Sheffield
Hallam
University

Efficient implementations of machine vision algorithms using a dynamically typed programming language

WEDEKIND, Jan

Available from Sheffield Hallam University Research Archive (SHURA) at:

http://shura.shu.ac.uk/6633/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version

WEDEKIND, Jan (2012). Efficient implementations of machine vision algorithms using a dynamically typed programming language. Doctoral, Sheffield Hallam University.

Repository use policy

Copyright © and Moral Rights for the papers on this site are retained by the individual authors and/or other copyright owners. Users may download and/or print one copy of any article(s) in SHURA to facilitate their private study or for non-commercial research. You may not engage in further distribution of the material or use it for any profit-making activities or any commercial gain.

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

Sheffield
Hallam
University

Efficient implementations of machine vision algorithms using a dynamically typed programming language

WEDEKIND, Jan

Available from Sheffield Hallam University Research Archive (SHURA) at:

http://shura.shu.ac.uk/6633/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version

WEDEKIND, Jan (2012). Efficient implementations of machine vision algorithms using a dynamically typed programming language. Doctoral, Sheffield Hallam University.

Repository use policy

Copyright © and Moral Rights for the papers on this site are retained by the individual authors and/or other copyright owners. Users may download and/or print one copy of any article(s) in SHURA to facilitate their private study or for non-commercial research. You may not engage in further distribution of the material or use it for any profit-making activities or any commercial gain.

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

Efficient Implementations of Machine Vision Algorithms using a Dynamically Typed Programming Language

Jan Wedekind

A thesis submitted in partial fulfilment of the
requirements of
Sheffield Hallam University
for the degree of Doctor of Philosophy

February 2012

Abstract

Current machine vision systems (or at least their performance critical parts) are predominantly implemented using statically typed programming languages such as C, C++, or Java. Statically typed languages however are unsuitable for development and maintenance of large scale systems.

When choosing a programming language, dynamically typed languages are usually not considered due to their lack of support for high-performance array operations. This thesis presents efficient implementations of machine vision algorithms with the (dynamically typed) Ruby programming language. The Ruby programming language was used, because it has the best support for meta-programming among the currently popular programming languages. Although the Ruby programming language was used, the approach presented in this thesis could be applied to any programming language which has equal or stronger support for meta-programming (*e.g.* Racket (former PLT Scheme)).

A Ruby library for performing I/O and array operations was developed as part of this thesis. It is demonstrated how the library facilitates concise implementations of machine vision algorithms commonly used in industrial automation. That is, this thesis is about a different way of implementing machine vision systems. The work could be applied to prototype and in some cases implement machine vision systems in industrial automation and robotics.

The development of real-time machine vision software is facilitated as follows

1. A just-in-time compiler is used to achieve real-time performance. It is demonstrated that the Ruby syntax is sufficient to integrate the just-in-time compiler transparently.

2. Various I/O devices are integrated for seamless acquisition, display, and storage of video and audio data.

In combination these two developments preserve the expressiveness of the Ruby programming language while providing good run-time performance of the resulting implementation.

To validate this approach, the performance of different operations is compared with the performance of equivalent C/C++ programs.

Publications

Refereed Journal Articles

- M. Boissenin, J. Wedekind, A. N. Selvan, B. P. Amavasai, F. Caparrelli, and J. R. Travis. Computer vision methods for optical microscopes. *Image and Vision Computing*, 25(7):1107–16, 07/01 2007 (Boissenin et al., 2007)

- A. J. Lockwood, J. Wedekind, R. S. Gay, M. S. Bobji, B. P. Amavasai, M. Howarth, G. Möbus, and B. J. Inkson. Advanced transmission electron microscope triboprobe with automated closed-loop nanopositioning. *Measurement Science and Technology*, 21(7):075901, 2010 (Lockwood et al., 2010)

Refereed Conference Publications

- Jan Wedekind, Manuel Boissenin, Balasundram P. Amavasai, Fabio Caparrelli, and Jon R. Travis. Object Recognition and Real-Time Tracking in Microscope Imaging. Proceedings of the 2006 Irish Machine Vision and Image Processing Conference (IMVIP 2006), pages 164–171, Dublin City University, 2006. (Wedekind et al., 2006)

- J. Wedekind, B. P. Amavasai, and K. Dutton. Steerable filters generated with the hypercomplex dual-tree wavelet transform. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 1291–4, Piscataway, NJ, USA, 24–27 Nov. 2007 2008. Mater. & Eng. Res. Inst., Sheffield Hallam Univ., Sheffield, UK, IEEE (Wedekind et al., a)

- J. Wedekind, B. P. Amavasai, K. Dutton, and M. Boissenin. A machine vision extension for the Ruby programming language. In *2008 International Conference on Information and Automation (ICIA)*, pages 991–6, Piscataway, NJ, USA, 20–23 June 2008 2008. Microsyst. & Machine Vision Lab., Sheffield Hallam Univ., Sheffield, UK, IEEE (Wedekind et al., b)

Formal Presentations

- Jan Wedekind. Real-time Computer Vision with Ruby. *O'Reilly Open Source Convention (OSCON)*, Portland, Oregon, USA, July 23rd 2008 (Wedekind, 2008)

- Jan Wedekind. Computer Vision Using Ruby and libJIT. *(RubyConf)*, San Francisco, California, USA, Nov. 19th 2009 (Wedekind, 2009)

- Jan Wedekind, Jacques Penders, Hussein Abdul-Rahman, Martin Howarth, Ken Dutton, and Aiden Lockwood. Implementing Machine Vision Systems with a Dynamically Typed Language. *25th European Conference on Object-Oriented Programming*, Lancaster, United Kingdom, July 28th 2011 (Wedekind et al., 2011)

Published Software Packages

- malloc[1]
- multiarray[2]
- hornetseye-alsa[3]
- hornetseye-dc1394[4]
- hornetseye-ffmpeg[5]
- hornetseye-fftw3[6]
- hornetseye-frame[7]
- hornetseye-kinect[8]
- hornetseye-linalg[9]
- hornetseye-narray[10]
- hornetseye-opencv[11]
- hornetseye-openexr[12]
- hornetseye-qt4[13]
- hornetseye-rmagick[14]

[1] http://github.com/wedesoft/malloc/
[2] http://github.com/wedesoft/multiarray/
[3] http://github.com/wedesoft/hornetseye-alsa/
[4] http://github.com/wedesoft/hornetseye-dc1394/
[5] http://github.com/wedesoft/hornetseye-ffmpeg/
[6] http://github.com/wedesoft/hornetseye-fftw3/
[7] http://github.com/wedesoft/hornetseye-frame/
[8] http://github.com/wedesoft/hornetseye-kinect/
[9] http://github.com/wedesoft/hornetseye-linalg/
[10] http://github.com/wedesoft/hornetseye-narray/
[11] http://github.com/wedesoft/hornetseye-opencv/
[12] http://github.com/wedesoft/hornetseye-openexr/
[13] http://github.com/wedesoft/hornetseye-qt4/
[14] http://github.com/wedesoft/hornetseye-rmagick/

- hornetseye-v4l[15]
- hornetseye-v4l2[16]
- hornetseye-xorg[17]

[15] http://github.com/wedesoft/hornetseye-v4l/
[16] http://github.com/wedesoft/hornetseye-v4l2/
[17] http://github.com/wedesoft/hornetseye-xorg/

Acknowledgements

First I would like to thank Bala Amavasai for his supervision, support, and his unshakable optimism. He developed a large part of the Mimas C++ computer vision library and organised the Nanorobotics grant. Without him I would not have been able to do this work. I am also very indebted to Jon Travis who has been a valuable source of help and advice when coming to the UK and while working at university.

I would also like to thank Ken Dutton, Jacques Penders, and Martin Howarth for continuing supervision of the PhD, for their advice and support and for giving me room to do research work.

I would also like to thank Arul Nirai Selvan, Manuel Boissenin, Kim Chuan Lim, Kang Song Tan, Amir Othman, Stephen, Shuja Ahmed and others for being good colleagues and for creating a friendly working environment. In particular I would like to express my gratitude to Georgios Chliveros for his advice and moral support.

Thanks to Julien Faucher who introduced 3D camera calibration to the research group.

A special thanks to Koichi Sasada for his research visit and for the many interesting and motivating discussions.

Thanks to Aiden Lockwood, Jing Jing Wang, Ralph Gay, Xiaojing Xu, Zineb Saghi, Günter Möbus, and Beverly Inkson for their valuable help in applying the work to transmission electron microscopy in context of the Nanorobotics project.

Finally I would like to thank my parents who sacrificed a lot so that I can achieve the best in my life. Without their support I would not have made it this far.

A seven year part-time PhD is a long time to work and make friends. My apologies but there is just not enough room to mention you all.

The work presented in this thesis was partially funded by the EPSRC Nanorobotics[18] project. I also received a student bursary of the Materials and Engineering Research Institute.

[18] http://gow.epsrc.ac.uk/ViewGrant.aspx?GrantRef=GR/S85696/01

Declaration

Sheffield Hallam University
Materials and Engineering Research Institute
Mobile Machines and Vision Laboratory

The undersigned hereby certify that they have read and recommend to the Faculty of Arts, Computing, Engineering and Sciences for acceptance a thesis entitled **"Efficient Implementations of Machine Vision Algorithms using a Dynamically Typed Programming Language"** by **Jan Wedekind** in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Date: February 2012

Director of Studies: ____________________
Dr. Martin Howarth

Research Supervisor: ____________________
Dr. Jacques Penders

Research Supervisor: ____________________
Dr. Jon Travis

Research Advisor: ____________________
Dr. Ken Dutton

Research Advisor: ____________________
Dr. Balasundram Amavasai

Disclaimer

Sheffield Hallam University

Author: **Jan Wedekind**

Title: **Efficient Implementations of Machine Vision Algorithms using a Dynamically Typed Programming Language**

Department: **Materials and Engineering Research Institute**

Degree: **PhD** Year: **2012**

Permission is herewith granted to Sheffield Hallam University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.

