct output domains:

- A and B ,
- $A - B$ and $B - A$,
- A_{-B} , $B - A$ and $A \cap B$,
- $A - B$, B_{-A} and $B \cap A$ or
- A_{-B} , B_{-A} , $A \cap B$ and $B \cap A$.

Note that in case $A \cap B = \emptyset$ the function outputs the two input domains. In case A and B do not interpenetrate then either $A - B = A$ or $B - A = B$ and also either $A \cap B = \emptyset$ or $B \cap A = \emptyset$. In any case, if both $A \neq \emptyset$ and $B \neq \emptyset$ the function $\text{separate}(A, B)$ produces at least two disjunct nonempty output domains.

Domains $A - B$ and A_{-B} will have the same colour function that A has, $B - A$ and B_{-A} will have the same colour function that B has, whereas $A \cap B$ and $B \cap A$ will have a mixture of the colour functions of A and B (see Figure 5.28).

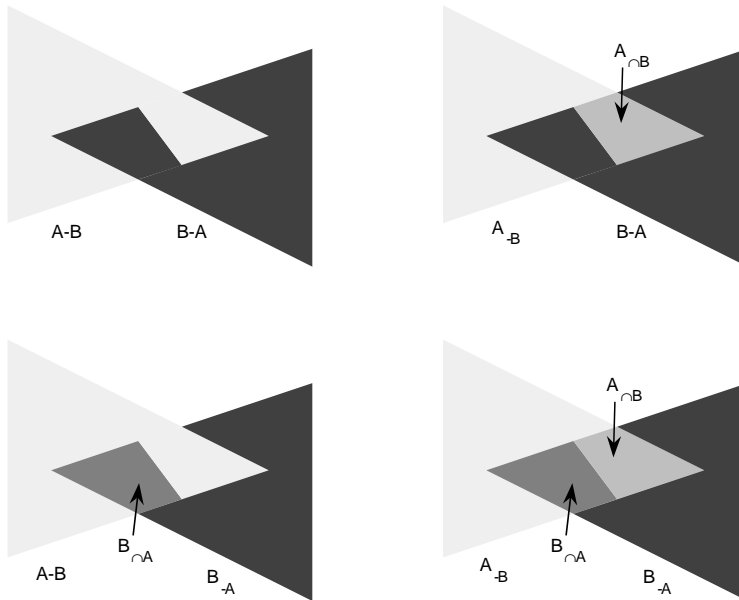


Figure 5.28: Hidden surface removal of two interpenetrating triangles (as shown in Figure 5.27) may result in two, three or four differently shaded areas, depending on opacity. In the upper left both triangles are opaque, in the upper right A is transparent and B is opaque, in the lower left A is opaque and B is transparent and finally in the lower right both triangles are transparent.

5.4.2. Basics of the Algorithm

The visibility calculation is handled in the following way. There exists a data structure containing all input domains, being the Medium Level Representation (MLR) defined in § 4.4. There also exists a so-called visibility map: a data structure that contains the complete set of disjunct visible domains, being the Low Level Representation (LLR) also defined in § 4.4. The domains in the LLR are stored in a Grid-List structure as described in the previous section.

Given this, the basics of the hidden surface removal algorithm is as follows:

```

for all I in the MLR do
  begin
    for each extent overlapping domain N in LLR do
      begin
        if N or I transparent and obscuring do
          begin
            insert  $N \cap I$  and  $I \cap N$  in LLR
             $N = N - I$ 
             $I = I - N$ 
          end
        else do
          begin
             $N = N - I$ 
             $I = I - N$ 
          end
        end
      end
    insert I in LLR
  end
end

```

From this algorithm we see that the LLR is built-up incrementally: the domains of the MLR are added one by one. Domains of the LLR that are (partly) obscured by a MLR domain and transparent domains that obscure that MLR domain are updated accordingly — that is, removed, partly removed or split. For each domain in the LLR the part of the MLR domain that is obscured by the LLR domain is removed. In this way we end up with just the visible area of the MLR domain. Note that this visible area makes up one domain that does not have to be one connected region. Finally, this visible area is added to the LLR. This process is repeated for all domains in the MLR. In this way we ensure that all domains of the LLR, or visibility map, are disjunct at all times.

Note that the inner `for`-loop of the algorithm is executed only for those domains of which extent overlaps the extent of I , by making use of the features of the Grid-List structure. Domains in the LLR of which parts are removed or that are split, are repositioned in the Grid-List structure, according to their updated extent.

The output domains $N - I$ and $I - N$ or N_{-I} , I_{-N} , $N \cap I$ and $I \cap N$ are calculated by the function `separate(I, N)`. Clearly only nonempty domains are inserted in the

visibility map.

We observe that the algorithm is quasi output sensitive. That is, the time complexity of the algorithm is related to the number of domains that will be visible — i.e., that are output —, yet the algorithm may spend time on domains that are invisible, depending on the ordering of input domains.

5.4.3. Results

Rather than looking at the time complexity of the algorithm, we measured the CPU time needed for the algorithm as a function of number of triangles when running on a SiliconGraphics Indigo equipped with a 50MHz processor. For these measurements we used three different test scenes. Two of the scenes are simply a stack of triangles, one of decreasing sizes (the scene tri-steps on the left side of Figure 5.29) and one of increasing sizes (the scene tri-spets on the right side). As a result, all triangles of the scene tri-steps are visible, whereas for the scene tri-spets only one triangle is visible, whereas for the scene tri-spets only one triangle is visible. These tests scenes (and their names) are similar to the scenes used in [Snippe92]. The third scene used is the Utah teapot looked upon from the side.

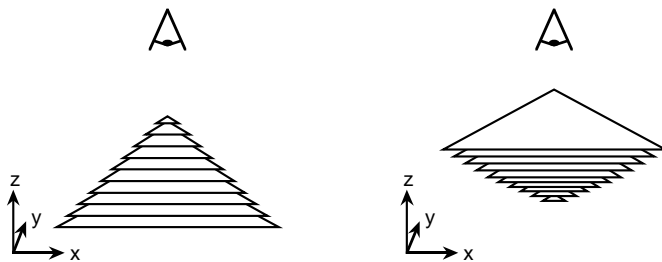


Figure 5.29: Test scenes used for timing of the hidden surface removal algorithm. On the left the scene “tri-steps” on the right the scene “tri-spets”.

In Figure 5.30 we present the results. The graph makes clear that the algorithm is indeed output sensitive. Presenting the triangles of the two scenes tri-steps and tri-spets in reversed order produced the exact same results. In the same graph we also show the results of the output sensitive algorithm presented in [Snippe92]. This algorithm takes a depth sorted list of polygons that are put in the leaves of a tree structure. The union of these objects are calculated bottom-up, next the visible parts are calculated top-down. The timing of this algorithm was done on similar equipment, yet the clock speed of the system was not specified.

We observe that our algorithm turns out to be exactly linear in case of the simple scene tri-spets and exactly quadratic in case of the worst case scene tri-steps. As could be expected, the time needed for the scene teapot falls in between the times needed for these two extremes. It is important to note that although there is a relatively high constant involved, the trend for the scene teapot is quite promising.

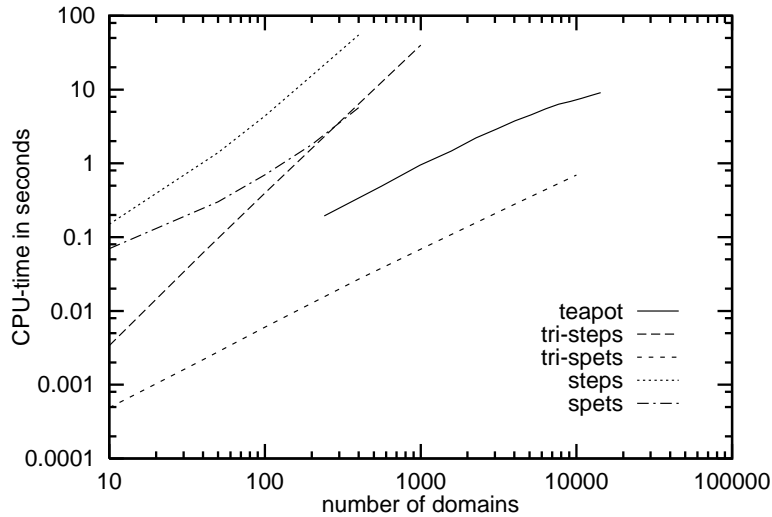


Figure 5.30: The time needed for hidden surface removal of the scene tri-steps, tri-spets and teapot. For comparison we also show the results of an output sensitive algorithm presented in [Snippe92] (steps and spets, which are test scenes similar to tri-steps and tri-spets).

When looking at the absolute values we conclude that we need to improve the implementation. We have not seriously analysed the implementation in order to optimise the code itself. We expect that a version based on the Sorted-Edge representation will do better because it does not require calculation of edge intersections at each scanline. Furthermore we now also have processors that run faster. Nevertheless, we clearly need techniques to reduce the complexity of the algorithm further.

5.4.4. Reducing Complexity

The complexity of the hidden surface removal algorithm can be reduced in several ways. Of course we can make use of techniques to reduce the number of input domains such as the obvious back-face culling and cone-of-vision techniques. In principle, also depth sorting domains prior to the HSR process can be of help. Clearly starting with large domains in front of the scene that obscure lots of small domains reduces the number of domains in intermediate visibility maps. Then the complexity of the algorithm will be reduced as well. However success cannot be assured. For some cases such as the test scenes tri-steps and tri-spets, depth sorting does not work out at all ¹²⁾. Because of this, pre-sorting should not be too costly.

Fortunately we do not need a full-fledged depth sorting algorithm that can handle cyclic overlaps, simply sorting on the average z-value for instance should do ¹³⁾.

A potentially better technique, that has to be looked into at more detail, is hierarchical hidden surface removal. Computer graphics applications in general display scenes of which the elements are ordered in some fashion. A scene may for instance comprise a number of high level objects such as a desk, a chair, a teapot. Each of these objects may be described by hundreds of polygons. These high level objects themselves are confined in space and so can have a 3D bounding volume associated with them, much like the 2D extent of the individual domains. Hidden surfaces can be removed on a per object bases, and the outline of these objects can be projected onto the front face of their bounding volume. This outline forms one domain. Then, hidden surface removal for the entire scene can be done based on these object projections as long as bounding volumes do not intersect. If bounding volumes of two objects do intersect, these objects have to be handled at the polygon level. The reduction on complexity that can be obtained in this way depends on the level of detail of the objects.

5.4.5. Support for Selection and Partial Updates

In conformity with Maxim 5, formulated in § 4.3, the hidden surface removal algorithm should be able to make changes incrementally. Only then the propagation of partial changes typical for interactions that occur in the MLR can be restricted, so that the effort needed to update the LLR can be minimised. The incremental changes of interest with respect to the hidden surface removal algorithm are changes of geometry and change of transparency. Both types of changes can be effectuated by removal of the visible parts of the domains of concern from the LLR, followed by repair of domains of which parts should become visible as a consequence of this removal (this is known as “damage repair”) and finally by adding updated domains. For damage repair we have to be able to find out in an efficient way which domains of the MLR had been obscured, or similarly, which domains of the MLR cover a certain area.

For selection of a display primitive by means of a pointing device — an essential basic interaction task mentioned in § 2.2 — we come to a similar requirement: it should be possible to find in an efficient way which domain in the MLR is visible at a certain position.

¹²⁾ For a scene with a large obstacle halfway and lots of small objects in front and in the back, it would be best to start with the large object, next the objects behind it and only then handle the objects in front.

¹³⁾ Depth sorting using for instance BSP trees [Fuchs80] — that are build-up by recursively splitting the 3D space — can be used to obtain a depth order for any viewpoint. However, this goes with splitting polygons as well, so that using these tree structures certainly does not pay off, especially when dealing with dynamic environments.

The LLR data structure is designed to be able to efficiently find the domain that covers a certain position. Yet, domains in the MLR are stored in a structured graph based on logical coherence, so that looking for domains in the MLR on the basis of a geometrical position is not efficient. Furthermore, numerous domains of the MLR may cover the same position, yet one of them is visible only (see for example the scene tri-spets, Figure 5.29). In the interest of the selection task and also to accommodate partial updates, we keep a simple administration of the relation between domains in the MLR and domains in the LLR, so that it is possible to find a domain in the MLR based on a geometrical position via its related domain in the LLR.

The relation between domains in the MLR and domains in the LLR is administered in the following way:

- Each domain of the MLR is augmented with a list of domains in the LLR that describe a visible part of that domain, the so-called “visibles”. This list is empty if the domain is obscured completely, it contains the reference to one domain in case the domain is opaque and it may contain the reference to several domains in case it is transparent, since then the visible part may obscure different other domains, and so will be split.
- Each domain of the MLR is also augmented with a list of domains in the LLR that (partly) obscure it, the so-called “invisibles”.
- Each domain of the LLR is augmented with a list of domains in the MLR that (partly) cover the area of the LLR domain, the so-called “parents”. This list is ordered by visibility (i.e., distance to the viewer) only up to the first domain that is opaque. This partial ordering is necessary and sufficient to be able to correctly handle the shading of the domains.

An administration like this is appropriate for the selection task. Given a position, a domain in the LLR that covers that position can be found efficiently due to the characteristics of both the grid-list structure and the domain representation. Based on this, the selection task can either report the first parent of that LLR domain—being the MLR domain that is visible at the given position—, but it can also report one of its other parents, such as the first opaque in the list, or the first domain behind the visible domain etc.

The administration as described also accommodates partial updates. Partial updates will involve addition and/or removal of domains. These actions will be initiated in the MLR. Consequently, upon these changes, the domains stored in the LLR as well as the administration have to be updated as follows.

Adding a Domain

The actions needed upon addition of a MLR domain are straightforward. Naturally the actions as described by the hidden surface removal algorithm (presented in § 5.4.2) have to be performed. In respect with the administration we have to add the following steps.

For each domain in the LLR that turns out to be obscured by the domain to be added, a reference to the MLR domain is simply put first in its list of parents. For each domain in the LLR that turns out to obscure the domain to be added, we have to determine the appropriate position of a reference to the MLR domain in its list of parents. If the obscuring domain is opaque, the reference in the list is simply inserted at an arbitrary position after the first parent. If the obscuring domain is transparent, the LLR domain may have been split — in which case both fragments will get a copy of the list of parents of the original domain¹⁴⁾. The appropriate position of a reference to the MLR domain, is found by checking the order of visibility of the MLR domain with respect to all transparent parents and the first opaque parent already in the list. This check includes a check for interpenetration. If interpenetration occurs, the domain has to be split.

For each domain in the LLR that either obscures or is obscured by a fragment of the MLR domain, a reference to that LLR domain is included in either the MLR domain's list of visibles or list of invisibles, depending on the order of visibility of that fragment.

Removing a Domain

Upon removal of a MLR domain, the visibles of that domain are removed from the LLR, and consequently also the references to them (note: if the MLR domain is opaque there is one visible at most). For such a visible, a reference to it can be found in the list of visibles of the domain first in the list of parents (this is the domain that is in the process of being removed), and in the list of invisibles of all of its other parents, so these references can be easily found and removed.

References to the MLR domain should be removed as well. Only the visibles and the invisibles of the MLR domain have such references in their list of parents, so these references can be easily found and removed.

Based on the administration, damage repair as a consequence of removing a MLR domain can be done at minimal cost. The parents of the visible(s) of the MLR domain, not being the MLR domain itself, are reissued. Prior to this the visibles of these parents are removed from the LLR, since reissuing the parents causes them to be replaced.

¹⁴⁾ Note that in principle this is not correct. After a split, some parents may not cover or partly cover the area of both fragments. However, this can occur for parents after the first opaque parent only. Therefore it is a simplification that may lead to some extra steps when removing the domain, yet it does not lead to incorrect images.

In this way we keep a consistent administration and restricted the actions needed for partial updates to those domains that could possibly be affected.

5.4.6. Discussion

In this section we presented an exact incremental hidden surface removal algorithm that does provide us the functionality needed for the layered object-space architecture presented in Chapter 4. If we look at the results presented in § 5.4.3 we conclude that the algorithm running on a single processor system will be able to handle 100-10000 primitives real-time, depending on the depth complexity of the scene. This may seem to be disappointing since often scenes will be more complex than this. On the other hand, incremental updates as occur in CAD-like applications may often be of that order of complexity. We will come back to this issue after a discussion on cost.

In the introduction of this chapter we noted that the cost of object-space hidden surface removal is compensated by not rasterising primitives that are not visible. In the next we elaborate on that. Details of the cost of rasterisation can be found in Appendix A.

From Figure 5.30 we estimate that the cost of the object-space hidden surface algorithm is in the order of 10K instructions per triangle¹⁵⁾. Note that this number is based on measurements on an implementation of the algorithm in C, which includes the cost of conversion of the polygon representation, memory management, function calls, exception handling, etc.