Signature of Author

Contents

Contents viii

Symbols xiii

Acronyms xv

List of Figures xviii

List of Tables xxii

Listings xxiii

1 Introduction 1
- 1.1 Interpreted Languages 1
- 1.2 Dynamically Typed Languages 3
- 1.3 Contributions of this Thesis 6
- 1.4 Thesis Outline 7

2 State of the Art 9
- 2.1 Object Localisation 9
- 2.2 Existing FOSS for Machine Vision 10
  - 2.2.1 Statically Typed Libraries 12
  - 2.2.2 Statically Typed Extensions 15
  - 2.2.3 Dynamically Typed Libraries 18
- 2.3 Ruby Programming Language 21
  - 2.3.1 Interactive Ruby 22
  - 2.3.2 Object-Oriented, Single-Dispatch 23
  - 2.3.3 Dynamic Typing 24
  - 2.3.4 Exception Handling 24
  - 2.3.5 Garbage Collector 26
  - 2.3.6 Control Structures 26
  - 2.3.7 Mixins 27
  - 2.3.8 Closures 27
  - 2.3.9 Continuations 28
  - 2.3.10 Introspection 29
  - 2.3.11 Meta Programming 29

2.3.12 Reification . . . 30
2.3.13 Ruby Extensions . . . 31
2.3.14 Unit Testing . . . 31
2.4 JIT Compilers . . . 32
2.4.1 Choosing a JIT Compiler . . . 32
2.4.2 libJIT API . . . 33
2.5 Summary . . . 36

3 Handling Images in Ruby **37**
3.1 Transparent JIT Integration . . . 38
3.2 Malloc Objects . . . 40
3.3 Basic Types . . . 41
3.3.1 Booleans . . . 42
3.3.2 Integers . . . 42
3.3.3 Floating-Point Numbers . . . 43
3.3.4 Composite Numbers . . . 44
3.3.5 Pointers . . . 44
3.3.6 Ruby Objects . . . 46
3.4 Uniform Arrays . . . 46
3.4.1 Variable Substitution . . . 47
3.4.2 Lambda Terms . . . 48
3.4.3 Lookup Objects . . . 48
3.4.4 Multi-Dimensional Arrays . . . 49
3.4.5 Array Views . . . 51
3.5 Operations . . . 52
3.5.1 Constant Arrays . . . 52
3.5.2 Index Arrays . . . 53
3.5.3 Type Matching . . . 53
3.5.4 Element-Wise Unary Operations . . . 55
3.5.5 Element-Wise Binary Operations . . . 57
3.5.6 LUTs and Warps . . . 59
3.5.6.1 LUTs . . . 59
3.5.6.2 Warps . . . 60
3.5.7 Injections . . . 61
3.5.8 Tensor Operations . . . 64
3.5.9 Argmax and Argmin . . . 64
3.5.10 Convolution . . . 66
3.5.11 Integral . . . 68
3.5.12 Masking/Unmasking . . . 71
3.5.13 Histograms . . . 72

3.6 JIT Compiler . . . 73
3.6.1 Stripping Terms . . . 75
3.6.2 Compilation and Caching . . . 75
3.7 Unit Testing . . . 77
3.8 Summary . . . 79

4 Input/Output **80**
4.1 Colour Spaces . . . 81
4.1.1 sRGB . . . 81
4.1.2 YC_bC_r . . . 82
4.2 Image Files . . . 85
4.3 HDR Image Files . . . 88
4.4 Video Files . . . 90
4.5 Camera Input . . . 94
4.6 Image Display . . . 95
4.7 RGBD Sensor . . . 97
4.8 GUI Integration . . . 99
4.9 Summary . . . 100

5 Machine Vision **101**
5.1 Preprocessing . . . 101
5.1.1 Normalisation and Clipping . . . 101
5.1.2 Morphology . . . 102
5.1.2.1 Erosion and Dilation . . . 102
5.1.2.2 Binary Morphology . . . 104
5.1.3 Otsu Thresholding . . . 105
5.1.4 Gamma Correction . . . 105
5.1.5 Convolution Filters . . . 106
5.1.5.1 Gaussian Blur . . . 106
5.1.5.2 Gaussian Gradient . . . 107
5.1.6 Fast Fourier Transform . . . 111
5.2 Feature Locations . . . 113
5.2.1 Edge Detectors . . . 113
5.2.1.1 Roberts' Cross Edge-Detector . . . 113
5.2.1.2 Sobel Edge-Detector . . . 114
5.2.1.3 Non-Maxima Suppression for Edges . . . 115
5.2.2 Corner Detectors . . . 115
5.2.2.1 Corner Strength by Yang *et al.* . . . 115
5.2.2.2 Shi-Tomasi Corner Detector . . . 116
5.2.2.3 Harris-Stephens Corner- and Edge-Detector . . . 118
5.2.2.4 Non-Maxima Suppression for Corners . . . 120

5.3 Feature Descriptors . . . 120
5.3.1 Restricting Feature Density . . . 120
5.3.2 Local Texture Patches . . . 122
5.3.3 SVD Matching . . . 123
5.4 Summary . . . 124

6 Evaluation **125**
6.1 Software Modules . . . 125
6.2 Assessment of Functionality . . . 127
6.2.1 Fast Normalised Cross-Correlation . . . 127
6.2.2 Lucas-Kanade Tracker . . . 127
6.2.3 Hough Transform . . . 132
6.2.4 Microscopy Software . . . 133
6.2.5 Depth from Focus . . . 134
6.2.6 Gesture-based Mouse Control . . . 136
6.2.7 Slide Presenter . . . 139
6.2.8 Camera Calibration . . . 141
6.2.8.1 Corners of Calibration Grid . . . 142
6.2.8.2 Camera Intrinsic Matrix . . . 146
6.2.8.3 3D Pose of Calibration Grid . . . 147
6.2.9 Augmented Reality . . . 148
6.3 Performance . . . 151
6.3.1 Comparison with NArray and C++ . . . 151
6.3.2 Breakdown of Processing Time . . . 153
6.4 Code Size . . . 155
6.4.1 Code Size of Programs . . . 155
6.4.2 Code Size of Library . . . 157
6.5 Summary . . . 157

7 Conclusions & Future Work **159**
7.1 Conclusions . . . 159
7.2 Future Work . . . 161

A Appendix **163**
A.1 Connascence . . . 163
A.2 Linear Least Squares . . . 164
A.3 Pinhole Camera Model . . . 164
A.4 Planar Homography . . . 165
A.5 “malloc” gem . . . 168
A.6 “multiarray” gem . . . 168
A.7 Miscellaneous Sources . . . 168

A.7.1 JIT Example 168
A.7.2 Video Player 169
A.7.3 Normalised Cross-Correlation 169
A.7.4 Camera Calibration 170
A.7.5 Recognition of a rectangular marker 173
A.7.6 Constraining Feature Density 174
A.7.7 SVD Matching 175

Bibliography **177**

Symbols

$:=$ "is defined to be"

$=:$ "defines"

$\equiv$ "is logically equivalent to"

$\in$ "is an element of"

$!$ "must be"

$\mapsto$ "maps to"

$\rightarrow$ "from ... to ..."

$\times$ product (or Cartesian product)

$\wedge$ "and"

$\ominus$ erosion

$\oplus$ dilation

$\otimes$ convolution

$\rightarrow_\beta$ β-reduction

$\forall$ "for all"

$\mathbb{B}$ Boolean set

B blue 72

C clipping

C_b chroma blue 83

C_r chroma red 83

$\exists$ "there exists"

G green 72

K_b weight of blue component in luma channel

K_r weight of red component in luma channel

$\mathcal{N}$ normalisation

$\mathbb{N}_0$ set of all natural numbers including zero

P_b chroma blue

P_r chroma red

$\mathbb{R}$	set of all real numbers
R	red . . . 72
$\mathcal{S}$	structure tensor
U	chroma blue . . . 83
V	chroma red . . . 83
Y	luma . . . 83
$\mathbb{Z}$	set of all integers

Acronyms

1D one-dimensional . . . 48
2D two-dimensional . . . 10
3D three-dimensional . . . 10
AAC Advanced Audio Coding . . . 90
ALSA Advanced Linux Sound Architecture
AOT ahead-of-time . . . 8
API application programming interface . . . 15
ASF Advanced Systems Format . . . 90
AVC Advanced Video Coding
AVI Audio Video Interleave . . . 90
BLUE best linear unbiased estimator
BMP Bitmap Image File . . . 86
CIE Commission internationale de l'éclairage
CPU central processing unit . . . 42
CRT cathode ray tube . . . 82
DFT discrete Fourier transform . . . 111
DICOM Digital Imaging and Communications in Medicine . . . 86
DLL dynamic-link library . . . 33
FFT Fast Fourier Transform . . . 111
FFTW Fastest Fourier Transform in the West
FFTW3 FFTW version 3 . . . 111
FLV Flash Video . . . 90
foldl fold-left . . . 61
foldr fold-right . . . 62
FOSS free and open source software . . . 10
GCC GNU Compiler Collection . . . 126
GIF Graphics Interchange Format . . . 86

GNU “GNU’s Not Unix!”

GPL GNU General Public License

GPGPU general purpose GPU 6

GPU graphics processing unit 7

GUI graphical user interface 8

H.264 MPEG-4 AVC standard 90

HDR high dynamic range 8

HSV hue, saturation, and value 73

I input 6

IIR infinite impulse response 106

IR Infrared 97

IRB Interactive Ruby Shell 22

JIT just-in-time 7

JPEG Joint Photographic Experts Group 83

LDR low dynamic range 8

LLVM Low Level Virtual Machine 32

LUT lookup table 59

MP3 MPEG Audio Layer 3 90

MPEG Motion Picture Experts Group

MPEG-4 MPEG standard version 4 90

MOV Apple Quicktime Movie 90

MR magnetic resonance

O output 6

OCR optical character recognition 15

Ogg Xiph.Org container format 90

PBM portable bitmap 86

PGM portable graymap 86

PNG Portable Network Graphics 86

PPM portable pixmap 86

RGB red, green, blue 44

RGBD RGB and depth 81

RANSAC Random Sample Consensus 10

SLAM	Simultaneous Localisation and Mapping	10
SO	shared object	73
sRGB	standard RGB colour space	81
SVD	singular value decomposition	123
TEM	transmission electron microscopy	134
Theora	Xiph.Org video codec	90
TIFF	Tagged Image File Format	86
Vorbis	Xiph.Org audio codec	90
VP6	On2 Truemotion VP6 codec	90
VM	virtual machine	8
V4L	Video for Linux	
V4L2	V4L version 2	
WMA	Windows Media Audio	90
WMV	Windows Media Video	90

List of Figures

1.1 Optical parking system for a car 2
1.2 Feedback cycle in a compiled and in an interpreted language 2
1.3 ARM Gumstix boards 3
1.4 Early vs. late method binding 4
1.5 Static typing vs. dynamic typing. Comment lines (preceded with "//") show the output of the compiler 4
1.6 Static typing and numeric overflow. Comment lines (preceded with "//") show the output of the program 5
1.7 Ariane 5 disaster caused by numerical overflow 5
1.8 Software architecture of machine vision system 7

2.1 Overview of a typical object localisation algorithm 9
2.2 Binary operations for different element types (Wedekind et al., b) 13
2.3 Low resolution image of a circle 15
2.4 Processing time comparison for creating an index array with GCC compiled code vs. with the Ruby VM 19
2.5 Interactive Ruby Shell 22
2.6 Mark & Sweep garbage collector 26
2.7 Conditional statements in Ruby (Fulton, 2006) 26
2.8 Loop constructs in Ruby (Fulton, 2006) 27

3.1 Pointer operations (compare Listing 3.5) 41
3.2 Abstract data type for 16-bit unsigned integer 43
3.3 Abstract data types for single-precision and double-precision floating point numbers 44
3.4 Composite types for unsigned byte RGB, unsigned short int RGB, and single-precision floating point RGB values 45
3.5 Abstract data type for pointer to double precision floating point number . 45
3.6 Shape and strides for a 3D array 50
3.7 Extracting array views of a 2D array 52
3.8 Type matching 54
3.9 Avoiding intermediate results by using lazy evaluation 56
3.10 Pseudo colour palette 60
3.11 Thermal image displayed with pseudo colours (source: NASA Visible Earth) 60

3.12 Visualised components of warp vectors 61
3.13 Warping a satellite image (source: NASA Visible Earth) 61
3.14 Recursive implementation of injection (here: sum) 63
3.15 Recursive implementation of argument maximum 65
3.16 Diagonal injection 66
3.17 Applying a moving average filter to an image 68
3.18 Recursive implementation of integral image 70
3.19 Computing a moving average filter using an integral image 70
3.20 Histogram segmentation example 73
3.21 Histograms of the red, green, and blue colour channel of the reference image 74
3.22 3D histogram 74

4.1 Input/output integration 80
4.2 Colour image and corresponding grey scale image according to sensitivities of the human eye 83
4.3 Colour space conversions (Wilson, 2007) 84
4.4 YV12 colour space (Wilson, 2007) 84
4.5 YUY2 colour space (Wilson, 2007) 84
4.6 UYVY colour space (Wilson, 2007) 85
4.7 Artefacts caused by colour space compression 85
4.8 Low resolution colour image using lossless PNG and (extremely) lossy JPEG compression 86
4.9 Examples of images in medical science and material science 87
4.10 Bracketing, alignment, and tone mapping 89
4.11 Decoding videos using FFmpeg 91
4.12 Encoding videos using FFmpeg 92
4.13 Logitech Quickcam Pro 9000 (a USB webcam) 94
4.14 Display, windows, and visuals on a standard X Window desktop 96
4.15 RGB- and depth-image acquired with an RGBD sensor 98
4.16 Work flow for creating Qt4 user interfaces 99
4.17 XVideo widget embedded in a GUI 100

5.1 Normalisation and clipping of RGB values 102
5.2 Grey scale erosion and dilation 103
5.3 Binary dilation according to Listing 5.2 104
5.4 Gamma correction 105
5.5 1D Gaussian blur filter 108
5.6 2D Gaussian blur filter 108
5.7 Gaussian blur ($\sigma = 3$, $\epsilon = 1/256$) applied to a colour image 109
5.8 1D Gauss gradient filter 110

5.9 2D Gauss gradient filter . . . 110
5.10 Gauss gradient filter ($\sigma = 3$, $\epsilon = 1/256$) applied to a colour image . . . 111
5.11 Spectral image of a piece of fabric . . . 112
5.12 Example image and corresponding Roberts' Cross edges . . . 114
5.13 Example image and corresponding Sobel edges . . . 115
5.14 Non-maxima suppression for edges . . . 116
5.15 Corner detection by Yang *et al.* . . . 117
5.16 Shi-Tomasi corner-detector . . . 118
5.17 Harris-Stephens response function . . . 119
5.18 Harris-Stephens corner- and edge-detector (negative values (edges) are black and positive values (corners) are white) . . . 119
5.19 Non-maxima suppression for corners . . . 120
5.20 Restricting feature density . . . 121
5.21 Computing feature locations and descriptors . . . 122
5.22 SVD tracking . . . 124

6.1 Normalised cross-correlation example . . . 128
6.2 Comparison of template and warped image . . . 128
6.3 Gradient boundaries of template . . . 129
6.4 Warp without and with interpolation . . . 131
6.5 Example of Lucas-Kanade tracker in action . . . 131
6.6 Line detection with the Hough transform . . . 132
6.7 Configuration GUI . . . 133
6.8 Closed-loop control of a nano manipulator in a TEM . . . 134
6.9 Part of focus stack showing glass fibres . . . 136
6.10 Results of Depth from Focus . . . 136
6.11 Human computer interface for controlling a mouse cursor . . . 137
6.12 Software for vision-based changing of slides . . . 139
6.13 Quick gesture for displaying the next slide . . . 140
6.14 Slow gesture for choosing a slide from a menu . . . 141
6.15 Unibrain Fire-I (a DC1394-compatible Firewire camera) . . . 141
6.16 Custom algorithm for labelling the corners of a calibration grid . . . 143
6.17 Result of labelling the corners . . . 144
6.18 Estimating the pose of the calibration grid . . . 149
6.19 Custom algorithm for estimating the 3D pose of a marker . . . 150
6.20 Augmented reality demonstration . . . 152
6.21 Performance comparison of different array operations . . . 152
6.22 Processing time of running "`m + 1`" one-hundred times for different array sizes . . . 153
6.23 Processing time increasing with length of expression . . . 153

6.24 Breakdown of processing time for computing “`-s`” where “`s`” is an array with one million elements . . . 154

7.1 The main requirements when designing a programming language or system (Wolczko, 2011) . . . 159

7.2 Vicious cycle leading to programming languages becoming entrenched . . . 162

A.1 Pinhole camera model . . . 165

List of Tables

2.1 Processing steps performed by a typical machine vision systems 10
2.2 Existing FOSS libraries for Machine Vision I/II 11
2.3 Existing FOSS libraries for Machine Vision II/II 11
2.4 Ruby notation 28
2.5 Just-in-time compilers 33

3.1 Directives for conversion to/from native representation 40
3.2 Methods for raw memory manipulation 41
3.3 Generic set of array operations 53

4.1 Different types of images 82
4.2 Methods for loading and saving images 87
4.3 Methods for loading and saving images 89

5.1 Non-maxima suppression for edges depending on the orientation 116

6.1 Processing times measured for tasks related to computing "`-s`" for an array 154
6.2 Size of OpenCV code for filtering images 157
6.3 Size of Hornetseye code for all array operations 157

Listings

2.1 Multi-dimensional "+" operator implemented in C++. Comment lines (preceded with "//") show the output of the program . . . 14
2.2 Integrating RMagick and NArray in Ruby. Comment lines (preceded with "#") show the output of the program . . . 16
2.3 Using OpenCV in Ruby. Comment lines (preceded with "#") show the output of the program . . . 16
2.4 Using NArray in Ruby. Comment lines (preceded with "#") show the output of the program . . . 17
2.5 Array operations in Python using NumPy. Comment lines (preceded with "#") show the output of the program . . . 17
2.6 Tensor operation with the FTensor C++ library . . . 18
2.7 Multi-dimensional "+" operator implemented in Ruby. Comment lines (preceded with "#") show the output of the program . . . 18
2.8 Arrays in Ruby. Comment lines (preceded with "#") show the output of the program . . . 19
2.9 Arrays in GNU Common Lisp. Comment lines (preceded with ";") show the output of the program . . . 20
2.10 Lush programming language. Comment lines (preceded with ";") show the output of the program . . . 21
2.11 Method dispatch in Ruby . . . 23
2.12 Methods in Ruby which are not overloadable . . . 23
2.13 Dynamic typing in Ruby . . . 24
2.14 Numerical types in Ruby . . . 25
2.15 Exception handling in Ruby . . . 25
2.16 Mixins in Ruby . . . 27
2.17 Closures in Ruby . . . 28
2.18 Continuations in Ruby . . . 28
2.19 Introspection in Ruby . . . 29
2.20 Meta programming in Ruby . . . 30
2.21 Reification in Ruby . . . 30
2.22 Example of a C-extension for Ruby . . . 31
2.23 Using the extension defined in Listing 2.22 . . . 31
2.24 Unit test for "`Array#+`" defined in Listing 2.7 . . . 32
2.25 Array operation implemented in C . . . 34

2.26 Array operation compiled with libJIT 35
3.1 Reflection using missing methods 38
3.4 Converting arrays to binary data and back 40
3.5 Manipulating raw data with Malloc objects 41
3.6 Boxing booleans 42
3.7 Constructor short cut 42
3.8 Template classes for integer types 43
3.9 Boxing floating point numbers 43
3.10 Composite numbers 44
3.11 Boxing composite numbers 45
3.12 Pointer objects 46
3.13 Boxing arbitrary Ruby objects 46
3.14 Variable objects and substitution 47
3.15 Lambda abstraction and application 48
3.16 Implementing arrays as lazy lookup 49
3.17 Uniform arrays 49
3.18 Multi-dimensional uniform arrays 50
3.19 Array views 51
3.20 Constant arrays 54
3.21 Index arrays 54
3.22 Type matching 55
3.23 Element-wise unary operations using "`Array#collect`" 55
3.24 Short notation for element-wise operations 56
3.25 Internal representation of unary operations 56
3.26 Element-wise binary operations using "`Array#collect`" and "`Array#zip`" 57
3.27 Internal representation of binary operations 58
3.28 Element-wise application of a LUT 59
3.29 Creating a pseudo colour image 59
3.30 Warp from equirectangular to azimuthal projection 61
3.31 Left-associative fold operation in Ruby 62
3.32 Internal representation of injections 62
3.33 Various cumulative operations based on injections 63
3.34 Concise notation for sums of elements 63
3.35 Tensor operations in Ruby (equivalent to Listing 2.6) 64
3.36 Argument maximum 65
3.37 One-dimensional convolutions in Ruby 67
3.38 Two-dimensional convolutions in Ruby 68
3.39 Moving average filter implemented using convolutions 69
3.40 Moving average filter implemented using an integral image 70
3.41 Conditional selection as element-wise operation 71

3.42 Injection with a conditional 71
3.43 Injection on a subset of an array 71
3.44 Element-wise operation on a subset of an array 72
3.45 Two-dimensional histogram 72
3.46 Histogram segmentation 73
3.47 Lazy negation of integer 75
3.51 The resulting C code to Listing 3.50 76
3.53 Some unit tests for integers 77
3.54 Tests for array operations 78
4.1 Handling compressed colour spaces 85
4.2 Loading and saving images 87
4.3 Converting an HDR image to an LDR image (no tone mapping) 89
4.4 Reading a video file 91
4.5 Writing a video file 93
4.6 Opening a V4L2 device and negotiating a video mode 95
4.7 Opening a V4L2 device and selecting a fixed mode 95
4.8 Loading and displaying an image 95
4.9 Displaying a video using Python and OpenCV 96
4.10 Displaying a video using Ruby and Hornetseye 97
4.11 Minimalistic video player 97
4.12 Hardware accelerated video output 97
4.13 Ruby example for accessing a Kinect sensor 98
5.1 Implementing dilation using diagonal injection 103
5.2 Implementing a structuring element using convolution and a lookup table 104
5.3 Otsu thresholding 105
5.4 Generating a gamma corrected gradient 106
5.5 Estimating the spectrum of a 2D signal 112
5.6 Roberts' Cross edge-detector 113
5.7 Sobel edge-detector 114
5.8 Non-maxima suppression for edges 115
5.9 Yang *et al.* corner detector 117
5.10 Shi-Tomasi corner detector 118
5.11 Harris-Stephens corner and edge detector 120
5.12 Non-maxima suppression for corners 121
5.13 Extracting local texture patches 122
5.14 SVD matching 123
6.1 Lucas-Kanade tracker 130
6.2 Hough transform to locate lines 132
6.3 Implementation of Depth from Focus 135
6.4 Human computer interface for controlling the mouse cursor 138

6.5 Lookup table for re-labelling 138
6.6 Vision-based changing of slides 140
6.7 Custom algorithm for labelling the corners of a calibration grid 145
6.8 Webcam viewer implemented using Python and OpenCV 155
6.9 Webcam viewer implemented using Ruby and Hornetseye 155
6.10 Sobel gradient viewer implemented using Python and OpenCV 156
6.11 Sobel gradient viewer implemented using Ruby and Hornetseye 156

1 Introduction

"Plan to throw one away; you will anyhow."
Fred Brooks - The Mythical Man-Month

"If you plan to throw one away, you will throw away two."
Craig Zerouni

"How sad it is that our PCs ship without programming languages. Every computer shipped should be programmable - as shipped."
Word Cunningham

"I didn't go to university. Didn't even finish A-levels. But I have sympathy for those who did."
Terry Pratchett

Machine vision is a broad field and in many cases there are several independent approaches solving a particular problem. Also, it is often difficult to preconceive which approach will yield the best results. Therefore it is important to preserve the agility of the software to be able to implement necessary changes in the final stages of a project.

A traditional application of computer vision is industrial automation. That is, the cost of implementing a machine vision system eventually needs to be recovered by savings in labour cost, increased productivity, and/or better quality in manufacturing. Most machine vision systems however are still implemented using a statically typed programming language such as C, C++, or Java (see Section 2.2). Development and maintenance of large scale systems using a statically typed language is much more expensive compared to when using a dynamically typed languages (Nierstrasz et al., 2005).

This thesis shows how the dynamically typed programming language *Ruby* can be used to reduce the cost of implementing machine vision algorithms. A Ruby library is introduced which facilitates rapid prototyping and development of machine vision systems. The thesis is organised as follows

- Section 1.1 discusses interpreted programming languages
- Section 1.2 introduces the notion of dynamically typed programming languages
- Section 1.3 states the contribution of this thesis
- Section 1.4 gives an outline of the thesis

1.1 Interpreted Languages

Historically software for machine vision systems was predominantly implemented in compiled languages such as assembler or C/C++. Most compiled languages map effi-

ciently to machine code and they don't use a run-time environment for managing variables and data types. Concise and efficient code is a requirement especially for embedded systems with limited processing power and memory (*e.g.* see Figure 1.1 for an example of an embedded system involving computer vision).