For the most simple shading technique (Gouraud shading) the costs for shading amounts to calculation of intensity at each vertex (i.e., evaluation of expression A.1 that can be found in Appendix A) and linear interpolation of these intensities across the polygon. For a 100 pixel triangle this requires 400 operations plus about 200 operations per light source. The z-buffer algorithm requires one frame buffer read (z-value), one compare and possible frame buffer updates (colour and z-value) per pixel. This cost will be in the order of 4 operations per pixel. With this, the cost of rasterising one Gouraud shaded z-buffer hidden surface removed 100 pixel triangle illuminated by one light source is in the order of 1000 operations.

For a better shading method (Phong shading) the intensity is calculated according to expression A.1 for each individual pixel, while the vectors involved are interpolated across the polygon. Assuming viewpoint and light sources at finite distances this method requires about 2850 operations plus 11700 operations per light source for a 100 pixel triangle. Again the z-buffer algorithm requires in the order of 400 operations, so the cost of rasterising one Phong shaded z-buffer hidden surface removed 100 pixel triangle illuminated by one light source is in the order of 15000 operations.

¹⁵⁾ Based on the fact that we measured the performance on a machine with a 50MHz R4000 CPU that has a peak floating point rating of 25MFLOPS.

As has been mentioned in the introduction of this chapter, the cost of objects space hidden surface removal and the alternative image space hidden surface removal method is based on different measures. Hence, there is a break-even point that depends on the average size of the triangles, the depth complexity of the scene, the number of light sources, and the complexity of the shading method used.

Based on the above estimated costs we conclude that object space hidden surface removal is not cost-effective for rendering an entirely Gouraud shaded scene, no matter what the depth complexity of the scene turns out to be (it would be cost-effective if there are > 50 light sources, which is not very realistic in CAD-like applications). Yet, since the object space hidden surface removal allows us to restrict update efforts to a mere fraction of the amount of primitives involved, the object space method may in cases be cost-effective for partial updates.

For the Phong shading method we conclude that for unexceptional conditions — that is, a common average size of the triangles, depth complexity, and number of light sources — object space hidden surface removal can be cost effective for rendering a scene ¹⁶⁾. In this case the object space method is almost certain to be cost-effective for partial updates, even if the scene is illuminated by one light source only.

The trend in computer graphics applications is to add more complexity to the shading method (e.g., texturing, transparency, etc.). It will be clear that in that case the conditions for which the object space hidden surface removal will be cost-effective are relaxed. If we also succeed to reduce the complexity of the object space hidden surface removal algorithm, as discussed in § 5.4.4, the break-even point would shift in favour of the object space hidden surface removal algorithm even further.

When comparing the cost of the object space hidden surface removal algorithm and the cost of rasterisation of Gouraud shaded triangles one might wonder why the object space hidden surface removal algorithm does not handle two orders of magnitude more primitives, since graphics systems can be found that handle in the order of 1 M Gouraud shaded triangles per second. Such graphics systems make use of specialised rasterisation hardware that operates on integers and that is based on pipelining and multiprocessor techniques. In a similar way we need specialised object space hidden surface removal hardware. This issue should be investigated further.

¹⁶⁾ Note, more efficient versions of the Phong shading method on which we based our cost estimate have been developed, at the cost of accuracy. On the other hand we did not include overhead costs in the cost estimation. The numbers presented here serve to indicate orders of magnitude.

5.5. Scan-Conversion

Scan conversion is the process of producing the component pixels of the rasterised image, also known as rasterisation. In this process we have to deal with the issue of converting a representation in a continuous space into a representation in a discrete space¹⁷⁾. On this issue F. C. Crow states:

"The simplest algorithm for determining the color of a pixel is to evaluate the intersections of all surface fragments with a ray emanating from the eye. The intersecting surface closest to the eye determines the color of the pixel. Areas lying in between the rays will be lost. Since such rays are infinitesimal thick, we can argue that virtually everything is lost." [Crow84]

This statement addresses one of the problems caused by the limited resolution of *point sampling*: the method may miss detail (see Figure 5.31). To reduce this problem, the sampling resolution can be increased by means of *supersampling*, that is, taking more samples per pixel area — in general assuming pixel areas to be cells of a rectangular grid. Subsamples are then combined to produce the final intensity. Supersampling in combination with an image space hidden surface removal algorithm implies that the resolution of the hidden surface removal algorithm has to be increased likewise. As a result, supersampling increases the effective resolution at the cost of a proportional increase in computation and storage demands. Supersampling reduces the problem of missing detail, yet it does not resolve it.

Increasing the number of subsamples to infinity leads to *area sampling* (that is an integration over the pixel area). For each pixel, the resulting intensity is determined by evaluating the area of all display primitives that cover a pixel area. Each primitive contributes to the pixel intensity proportional to the amount of area covered. This technique — that resolves the problem of unseen detail — is known as *unweighted area sampling*. A pixel projected onto the CRT surface is a nonhomogeneously illuminated spot. An exact convolution calculation that corresponds to the characteristics of that spot is far too complex and even display dependent. A commonly used approximation is assuming pixel areas to be nonoverlapping cells of a rectangular grid. Note that area sampling does not go with image space hidden surface removal algorithms. For area sampling an exact (that is object space) hidden surface removal algorithm is a necessity.

Area sampling may resolve the problem of unseen areas, however, it does not resolve aliasing that occurs because of the limited spatial sampling frequency (see Figure 5.31). Aliasing is reduced by applying appropriate convolution filters to the image. This corresponds to *weighted area sampling*. Weighted area sampling typically introduces a dependency of the influence on the intensity per unit area based on the distance from the center of the pixel¹⁸⁾. The weighting functions of

¹⁷⁾ A continuous space implies a space in which a continuum of positions occur, whereas in a discrete space a discrete set of positions can be found.

adjacent pixels overlap to guarantee a continuous transition for objects that move from one pixel to the next. More on this subject can be found in [Foley90, Crow81, Crow84].

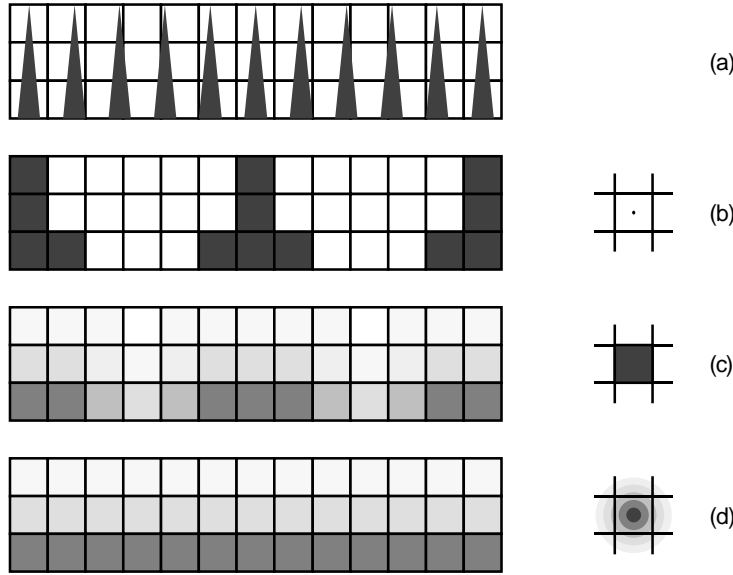


Figure 5.31: The result of different sampling techniques. A regular spaced spike structure (a) that is rasterised by means of point sampling (b), unweighted area sampling (c), and (d) weighted area sampling using a cone filter. On the left we find a graphical representation of the distribution of the sampling functions used.

The low level representation (LLR) from which the display is refreshed is the result of an object-space hidden surface removal algorithm. As a result, the patterns in the LLR are nonoverlapping. The geometrical properties of these patterns are described by the *domain function* $D(x, y)$, whereas the colour properties of patterns are described by the *colour function* $C(x, y)$. The colour on a certain position can be found by:

$$C(x, y) = \sum_{k=1}^n \int D_k(x, y) C_k(x, y) dx dy \quad (5.5)$$

in which D_k and C_k are the domain and colour function of pattern k and in which n denotes the number of patterns that contribute to the image. In case the contribution of domains to a pixel is determined by unweighted area sampling, this reduces to:

$$C(x, y) = \frac{1}{A_P} \sum_{k=1}^n A_k(x, y) C_k(x, y) \quad (5.6)$$

¹⁸⁾ Unweighted area sampling corresponds to filtering the image using a box filter, so that

The normalisation factor A_p is the inverse of the amount of area covered by a pixel cell. $A_k(x, y)$ is that part of the area of a pixel cell located at (x, y) that is covered by pattern k .

A discussion on the organisation of an implementation of the scan-conversion process can be found in § 7.2.

5.6. Conclusions

Scan-conversion as well as hidden surface removal in a great extent involves geometry-based sorting. We formulated a representation of objects, named domains, in which the geometrical data of the objects are sorted. This domain representation reduces the complexity of both the hidden surface removal and the scan-conversion process. We demonstrated how to store objects in a data structure that orders the objects based on their geometry. This data structure significantly reduces the search space for geometry-based object identification.

The incremental object-space hidden surface removal algorithm presented in this chapter is based on binary operations on domains. It can handle a mixture of opaque and transparent objects. By keeping an administration of relations between objects of the scene and visible parts of these objects, the binary operations can be used to add and delete individual objects so that incremental changes affect only those objects of which the visibility is changed.

Object-space hidden surface removal preserves the information on pixel coverage. This can be used to improve the image quality by applying unweighted or weighted area sampling.

Based on the measured cost of the current implementation we conclude that object-space hidden surface removal can be cost-effective for the more advanced shading methods. It will be cost-effective in particular for partial updates. Optimisation and parallelisation of object-space hidden surface removal algorithms should be further investigated. It is also worth investing in research on specialised object space hidden surface removal hardware.

within the pixel cell there is no dependency of the influence on the intensity per unit area.

Illumination and Shading of Polygonal Models

Synopsis.

Dynamic graphics applications are quite demanding on the visual quality of smooth shading of planar polygon meshes. In this chapter we will review different shading methods. We introduce phased shading and present an efficient high quality method based on angular interpolation. We will show how spherical trigonometry leads to a linear expression of the angle between the vectors involved in the shading calculations. The outcome is a parameterised piecewise quadratic expression in terms of pixel position. This expression can efficiently be computed by means of forward differencing at the cost of two additions per pixel.

6.1. Introduction

By looking around us in the real world we observe the result of rather complicated physics: the interaction of photons with the inhomogeneous entities that make up the physical environment. This reality is far too complicated to simulate accurately at all, let alone in real-time. Therefore computer generated images are produced by using a simplified *illumination model* that describes the interaction between light and the elements of the simulated 3D environment.

"Illumination models are often discussed in the computer graphics literature in a theoretical sense, however, when put into practice the models are greatly compromised by the limitations of shading technique, of image display, and of the knowledge of physical properties required to apply the model. The interrelationships of perception, physics, rendering techniques, and the display techniques coupled with the current state of the art make it nearly impossible to make significant new advances by addressing only one of these topics in isolation." [Hall88]

The visual quality of computer generated images of 3D scenes is directly related to the cost of the shading technique used to generate these images. Our concern is not just the quality of individual images, it involves temporal aspects of interaction as well. As a result, the extent to which we can strive for image quality for interactive graphics applications is often limited.

For instance, in applications in which the user can directly manipulate the objects in the scene, we have to rule out *global illumination* methods such as ray-tracing or radiosity, since global illumination methods do not only take into account the direct illumination of a surface by light sources, they also take into account the indirect illumination by all other surfaces in the scene. In doing so, the complexity of these methods is orders of magnitude higher than the complexity of *local illumination* methods that simply leave out these surface-surface interactions. Yet, in case of more restricted user control — such as in architectural walkthroughs — it may very well be possible to include global illumination elements by preprocessing viewpoint independent parts of the shading calculation. This has briefly been mentioned in §4.5.2.

For the application area of concern — being CAD-like applications in which the user can directly manipulate the objects in the scene — we need a local illumination model and a shading method based on this model to smoothly shade curved surfaces approximated by planar polygonal meshes, preferably producing the best quality image. This also has to be brought in relation with Maxim 1 which states that we have to reduce the effort to perform interaction tasks. This issue will be addressed in the next section.

6.2. Illumination & Shading

6.2.1. Illumination Model

Most popular shading methods are based on the illumination model developed by Phong Bui-Tuong [Phong 75] that — in spite of its simplicity — has the potential to produce remarkably realistic results. This model incorporates *ambient*, *diffuse* and *specular* components. More light sources may contribute to an individual polygon, in which case the diffuse and specular components for each of the individual light sources have to be summed. The intensity vector I ¹⁾ is calculated using the expression :

$$I = I_{\text{ambient}} + \sum_{\text{light sources}} I_{\text{light source}} \cdot ((N \cdot L) + (E \cdot R)^n) \quad (6.1.a)$$

In this expression details such as the coefficients that describe the properties of the surface and the distance attenuation function are left out. These details do not influence the core of the methods that are used to calculate the light

¹⁾ Each intensity vector is a vector in the 3D RGB color space. Therefore I has three colour components I_R, I_G and I_B .

intensity across an individual polygon. For more information on surface property issues and attenuation we refer to, for instance [Foley 90] or [Hall 88].

$I_{\text{light source}}$ is the intensity vector of the light source, n is a coefficient which relates to the reflectivity of the surface. The caption of Figure 6.1 describes what the vectors used in 6.1.a stand for. The vectors N , L , E and R are normalised. With the angles α and β that relate to these vectors as shown in Figure 6.1 we can reformulate this expression and come to the equivalent form:

$$I = I_{\text{ambient}} + \sum_{\text{light sources}} I_{\text{light source}} \cdot (\cos \alpha + \cos^n \beta) \quad (6.1.b)$$

In this illumination model, the ambient term (I_{ambient}) and the diffuse term ($I_{\text{light source}} \cdot (N \cdot L)$, or its alternative form $I_{\text{light source}} \cdot \cos \alpha$) account for surfaces that are ideal diffuse reflectors, i.e., reflectors for which the intensity of the reflected light is equal in all directions. The ambient term replaces the surface-surface interactions that are handled in more detail by global illumination models: it represents the amount of energy of the indirect light cast upon the surface area by the environment. The diffuse term represents the amount of energy of the direct light that each light source casts upon the surface area. According to Lamberts law (see [Foley 90]) the amount of energy the direct light produces is proportional to $\cos \alpha$.

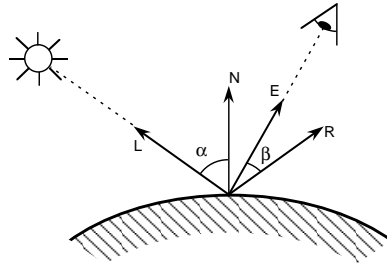


Figure 6.1: Vectors involved in Phong's Illumination model. The diffuse component relates to α (i.e., the angle between the surface normal N and the direction of the light source L). The specular component relates to β (i.e., the angle between the direction of reflection R and the direction of the viewpoint E).

The specular component accounts for nondiffuse reflection. A perfect mirror is the ultimate nondiffuse reflector. It reflects 100% of the direct light in the direction of the reflection vector (see Figure 6.1). Less perfect reflectors tend to disperse the energy in a gaussian-like distribution around that direction. This effect is simulated by the term $I_{\text{light source}} \cdot (E \cdot R)^n$, or its alternative form $I_{\text{light source}} \cdot \cos^n \beta$. This term produces the highlight caused by a reflection of the light source that is characteristic for shiny surfaces.

Often the specular component is calculated by making use of the vector H that represents the surface normal that would produce the mirror reflection

from the light to the eye [Blinn77]. This vector can be calculated using the following expression :

$$H = \frac{L + E}{|L + E|}$$

Or in words : H is the normalized vector halfway between L and E (see Figure 6.2). This vector was introduced by Blinn to be able to replace $(E \cdot R)$

An Object-Space Based Display Controller

Synopsis.

In this chapter we present the architecture of a display controller that refreshes a raster display from an object-space frame representation. The display controller scan-converts two-dimensional areas and produces scanline-based video signals. Each refresh cycle is characterised by repeated actions that relate to two different time scales. This logically led to the design of a two-phased display controller architecture with two classes of processing elements and interconnections. The custom modules designed for this architecture are based on an abstraction that is not restricted to a specific type of primitive or a specific type of rendering. The two-phased modular structure makes it possible to scale the system's resolution and maximum image complexity independently. The implementation of a one-phase prototype that serves to demonstrate the image quality of the system is described.

7.1. Introduction

High quality visualisation and interaction facilities are essential features of today's workstations. Recognising this, system designers paid special attention to the image generation pipeline in order to improve both image quality and interaction behaviour. By improving the image generation pipeline, the frame buffer access bottleneck became more and more apparent. To overcome this problem, all sorts of partitioning strategies have been developed, seemingly without much questioning of the function of the frame buffer and the effectiveness of the frame representation itself.

The function of a frame buffer in graphics systems is to uncouple the real-time refresh process from the computationally intensive image synthesis process. In order to separate these two processes, storage of the image is needed in a representation suitable for the refresh process. The enormous evolution of hardware that occurred during the last decade did not produce an

image synthesis system that meets the timing requirements imposed by the refresh process. Throughout the years, the increased capacity of processing systems has largely been negated by the ever increasing demands on image quality and image complexity. More fundamentally, since there is no limit on the scene complexity, image synthesis by definition is not a real-time process. It cannot be guaranteed that the image synthesis process will meet the real-time requirements of the refresh process. Therefore, uncoupling of the image synthesis process and the refresh process is inevitable.