Figure 1.1: Optical parking system for a car

The downside of using a compiled language is that a developer is required to make changes to the source code, save them in a file, compile that file to create a binary file, and then re-run that binary file. In contrast, **interpreted languages** offer considerable savings in development time. In an interpreted language *the developer can enter code and have it run straight away*. Figure 1.2 shows that the feedback cycle in an interpreted language is much shorter than the one of a compiled language.

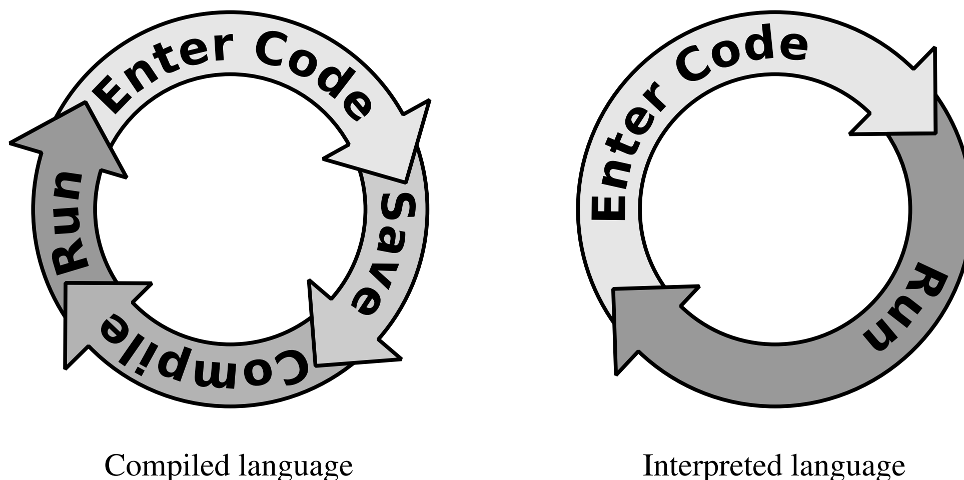


Figure 1.2: Feedback cycle in a compiled and in an interpreted language

A shorter feedback cycle consumes less time as the developer does not need to spend time waiting for the result of the previous change. The immediate feedback also fits well with the human learning process. Immediate feedback about the progress being made is a requirement for the human mind to enter a state of "flow" where it operates at full capacity (Nakamura and Csikszentmihalyi, 2002; DeMarco and Lister, 1987).

Even though interpreted languages have been applied to machine vision as early as 1987 (see Mundy (1987)), machine vision systems are still predominantly implemented using compiled languages. The reason is that if an embedded system is produced in large quantities, it is possible to offset the considerable software development cost against small per-unit savings in hardware cost. However this trade-off might become less important with the advent of modern embedded hardware (Figure 1.3 for example shows the Gumstix board which is an embedded computer capable of running an operating system).

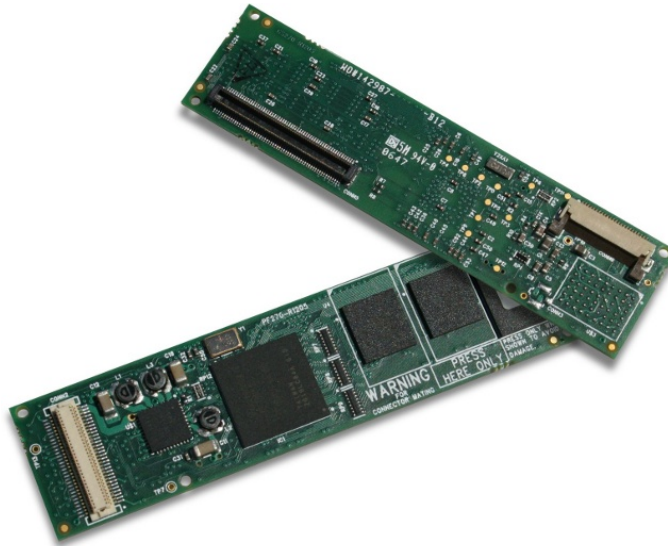


Figure 1.3: ARM Gumstix boards

It can be argued that the widespread adoption of compiled languages is currently hampering innovation (Nierstrasz et al., 2005). The publication by Roman et al. (2007) demonstrates that robotic projects can greatly benefit from the properties of the interpreted programming language Ruby. Interpreted languages not only allow for concise code, they also make interactive manipulation of data possible where one can confirm the results immediately.

1.2 Dynamically Typed Languages

The benefits of using an interpreted language are quite obvious. A less visible but nevertheless important issue is the difference between *statically typed languages* and *dynamically typed languages*. Note that this issue should not be confused with the issue of strong typing versus weak typing. A language is statically typed if all type checks are performed at compile-time. Dynamically typed languages on the other hand perform type checks at run-time and allow to define new types at run-time. Dynamic typing however makes early method binding impossible which has a negative impact on run-time performance. Figure 1.4 gives an example. In C++ the "+" operation can be compiled to a machine instruction (*e.g.* "`ADD AX, 1`"). The method "`test`" is limited to processing integers. In

Ruby however it is in general impossible to determine whether the value of "`x`" always will be an integer. For example the value might be a floating point number or a rational number.

```
int test(int x)
{
  return x + 1;
}
// ...
  int y = test(42);
// ...
```

C++ (early method binding)

```
def test(x)
  x + 1
end
# ...
y = test 42
z = test Complex::I
```

Ruby (late method binding)

Figure 1.4: Early vs. late method binding

Type safety is a term to describe the fact that static typing prevents certain programming errors such as type mismatches or misspelled method names from entering production code. With static typing it is possible to reject these kind of errors at compile time. Statically typed languages are engrained in safety critical systems such as nuclear power plants, air planes, and industrial robots because of increased type safety. Figure 1.5 gives an example where the bug in the C++ program is rejected by the compiler. The equivalent Ruby program however discovers the error only at run-time and only for certain input.

```
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int x = atoi(argv[1]);
  if (x == 0) x += "test";
  // error: invalid conversion from
  //   'const char*' to 'int'
  return 0;
}
```

C++ (static typing)

```
x = ARGV[0].to_i
x += "test" if x == 0
```

Ruby (dynamic typing)

Figure 1.5: Static typing vs. dynamic typing. Comment lines (preceded with "//") show the output of the compiler

However statically typed implementations tend to become inflexible. That is, when a developer wants to modify one aspect of the system, the static typing can force numerous rewrites in unrelated parts of the source code (Tratt and Wuyts, 2007). Development and maintenance of large scale systems using a statically typed language is much more expensive compared to when using a dynamically typed languages (Nierstrasz et al., 2005). Heavy users of statically typed languages tend to introduce custom mechanisms to deal with the absence of support for reflection and meta-programming in their language (see the CERN's C++ framework for example Antcheva et al. (2009)).

Though offering some safety, static typing does not prevent programming errors such as numerical overflow or buffer overflow (Tratt and Wuyts, 2007). That is, the efficiency gained by using C or C++ is at the cost of security (Wolczko et al., 1999). Figure 1.6 shows two programs where numerical overflow occurs if a native integer type of insufficient size is chosen. A well known example is the failure of the first Ariane 5 (shown in

```
#include <iostream>
using namespace std;
int main(void)
{
  int x = 2147483648;
  x += 1;
  cout << x << endl;
  // -2147483647
  return 0;
}
```

32-bit integer

```
#include <iostream>
using namespace std;
int main(void)
{
  long x = 2147483648;
  x += 1;
  cout << x << endl;
  // 2147483649
  return 0;
}
```

64-bit integer

Figure 1.6: Static typing and numeric overflow. Comment lines (preceded with "//") show the output of the program

Figure 1.7) due to an arithmetic overflow (see talk by Fenwick, 2008). That is, even when

Figure 1.7: Ariane 5 disaster caused by numerical overflow

using static typing, it is still necessary to use techniques such as software assertions or unit tests to prevent runtime errors from happening.

Dynamic typing on the other hand allows to combine integers, rational numbers, complex numbers, vectors, and matrices in a seamless way. The Ruby core library makes use of dynamic typing to represent integers, big numbers, floating point numbers, complex numbers, and vectors work together seamlessly (see Section 2.3.3 for more details).

Ruby data types do not map well to native data types (*i.e.* the integer and floating-point registers of the hardware). For example Ruby integers do not exhibit numerical overflow

and the boundaries of Ruby arrays are resized dynamically. Furthermore dynamic typing requires late binding of method calls which is computationally expensive on current hardware (Paulson, 2007). This thesis tries to address these problems by defining representations of native types in a Ruby extension[1] (see Section 1.3).

1.3 Contributions of this Thesis

The title of this thesis is "Efficient Implementations of Machine Vision Algorithms using a Dynamically Typed Programming Language", The Ruby extension implemented in the context of this thesis makes it possible for researchers and developers working in the field of image processing and computer vision to take advantage of the benefits offered by this dynamically typed language. The phrase "efficient implementation" was intentionally used in an ambiguous way. It can mean

- **machine efficiency**: The run-time performance of the system is sufficient to implement real-time machine vision systems.
- **developer efficiency**: The programming language facilitates concise and flexible implementations which means that developers can achieve high productivity.

The contribution of this thesis is a set of computer vision extensions for the existing Ruby programming language. The extensions *bring together performance and productivity in an unprecedented way*. The Ruby extensions provide

- extensive input (I)/output (O) integration for image- and video-data
- generic array operations for uniform multi-dimensional arrays
  - a set of objects to represent arrays, array views, and lazy evaluations in a modular fashion
  - optimal type coercions for all combinations of operations and data types

The work presented in this thesis brings together several concepts which previously have not been integrated in a single computer vision system:

- *expressiveness:* An library for manipulating uniform arrays is introduced. A generic set of basic operations is used to build computer vision algorithms from the ground up.
- *lazy evaluation:* Lazy evaluation of array operations makes it possible to reduce memory-I/O. This facilitates the use of general purpose GPU (GPGPU) (not done as part of this thesis) where memory-I/O is the performance-bottleneck.

[1] Ruby libraries are generally called "Ruby extensions"

- *array views:* Shared references make it possible to extract sub-arrays without making a "deep copy" of the array.
- *transparent just-in-time (JIT) compilation:* A JIT compiler and a cache are integrated transparently to achieve real-time performance.
- *I/O integration:* The implementation also provides integration for image- and video-I/O (see Figure 1.8) as well as the necessary colour space conversions.

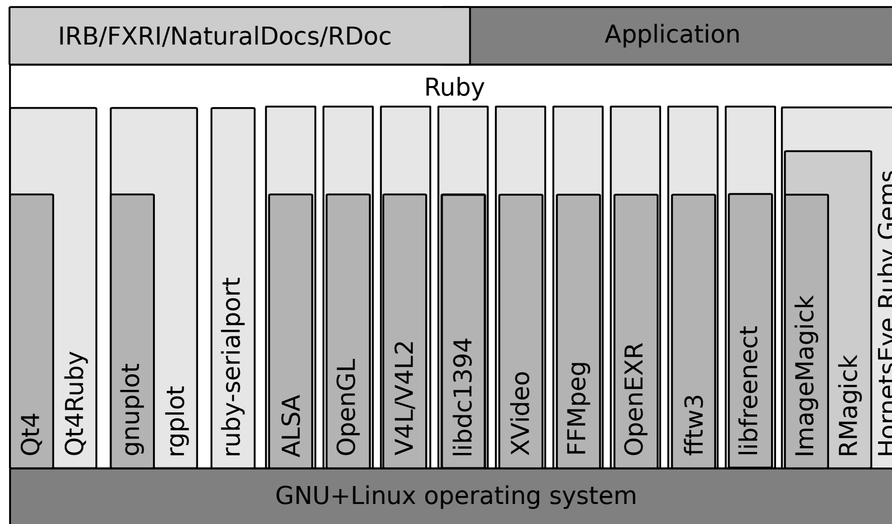


Figure 1.8: Software architecture of machine vision system

The functionality was implemented in a modular way (see Section 3.4). The result is a comprehensive approach to implementing computer vision algorithms.