Frames, stored in the frame buffer, are by default represented in the form of individual pixels. In Chapter 4, an architecture for an image synthesis system has been presented that is designed to have optimal interaction support for realistic 3D graphics. There it was stated that —from an interaction point of view— there is no need to have access to an image representation in the form of individual pixels. The performance and resolution of image synthesis systems have, to a major extent, been limited by the bare pixel update speed as determined by memory access times. The availability of powerful single-chip processing systems makes it in principle possible to store frames in a compressed form and decompress the compressed representation during the real-time refresh process. This approach indeed reduces the bandwidth requirements for frame buffer access. However, for the purpose of interaction and partial updates it should still be possible to relate a compressed output representation with the screen position (see § 4.3. and § 5.4.5). This property does not hold for most well known image compression algorithms.

These observations led us to investigate whether a CRT display could be refreshed directly from an object-based representation of the frame. As stated in Chapter 4, an object-based frame store is more suited for interaction purposes than the conventional pixel-based frame store. It can also be much more compact, thereby reducing the frame buffer access problem. In this chapter, the result of this investigation is presented.

7.2. Unfolding the Scan-Conversion process

In the proposed architecture, the low level representation (LLR) described in § 4.4 from which an object-space display controller refreshes the display, contains a set of nonoverlapping domains. Domains describe the geometry of a pattern projected onto the 2D image plane in a nonrasterised form (see § 5.2). This implies that the real-time refresh process involves scan-conversion of these patterns. Real-time scan-conversion requires a considerable amount of computational resources. As a result, any reduction of costs such as the exploitation of coherence is essential. For each pattern, the colour of adjacent pixels, as well as the intersections of the edges of a domain with the next pixel-row can be calculated incrementally. Making use of coherence reduces the processing requirements of the display controller, but even then, real-time scan-conversion requires in the order of several thousand MIPS.

In the process of converting 2D areas into 0D points, a scan-conversion algorithm that exploits spatial coherence will in essence be based on two nested loops, each scanning in one of the two orthogonal directions thereby reducing the dimensionality by 1. The lean production strategy (see Chapter 4), prescribes that pixels should preferably be produced just in time to avoid unnecessary memory access. Then the path of the electron beam scanning the display area (see §3.2.1) determines the order of the nested loops of the scan-conversion process: the outer loop should make incremental steps in the vertical (y) direction, whereas the inner loop should make incremental steps in the horizontal (x) direction.

```

for ( y = 0; y < VERT; y++ ) do
  begin
    ...
    for ( x = 0; x < HOR; x++ ) do
      begin
        ...
      end
    ...
  end
end

```

In our implementation of the scan-conversion process, the cycle of the outer loop of the algorithm for scanline s will for a pattern k explicitly produce the following result:

$$C_s(x) = \frac{1}{A_s} D_{k,y}(x) C_{k,y}(x) \quad (7.1)$$

The normalisation factor $\frac{1}{A_s}$ is the inverse of the amount of area covered by a scanline. $D_{k,y}(x)$ describes the amount of pixel area covered by pattern k along scanline y (see the discussion on area sampling in §5.5). $C_{k,y}$ is the colour function of pattern k .

The result for all N domains in the LLR for scanline s is:

$$C_s(x) = \frac{1}{A_s} \sum_{k=1}^N D_{k,y}(x) C_{k,y}(x) \quad (7.2)$$

This set of functions along a scanline is an intermediate result of the scan-conversion process. For each pixel of the scanline, the individual terms of this expression have to be evaluated and summed to produce the colour value (or intensity) of the individual pixels — which in effect is execution of the inner loop of the scan-conversion process. This two-phase approach goes with two different classes of requirements on computing resources. This difference will reflect in the architecture of the display controller.

7.3. Architecture of the Display Controller

In the design of the display controller, the following requirements have been taken into account.

- The system should be scalable. That is, it should be possible to configure a system for a particular resolution and maximum image complexity ¹⁾.
- The scalability should show a performance improvement linear with the increase of hardware.
- Critical elements should be implementable in custom VLSI.
- It should be possible to use the same architecture for increased packing densities (i.e., the architecture should not be determined by current VLSI technology).

These requirements and the two-phase approach of scan-conversion, as suggested in the previous section, resulted in a display controller architecture with two different types of processing elements : the x-processors and the y-processors (see Figure 7.1).

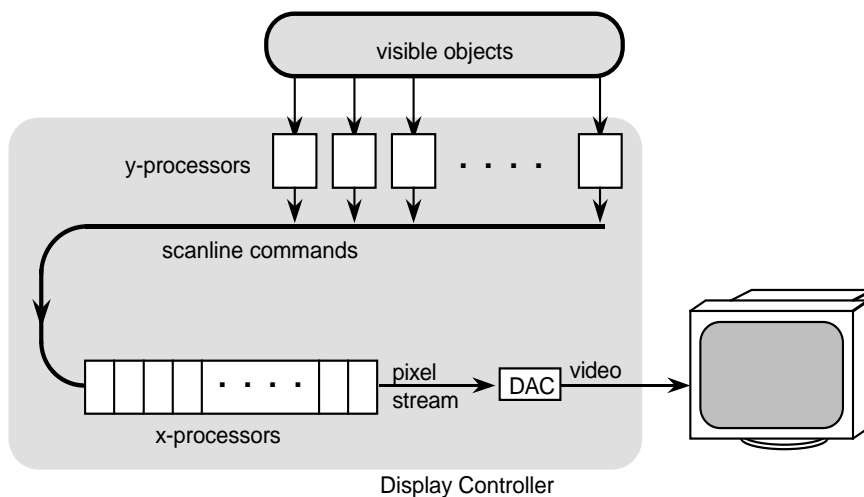


Figure 7.1 : Block diagram of a grey-level Display Controller. The y-processors scan objects and generate scanline-based commands for the x-processor array that produces a pixel stream. A full colour system requires 3 pixel streams.

¹⁾ A measure of image complexity is the number of patterns as well as the complexity of the colour functions of the individual patterns (that is number of light sources, diffuse only or diffuse and specular reflection, transparency etc.). We insist that the system should be able to handle frames that consist of lots of small patterns with rather simple colour functions just as well as frames that consist of fewer patterns with more complex colour functions.

In this architecture the array of y-processors perform the calculations involved in the outer loop of the scan-conversion process — that is, calculation of the intersections of the domain of a pattern k with a scanline and the colour function along that scanline.

```

for (  $y = 0$ ;  $y < \text{VERT}$ ;  $y++$  ) do
  begin
    calculate  $D_k(x, y) C_k(x, y)$ ;
  end

```

The result of this comes in the form of *scanline commands* (i.e., the terms of expression 7.2), on a per scanline basis. These commands are sent to a one-dimensional systolic array ²⁾ of x-processors. The x-processors sort and process the scanline commands and produce a stream of digital pixel values that are converted into analogue video signals.

Note that the display controller architecture is a hybrid (§3.3); it is a combination of image-space parallelism (the x-processors) and object-space parallelism (the y-processors). The display controller can be configured with any number of x-processors to accommodate any display resolution desired ³⁾. The performance of the system in terms of image complexity is proportional to the number of y-processors ⁴⁾. In this way, the two-phase approach allows us to choose different levels of parallelism for the two tiers, and thereby tune different characteristics of the display controller independently.

7.4. The X-Processor Array — or Difference Engine

The x-processor array is the most time critical element of the display controller. This array has to produce pixels at the pixel clock rate. For high resolution systems (> 1M pixels) this rate is close to 100 MHz. There are no general purpose elements that can be used for this purpose, so we had to implement these elements in custom VLSI.

²⁾ A systolic array is a dedicated multi-stream processor in which data and instructions flows between processing elements in a pipelined fashion (see §3.3.2).

³⁾ That is, within the range of current display technology. The throughput rate of any physical device is limited and so would be the maximum resolution of this system. In Appendix B we present 'virtual arrays' as a means to go beyond this physical limit.

⁴⁾ The number of y-processors determines the bandwidth to the (distributed) frame store (which relates to the maximum number of patterns that can be handled) as well as the available processing power (which relates to the maximum complexity of the colour function that can be handled).

7.4.1. Basics of the X-Processor Array

The high throughput requirements of the system at the level of the x-processor array made it clear that processing of scanline commands as well as the data transport mechanism should be simple. Inspired by the SAGE⁵⁾, presented by Gharachorloo et.al [Gharachorloo 88], we opted for a linear systolic array. The inner loop of the scan-conversion process

```

for ( x = 0; x < HOR; x++ ) do
  begin
    calculate  $D_{k,y}(x)$   $C_{k,y}(x)$ ;
  end

```

is unfolded so that each processor of the array executes one cycle of the loop. Consequently each x-processor is assigned to a pixel column: there are as many x-processors as there are pixels on the scanline. The scanline commands produced by the y-processors comprise all the information for execution of the inner loop. Evaluation of $D_{k,y}$ is basically a range check, so that the linear systolic array automatically sorts commands in linear time. As a result, the scanline commands of a particular scanline can be accepted in arbitrary order.

Each x-processor performs incremental calculations on the scanline command it receives. The result is then passed on to its right neighbour. The left-most x-processor accepts the scanline commands generated by the array of y-processors. The 'heart-beat' of the systolic array is equal to the pixel clock rate: opcode and operand of the commands are passed on at the rate at which pixels are produced.

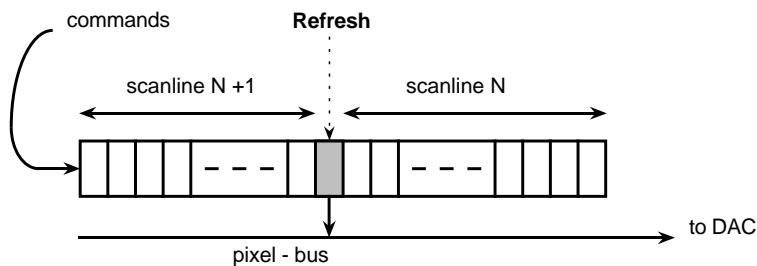


Figure 7.2: The x-processors of the array operate on commands of at most two successive scanlines. The commands for one scanline and those for the next one are separated by a Refresh command.

⁵⁾ SAGE stands for Systolic Array Graphics Engine.

Commands from one scanline are separated from commands of the next scanline by, a *Refresh* command (see Figure 7.2). Upon receiving a Refresh command an x-processor outputs the accumulated value — i.e., the result of evaluation of all the commands executed after the preceding Refresh command — onto the 'pixel bus'. In the meantime the processor resets its internal registers to be ready to accept the commands for the next scanline. In this way, a Refresh command directly controls the pixel flush. Consequently Refresh commands will have to be issued at regular intervals, equal to the total line time. Since processors of the array should output accumulated pixel values one at a time, there can be only one Refresh command active at a time. There is no Refresh command active during the horizontal and vertical retrace times (see § 3.2.1), since then the array does not have to produce pixels.

7.4.2. The Command Set

For high quality shading models, the intensity along a scanline appears to be a sum of Gaussian-like features. For evaluation of such colour functions we have considered various alternatives. In Chapter 6 we demonstrated how the ambient, diffuse, and specular components of Phong's illumination model can be approximated by piecewise second order polynomials. Such low order polynomials are well suited to be evaluated by means of forward differencing. In turn, forward differencing is well suited to be implemented in VLSI timing requirements can be met at minimal cost in terms of chip area.

We therefore decided to let the system evaluate colour functions by means of forward differencing. Consequently we use the term "Difference Engine" if we refer to the integrated systolic array of x-processors ⁶⁾. We anticipated that for evaluation of the illumination model it would be useful if partial contributions of multiple light sources could be summed by the Difference Engine. We noted that efficient generation of regular patterns can be obtained by (re)setting the intensity and/or first and/or second order difference at fixed intervals. Furthermore we anticipated that a command to reproduce rasterised images efficiently would be useful. Given this, we came up with the command set of the x-processor shown in Table 7.1. At the CWI this command set was examined by means of a structural simulator. This structural simulator and some of its implementation issues will be described in § 7.4.4.

⁶⁾ The term *Difference Engine* is not new in the history of computing. In 1882 Charles Babbage built an experimental model of a mechanical calculator that he called Difference Engine because it was "based on a mathematical principle known as the method of finite differences". It was supposed to be part of a machine that could calculate and print astronomical tables. An attempt of Babbage to build this first mechanical computer failed after a dispute between Babbage and his chief engineer. In November 1991 a full-size computing engine based on Babbages original design built by Swade and his colleagues at the Science Museum in London flawlessly performed its first major calculation [Swade 93]. (Also we were inspired to use the name as result of a Science Fiction novel by William Gibson & Bruce Sterling).

Mnemonics	Operands	Operation
Eval0	X,dX,I	Set the intensities between the pixel locations X and X+dX at I and disable the accumulation of intensities until the next Refresh command
Eval1	X,dX,I	Accumulate the intensities between the pixel locations X and X+dX.
Eval2	X,dX,I,dI	Interpolate and accumulate the intensities between the pixel locations X and X+dX by first order forward differencing.
Eval3	X,dX,I,dI,ddI	Interpolate and accumulate the intensities between the pixel locations X and X+dX by second order forward differencing.
SetI	X,I	Set the intensity at pixel location X to I
SetdI	X,dI	Set the first forward difference of the intensity at pixel location X at dI
SetddI	X,ddI	Set the second forward difference of the intensity at pixel location X at dI
SetPI	X,dX,I	Set the intensity at pixel locations X, X+dX, X+2dX.... to I
SetPdI	X,dX,dI	Set the first forward difference of the intensity at pixel locations X, X+dX, X+2dX.... at dI
SetPddI	X,dX,ddI	Set the second forward difference of the intensity at pixel locations X, X+dX, X+2dX.... at ddI
Dis	X, dX	Disable the accumulation of the intensities between the pixel locations X and X+dX until the next Eval command.
Acc_mode	-	Enable/disable accumulation of negative intensities.
Refresh	-	Output the accumulated intensity and reset the processor.
Nop	-	No operation

Table 7.1 : Command set of the x-processors.

The number of bits of the data that goes with the commands listed in Table 7.1 is as follows.

For specification of the position and the width of a span we decided that 12 bits should be sufficient. This implies that the Difference Engine can cope with screen resolutions of up to 4 Kpixels in horizontal direction.

Mostly in graphics systems pixels are 8 bits per colour component —i.e., 3×8 bits for a full colour system. For high quality applications, such as film production, we may find systems that produce pixels of 12 bits per colour component. We decided that the Difference Engine should be able to accommodate this high resolution standard. Using 12 bits per component also simplified the design of the data path since it matches nicely with the 12 bit pixel addressing scheme —that uses the same data path as we will see when we describe the implementation. Furthermore it allows us to allocate sign-, overflow- and extra precision-bits when 8 bit colour components are sufficient.

Evaluation by means of forward differencing implies that round-off errors accumulate. Second order differencing implies that the error can at maximum be $\text{resolution}^2 \times \text{round-off error}$. Since the maximum number of steps is 4 K, the intensity and related differences should be specified with 12 bits + 24 precision bits to guarantee a correct 12 bits pixel value up to the last pixel in the span. In other words: I, dI and ddI have to be 36 bit fixed-point numbers.

7.4.3. Implementation of the Difference Engine

In this subsection, the implementation of the x-processor elements that make up the Difference Engine will be described. The actual chip design and verification has been done by our colleagues in Twente. For more detail on the operation and inner structure of the x-processor, we refer to [Jayasinghe 89]. For more information on the theoretical justification of the design, we refer to [Jayasinghe 91c].

As already explained in the previous subsection, the intensity and related differences are represented by 36 bit fixed-point numbers. Feeding I, dI and ddI into the system simultaneously would involve 216 pins per package for the data path alone. We had to limit the number of pins per package. Therefore the chip has one 36 bit data input and one 36 bit data output port. This implies that I, dI and ddI have to be sent serially. Since 12 bits are sufficient for the representation of the X and dX, we can send both X and dX simultaneously. Table 7.2 shows the serialised data as it is sent into the Difference Engine. The prescribed sequential ordering has been optimised to maximise the throughput of the x-processor. Throughout the rest of this chapter we will indicate the serialised atomic units of the commands as being *instructions*. These instructions hop from one processor to the next at the pixel clock rate.

Note that as an exception, the intensity that goes with the Eval0 command is specified by a 12 bit integer value. The Eval0 command — that serves to reproduce rasterised images efficiently — does not allow for accumulation, hence we do not need precision bits at all. As a result, the Eval0 command occupies 1 time slot only.

Command	Time slot #1	Time slot #2	Time slot #3	Time slot #4	Time slot #5
Eval0(X, dX, I)	X, dX, I				
Eval1(X, dX, I)	X, dX	I	Accumulate		
Eval2(X, dX, I, dI)	X, dX	dI	I	Accumulate	
Eval3(X, dX, I, dI, ddI)	X, dX	ddI	dI	I	Accumulate
SetI(X, I)	X	I			
SetdI(X, dI)	X	dI			
SetddI(X, ddI)	X	ddI			
SetPI(X, dX, I)	X, dX	I			
SetPdI(X, dX, dI)	X, dX	dI			
SetPddI(X, dX, ddI)	X, dX	ddI			
Dis(X, dX)	X, dX				
Acc_mode	Enable/Disable				
Refresh()	Refresh				
Nop()	No operation				

Table 7.2: Serialisation of the commands in atomic instructions.

Figure 7.3 shows the architecture of the x-processor at register transfer level. As is shown in this figure, the x-processor has four registers and three main busses that carry data, instructions and pixel values. Registers A, B and C serve to store respectively the intensity, the first difference and the second difference of

the intensity function. Register D serves to hold the accumulated intensity. Furthermore, the x-processor has a 36 bit so-called 'block carry' adder, a control unit and some multiplexers and buffers. For increased clarity, Figure 7.3 does not show the clock distribution, control lines and other details of the processor. For this type of information, we refer to [Jayasinghe 89].

Figure 7.3 does show that the 36 bit data is broken in three blocks of 12 bits each. In doing so, the addition could be pipelined and delay caused by carry propagation is nonexistent. Because of the pipelining, the data that enters the array should be skewed in time. Consequently, the pixel data leaving the array will have to be de-skewed again. From a conceptual point of view data skewing does not influence the operation of the processors, so in the rest of this exposition we ignore the fact that the data is skewed.

The data flow through the processor is handled by a control unit, that also keeps track of write-protect information. Furthermore the control unit sends signals to downstream processors via the instruction bus.

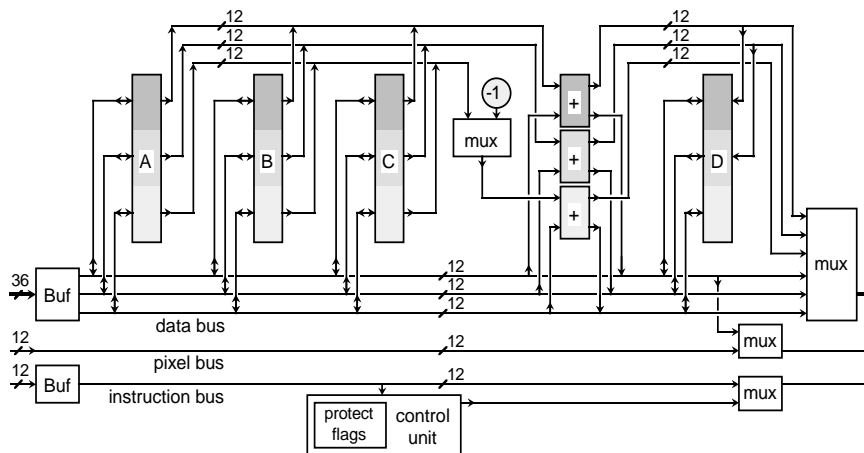


Figure 7.3: Register level architecture of the x-processor. The arrows indicate the directions for read/write access. To allow for pipelined block carry addition, the data bus and the 36 bit registers are subdivided in three 12 bit sections, in the figure indicated by different shades of grey.

Each clock cycle, 36 bit data is clocked in by the x-processor, together with a 12 bit instruction identifier. To improve the speed of the system, time-critical signals are available directly via some bits of the instruction identifier. These bits (or signals) typically control the data flow through the processor. As a result, the control unit could be kept rather simple, which also reflects in the amount of real-estate. Other bits of the instruction identifier are decoded by the control unit to produce less time-critical signals.

Addressing of the processors (which corresponds to pixel positions) is relative. That is, to be able to execute commands, the processors do not need

to know to which absolute pixel column they are assigned to. Relative addressing is handled in the following way. Each processor decrements the X value of the command that comes along (in fact adds -1). As long as $X > 0$ the commands are not active. The first processor that finds $X = 0$ will start to perform the action that goes with the command. All its downstream processors then start to decrement dX in a similar way. The processor that finds $dX = 0$ turns the command into a Nop instruction, so that no further action is performed by all remaining downstream processors. Relative addressing makes the system very flexible; modules can be simply interchanged or replaced and faulty processors can be set in an 'idle' state (see § 7.4.5).

In the next we will describe the actions performed by the processors for each of the commands.

Eval

The Eval0 command differs from the other Eval commands. It serves to write 12 bit intensity values directly into the D registers of the processors between position X and $X+dX$. It thereby overwrites the content. The content of the D registers then also cannot be changed until after a Refresh command. The Eval0 command — that takes one instruction only — is designed to be able to reproduce rasterised images efficiently.

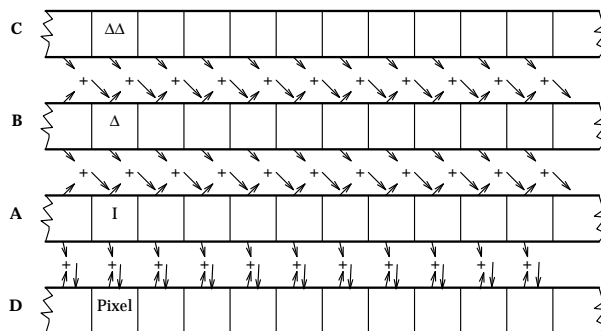


Figure 7.4: Principle of second order forward differencing. The content of two registers are added, the resulting difference is passed on to the right-neighbouring processor, and the resulting intensity is accumulated in the local register D. Note that this is an overview of actions that in the Difference Engine occur sequentially in time.

The other Eval commands are the commands upon receipt of which the processors perform forward differencing and produce the intensity values that are accumulated. All Eval commands are active for the processors between position X and $X+dX$. Figure 7.4 illustrates how the processors of the array cooperate to perform second order forward differencing (i.e., handle an Eval3). It shows how data flows from registers A, B and C via the accumulator to register D and registers A, B and C of the right-neighbouring processor⁷⁾.

Set

The Set commands serve to set one of the values I, dI or ddI at a specific location. A set value will override the value that would otherwise be the result of forward differencing performed by upstream processors. Setting the intensity will in general result in a discontinuity. Setting the first difference will result in a continuous intensity curve with a discontinuity of the first derivative of the curve. Setting of the second difference will result in a smooth change of the intensity curve (see Figure 7.5). The Set commands come in two 'flavours': the nonperiodic, which are effective at one position only, and the periodic, which repeat the setting after every dX positions.

The Set commands write data in the appropriate registers and raise a corresponding protect flag. When this protect flag is true, the Eval commands will not overwrite the content of the register. All protect flags are reset during the last cycle of each Eval command.

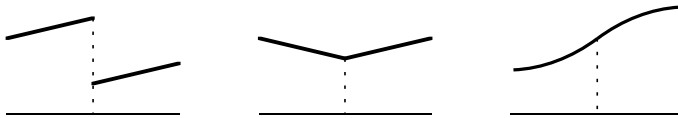


Figure 7.5: Results of a SetI (left) a SetdI (middle) and a SetddI (right).

Dis

The Dis command disables accumulation of the intensity values produced by the Eval commands —that is, disables writing into register D—, for processors between position X and X+dX, yet all other actions still take place. As a result, for an Eval command of which the range goes beyond position X+dX as given by the operand of the Dis command, the processors that come after this position will produce pixel values just as if the span X, X+dX would not have been disabled. The disabled state is reset after execution of the first Eval command that comes along.

The Dis command serves to reduce the cost of rendering of partly obscured patterns twofold. Firstly, the Dis command takes one time slot only, whereas setting up the forward differencing command for a separate section may take as

⁷⁾ By reusing the registers it is possible to perform forward differencing of any order. Note that in that case, the maximum span for which full precision can be guaranteed is limited. For instance if the most significant bit is a sign bit and 8 bits precision is

sufficient, the maximum span of which the results will be exact is $2^{\frac{27}{p}}$ in which p is the order of the forward differencing. As can be deduced from Table 7.2, the number of time slots needed to supply the necessary information is $p + 3$. For higher order forward differencing the Set commands can be used to set the two highest order differences only.

much as 5 time slots (in case of an Eval3). Secondly, the Dis command simplifies the task of the y-processors, since the initial values of the intensity and all the differences for such a separate section do not have to be computed.

Acc_mode

The Acc_mode command serves to be able to disable accumulation of negative intensities in the D register. Evaluation of piecewise second order polynomials can be done efficiently by setting the second order difference at specific locations. In doing so, we can even position the intensity curve with sub-pixel precision —especially useful for confined specular highlights—, yet at the cost of introducing round-off errors that may result in small negative intensity values at the end of the curve. With Acc_mode set we can allow for such small round-off errors.

Refresh

Upon the Refresh command, the content of the D register will be transferred to the pixel output circuitry, and all registers and their flags will be cleared to be ready to process the scanline commands for the next pixel row. As stated in § 7.4.1, this Refresh command has to be issued at regular intervals.

Nop

In this architecture the Nop command (all instruction identification bits are 0) is indispensable. It does not change the state of the processor, and can therefore be safely used to fill-up possible empty time slots before the Refresh command.

7.4.4. Software Simulation of the Difference Engine

In the process of developing custom (VLSI) hardware, simulation tools have to be used to verify a design at different levels of accuracy. These levels include at least the following :

- a) Electronic level simulation : analogue simulation of the digital hardware.
- b) Logic level simulation : technology independent, gate-level simulation.
- c) Structural simulation ; simulation of functionality, registers, flags and logic.
- d) Functional level simulation ; data and control may differ from that of the actual hardware.

Together these levels represent an integrated set of layered simulation tools. The reason for using less accurate simulators (c & d) at all, is that accuracy (that is, the amount of detail) is very expensive in terms of computing resources. As a result, accurate simulators (a & b) are impractical for testing software. Therefore we implemented two levels of simulators for the Difference Engine. A functional level simulator that was optimised for speed and a structural level simulator to validate the hardware design, and to visualise the result of the software that should run on it.

Functional level simulation

The functional level simulator (d) is used primarily to verify that algorithms will perform as expected in practice. The simulator accepts the same instruction set as the Difference Engine, however, it employs instruction codings and word sizes appropriate for fast execution on workstation platforms. As a result it has sufficient speed to allow simple graphical results to be visualised, yet the numerical precision differs from the targeted precision. This simulator is used for exploring high level ideas and testing instruction sets. Alternative shading algorithms and the investigation of wavelet image decoding (see § 7.2.2) was done exclusively on this functional level simulator.

Structural level simulation

In a layered set of graphics hardware simulators, a structural level simulator (c), bridges the gap between hardware fidelity on the one side, and sufficient performance to visualise graphics algorithms on the other. A structural level simulator allows the applications programmer to verify that the output from a shading algorithm will be accurate on the target hardware. It also allows the hardware specialist to trace data and commands and to interrogate the state of any part of the system at any stage.

It is this combination of interests that makes this level of simulation a useful adjunct in dealing with one of the problems which crop up in hardware design: the communication between applications programmers and hardware specialists. Exact simulators (that is, simulators of the levels a & b in the list above) are not the right tools to help the communication between the two different disciplines, since they cannot execute realistic code, and they are comprehensible only to hardware designers. On the other hand, functional level simulators do not provide the information which hardware designers can use. The intermediate structural level simulator is particularly useful in giving the system designers a concrete idea of what the hardware designers have implemented. On the basis of the simulation, questions can be formulated by the applications programmer in terms which can readily be understood by the hardware designers.

A structural level simulator should have a sophisticated user interface to allow for interactive inspection and debugging of executing programs; both the exact digital state of the hardware, and the graphical output should be visualised interactively. Therefore, we designed and implemented "XInPosse", a structural simulator that models the elements of the Difference Engine at a functional level with fidelity and granularity sufficient to predict and evaluate their behaviour [Guravage 93]. Its graphical user interface (see Figure 7.6) allows one to initiate, control, and observe the execution of commands in the array. The simulator provides the user with a means of tracing commands within the array while interactively setting breakpoints. Individual processors can be displayed and their states observed while commands pass through them. An intensity graph for each scanline is drawn, along with a complete output image.

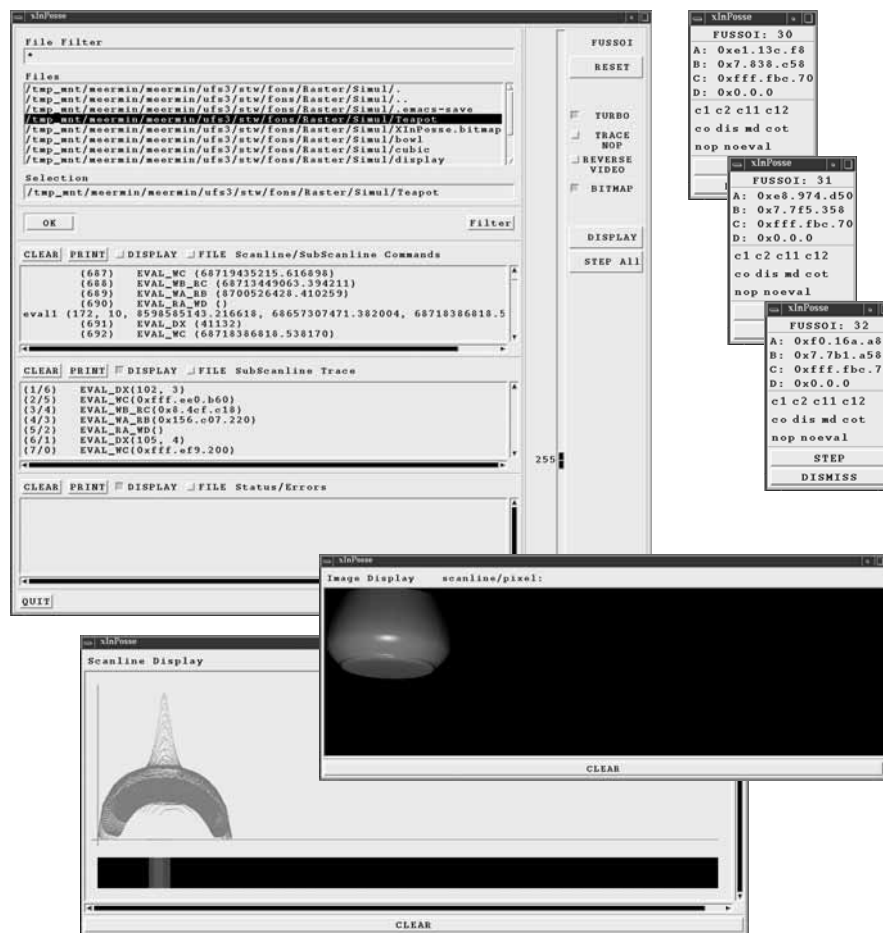


Figure 7.6: Graphics interface of the XInPosse simulator. The main window is used for control of the simulation (file selection, tracing, etc.). The three small windows display the state of individual processors (here processors 30, 31 and 32). The 'Scanline Display' window serves to display the intensity graphs of scanlines and the 'Image Display' window serves to display the complete output image.

We opted for an object-oriented design of the simulator process's components and communications. The user interface was written with Xt and the MOTIF widget set, both C libraries. The simulator, on the other hand, was written in C++. While the proposed hardware generates pixel intensities at the rate required for the screen refresh (up to 85 MHz), the simulator draws pixels between 6 and 10 per second. Though the actual hardware is a systolic array, the processors are simulated serially.

November 24, 1995

The structural simulator is the only medium, short of the hardware itself, which offers sufficient numerical accuracy to compute the pixel intensities based on forward differencing. With it, we verified that the hardware could execute the graphics algorithms correctly and that the limitations on numerical accuracy and range were graphically acceptable.

Straight away, the simulator —that was built based on the hardware documentation — revealed several errors. All were attributed to errors in the documentation. By testing sequences of instructions rather than testing the functionality of individual hardware instructions, we can observe the relationships (and problems) which arise only among multiple commands. The simulator has served as a liaison between software and hardware engineers. It has been used to formulate and specify questions about the hardware before asked of the hardware designers; and to evaluate their answers. A number of nontrivial ambiguities have been resolved this way.

7.4.5. Features and Estimated Performance

The implementation of the Difference Engine resulted in a chip which houses 9 processing elements (see Figure 7.7). To be able to increase the yield, it is made possible to configure the system and bypass faulty processors —provided the bypass itself is not faulty. For this purpose the processors have a bit-serial shift register. Due to the relative addressing scheme, configuration of the system can be done without any consequences for the commands that have to be issued. For the configuration procedure the normal operation mode has to be interrupted. Normally this will be done once only during system setup. However, it can be done unnoticeably during vertical retrace time also.

A second serial shift register is used for off-line testing purposes. For all commands, controller outputs are loaded into this shift register. In a dedicated test circuit the contents of the register of an individual chip can be shifted out to be able to observe the behaviour of the controller.