The type system and the expressions presented in Chapter 3 constitute the library which was developed as part of this thesis. If some of the expressions appear to be part of the Ruby syntax at first sight, it is due to the dynamic nature of the programming language. Although the Ruby programming language was used, this approach could be applied to other dynamically typed languages with sufficient meta-programming support. The approach presented in this thesis could also be used to provide transparent integration of graphics processing units (GPUs) for parallel processing. Finally the facilitation of succinct implementations of various computer vision algorithms allows for a more formal understanding of computer vision.

1.4 Thesis Outline

Chapter 1 (this chapter) showed that there is sufficient motivation to address the performance issues involved with applying a dynamically typed language to the problem of

implementing machine vision algorithms. Apart from offering productivity gains, dynamically typed languages also make it possible to combine various types and operations seamlessly.

Chapter 2 gives an overview of the state of the art in machine vision software, illustrating the difficulty of achieving performance and productivity at the same time. It will be shown that the performance of the Ruby virtual machine (VM) is significantly lower than the performance achieved with GNU C. But it will also be argued that ahead-of-time (AOT) compilation is incompatible with the goal of achieving productivity.

Chapter 3 is about the core of the work presented in this thesis. Starting with memory objects and native data types, a library for describing computer vision algorithms is introduced. It is demonstrated how this approach facilitate succinct implementations of basic image processing operations. JIT compilation is used to address the issue of performance.

Chapter 4 covers key issues in implementing interfaces for input and output of image data. Image I/O involving cameras, image files, video files, and video displays is discussed. The key issues are colour space compression, image and video compression, low dynamic range (LDR) versus high dynamic range (HDR) imaging, and graphical user interface (GUI) integration.

In Chapter 5 it is shown how different algorithms which are common in the field of computer vision can be implemented using the concepts introduced in chapter Chapter 3 and Chapter 4.

Chapter 6 shows some examples of complete applications implemented using the Hornetseye Ruby extension which was developed as part of this thesis (see page iii). Furthermore a performance comparison is given.

At the end of the thesis Chapter 7 offers conclusions and future work.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Sir Charles Antony Richard Hoare

"I am a historian and a computer programmer, but primarily I am a lawyer. My research, ongoing for a decade, follows a purely experimental paradigm:

1. *Try to create freedom by destroying illegitimate power sheltered behind intellectual property law.*
2. *See what happens.*

Early results are encouraging."

Eben Moglen

2 State of the Art

This chapter gives an overview of the state of the art in machine vision systems, it discusses the features of the Ruby programming language, and available JIT compilers are discussed

- Section 2.1 shows the typical structure of an object localisation system
- Section 2.2 gives an overview of a typical object localisation algorithm and how it is implemented
- Section 2.3 characterises the Ruby programming language by describing the paradigms it supports
- Section 2.4 points out different JIT compilers and their properties
- Section 2.5 gives a summary of this chapter

2.1 Object Localisation

The task of an object localisation algorithm is to determine the pose of known objects given a camera image as input. Figure 2.1 shows an overview of a typical object localisation algorithm. The processing steps are explained in Table 2.1. The processing steps

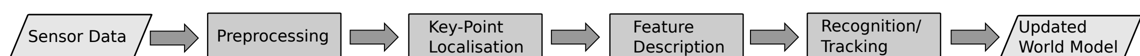


Figure 2.1: Overview of a typical object localisation algorithm

Table 2.1: Processing steps performed by a typical machine vision systems

Processing step	**Details**
preprocessing	basic operations such as filtering, thresholding, morphology and the like are applied to the image
key-point localisation	a feature extraction method defines feature locations in the image
feature description	the descriptors for the local feature context are computed
recognition/tracking	the features are used to recognise and track known objects in the scene

are not mandatory. Some algorithms do not use feature descriptors (*e.g.* Geometric Hashing (Lamdan and Wolfson, 1988)). Some algorithms for two-dimensional (2D) object localisation do not even use features at all (*e.g.* Fast Normalised Cross-Correlation (Lewis, 1995)).

Current three-dimensional (3D) object recognition and tracking algorithms however are predominantly based on feature extraction and feature matching (*e.g.* spin image features by Johnson and Hebert (1999), Geometric Hashing (Lamdan and Wolfson, 1988), Bounded Hough Transform (Greenspan et al., 2004), Random Sample Consensus (RANSAC) (Shan et al., 2004)). Approaches based on feature matching are furthermore used to deal with related problems such as real-time Simultaneous Localisation and Mapping (SLAM) (*e.g.* Davison, 2003; Pupilli, 2006) and 3D modelling (*e.g.* Pan et al., 2009; Pollefeys et al., 2004; Tomasi and Kanade, 1992; Yan and Pollefeys, 2006).

There are more unconventional techniques (*e.g.* tensor factorisation (Vasilescu and Terzopoulos, 2007), integral images (Viola and Jones, 2001)) but they are mostly applied to object detection. That is, the algorithms detect the presence of an object but do not estimate its pose.

2.2 Existing FOSS for Machine Vision

A survey of existing free and open source software (FOSS) for machine vision has been conducted in order to find out about commonalities of current algorithms in use and how current computer vision systems are implemented.

Table 2.2 and Table 2.3 give an overview of noticeable computer vision libraries. The libraries where checked against a set of features. Each check mark signifies a feature being supported by a particular library. One can see that no library completely covers all the features which are typically required to develop an object recognition and tracking system as shown in Figure 2.1.

One can distinguish three different kinds of libraries: statically typed libraries, stati-

Table 2.2: Existing FOSS libraries for Machine Vision I/II

feature	Blepo	Camellia	CMVision	libCVD	EasyVision	Filters	Framewave	Gamera	Gandalf
Camera Input	✔		✔	✔	✔				
Image Files	✔	✔		✔	✔	✔	✔	✔	✔
Video Files				✔			✔		
Display	✔		✔	✔	✔			✔	✔
Scripting		✔			✔			✔	
Warps				✔			✔		✔
Histograms			✔			✔		✔	✔
Custom Filters	✔			✔		✔	✔	✔	✔
Fourier Transforms	✔								✔
Feature Extraction	✔			✔	✔	✔	✔	✔	✔
Feature Matching	✔				✔				
GPL compatible	✔	✔	✔	✔	?	✔	✔	✔	✔

Table 2.3: Existing FOSS libraries for Machine Vision II/II

feature	ITK/VTK	IVT	LTIlib	Lush	Mimas	NASA V. W.	OpenCV	SceneLib	VIGRA
Camera Input		✔	✔	✔	✔		✔	✔	
Image Files	✔	✔	✔	✔	✔	✔	✔		✔
Video Files		✔			✔		✔		
Display	✔	✔	✔	✔	✔		✔	✔	
Scripting				✔			✔		
Warps	✔			✔	✔	✔	✔		
Histograms	✔	✔	✔	✔			✔		
Custom Filters	✔	✔	✔	✔	✔	✔			✔
Fourier Transforms	✔				✔		✔		✔
Feature Extraction	✔	✔	✔	✔	✔		✔	✔	✔
Feature Matching		✔	✔				✔	✔	
GPL compatible	✔	✔	✔	✔	✔		✔	✔	✔

cally typed extensions for a dynamically typed language, and dynamically typed libraries.

2.2.1 Statically Typed Libraries

Most computer vision libraries are implemented in the statically typed C/C++ language. However C++ has a split type system. There are primitive types which directly correspond to registers of the hardware and there are class types which support inheritance and dynamic dispatch. In C++ not only integers and floating point numbers but also arrays are primitive types. However these are the most relevant data types for image processing. To implement a basic operation such as adding two values so that it will work on different types, one needs to make extensive use of template meta-programming. That is, all combinations of operations, element-type(s), and number of dimensions have to be instantiated separately. For example the FrameWave[1] C-library has 42 explicitly instantiated different methods for multiplying arrays.

For this reason most libraries do not support all possible combinations of element-types and operations. Assume a library supports the following 10 binary operations

- addition ("+")
- subtraction ("-")
- division ("/")
- multiplication ("*")
- exponent ("**")
- greater or equal (">=")
- greater than (">")
- less or equal ("<=")
- less than ("<")
- equal to ("==")

Furthermore assume that it supports the following types as scalars and array elements

- 6 integer types: 8-,16-, and 32-bit, signed/unsigned
- 2 floating-point types: single/double precision

Finally for every binary operation there are the following variations

- scalar-array operation

[1] http://framewave.sourceforge.net/

- array-scalar operation
- array-array operation

This results in $10 \cdot 8 \cdot 8 \cdot 3 = 1920$ possible combinations of operations and element-types. That is, to fully support the 10 binary operations on these element-types requires 1920 methods to be defined either directly or by means of C++ template programming. That is, static typing and ahead-of-time compilation leads to a explosion of combinations of basic types and operations. Listing 2.1 shows how much code is required when using C++ templates to implement a element-wise "+" operator (array-array operation only) for the "`boost::multi_array`" data types provided by the Boost library. The implementation works on arrays of arbitrary dimension and arbitrary element-type.

Static typing not only leads to an explosion of methods to instanciate. A related problem caused by static typing is that when a developer wants to modify one aspect of the system, the static typing can force numerous rewrites in unrelated parts of the source code (Tratt and Wuyts, 2007). Static typing enforces unnecessary "connascence" (a technical term introduced by Weirich (2009), also see Appendix A.1) which interferes with the modularity of the software. In practise this causes problems when implementing operations involving scalars, complex numbers, and RGB-triplets (Wedekind et al., b). Figure 2.2 shows that binary operations are not defined for some combinations of the argument types involved. That is, it is not sufficient to simply use C++ templates to in-

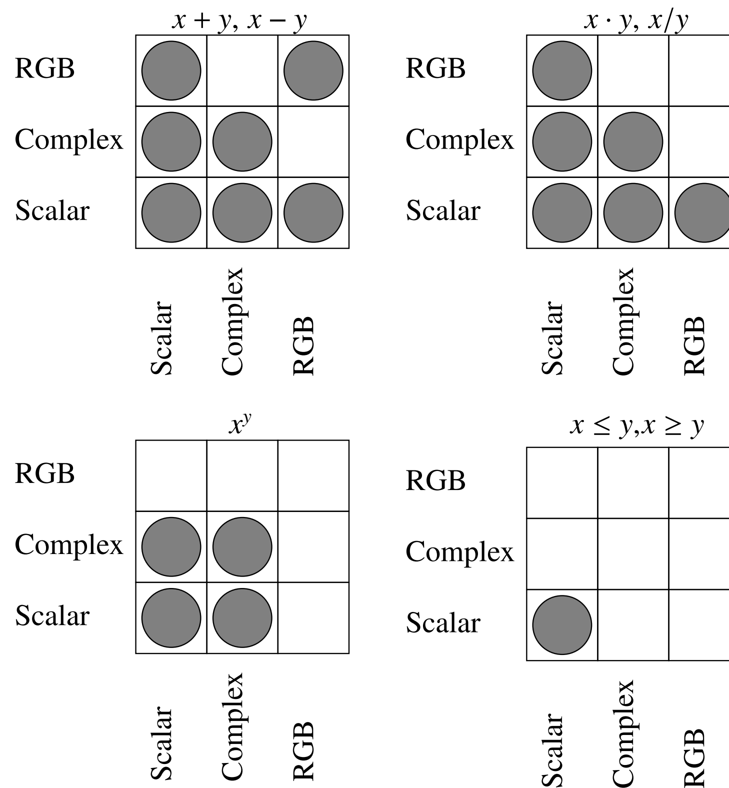


Figure 2.2: Binary operations for different element types (Wedekind et al., b)