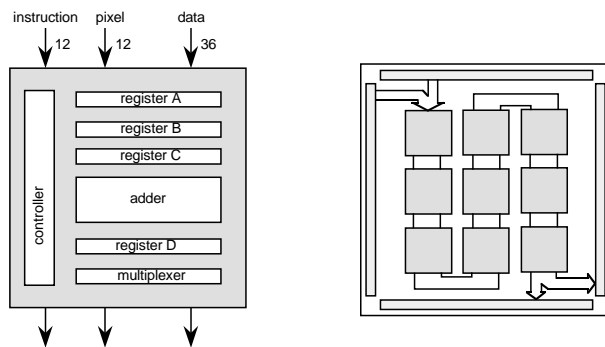


Figure 7.7: Floorplan of one x-processor (left) and of a complete chip that houses 9 x-processors (right).

The actual design of the chip took place at the University of Twente using CAMELEON for the symbolic layout design, the procedural design language GrapMG to build the processor in a hierarchical form and DRACULA for design rule and electrical rule checking.

The Difference Engine is implemented using a 1.6μ CMOS process (technology : C3DM). The 9-processor chip has about 80 K transistors, the die size is $11.4\text{ mm} \times 10.2\text{ mm}$. The chip is housed in a ceramic 144 pin grid array package. Of the 144 pins, 48 pins are for the input bus (data + instruction identifier), 12 for pixel input, 48 for the output bus and 12 for pixel output. The remaining 24 pins are for clock input (4), system configuration (4), testing (2), sign in/out (2) and power supply (12). The chip is designed to operate at 5 V. The power dissipation is estimated to be 78 mW per MHz.

Extensive simulation has indicated that the circuitry of the processors will run at 12–15 ns cycle time (66–85 MHz). In practice it turned out that by mistake the driving capacity of the bond-pads was too low ⁸⁾. The cooling capacity of the package (that was selected on price) is also a limiting factor. The current batch turns out to have a maximum clock rate of 25 MHz.

Due to the (potential) high clock rate and the massive parallelism, the Difference Engine is a very powerful machine. Its performance relates directly to the pixel clock rate. For the targeted high resolution systems (1280×1024) the performance would be as high as 108 GOPS (85 MHz pixel rate \times 1280 processors). For low resolution systems (640×485) the performance would be 13.8 GOPS (21.5 MHz pixel rate \times 640 processors). Note that the processors continue to operate usefully during horizontal retrace time, in spite of the fact that during that time no pixels are produced.

Of main interest however, is the maximum number of patterns that can be rasterised — i.e., that can contribute to pixels on the screen ⁹⁾. This measure is somewhat hard to quantify, since there is no direct one-to-one correspondence with the number of instructions that can be handled by the Difference Engine. The number of instructions per pattern depends on the number of light sources, the average size of the patterns and the order of forward differencing involved in the rasterisation. To be able to give an indication of the performance of the Difference Engine anyway, we assume that there is one light source, that 10% of the patterns fall within the range of the specular term, and that 10% of the patterns fall into two subsequent sections of the piecewise quadratic function used for colour evaluation. Based on these moderate assumptions, we can

⁸⁾ The bond pad design used has a throughput rate of typical 30–35 MHz. This bond pad design was extracted from a library and therefore unfortunately not simulated.

⁹⁾ N.B. The capacity of a graphics system is typically given in number of primitives that can be handled by the system, in other words, the number of primitives that enter the graphics pipeline. In general that does not indicate how many of these primitives actually contribute to pixels on the screen — i.e., are visible.

produce the number of patterns that can be rasterised at maximum ¹⁰⁾. In Table 7.3 these numbers are given for low resolution systems (running at 21.5 MHz), as well as for high resolution systems (running at 85 MHz), for patterns of on average 10×10 pixels and for patterns of on average 100×100 pixels.

Interpolation	patterns / second			
	screen resolution 640×485		screen resolution 1280×1024	
	size 10×10	size 100×100	size 10×10	size 100×100
Quadratic	331 K	33 K	1.5 M	146 K
Linear	497 K	50 K	2.2 M	220 K
Constant	663 K	66 K	2.9 M	292 K
Eval0	2.0 M	199 K	8.8 M	880 K

Table 7.3: The rasterisation capacity of the Difference Engine in terms of maximum number of patterns per second for various orders of forward differencing (Quadratic implies Phong shading, and Linear implies Gouraud shading). Note that these numbers refer to patterns that really contribute to pixels on the screen, i.e., that are actually visible.

The numbers in Table 7.3 are defined by the input bandwidth of the system. We can multiply these numbers by the number of pixels the patterns cover, and divide that by the number of pixels that are output by the system per second. This gives us the ratios shown in Table 7.4. Under the conditions given in the above, these ratios turn out to exceed 1. Hence, there are not enough pixels on the screen to display all the patterns that can be handled. Apparently the capacity of the Difference Engine is such that it is possible to accumulate several intensity values for each pixel on average ¹¹⁾.

Table 7.4 illustrates that for large patterns, the maximum number of intensity values that can be produced per pixel, is higher than that for small patterns. Considering that the smaller the pattern the harder it is to distinguish individual contributions of a large number of light sources we conclude that this behaviour correlates with the lean production strategy; lean production prescribes that the system should not produce more information than what can be seen. This implies that for small patterns the shading calculation should preferably be kept simple (e.g., calculations based on the summed influence of a number of light sources instead of calculations based on the influence of individual light sources).

¹⁰⁾ The geometry of patterns can be quite complex. Here the only restriction on complexity we assumed is that intersection with a scanline leads to one continuous span (e.g., triangles and other convex polygons).

¹¹⁾ Note that we also assumed that the quadratic interpolation implies accumulation of the diffuse and the specular intensity.

Interpolation	average # intensity values / pixel			
	screen resolution 640×485		screen resolution 1280×1024	
	size 10×10	size 100×100	size 10×10	size 100×100
Quadratic	2.1	21.4	2.2	22.4
Linear	3.2	32.0	3.3	33.6
Constant	4.3	42.9	4.5	44.9
[Eval0]	[12.8]	[128.0]	[13.4]	[134.0]

Table 7.4 : The capacity of the Difference Engine in terms of average number of intensity values that can be accumulated per pixel. Note that the Eval0 instruction does not allow for accumulation.

Observe that for the Eval0 instruction, the ratio of intensity values versus pixels is highest of all. However, this instruction is used typically for (re)production of incoherent and/or run-length encoded images for which coherent areas of 10×10 pixels are most unlikely. Furthermore using an Eval0 instruction implies that intensity values cannot be accumulated.

From Table 7.3 we learned that there is a relation between the system resolution and maximum number of patterns that can be processed. This is caused by the fact that the clock speed of the Difference Engine is fixed with respect to the pixel rate. This relation is a useful one. It makes sense that a high resolution system is able to process more patterns —and hence produce more detail — than a low resolution system.

In Appendix B we demonstrate the flexibility of the system by showing some nonstandard configurations that provide ways to adapt —i.e., increase or decrease — the maximum number of patterns that can be processed for a given resolution. Just like the standard configuration, these nonstandard configurations have one input port for the 48 bit instructions and one output port for the 12 bit pixel values. These type of configurations are called *virtual arrays*. Virtual arrays allow us to configure a system to obtain any required performance level, not restricted by the limitations of the physical x-processor implementation. Virtual array configurations may introduce some extra delay, but this will at most be in the order of 0.1 msec, which is insignificant.

7.5. Driving the Difference Engine

In the previous section we discussed the operation and implementation of the x-processors. These x-processors could be kept quite simple, yet they operate at a very high speed. The uninterrupted data stream and the simple operations on fixed-point numbers made it possible to pipeline the executions. Because of this, the maximum throughput rate of the x-processors could meet the requirements (up to 85 MHz).

The y-processors operate at a different time-scale. The complexity of their task is in a different scale also. They operate on floating point numbers and deal with irregular data streams. Because of the level of complexity involved we decided to rely on general purpose hardware. General purpose processors are produced in large quantities, so they can be implemented economically on the most advanced technologies and developed using the most advanced design tools that produce highly optimised designs. As a result it is extremely hard to be competitive in terms of performance. In spite of the advanced technology of current general purpose processors, the required input rate of the Difference Engine is too high to allow for a direct coupling of the Difference Engine and such general purpose processors.

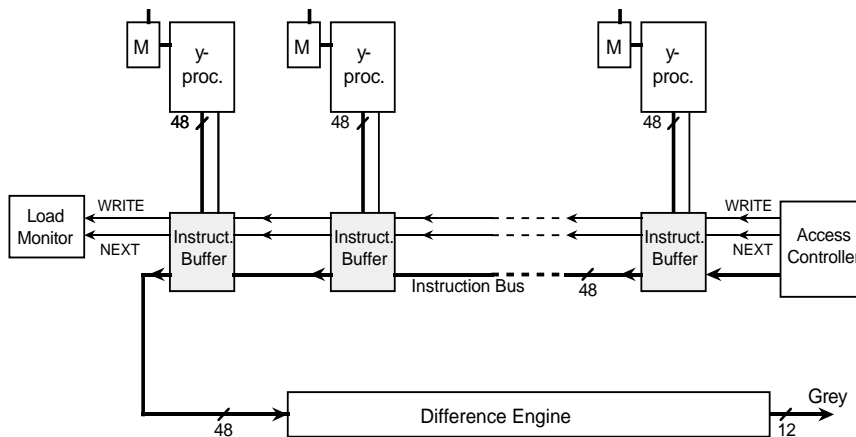


Figure 7.8: Block diagram of a grey-level display controller. The Access Controller, Load Monitor, and Instruction Buffer handle bus arbitration and reduce requirements on the output rate of the y-processors.

In Figure 7.8 we find a more detailed version of the block diagram shown in Figure 7.1. It shows that, beside the Difference Engine and the y-processors, a grey-level display controller comprises an Access Controller, a Load Monitor, and one Instruction Buffer per y-processor. These elements handle bus arbitration and bridge the difference in maximum output rate of the y-processors and required input rate of the Difference Engine. A detailed description of these elements can be found in Appendix C. In short :

The Access Controller is the unit that issues the Refresh command and the control signals NEXT and WRITE involved in bus access at regular intervals.

The Load Monitor is an optional device that provides information on the total number of instructions per scanline. Based on this information, higher level processes may adapt the level of detail or the quality of the illumination model.

The Instruction Buffer accepts instructions at a rate at which the y-processor is able to write. It stores these instructions until access to the Instruction Bus is granted. Then the instructions are written onto the Instruction Bus at the high rate required by the Difference Engine.

The y-processors are general purpose processors that generate the instructions for the Difference Engine ¹²⁾. For synchronisation of the scanning process, the y-processor 'listens' to the bus control signals.

Both the Access Controller and Load Monitor can be built with off-the-shelf hardware. The Instruction Buffer is the main element needed to drive the Difference Engine for which no off-the-shelf hardware could be found. The VLSI design team at the University of Twente did some design studies and simulations that demonstrated the feasibility of the design. In Appendix C we describe: the architecture of a full colour system, the principle of the bus arbitration mechanism, the design and special features of the instruction buffers, and the y-processors.

7.6. Building a Prototype

The feasibility of the object-space display controller described in this chapter can be demonstrated convincingly by actually building a prototype system. The Difference Engine is the fundamental, and technologically most challenging part of the display controller. Therefore this part of the display controller was selected to be implemented in full custom VLSI. To complete a working prototype, the rest of the system was implemented using standard components, necessitating compromises with overall system performance. The most radical compromise is that the instructions for the Difference Engine are stored in a buffer rather than being generated in real-time. From a technological standpoint it is relatively simple to replace this instruction store by an y-processor array to obtain an object-space display controller system in which the instructions are generated in real-time. (Details on this can be found in Appendix C.). Another compromise is that we decided to implement a grey-level system. A full colour system would not more convincingly demonstrate the feasibility of the system; it merely involves more components.

¹²⁾ The display controller can in principle combine different types of specially configured y-processors. One can think for instance of a dedicated "character-generator" that handles a database of instruction tables for characters of different typefaces. Similarly one can think of a dedicated texture-processor.

7.6.1. System Components

The prototype system comprises the following three sub-systems shown in Figure 7.9:

- a SUN 4/100 Workstation,
- an INMOS B408/B409 graphics system,
- the Difference Engine.

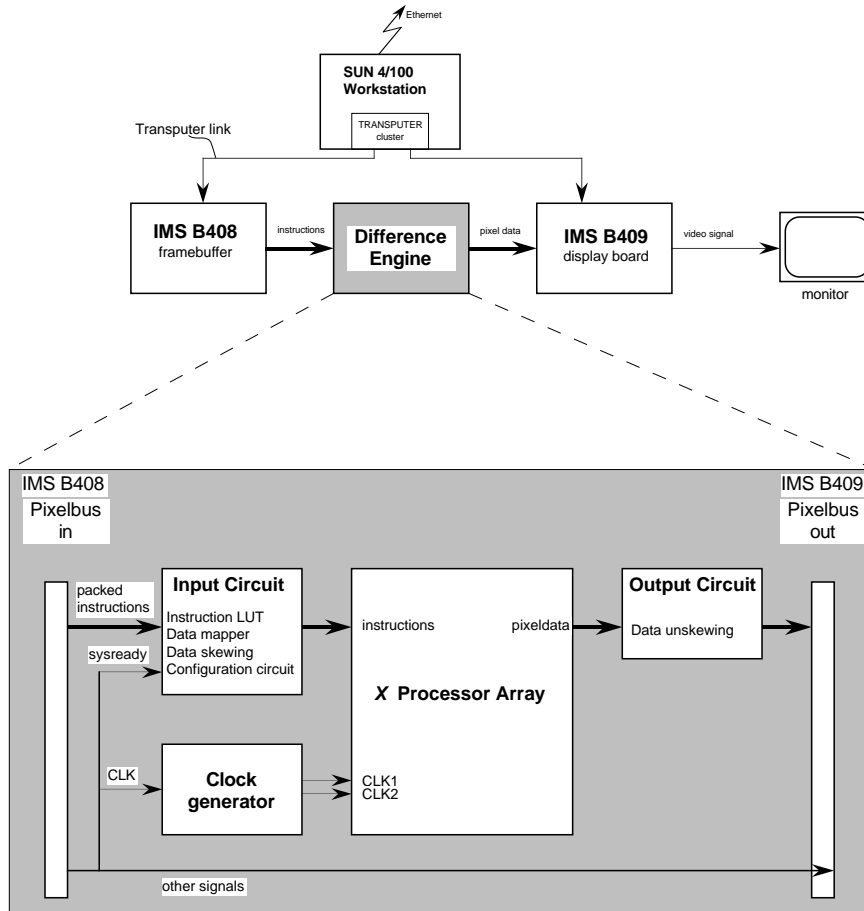


Figure 7.9: Block diagram of the prototype Difference Engine System. A frame store module (IMSB408) is used to store encoded instructions. These instructions are decoded by the Mapper. The x-processor array produces pixel values that are converted into video signals by the display board (IMSB409).

The Host

The SUN workstation serves as a host computer on which program development is done, data is stored and via which the graphics display system communicates with the rest of the world. The connection with the INMOS B408/B409 graphics system modules takes place via a NiCHE NT1000. The NT1000 is a transputer system motherboard which plugs directly into the Sun VMEbus¹³. It offers a range of I/O interfaces, plus sites for up to 32 transputer based modules (TRAMS)¹⁴. The transputer networks on the NT1000 are software configurable [Niche 88].

The INMOS Graphics System

The INMOS graphics system consists of two TRAM modules : the frame buffer (IMSB408) and the display card (IMSB409). These INMOS modules were selected to drive the Difference Engine because they closely met our needs with respect to functionality. The B408 and B409 are meant to be building blocks for a distributed graphics system (see [INMOS 89]). In normal mode of operation these modules are connected via a pixel bus. For our application we break the pixel bus up and write instructions into the frame buffer memory, instead of pixel values. The Difference Engine that connects the two ends of the pixel bus turns these instructions into pixel values.

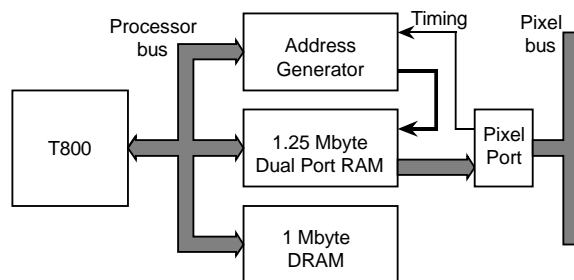


Figure 7.10 : Block diagram of the IMSB408 Frame Store Module. Instructions for the Difference engine are stored in the Dual Port RAM.

The B408 is a frame store TRAM that houses a T800 transputer, 1 Mbyte DRAM, 1.25 Mbyte Dual Port RAM organised in 32 bit words, and an Address Generator (see Figure 7.10). Controlled by signals on the pixel bus, the Address Generator handles the output of data at a rate of up to 100 Mbytes/second. The Dual Port RAM is used to store the instructions to be

¹³) A transputer is a single VLSI device with processor, memory and communication links for direct connection to other transputers.

¹⁴) The TRAM is in effect a board level transputer module with a standardised interface and possibly RAM (see [INMOS 91]).

executed by the Difference Engine. This instruction store is accessible via the T800 transputer. This transputer can either run a program that generates instructions locally, or it can setup a communication link and receive instructions generated by external processors (being the SUN host or other transputers). Data is transferred via a transputer link.

The pixels generated by the Difference Engine are handled by the IMSB409 display board (Figure 7.11) and displayed on a separate monitor. The B409 houses a T222 transputer, a Video Timing Generator —that produces the signals to control the Address Generator of the B408— and three 6 bit DAC's. It accepts data from the pixel bus at a maximum rate of 25 MHz.

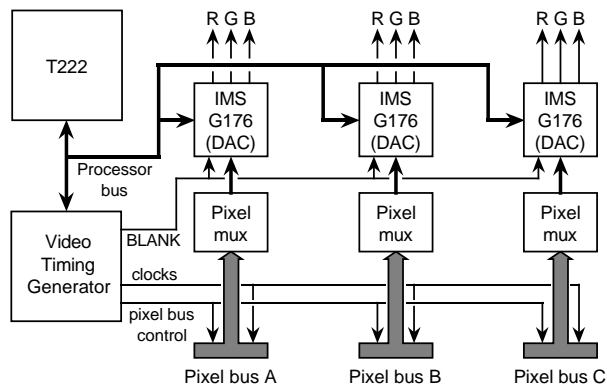


Figure 7.11 : Block diagram of the IMSB409 Display Module. One of the DAC's is used to convert 6 bit pixel values.

The above specifications of the B408/B409 graphics system indicate that a prototype display controller based on these modules will not exploit the full potential of the Difference Engine. The Dual Port RAM of the B408 is organised in 32 bit words and the pixel bus between the INMOS modules is 32 bit wide, whereas the Difference Engine needs a 12 bits instruction identifier and a 36 bits operand. Instructions for the Difference Engine come with two types of operands : one where the operand is one 36 bit wide intensity, and one where the operand is formed by three optional 12 bit items (see Figure 7.12).

Since the pixel bus between the B408 and B409 modules has a maximum clock rate of 25 MHz and since the DAC's have a resolution of 6 bits only, we observe that :

- The resolution of the prototype will have to be less than $1K \times 1K$, so that 10 bits for representing X and dX is sufficient.
- The intensity of the Eval0 instruction can be represented by 6 bits.
- 7 bits (that is 6 + a sign bit) + 2×10 precision bits (needed for second order interpolation) is sufficient for representing intensities that go with all other Eval commands.

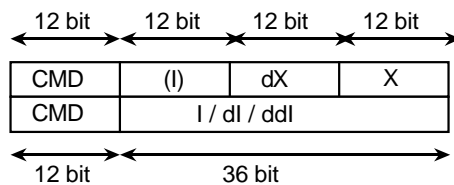


Figure 7.12 : The two types of data formats of the instructions of the Difference Engine.

From this we conclude that all instructions can do with 27 bits for the operand. This leaves us 5 bits for the instruction identifier. The x-processor has 25 "atomic" instructions (see Table 7.2) that make up the 14 different commands indicated in Table 7.1 (including the Nop instruction). Therefore, we can include a look up table between the pixel bus and the x-processor array and use a 5-bit index to refer to the appropriate 12 bit atomic instruction. As a result we find that the limited resolution enforced by the B408 and B409 module specifications complies well with a 32 bit representation of the instructions in the frame store.

Using the B408/B409 graphics system modules we can build a 640×480 pixel 64 grey-level prototype display controller. Such a low resolution prototype does not exploit the full potential of the Difference Engine, yet it still does demonstrate the feasibility of the architecture. Therefore we decided not to invest in the design of better modules.

7.6.2. Board Level Components

In this subsection we will discuss the board-level circuitry of the Difference Engine, that is, the input, the clock, and the output circuit shown in Figure 7.9 (more detailed information can be found in [Steffens95]).

Input Circuit

The elements of the input circuit are shown in a block diagram (Figure 7.13). The input circuit consists of an instruction look up table, a data mapper, a delay circuit and a configuration circuit.

Instruction Look-up Table. Input of the instruction look-up table is 5 bits coded instructions that are stored in the frame store. Output of the instruction look-up table is the 12 bit instruction identifier required by the x-processors and one extra bit, Type, indicating the type of the instruction. This signal is used in the mapper circuit to be able to handle the two different types of data formats that go with the instructions differently.

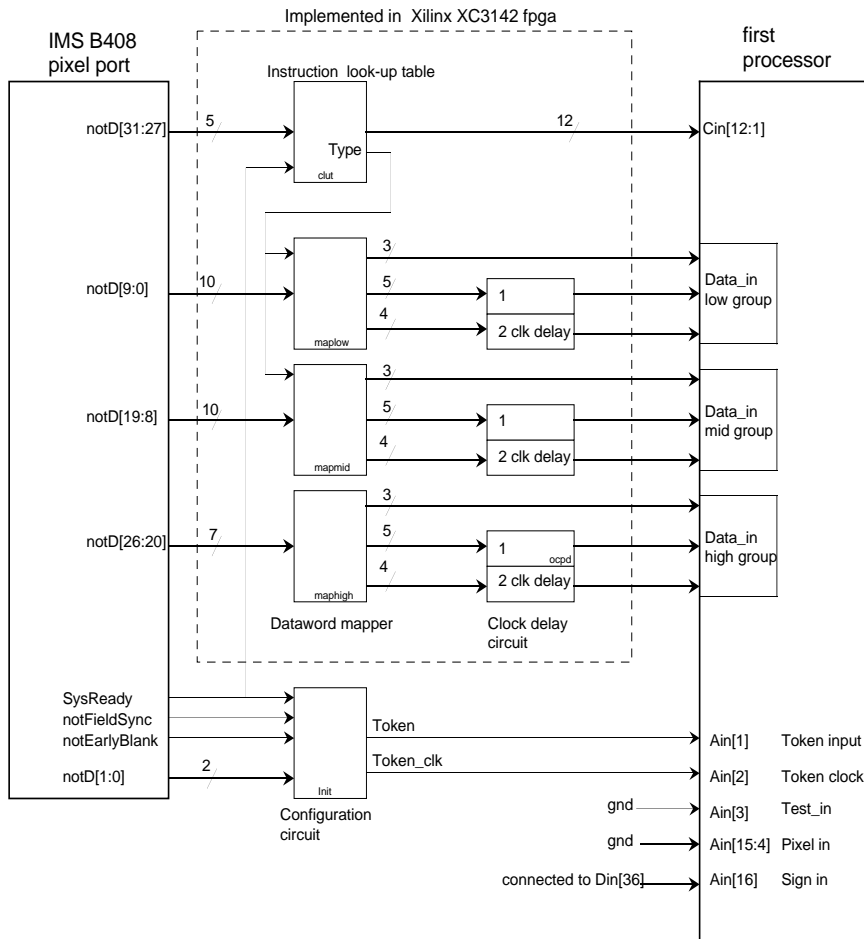


Figure 7.13 : Block diagram of the input circuit. The instruction look-up table, the data mapper, and the delay circuit are implemented in one FPGA.

Data Mapper. The input to the mapper circuit are Type generated by the instruction lookup table and the 27 bits data that go with the 5 bits coded instructions.

For instructions that need 36 bit intensity values, the 27 bits data stored in the frame store module are expanded to 36 bit by expanding the sign bit and padding low bits to zero. Note that we are free to choose which range of 6 bits of the 12 bit pixels that are produced by the Difference Engine will be connected to the DAC —these are not necessarily the 6 least significant bits. This choice will determine the expansion of the sign bit (which therefore may involve 1 to 5 bits) and consequently also the number of least significant bits that are padded to zero (involving 8 to 4 bits).

For the Eval0 instruction — that goes with a 12 bit intensity value — the sign bit of the 7 bit intensity stored in the frame store module (that is supposed to be zero always) is expanded to 1 to 5 bits in the same way as for the 36 bit intensity values. The least significant bits are padded to zero, yet in this case this may involve 4 to 0 bits.

The 10 bit pixel address and 10 bit pixel span, stored in the frame store module, form the 10 least significant bits of the 12 bit pixel address and the 12 bit pixel span. The two most significant bits of the pixel address and pixel span are set to zero.

Delay Circuit. Following the data mapper we find an input delay circuit. This delay circuit serves to delay specific bits of each of the three 12 bit sections by one or two clock cycles. This delay is applied because the x-processors operate on skewed 12 bit data words. By operating on skewed data, the 12 bit adders of the x-processors could be pipelined, so that the x-processors can run at a maximum speed of 11 ns cycle time.

Configuration Circuit. A fully configured prototype Difference Engine consists of 80 IC's housing 9 processors each. For such a large number of processors it is not unlikely that some of them are not fully functional, but are still capable of passing on data. By switching off defective processors, we can still make use of the rest of the processors on the chip. Programming of the array (that is, switching off malfunctioning processors) is done by activating a configuration circuit by means of the "SysReady" signal generated by the B408 module. The configuration circuit then sets the array in programming mode and passes on the data put on the pixel bus. When programming the array, the data put on the pixel bus is data from a "configuration file" stored in a dedicated part of the frame store. In this way the system is fully software configurable.

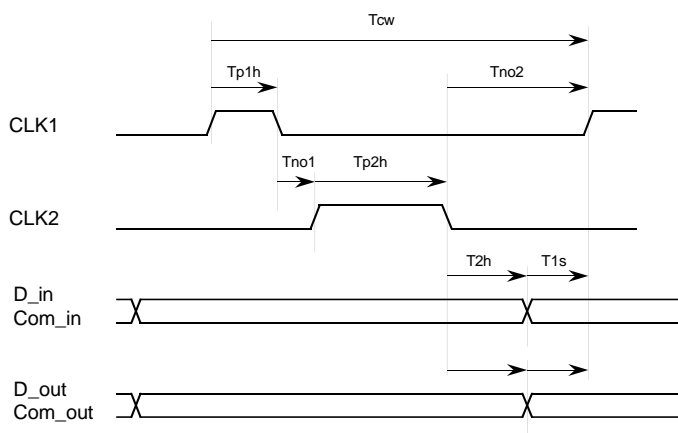


Figure7.14 : Timing diagram of the two phase processor clock and data in/out.

Clock Circuit

The Difference Engine requires a two-phase clock that is, two phase pulses and their inverse. In Figure 7.14 the timing of the clock pulses is shown. At the time of design of the prototype an indication of the required pulse widths and intervals were given as result of measurements in a test circuit and analysis of the chip design. The measurements made clear that timing was critical. It was indicated by the chip designers that the appropriate clock timing could only be established by tuning the clock parameters in a populated (or even partly populated) prototype. As a consequence of this we designed a clock circuit of which the width and phases are fully programmable.

The circuit has been built using Analog Devices AD9500 programmable delay generators (PDG). As shown in Figure 7.15, two such programmable delay generators drive one D-flip-flop (a 10135 JK). One PDG sets the output of the flip-flop after a programmed delay of t_1 , the second PDG resets the flip-flop after a programmed delay of t_2 . This creates a programmed pulse width of $t_2 - t_1$. These two PDG's are triggered by means of a PDG which in turn is triggered by the clock signal of the B408/B409 pixel bus. In this way we can shift the clock pulses relative to the clock of the pixel bus. Two of these circuits produce the fully programmable, two phase clock. Inverse phases are obtained by using the inverse output of the D-flip-flops.

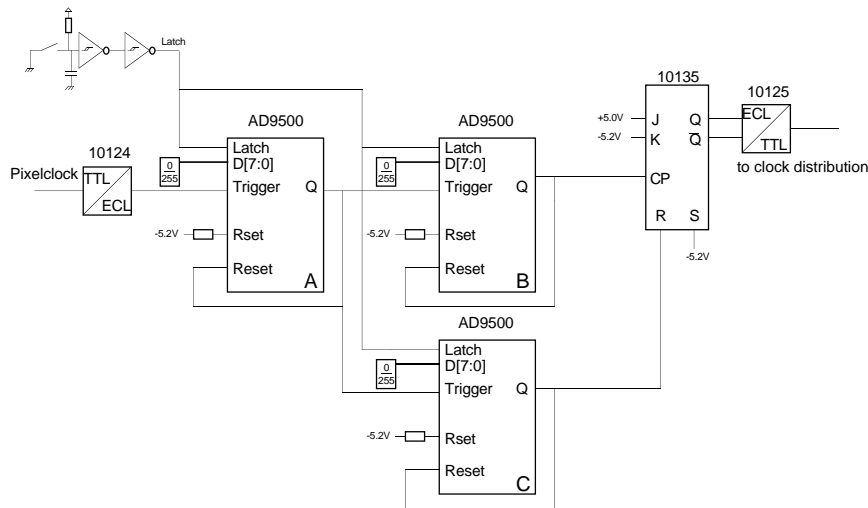


Figure 7.15 : Block diagram of one programmable pulse width clock generator.

The delay of the AD9500 is set by an 8-bit value, which is provided by two hexadecimal rotary switches, and latched manually by pressing a switch. The resolution of the delay can be set between 10 ps and 10 μ s by means of an external resistor and capacitor. With Rset being 150 Ω and Cset not connected we obtain a maximum delay of 76.8 ns with a resolution of 0.3 ns.

The Trigger and Reset of the PDG's expect differential ECL signal levels, so that TTL/ECL conversion and visa versa is necessary. The delay setting however is TTL compatible.

The prototype system is implemented on multi-layer printed circuit boards. Two of the layers serve for distribution of the power (a +5 v and a ground plane). The AD9500 programmable delay generator is an analogue device that turned out to be rather sensitive to even small variations in the power as caused by switching of other digital components. For reasons of stability, the clock circuit required a separated ground plane and +5 V, -5 V power plane. Also, the clock circuit has a separate power supply.

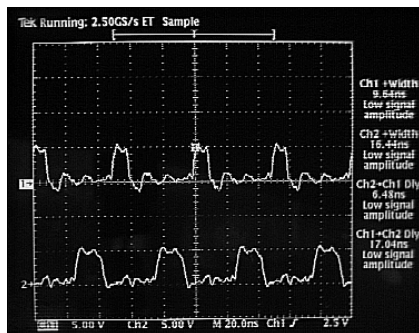


Figure7.16 : A 20 MHz two phase clock as produced by the clock generator.

Output Circuit

A delay circuit was needed to skew certain bits of the input data with one and two clock cycles. Similarly an output circuit is needed to unskew the pixel values that leave the last processor of the processor array (see Figure 7.17). The delay circuit that does this is built using 74AS574 registers. 6 bits of the 12 bit pixels that are produced by the Difference Engine will form the input for the DAC. Selection of these 6 bits is done by a GAL26cv12. The output is buffered by 74AS642 buffers and terminated conform the pixel bus specifications.

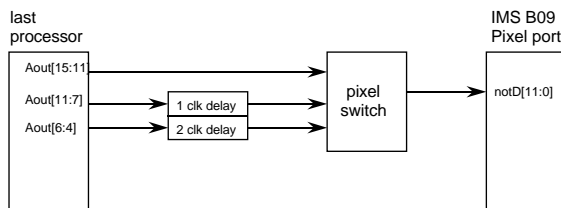


Figure7.17 : Block diagram of the output delay circuit. Pixel data is unskewed and a (fixed) 6 bit range of the 12 bits produced is selected.

7.6.3. Physical Design

From a logical point of view, the system may look quite simple. The board level circuits just described are not complicated, and the x-processor chips simply have to be linked up. The main issue of the hardware implementation is the size of the system: a fully configured system comprises a pipeline of 720 processors. The size of the system is determined primarily by the chip carrier in which the x-processors are housed. This chip carrier is a 144 pin, 15×15 , pin grid array (PGA). Building the pipeline with PGA's mounted flat on a printed circuit board, requires a lot of area. This would imply that a fully configured pipeline would have to be spread out over several boards. This is likely to cause problems, since the x-processor are synchronously clocked. Therefore it is important to reduce the distance covered by the processor pipeline and so make the length of the clock lines to be as short as possible.

Board Layout

A workable solution was found by placing two x-processor IC's on a small multi-layered printed circuit board. A motherboard was designed that contains 2×20 slots to insert these two-chip processor boards. The layout of this multi-layered printed circuit board is shown in Figure 7.18. On the motherboard itself we find the clock generator and clock drivers. A power connector is placed at the rear edge. The data mapper, the configuration circuit, and the input and output delay circuits can be found on a small board (the input/output circuit board) that is inserted in a dedicated slot. Also the two connectors that connect to both ends of the interrupted pixel bus are situated on this board.

The space needed for routing interconnections of the x-processor chips has been kept entirely on the two-chip processor boards. In this way we could minimise the span of the pipeline: the connectors on the motherboard for the two-chip processor boards are placed close to each other, leaving a small separation between the processor boards for the purpose of cooling.

A further reduction of the extent of the processor pipeline has been obtained by breaking up the pipeline in two sections. The clock lines can be found in the area between these two sections. This particular layout reduces the length of the clock lines — and thereby the propagation delay — by half. The total length of each of the four clock lines is 273 mm, which — given the clock rate — can be considered as being rather long. Driving wires of this length at 25 MHz does not have to be a problem as long as the usual termination techniques are applied.