Listing 2.1: Multi-dimensional "+" operator implemented in C++. Comment lines (preceded with "//") show the output of the program

```
#include <boost/multi_array.hpp> // 3726 lines of code
#include <iostream>
using namespace boost;
template< typename T >
T &multi_plus(T &a, const T &b, const T &c) {
  a = b + c;
  return a;
}
template< template< typename, size_t, typename > class Arr, typename Alloc,
          typename T, size_t N >
detail::multi_array::sub_array< T, N > multi_plus
  (detail::multi_array::sub_array< T, N > a, const Arr< T, N, Alloc > &b,
   const Arr< T, N, Alloc > &c) {
  typename Arr< T, N, Alloc >::const_iterator j = b.begin(), k = c.begin();
  for (typename detail::multi_array::sub_array< T, N >::iterator i =
       a.begin(); i != a.end(); i++, j++, k++)
    multi_plus(*i, *j, *k);
  return a;
}
template< template< typename, size_t, typename > class Arr, typename Alloc,
          typename T, size_t N >
Arr< T, N, Alloc > &multi_plus
  (Arr< T, N, Alloc > &a, const Arr< T, N, Alloc > &b,
   const Arr< T, N, Alloc > &c) {
  typename Arr< T, N, Alloc >::const_iterator j = b.begin(), k = c.begin();
  for (typename Arr< T, N, Alloc >::iterator i = a.begin();
       i != a.end(); i++, j++, k++)
    multi_plus(*i, *j, *k);
  return a;
}
template < template< typename, size_t, typename > class Arr, typename Alloc,
           typename T, size_t N >
multi_array< T, N > operator+
  (const Arr< T, N, Alloc > &a, const Arr< T, N, Alloc > &b) {
  array< size_t, N > shape;
  std::copy(a.shape(), a.shape() + N, shape.begin());
  multi_array< T, N > retVal(shape);
  multi_plus(retVal, a, b);
  return retVal;
};
int main(void) {
  multi_array< int, 2 > a(extents[2][2]);
  a[0][0] = 1; a[0][1] = 2; a[1][0] = 3; a[1][1] = 4;
  multi_array< int, 2 > b(extents[2][2]);
  b[0][0] = 5; b[0][1] = 4; b[1][0] = 3; b[1][1] = 2;
  multi_array< int, 2 > r(a + b);
  std::cout << "[[" << r[0][0] << ", " << r[0][1] << "], ["
            << r[1][0] << ", " << r[1][1] << "]]" << std::endl;
  // [[6, 6], [6, 6]]
  return 0;
}
```

stantiate all combinations of operations and argument types. One also has to address the problem that binary operations usually only are meaningful only for some combinations of element-types.

Finally using a combination of multiple libraries is hard, because each library usually comes with its own set of data types for representing images, arrays, matrices, and other elements of signal processing.

2.2.2 Statically Typed Extensions

Some computer vision libraries come with bindings in order to use them as an extension to a dynamically typed language. For example for the OpenCV[2] library there are Python bindings (PyCV[3]) as well as Ruby bindings (opencv.gem[4]). Some projects (*e.g.* the Gamera optical character recognition (OCR) software (Droettboom et al., 2003) and the Camellia[5] Ruby extension) use the Simplified Wrapper Generator (SWIG[6]) to generate bindings from C/C++ header files. This allows one to use a statically typed extension in an interpreted language and it becomes possible to develop machine vision software interactively without sacrificing performance.

Open classes and dynamic typing make it possible to seamlessly integrate the functionality of one library into the application programming interface (API) of another. For example Listing 2.2 shows how one can extend the NArray[7] class to use the RMagick[8] library for loading images. The method "`NArray#read`" reads an image using the RMagick extension. The image is exported to a Ruby string which in turn is imported into an object of type "`NArray`". The image used in this example is shown in Figure 2.3.

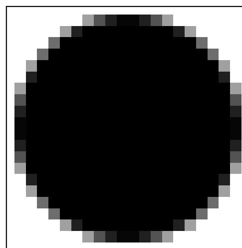


Figure 2.3: Low resolution image of a circle

However supporting all possible combinations of types and operations with a statically typed library is hard (see Section 2.2.1). In practise most computer vision extensions only provide a subset of all combinations. Listing 2.3 shows that the OpenCV library for example supports element-wise addition of 2D arrays of 8-bit unsigned integers (line 3).

[2] http://opencv.willowgarage.com/
[3] http://pycv.sharkdolphin.com/
[4] http://rubyforge.org/projects/opencv/
[5] http://camellia.sourceforge.net
[6] http://swig.org/
[7] http://narray.rubyforge.org/
[8] http://rmagick.rubyforge.org/

Listing 2.2: Integrating RMagick and NArray in Ruby. Comment lines (preceded with "#") show the output of the program

```
require 'narray'
require 'RMagick'
class NArray
  def NArray.read(filename)
    img = Magick::Image.read(filename)[0]
    str = img.export_pixels_to_str 0, 0, img.columns, img.rows, "I",
          Magick::CharPixel
    to_na str, NArray::BYTE, img.columns, img.rows
  end
end
arr = NArray.read 'circle.png'
arr / 128
# NArray.byte(20,20):
# [ [ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1 ],
#   [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1 ],
#   [ 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1 ],
#   [ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 ],
#   [ 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 ],
#   [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
#   [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 ],
#   [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
#  ...
```

But trying to add elements of 8-bit unsigned and 16-bit unsigned will cause an exception

Listing 2.3: Using OpenCV in Ruby. Comment lines (preceded with "#") show the output of the program

```
1 require 'opencv'
2 include OpenCV
3 CvMat.new(6, 2, CV_8U) + CvMat.new(6, 2, CV_8U)
4 # <OpenCV::CvMat:2x6,depth=cv8u,channel=3>
5 CvMat.new(6, 2, CV_8U) + CvMat.new(6, 2, CV_16U)
6 #(irb):4: warning: OpenCV error code (-205) : cvAdd (840 in cxarithm.cpp)
7 #OpenCV::CvStatusUnmatchedFormats:
8 #        from (irb):4:in '+'
9 #        from (irb):4
```

(line 5). Other libraries such as EasyVision[9] (an extension for Haskell) even have different method names depending on the types of arguments involved. For example "**absDiff8u**" to compute the element-wise absolute difference of arrays of 8-bit unsigned integers or "**sqrt32f**" to compute the element-wise square root of arrays of 32-bit floating point values.

In contrast to the previously mentioned libraries, the NArray[10] (Tanaka, 2010a,b) Ruby extension supports adding arrays with different element-types (see Listing 2.4). The library also does optimal return type coercions. For example adding an array with

[9] http://perception.inf.um.es/easyVision/
[10] http://narray.rubyforge.org/

Listing 2.4: Using NArray in Ruby. Comment lines (preceded with "#") show the output of the program

```
require 'narray'
a = NArray.byte 6, 2
# NArray.byte(6,2):
# [ [ 0, 0, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 0, 0 ] ]
b = NArray.sint 6, 2
# NArray.sint(6,2):
# [ [ 0, 0, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 0, 0 ] ]
a + b
# NArray.sint(6,2):
# [ [ 0, 0, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 0, 0 ] ]
2 * a + b
# NArray.sint(6,2):
# [ [ 0, 0, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 0, 0 ] ]
```

Listing 2.5: Array operations in Python using NumPy. Comment lines (preceded with "#") show the output of the program

```
1 from numpy import *
2 a = array([[1, 2], [3, 4]], dtype = int8)
3 b = array([[1, 2], [3, 4]], dtype = uint16)
4 a + b
5 # array([[2, 4],
6 #        [6, 8]], dtype=int32)
7 2 * a + b
8 # array([[3,  6],
9 #        [9, 12]], dtype=int32)
```

single precision complex numbers ("`NArray::SCOMPLEX`") and an array with double precision floating point numbers ("`NArray::DFLOAT`") will result in an array of double precision complex numbers ("`NArray::DCOMPLEX`"). In contrast to OpenCV however, the NArray library does not support unsigned integers.

A similar but more sophisticated library is NumPy[11] for Python (also see Oliphant, 2006). NumPy also offers a C-API which makes it possible to define custom element-types. Listing 2.5 shows that NumPy supports unsigned integer as well as signed integer types. In contrast to NArray the result of the type coercion is an array of 32-bit integers (line 4). Similar to the NArray library, NumPy is implemented in C and uses tables of function pointers to do operations on combinations of elements.

The problem with this approach is that the last operation shown in Listing 2.4 as well as Listing 2.5 (multiplying an array with two and adding another array) creates an array as intermediate result. That is, the result of the scalar-array multiplication is written to memory and then read back again when the array-array addition is performed. Since the

[11] `http://numpy.scipy.org/`

Listing 2.6: Tensor operation with the FTensor C++ library

```
Index< 'i', 3 > i;
Index< 'j', 3 > j;
Index< 'k', 3 > k;
Tensor2< double, 3, 3 > r, a, b;
r(i, k) = a(i, j) * b(j, k);
```

array operations are just calls to a static library, there is no means of optimising them. For this reason one cannot really use this array operations in order to build higher-level functions without sacrificing performance!

In general it is not possible to instantiate efficient implementations of all the possible combinations of operations and compile them ahead-of-time. For example the FTensor C++ library (see Listing 2.6 for example) allows one to instantiate various tensor operations using C++ templates (Landry, 2003). However the library contains code specific to the dimensions $0, 1, \ldots, 4$. That is, the functionality of the library is diminished for dimensions higher than 4.

2.2.3 Dynamically Typed Libraries

Listing 2.7: Multi-dimensional "+" operator implemented in Ruby. Comment lines (preceded with "#") show the output of the program

```
class Array
  def +(other)
    zip(other).collect { |x,y| x + y }
  end
end
a = [[1, 2], [3, 4]]
b = [[5, 4], [3, 2]]
puts (a + b).inspect
# [[6, 6], [6, 6]]
```

Listing 2.7 shows an implementation of an element-wise "+" operator (array-array operation) for the "`Array`" data type of the Ruby standard library. It is much shorter than the equivalent C++ implementation shown in Listing 2.1. That is, implementing a computer vision library in Ruby is straightforward but there is a performance issue.

Figure 2.4 shows a performance comparison of GNU C compiled code and code interpreted with two different versions of the Ruby VM. One can see that in the best case the Ruby example is 30 times slower than the equivalent C++ example.