The setup of the system makes it possible to bypass empty slots and run the system with even a partly populated system. This is done simply by means of a flat cable with connectors at both ends. Any of the processor boards can also be replaced by an adapter card that connects up to a logic analyzer. In this way the data stream through the system can be monitored.

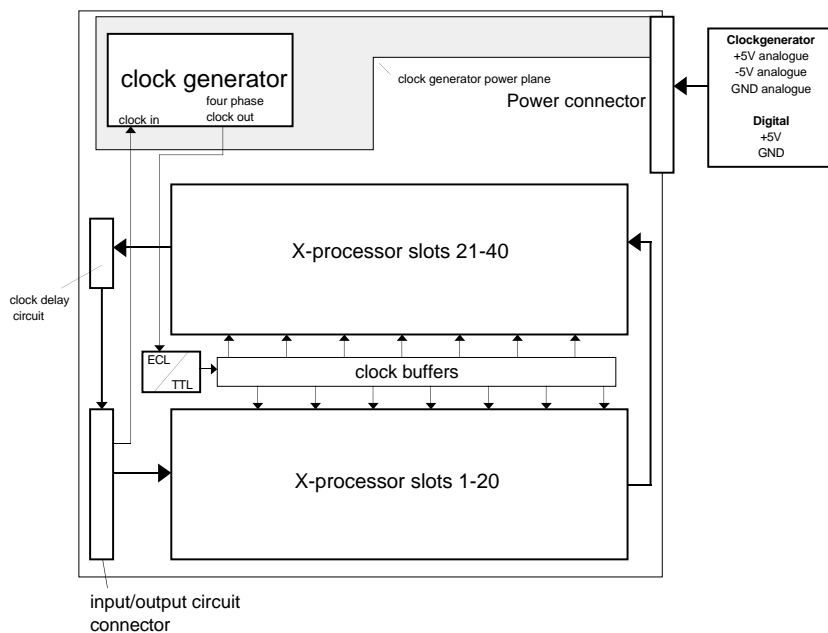


Figure 7.18 : Layout of the mother board of the Difference Engine. The x-processor chips can be found on daughter boards inserted in the slots.

Clock distribution

The clock lines for each two-chip processor board are buffered individually to avoid fan-out problems and to avoid driving too much capacitance, since this would lead to degrading rise and fall times. Regarding the minimum time interval between the two clock phases (min. 4 ns) and the set up times between two x-processors in the pipeline, a skew —i.e., a difference of propagation delay between any two outputs— in the order nanoseconds is unacceptable. Therefore we make use of 49FCT805 clock buffers of which the maximum skew is 0.7 ns. The length of the wires between the buffer and the clock pins of the chips on the processor board ranges from 6.9 to 8.9 inch. All clock lines are terminated by parallel termination resistors.

Each clock buffer chip consists of two banks of drivers (see Figure 7.19). Each bank drives five outputs from one TTL compatible CMOS input, divided over the eight sections of the processor pipeline. The skew between each section due to propagation delay on the clock lines is depended on the length between the points where the signal is connected to the main clock line. For this board the typical delay on a wire is 0.15 ns per inch. The maximum distance between the connections is 1.1 inch which relates to 0.165 ns delay between each section. Such a clock skew is insignificant.

As we can see in Figure 7.18 the clock generator is relatively far away from the processor array. This also implies that we have to deal with relatively long clock lines. For this reason the ECL/TTL converter is positioned at the beginning of the "clock distribution highway" between the two sections of the x-processor slots rather than near the clock generator circuit itself. Naturally the ECLwires are terminated as usual for ECL.

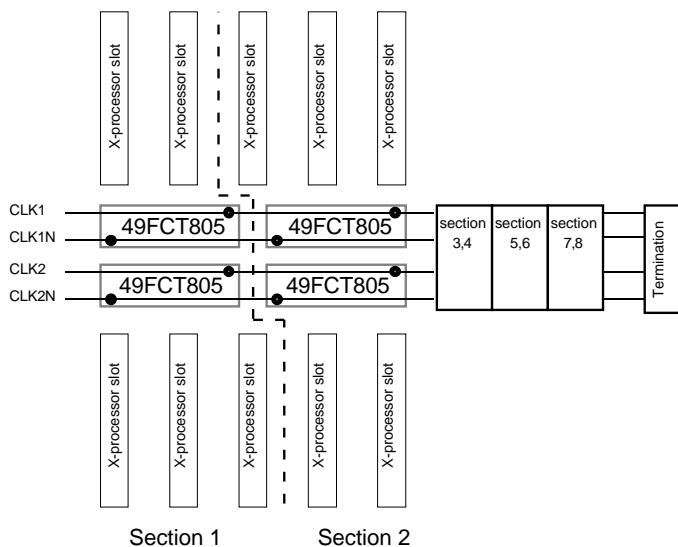


Figure 7.19 : Distribution of the four clock signals. It takes two 49FCT805 clock buffers to drive five slots.

7.7. Results

Several instruction sets in various orders of complexity were used to test the prototype system. The most complex set used is a set of instructions to render a Phong shaded image of the body of the Utah teapot (upside down, as is shown in Figure 7.20 on the left). Also we implemented various utilities to manipulate the content of the instruction buffer for system debugging purposes. The first results on a partly populated system (about 180 processors) made clear that there was a bug in the hardware implementation (Figure 7.20 on the right).

By analysing the data flowing through the processor array we found this to be an error in carry propagation. As it turns out, the carry produced by the 12 bit adder that handles the middle most bits (see § 7.4.3) is propagated to the second least significant bit of the highest 12 bit byte. As a result the forward differencing performed by the array is incorrect (see Table 7.5).

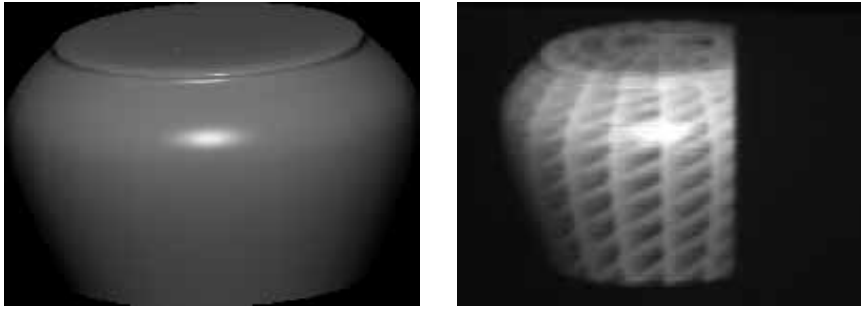


Figure 7.20 : The upside down body of the Utah teapot generated by the simulator (left), and generated by the partly populated Difference Engine (right image has been grabbed from the screen by using a video camera).

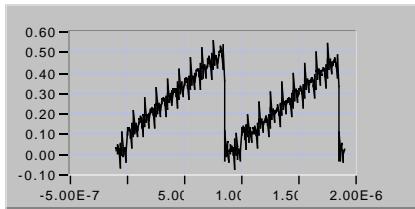
correct		erroneous		compensated	
dl	l	dl	l	dl	l
000.000000	000.000000	000.000000	000.000000	000.000000	000.000000
000.800000	000.000000	000.800000	000.000000	000.400000	000.000000
001.000000	000.800000	002.000000	000.800000	000.800000	000.400000
001.800000	001.800000	002.800000	002.800000	000.c00000	000.c00000
002.000000	003.000000	004.000000	006.000000	002.000000	002.800000
002.800000	005.000000	004.800000	00a.000000	002.400000	004.800000
003.000000	007.800000	006.000000	00e.800000	002.800000	006.c00000
003.800000	00a.800000	006.800000	014.800000	002.c00000	00a.400000
004.000000	00e.000000	008.000000	01c.000000	004.000000	00e.000000

Table 7.5 : Example of second order forward differencing of 36 bit numbers as intended (two leftmost columns). A bug in carry propagation makes the array produce the result shown in the two columns in the middle. A compensation for this error leads to the result shown in the two rightmost columns.

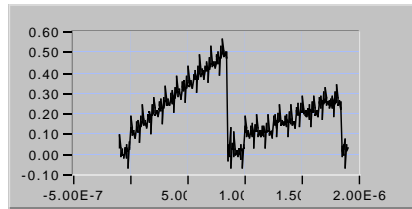
This error in carry propagation can largely be compensated by down shifting the 24 least significant bits one position ¹⁵⁾. In case the difference is negative, the most significant of the 24 least significant bits has to be set to 1. In doing so, the resulting 12 bit intensity is almost correct, except for the least significant bit (as is shown in Table 7.5).

The Difference Engine produces 12 bit intensities, yet the prototype has 6 bits DAC's. The range of 6 bits used can be chosen arbitrarily — provided that the mapping on the input side corresponds to this choice (see § 7.6.2).

¹⁵⁾ To which bits of the data in the B408's frame store this applies depends on how the 27 bit fixed point representation maps onto the 36 bits of the Difference Engine.

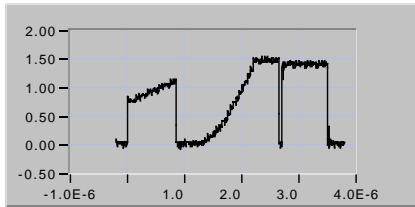


(a)

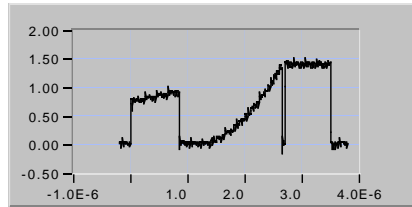


(b)

Eval2 (1, 17, 0.0625, 0.015625);
Eval2 (21, 17, 0.0625, 0.0078125);

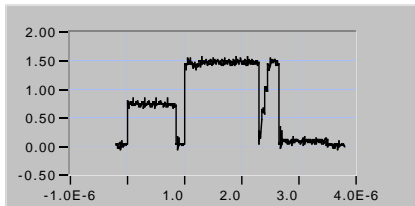


(c)

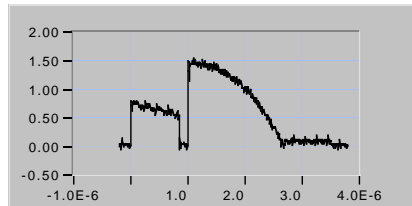


(d)

Eval2 (1, 17, 0.5, 0.0078125);
Eval3 (21, 33, 0.0, 0.0, 0.001953125);
Eval1 (55, 16, 0.9375.0);



(e)



(f)

Eval2 (1, 17, 0.5, -0.0078125);
Eval3 (21, 33, 0.984375, 0.0, -0.001953125);
Eval1 (55, 16, 0.046875);

Figure 7.21 : Measured video signals (using a digital oscilloscope) and the commands used to generate these signals. In this case the DAC is connected to the 6 least significant bits of the 12 bits that are produced by the Difference Engine. The parameters of the commands are : Eval1(x, dx, I); Eval2(x, dx, I, dI); Eval3(x, dx, I, dI, ddI) — see also Table 7.1. Graphs (a), (c) and (e) show the result of unmodified data, whereas (b), (d) and (f) show the result of compensation for the carry bug.

Due to this, the error in the least significant bit of the 12 bits produced that (vaguely) shows up in Figure 7.21 can be made invisible by not using the least significant bit. We therefore use the 6 middlemost bits of the 12 bits that are produced (see the result shown in Figure 7.22).

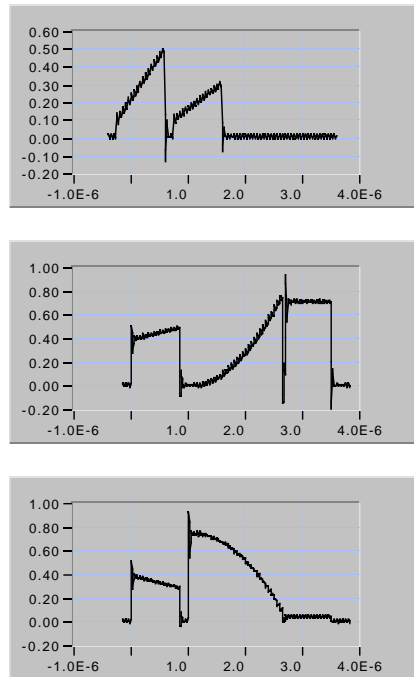


Figure 7.22 : Measured video signals showing the result of compensation for the carry bug. In this case the DAC is connected to the 6 middlemost bits of the 12 bits that are produced by the Difference Engine. As a result, the error in the least significant bit of the 12 bits produced does no longer show up.

In this way we produced the images shown in Figure 7.23. These images are mostly correct, yet some of the data appeared to be lost (i.e., data was not clocked in properly). In time this got worse (Figure 7.23 on the right), which indicated a temperature dependence of the input circuitry. This problem could be solved by a redesign of the motherboard. The clock circuit was better isolated electromagnetically by separating both the ground and power plane from the rest of the system. Furthermore the gap between the processor boards was increased to improve the cooling capacity of the system.

We verified that the modifications were successful and configured the system with 640 processors. In Figure 7.24 we see the result of this. The graphs (a) and (b) show two scanlines of the image of the teapot body before modification (generated by a partly populated system), the graphs (c) and (d) show (other) scanlines of the same image after the modification.

November 24, 1995

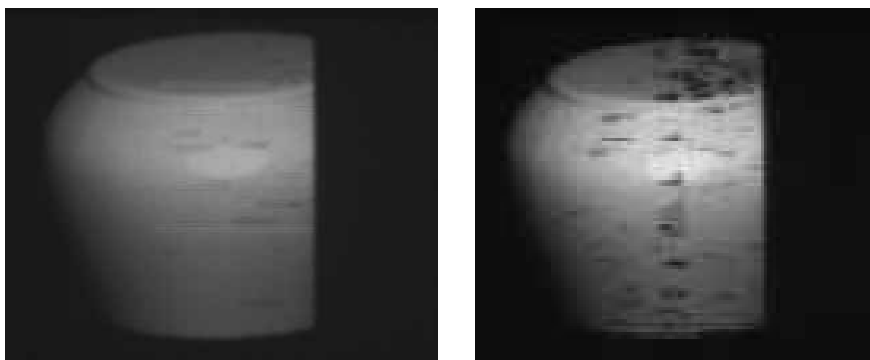


Figure 7.23 : Images grabbed from the screen showing the result of compensation for the carry bug. The image on the right was captured several hours later, indicating a temperature drift.

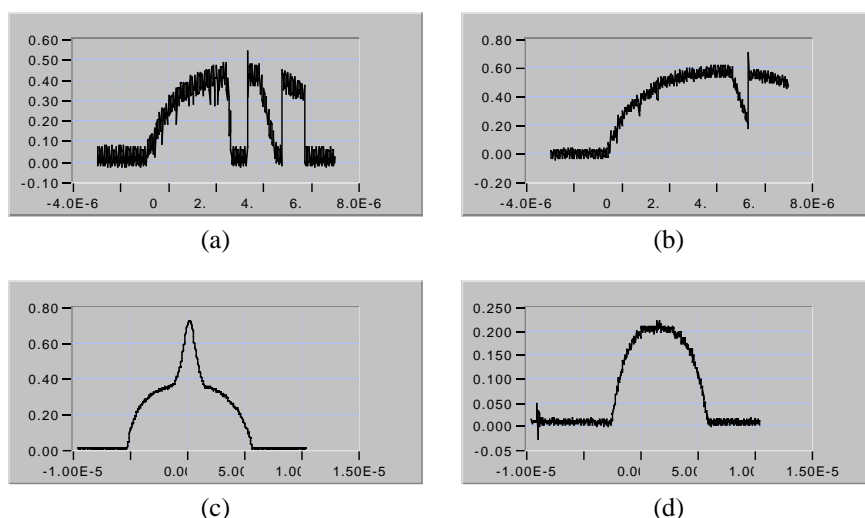
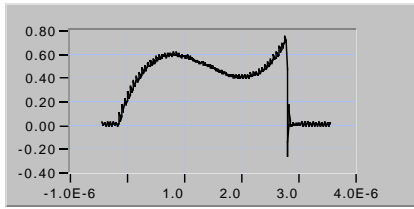


Figure 7.24 : Scanlines grabbed from the image of the teapot body before modification of the motherboard (a and b) and generated with a partly populated system (about 180 processors). Graphs (c and d) show the result after modification of the motherboard (with 640 processors installed).

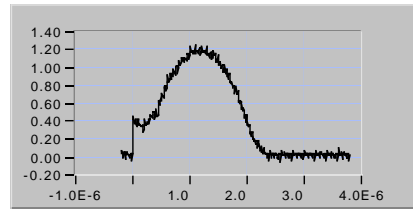
More results are shown in Figure 7.25 : a cubic spline that is generated by one single command (a), a quadratic spline with two set instructions (b), a combination of a linear and quadratic spline (c), the combination of a regular pattern and a linear interpolation (d), and the different results obtained with accumulate mode off (e) and accumulate mode on (f). In Figure 7.26 we show the resulting test image grabbed from the screen by means of a video camera.