Ruby arrays can contain elements of arbitrary type. For example Listing 2.8 shows the definition of the array of integers "`a`" in line 1. In line 9 however one element of "`a`" is set to a string value. Also Ruby arrays support dynamic resizing and they do not suffer from buffer overrun. If the array index is out of bounds, a "`nil`" object is returned (*e.g.* see line 7 of Listing 2.8). This means that Ruby code involving arrays is difficult to optimise. In

```c
#include <stdlib.h>
#define SIZE 10000000
int main(void)
{
  int i, *arr = (int *)
    malloc(SIZE * sizeof(int));
  for (i = 0; i < SIZE; i++)
    arr[i] = i;
  free(arr);
  return 0;
}
```

gcc 4.2.4 0.06s

```ruby
SIZE = 10000000
arr = (0 ... SIZE).collect { |i| i }
```

ruby 1.8.6 76.3s
ruby 1.9.1 1.8s

Intel® Core™2 CPU T5600 @ 1.83GHz
Linux 2.6.24-24-generic SMP i686 GNU/Linux

Figure 2.4: Processing time comparison for creating an index array with GCC compiled code vs. with the Ruby VM

Listing 2.8: Arrays in Ruby. Comment lines (preceded with "#") show the output of the program

```ruby
a = [[2, 3, 5], [7, 11, 13]]
# [[2, 3, 5], [7, 11, 13]]
a[0]
# [2, 3, 5]
a[0][2]
# 5
a[0][3]
# nil
a[0][2] = 'x'
# "x"
a
# [[2, 3, "x"], [7, 11, 13]]
```

general it is not possible to remove the dispatcher code and the boundary checks. That is, the current implementation of Ruby arrays is not suitable for real-time machine vision.

A noteworthy example of a machine vision library implemented in a dynamically typed language is Lush[12] which is based on Common Lisp. The Common Lisp programming language supports multi-dimensional arrays (Graham, 1994). Line 8 of Listing 2.9 shows that the array type does not support dynamic resizing which makes it possible to generate more efficient code. However the array type still supports elements of arbitrary

Listing 2.9: Arrays in GNU Common Lisp. Comment lines (preceded with ";") show the output of the program

```
(setq m (make-array '(2 3) :element-type 'integer
  :initial-contents '((2 3 5)(7 11 13))))
; #2A((2 3 5) (7 11 13))
(aref m 0)
*** - AREF: got 1 subscripts, but #2A((2 3 5) (7 11 13)) has rank 2
(aref m 0 2)
; 5
(aref m 0 3)
*** - AREF: subscripts (0 3) for #2A((2 3 5) (7 11 13)) are out of range
(setf (aref m 0 2) "x")
; "x"
m
; #2A((2 3 "x") (7 11 13))
```

type even if an element-type was specified (*e.g.* line 10 of Listing 2.9). GNU Common Lisp doesn't seem to support extracting array slices.

It is possible however to introduce uniform arrays as data types and use just-in-time compilation to generate efficient code at run-time. The Lush language demonstrates this approach (Lush was used to implement OCR software for example (Lecun et al., 1998)). Listing 2.10 shows some operations involving 2D arrays. Lush does not support dynamic resizing (*e.g.* line 6 of Listing 2.10) and it enforces uniform arrays (*e.g.* line 8 of Listing 2.10). Although Lush does just-in-time compilation, it defaults to double-precision floating point numbers (*e.g.* line 13 of Listing 2.10) instead of doing optimal coercions like NArray (Listing 2.4). The problem is that implementing support for coercions of native types requires a substantial amount of work and there is insufficient incentive for a developer to invest the time given the low acceptance of the Lisp programming language in the image processing community.

Furthermore Lush is not a Lisp library but it is a Lisp dialect (*i.e.* it is an independent programming language). That is, there are potential integration issues when using other software developed in different programming languages. A library for a language with broader adoption such as Lisp, Racket (former PLT Scheme developed by Steele and Sussman (1979)), or Clojure[13] would be more desirable.

[12] http://lush.sourceforge.net/
[13] http://clojure.org/

Listing 2.10: Lush programming language. Comment lines (preceded with ";") show the output of the program

```
(setq m [i [2 3 5][7 11 13]])
($*0 m 0)
; [i     2     3     5]
(m 0 2)
; 5
(m 0 3)
*** validate-subscript : invalid subscript : 3
(m 0 2 "x")
*** m : not a number : "x"
m
; [i[i     2     3     5]
;   [i     7    11    13]]
? (+ m 1)
= [d[d   3.0000   4.0000   6.0000]
    [d   8.0000  12.0000  14.0000]]
? (* m m)
= [d[d   4.0000   9.0000  25.0000]
    [d  49.0000 121.0000 169.0000]]
```

2.3 Ruby Programming Language

Ruby is a multi-paradigm language and it is inspired by Perl, Python, Smalltalk, Eiffel, Ada, and Lisp. Ruby supports the following language features

- object-oriented, single-dispatch
- dynamic typing
- exception handling
- garbage collection (*i.e.* managed environment)
- mixins
- closures
- continuations
- introspection
- meta programming
- reification

The Ruby programming language[14] was designed by Yukihiro Matsumoto (see article by Matsumoto, 2000 for a short introduction to Ruby; see Fulton, 2006, Matsumoto, 2002, or Cooper, 2009 for a thorough introduction). He first released his implementation

[14] http://www.ruby-lang.org/

of a Ruby interpreter as free software in 1995. With Ruby version 1.9 Koichi Sasada's implementation has become the current Ruby VM in use (Sasada, 2008).

The design philosophy of the Ruby programming language follows the following principles (Matsumoto, 2007)

- **Brevity:** The language is expressive so that programs written in that language are succinct.
- **Conservatism:** Ruby sticks to traditional control structures to reduce the cost of adoption.
- **Simplicity:** The Ruby programming language supports simple solutions.
- **Flexibility:** Ruby should adapt to the user instead of the user adapting to Ruby.
- **Balance:** The Ruby programming language tries to achieve a balance between all previous concepts.

A brief introduction to the language features of Ruby follows.

2.3.1 Interactive Ruby

The Interactive Ruby Shell (IRB) provides a command-line interface to develop programs interactively. Figure 2.5 shows IRB running in an X-Terminal. IRB accepts Ruby expres-

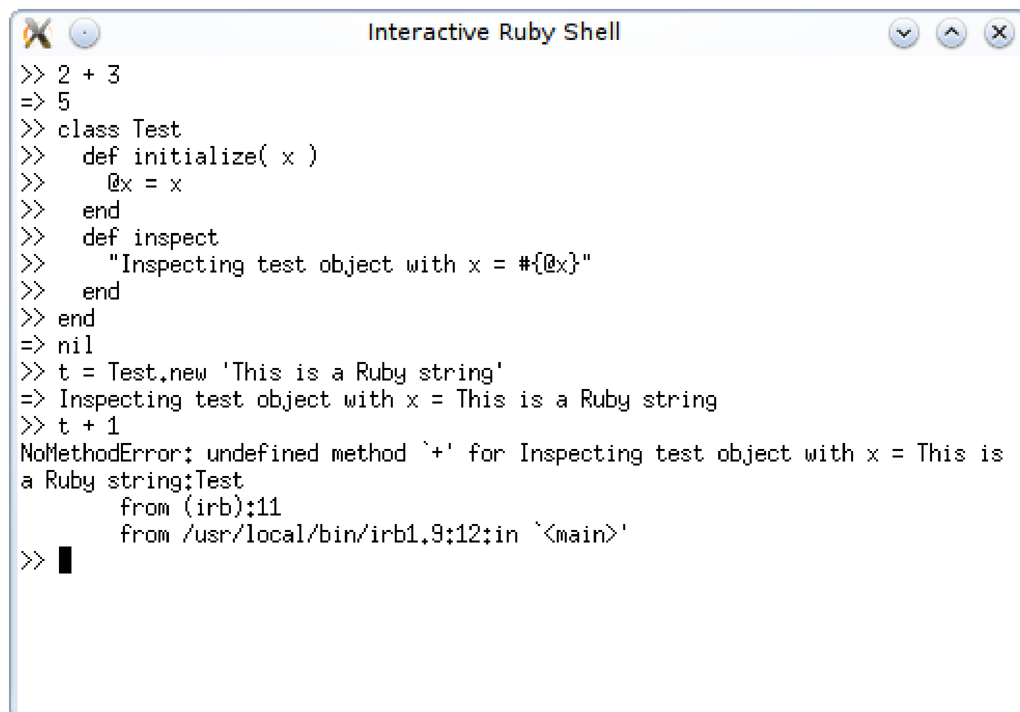


Figure 2.5: Interactive Ruby Shell

sions and interprets them in the same context. After evaluating the expression, IRB calls

"#inspect" on the result and prints the string returned by that method. For example if the user inputs "2 + 3", IRB will call "5.inspect" which returns ""5"". The unquoted string then will be printed to standard output (here prefixed with "=>"). Figure 2.5 furthermore illustrates how in the case of an error, IRB simply catches the exception and prints it to standard output instead.

2.3.2 Object-Oriented, Single-Dispatch

Ruby is object-oriented with single-dispatch. That is, every object is an instance of a class and the class defines which methods and attributes an object supports. Ruby is purely object-oriented in the sense that everything including arrays, integers, floating point numbers, and classes is an object.

Ruby has open classes. That is, it is possible to add methods to a class at any point in time. Listing 2.11 shows an example where the already existing "Numeric" class is extended with a "plus" method (lines 1 to 5). Afterwards the method is called (line 6).

Listing 2.11: Method dispatch in Ruby

```
1  class Numeric
2    def plus(x)
3      self.+ x
4    end
5  end
6  y = 5.plus 6
7  # 11
```

However there are operations in Ruby which are not overloadable. Listing 2.12 gives several examples of operations (logical and, logical or, conditional statements) which have a behaviour which cannot be changed from within the Ruby language. The advantage of

Listing 2.12: Methods in Ruby which are not overloadable

```
1  false and true
2  # false
3  false or true
4  # true
5  3 < 4 ? '3 is less than 4' : '3 is not less than 4'
6  # "3 is less than 4"
7  if true
8    1
9  else
10   2
11 end
12 # 1
```

this is that programs are easier to understand since certain statements always have the expected behaviour. However this limits the meta programming capabilities of Ruby. This problem will be revisited in Section 3.1.

2.3.3 Dynamic Typing

Ruby uses dynamic typing. In Listing 2.13 the method "`test`" is defined. Since Ruby

Listing 2.13: Dynamic typing in Ruby

```
def test(a, b)
  a + b
end
x = test 3, 5
# 8
x = test 'a', 'b'
# 'ab'
x = test 3, 'b'
# TypeError: String can't be coerced into Fixnum
```

is a dynamically typed language, it is possible to pass objects of any type as parameters. However the object passed for parameter "`a`" must support the method "+" and this method must accept one argument (the definition of method "`test`" is based on this). In a statically typed language an erroneous argument would cause an error message at compile time. In a dynamically typed language it will cause an exception at runtime (line 8).

Listing 2.14 shows how the Ruby standard library allows to combine integers, rational numbers, complex numbers, vectors, and matrices in a seamless way. For example line 6 shows a complex number with rational numbers as components. Note that Ruby uses dynamic typing to switch between native integers and big number representations in order to prevent numeric overflows. Implementing a comparable library in a statically typed language is hard because the type system has to reflect exactly which combinations of data types and operations are supported. If there are n operations, there are potentially 2^n possible composite types, each of them supporting a different subset of operations.

Note that dynamic typing and weak typing are two different properties! Ruby uses strong, dynamic typing.

2.3.4 Exception Handling

Like many other programming languages, Ruby supports exceptions as a means of handling errors without using old-fashioned return codes. Thus the "spaghetti logic" that results from checking return codes can be avoided. That is, exception handling facilitates separation of the code that detects the error and the code for handling the error without the semantic overhead of checking return values (Fulton, 2006). Exception handling is state-of-the art and supported by most modern programming languages (*e.g.* C++, Java, Python, Ruby, and Smalltalk all support it).

Listing 2.15 shows an example where the call to "`File.new`" in line 5 can potentially raise an exception. The exception is handled in the block starting after the "`rescue`" statement (lines 7 to 9). That is, if an error occurs during the execution of lines 2 to 6, the program flow will continue in the rescue clause (lines 7 to 9).