November 24, 1995



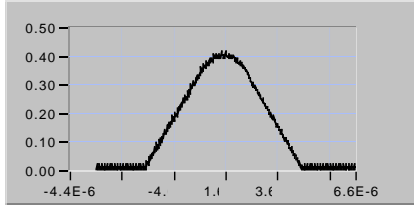
Eval4 (1, 52, 0.0, 0.18,-0.0127, 0.000427);

(a)



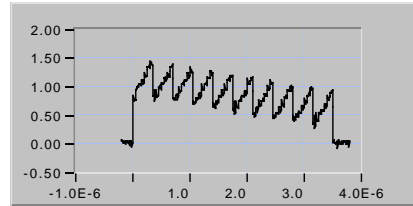
Setddi (10, 1,-0.003906);
Setddi (40, 1, 0.0078125);
Eval3 (1, 48, 0.25,-0.005859, 0.0078125);

(b)



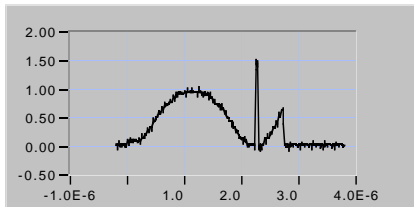
Eval2 (2, 40, 0.0, 0.01);
Eval3 (42, 41, 0.4, 0.01,-0.0005);
Eval2 (83, 40, 0.4, -0.01);

(c)



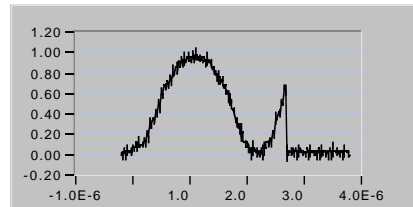
Setpi (8, 7, 0.0);
Eval2 (1, 100, 0.0, 0.0703);
Eval2 (1, 100, 0.59375,-0.00586);

(d)



Setddi (9, 1,-0.0078125);
Setddi (38, 1, 0.008789);
Eval3 (1, 55, 0.0957, 0, 0.008789);

(e)



Acc_mode ();
Setddi (9, 1,-0.0078125);
Setddi (38, 1, 0.008789);
Eval3 (1, 55, 0.0957, 0, 0.008789);

(f)

Figure7.25 : Measured results of several instruction sets. A description of the operation and the operands that go with the commands used can be found in Table 7.1.



Figure 7.26 : Test image generated by the Difference Engine (grabbed from the screen by using a video camera).

7.7.1. Displaying Animated Sequences

The prototype object-space display controller was build to demonstrate the feasibility of the architecture. In this prototype system, the instructions for the Difference Engine are stored in a buffer rather than being generated real-time. In this way, the system is able to display even complex static images, that is, images that require a substantial amount of instructions per scanline.

Given the limited capacity of the 25 MHz T800 transputer that drives the Difference Engine, it should be clear that building up all instructions for a complex image takes some time ¹⁶⁾. However, when building up instructions involves a few bytes per scanline, or if we can do with changes that involve a few bytes per scanline, frame rates of about 50 Hz can be obtained.

We have some animated sequences that cover a large portion of the screen area, yet the changes between successive frames involve a few instructions per scanline only (see the example shown in Figure 7.27). The fact that we can generate these animation sequences real-time demonstrates the result of one of the main objectives of the architecture : due to the object-space approach we can make incremental changes effectively.

The 1.25 Mbyte Dual Port Memory in which the instructions are stored is not big enough for double buffering. Synchronisation of frame updates with the vertical sync is possible, yet for most of the animated sequences we have, the time needed to update the instructions is more than the vertical retrace time. As a result, when using this prototype system for animated sequences, partly updated frames may be displayed.

¹⁶⁾ Note that in this prototype system the transputer that drives the Difference Engine is the bottleneck. The Difference Engine itself can at all times handle all instructions stored in the buffer.

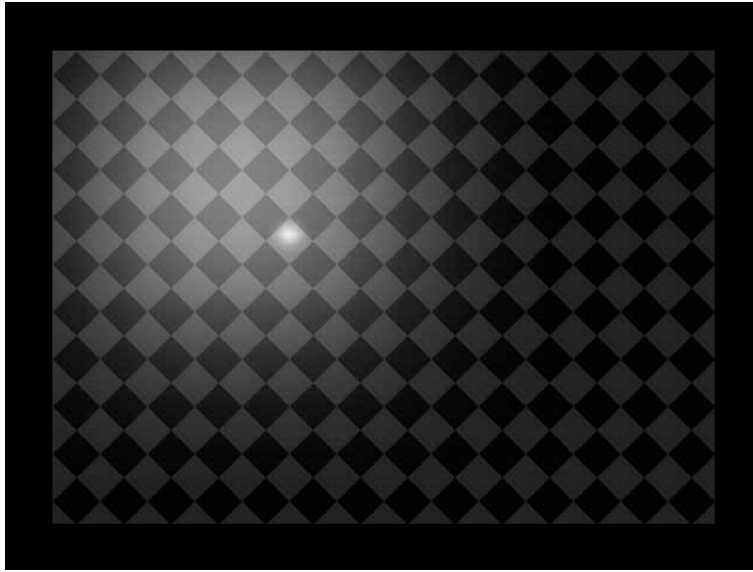


Figure 7.27 : Frame from a real-time animation running on the prototype architecture (simulated). It shows one (!) pattern that is textured and illuminated by a nearby light source (that is not visible). In the animation, this light source moves continuously towards randomly chosen positions.

7.7.2. Reconstruction of Encoded Signals

The Difference Engine can interpolate any spline (polynomial) curve. Thus any signal that is expressed in terms of a spline basis can be reconstructed. Not only that, the architecture with its accumulator allows one to sum over incrementally generated output so that the splines of a multiresolution representation can be summed over different scales to produce the final image to any required accuracy. The reconstruction time depends not on the spacing of the knots in the splines (the lengths of the interpolation spans) but only on the number of knots. As a result, an image can be decompressed even at video rates provided that the number of knots are less than the number of pixels to be generated (by some fixed overhead per scanline).

This has opened the way for using the Difference Engine to reconstruct images that have been coded with a *wavelet transform*. The wavelet transform produces a multiresolution description of an image that can be decoded to yield more and more accurate reconstructions of that image. It is argued that wavelet transforms perform better than the discrete cosine transform advocated by the JPEG standard, it fits in better with human perceptual aptitudes and is a more compact coding.

For a detailed discussion of the use of this system for wavelet reconstruction see [Marais 94].



Figure 7.28 : Reconstruction of a wavelet encoded image (simulated). The multi resolution description is transformed directly into instructions for the Difference Engine.

7.8. Conclusions

The display controller presented in this chapter has a two-phased structure that makes it possible to independently scale two important characteristics of the system. The number of y-processors can be selected to meet the requirements on the maximum image complexity. The number of processing elements in the Difference Engine can be scaled to accommodate different resolutions (up to 4 Kpixels per scanline).

The Difference Engine is “user-friendly” in that it accepts sets of related instructions in arbitrary order. The time to process instructions is independent of the length of the interpolation span and results of different interpolations can be accumulated. The number of instructions per scanline that can be handled is related to the number of pixels that have to be generated — so, the higher the resolution of the system, the more instructions can be handled, hence, the more complex the image can be.

We note that the input bandwidth of the Difference Engine is limited. Yet, also the output bandwidth is limited — simply because that relates to the limited number of pixels on the screen. By selecting the appropriate instructions (on the one end of the spectrum we can opt for accumulation of high order interpolation, on the other end we can opt for setting individual pixels) we can always minimise the input stream such that the number of instructions needed is less than what can be input. Furthermore the input capacity can even be increased as is shown in Appendix B.

The designed maximum throughput rate of the system is sufficient to support high resolution systems. Due to the (potential) high clock rate and the massive parallelism, the Difference Engine is a very powerful machine. We have demonstrated that a 13.8 GOP system can be built that drives a 640×480 pixel display. It can handle the rasterisation of ~ 330 K high quality shaded polygons per second. The same architecture can be used for high resolution systems (1280×1024). In that case the performance would be as high as 108 GOPS (85 MHz pixel rate \times 1280 processors) and the rasterisation of ~ 1.5 M high quality shaded polygons per second can be handled.

In spite of the massive parallelism and the 1.2 micron technology used, we needed 'only' 72 chips for a 640×485 display. This number can be further reduced by using better design tools and better technologies. Then it is possible to fit more processing elements on the same die with even a slight improvement in speed.

In a multi-disciplined project like this, the different "worlds" of application programmers and technicians was foreseen. It was somewhat naively expected that VLSI designers would — by nature — at least anticipate on the problems board-level system designers have to solve. Unfortunately, the selected chip carrier, the pin assignment, and the extreme demands imposed on the board-level design by the complex two phase bipolar clock signals required by the processor array suggest otherwise.

The development of software simulators was done in parallel with hardware specification and not prior to such specification. This reduced the delay in getting feedback on the feasibility implementing ideas in hardware from our VLSI designers but meant that proper simulation of the system was occasionally neglected. One useful purpose of the structural simulator discussed in § 7.4.4 was that it served as a communication tool between the architecture designers/specifiers at the CWI and the VLSI design/implementation team. Some bugs showed up in the hardware implementation. These were not related to improper specification. It appeared that only the individual modules of the design were checked, not the fully integrated design. The bugs found all related to module interconnections.

By exploiting massive parallelism and RISC-like technology, we designed a multiprocessor display controller for real-time refresh from the low level representation defined in Chapter 4. A system based on such a display controller is as responsive as can be for interactions taking place at this level. Since the frame representation at this level consists of a structured list of patterns, it is both very compact as well as well suited for partial updates. As a result, the access requirements of the frame store memory are reduced considerably. The real-time refresh process includes advanced scan-conversion. Complex shading methods as well as regular patterns and multiple light sources are efficiently supported.

Although designed for computer synthesised images, the Difference Engine turned out to be well suited for high speed image reconstruction. Without going into a contemplation of the meta-levels of the design process it is interesting to observe that this generality of application resulted from bottom-up design. The initial top-down design produced an architecture for raster graphics (only). The bottom-up design that followed concentrated on extracting the lowest common denominator of primitive operations for synthesising pixels — a language for manipulating related pixels. This vocabulary can now be used for expressing other facts about images. This may lead to one integrated system for image reconstruction and image synthesis (see Figure 7.29).

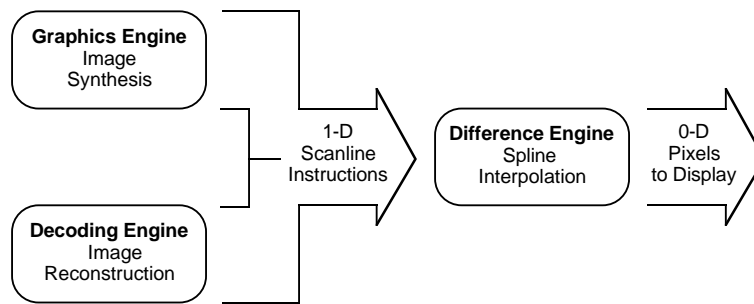


Figure 7.29 : The Difference Engine can be used as rasteriser for image synthesis as well as for image reconstruction, even at the same time.

Conclusions

Synopsis.

This dissertation is on the development of an interactive three-dimensional raster graphics system for CAD-type of applications. The aim of the research described in this dissertation has been to develop a new architecture for an interactive graphics workstation, targeted for displaying three-dimensional polygonal models. We concentrated on reducing the amount of calculation involved in interaction tasks and concluded that the amount of calculation can be kept low by providing exactly that functionality required for interaction. We argued that a layered system and a consistent raster independent object-space approach allows us to make optimal use of incremental algorithms. A consequence of this approach is that we have to make use of nonstandard image generation algorithms and nonstandard image generation hardware.

Computer graphics involves computing tasks of vastly differing order of granularity and complexity. These tasks relate to operations on frames, objects, patches, polygons, vertices and pixels. For specific tasks — such as geometric transformations — a dedicated hardware configuration on which the algorithm can be mapped appropriately can be found in a confined design space. Yet, for the entire image generating pipeline many and various configurations should be considered. Consequently, we may find implementations of graphics systems based on almost all possible task subdivision strategies that can be conceived.

Differing implementation strategies result in systems with differing characteristics. Normally a system is designed for a targeted type of rendering and for a targeted class of images. However, in computer graphics we find several types of graphics primitives, hidden surface removal algorithms, shading methods and image processing techniques. This causes a great variety in intrinsic characteristics of tasks involved in image generation. In addition, the characteristics of interactive applications vary considerably, even at run-time. As a result some systems available today may excel in certain types of applications, and yet be mediocre in others: they are too specific and lack the flexibility required for high speed interaction.

Changing pictures form the key to the architecture proposed here. In an interactive computer graphics application a user interacts with a three-dimensional model at several levels of abstraction. For an efficient support of interactive editing and incremental updates a raster independent representation of the three-dimensional model should be immediately available at each of these levels. Fundamental actions that underly every change a user makes are pointing and identification. Actual pixels are not needed for interaction.

By taking these observations seriously and ruthlessly pare away other elements we get a radical prescription for a graphics architecture. One where all visible surfaces are identified explicitly and without any mandate for a pixel store. We argue that such a geometry based, raster independent pipeline organisation leads to a lean, yet flexible, computation model.

As a consequence of the raster independent object-space paradigm, the visibility calculation —that is clipping and hidden surface removal — takes place in object-space. The transition from object-space to image-space (i.e., rasterisation) is included in the real-time refresh process. The feasibility of the approach is demonstrated by the design and implementation of an incremental object-space hidden surface removal algorithm, by the formulation of a shading method optimised for fast rasterisation, and by the design and implementation of an object-space display controller.

Our architecture uses a pre-sorted representation of objects and a geometry-based data structure to store these objects. The specific representation of objects reduces the complexity of both the hidden surface removal and the scan-conversion process. The data structure used to store objects is designed to reduce the search space for geometry-based object identification. The hidden surface removal algorithm presented in this dissertation includes a set of binary operations on projected objects. These operations can be used to add and delete individual objects so that incremental changes affect only those objects of which the visibility is changed. Thus a firm basis for powerful interaction and animation is established.

We formulated a shading method, much like Phong shading, that is optimised for high speed rasterisation. This 'reformulated Phong' shading method interprets interpolation of vectors involved in the calculation as rotations. Spherical trigonometry leads to a linear expression of the angle between the vectors along a scanline. The outcome is a parameterised piecewise quadratic expression for each intensity term. These terms can be handled directly by forward differencing. The image quality obtained is as good as the quality of Phong shaded images.

A display controller handles the refresh process of a graphics display device. The display controller in the object-space based architecture proposed here reads 'area drawing instructions' collected in a structured list and produces scanline based video signals. We designed a two-phased display controller architecture with two differing classes of processing elements. The

actual rasterisation —i.e., the second phase of the scan-conversion process — involves forward differencing. This goes with relatively simple operations repeated numerous times. For this task we opted for a high speed 36 bit forward differencing engine, implemented as a highly pipelined systolic array.

The implementation of an incremental object-space hidden surface removal algorithm and a prototype object-space display controller demonstrates the feasibility of the raster independent object-space approach.

We found that the cost of object-space hidden surface removal (based on a non-optimised version of the algorithm) is in the same order of magnitude as the cost of rendering "Phong-like" shaded objects. Then, for sufficient complex scenes (that is, in terms of depth complexity and number of light sources), the current version of the object-space hidden surface removal algorithm can already be cost effective. The superiority for partial updates should be clear. However, optimisation techniques should be investigated and research in specialised object-space hidden surface removal hardware in particular is needed to be able to reach the level of performance of the current generation specialised rasterisation hardware.

For the design of the object-space display controller we concentrated on extracting the lowest common denominator of primitive operations for synthesising pixels. This led to a vocabulary that can be used for expressing facts about images in a most general way. The generality of application of the architecture is demonstrated by using the prototype display controller to reconstruct images that have been coded with a wavelet transform. This led to new ideas on functionality and further optimisation for the next generation of rasterisation hardware that, given the interest in multi-media applications, seems to be worth investigating.

January 10, 1996

Acknowledgements

Many people have been involved in this work in one way or another. First of all I would like to thank Paul ten Hagen for introducing me to the subject and backing me all this time. My (ex) colleagues Toos Trienekens, Varol Akman, Behr de Ruiter, Vincent Disselkoen, Michael Guravage and Tanja van Rij contributed to the research in various ways. Of the VLSI design team at the University of Twente I thank Jaap Smit and in particular Kapila Jayasinghe for his enthusiasm and discernment. I thank prof. L.O. Herzberger (University of Amsterdam) for providing the facilities and manpower to help build the prototype. Edwin Steffens did a superb job in building the prototype display controller. Also I thank STW, and Jean Pierre Veen in particular, for supporting this project. Furthermore I acknowledge Patric Marais (guest researcher from the University of Capetown) for his work with respect to wavelet-based image reconstruction. Naturally I have to thank Edwin Blake for being an inspiring colleague, for never failing to believe, and for his many useful suggestions on this dissertation.

Finally I thank my wife and children. They will be happy to find that I just ran out of excuses.

January 10, 1996