Listing 2.14: Numerical types in Ruby

```ruby
require 'mathn'
require 'complex'
require 'matrix'
x = -8 / 6
# -4/3
y = x * Complex(1, 2)
# Complex(-4/3, -8/3)
z = 2 ** 40
# 1099511627776
y + z
# Complex(3298534883324/3, -8/3)
m = Matrix[[1, 2], [3, 4]]
# Matrix[[1, 2], [3, 4]]
v = Vector[1/2, 1/3]
# Vector[1/2, 1/3]
m * v
# Vector[7/6, 17/6]
z ** 2
# 1208925819614629174706176
```

Listing 2.15: Exception handling in Ruby

```ruby
begin
  print "Enter filename: "
  STDOUT.flush
  file_name = STDIN.readline.delete("\n\r")
  file = File.new file_name, 'r'
  # ...
rescue Exception => e
  puts "Error: #{e.message}"
end
```

2.3.5 Garbage Collector

The Ruby VM uses the *Mark & Sweep* algorithm as a garbage collector. Figure 2.6[15] il-

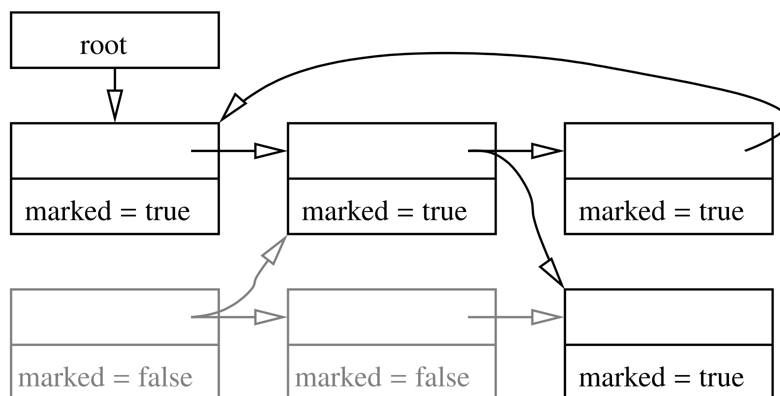


Figure 2.6: Mark & Sweep garbage collector

lustrates the algorithm. Every object has a mark which initially is set to "`false`". Starting from the root context, the graph of references is traversed recursively and every object encountered is marked with "`true`". Afterwards all objects which are still marked "`false`" are deallocated. Cyclical references are not a problem since only objects connected to the root context are marked as "`true`".

2.3.6 Control Structures

Ruby supports control structures for branching (see Figure 2.7) and looping (see Figure 2.8). The syntax of Ruby offers many different ways to write code with the same semantics. This requires the software developer to make frequent choices. But the advantage is that the software developer has more control over the appearance of the code.

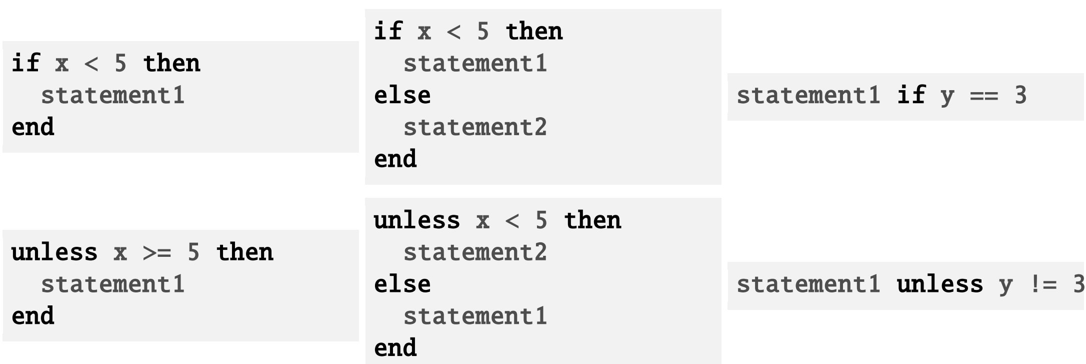


Figure 2.7: Conditional statements in Ruby (Fulton, 2006)

[15] http://www.brpreiss.com/books/opus5/html/page424.html

`while cond do`
`  statement`
`end`
	`until cond do`
	`  statement`
	`end`
		`for x in array do`
		`  statement`
		`end`

`array.each do |x|`
`  statement`
`end`
	`loop do`
	`  statement`
	`  break if cond`
	`end`
		`loop do`
		`  statement`
		`  break unless cond`
		`end`

`for i in 0 .. n - 1 do`
`  statement`
`end`
	`for i in 0 ... n do`
	`  statement`
	`end`
		`array.each_index do |i|`
		`  statement`
		`end`

Figure 2.8: Loop constructs in Ruby (Fulton, 2006)

2.3.7 Mixins

Ruby mixins are a unifying concept for namespaces and interfaces. That is, the "module"-statement in Ruby can be used to declare a namespace. For example the "Math"-module in Ruby contains constants such as "Math::PI" and methods such as "Math::cos" (or "Math.cos"). However Ruby modules can also be used to "mix" methods into a class (Fulton, 2006). When a Ruby module is included in a class, all the module's instance methods become available as methods in the class as well. In that case the mixed-in module effectively behaves as superclass (Thomas et al., 2004). Listing 2.16 shows an example where the mixin "TimesThree" providing the method "three_times" is mixed into the "String" class.

Listing 2.16: Mixins in Ruby

```
module TimesThree
  def three_times
    self + self + self
  end
end
class String
  include TimesThree
end
'abc'.three_times
# "abcabcabc"
```

2.3.8 Closures

Closures are code blocks retaining the variable scope. Listing 2.17 gives an example where a function returns a closure. Even after returning from the method "inc" (lines 1 to 5), the closure returned by that method (lines 2 to 4) still has access to the method parameter "i". Note that while the vertical bar "|" is used to represent the absolute value in mathematical notation, in Ruby syntax it is used to denote the parameters of a block (see Table 2.4 for more detail).

Listing 2.17: Closures in Ruby

```
def inc(i)
  proc do |v|
    v + i
  end
end
t = inc 5
# <Proc:0xb742ed2c@(irb):3>
t.call 3
# 8
[1, 2, 3].collect do |x|
  x ** 2
end
# [1, 4, 9]
[1, 2, 3].inject do |v,x|
  v + x
end
# 6
```

Table 2.4: Ruby notation

	Ruby syntax	mathematical notation
function	“`f = proc { |x| x * x }`”	$f(x) := x^2$
absolute value	“`x.abs`”	$\lvert x \rvert$

2.3.9 Continuations

Ruby supports continuations. A continuation captures the current state of the process in a variable. That is, the continuation offers a way to save the current program pointer and context. Listing 2.18 gives an example where two continuations are used to jump into and out of a method. The order of execution is as follows:

Listing 2.18: Continuations in Ruby

```
require 'continuation'
def test(c2)
  callcc do |c1|
    return c1
  end
  c2.call
end
callcc do |c2|
  c1 = test(c2)
  c1.call
end
```

1. The method “`test`” is defined (lines 2–7) and later called in line 9
2. In line 4 the continuation “`c1`” is returned and execution resumes in line 10

3. Line 10 calls the continuation so that execution resumes in line 6
4. Line 6 calls the continuation "`c2`" so that execution resumes after line 11 (*i.e.* the program terminates)

Basic language features such as exception handling, fibers (cooperative multi-threading), and break statements can all be implemented using continuations.

2.3.10 Introspection

Listing 2.19: Introspection in Ruby

```
x = 5
# 5
x.class
# Fixnum
x.class.class
# Class
x.class.superclass
# Integer
x.is_a?(Fixnum)
# true
Fixnum < Integer
# true
5.respond_to?(:+)
# true
5.methods.grep(/^f/).sort
# ["floor", "freeze", "frozen?"]
```

Introspection allows the program to "see" itself. Listing 2.19 gives some examples querying information about objects and their types. Using the method "`methods`" it is possible to get an array with the names of the methods supported by an object (line 15).

2.3.11 Meta Programming

Ruby supports meta programming. That is, the interpreter provides means of modifying the program during run-time. Listing 2.20 gives a few examples. Using "`eval`" one can evaluate strings (line 1), using "`instance_eval`" (lines 4–6) and "`class_eval`" (lines 10–14) one can evaluate code blocks in the context of a particular object or class, and using "`define_method`" one can create a method with the specified name and code block (lines 11–13).

Ruby meta programming is not as powerful as meta programming in Lisp or Smalltalk. For example some control structures such as while-loops and if-then-else statements cannot be overloaded which means that their behaviour cannot be changed.

Listing 2.20: Meta programming in Ruby

```
eval 'x=5'
# 5
a = [1]
a.instance_eval do
  push 2
end
# [1, 2]
a.send :push, 3
# [1, 2, 3]
Object.const_get('String').class_eval do
  define_method 'test' do
    reverse
  end
end
'abc'.test
# 'cba'
```

2.3.12 Reification

Ruby supports some means of reification. Reification means that the program can modify the behaviour of the interpreter. By overloading the standard method "`method_missing`" one can implement a different behaviour for the case an unknown method was called. For example Listing 2.21 shows a definition of "`Numeric#method_missing`" (lines 2 to 10) which tries to find and call a method with the same prefix as the missing method. When

Listing 2.21: Reification in Ruby

```
class Numeric
  def method_missing(name, *args)
    prefix = Regexp.new("^#{name}")
    full_name = methods.find { |id| id =~ prefix }
    if full_name
      send(full_name, *args)
    else
      super
    end
  end
end
5.mod 2
# calls 5.modulo 2
```

the missing method "`Numeric#mod`" is called in line 12, "`Numeric#method_missing`" is called which will call "`Numeric#modulo`" instead.

There are other hooks for handling missing constant ("`const_missing`"), inheritance changes ("`inherited`"), module inclusions ("`extend_object`"), and method definitions ("`method_added`") (Fulton, 2006).

2.3.13 Ruby Extensions

Ruby has a C-API for developing Ruby extensions[16]. Compared to other APIs such as the Java Native Interface (JNI) it is very easy to use. The problem of the current API is that it does not allow for relocation of allocated memory (Sasada, 2009a) (such as required by compacting garbage collectors).

Listing 2.22 shows a small Ruby extension which upon loading registers the method "`Numeric#logx`". The Ruby native interface provides type definitions (*e.g.* "`VALUE`"), macros (*e.g.* "`NUM2DBL`", "`RUBY_METHOD_FUNC`"), and methods (*e.g.* "`rb_define_method`", "`rb_float_new`", ...) to manipulate the objects of the Ruby interpreter. The Ruby extension needs to define a method where the method's name is the base name of the library prefixed with "`Init_`" (here "`Init_myextension`"). When the library is loaded using the "`require`" statement (see Listing 2.23), this method is called so that the Ruby extension can register new methods (here: the method "`Numeric#logx`") with the Ruby interpreter.

Listing 2.22: Example of a C-extension for Ruby

```
1  // gcc -shared -fPIC -I/usr/lib/ruby/1.8/x86_64-linux \
2  //     -o myextension.so myextension.c
3  #include <ruby.h>
4  #include <math.h>
5
6  VALUE wrap_logx(VALUE self, VALUE x)
7  {
8    return rb_float_new(log(NUM2DBL(self)) / log(NUM2DBL(x)));
9  }
10
11 void Init_myextension(void) {
12   VALUE numeric = rb_const_get(rb_cObject, rb_intern("Numeric"));
13   rb_define_method(numeric, "logx", RUBY_METHOD_FUNC(wrap_logx), 1);
14 }
```

Listing 2.23: Using the extension defined in Listing 2.22

```
require 'myextension'
# true
1024.logx 2
# 10.0
```

2.3.14 Unit Testing

Unit testing is a common practise in the Ruby community (Martin, 2009). There are several unit testing tools for Ruby. The basic Test::Unit[17] framework is part of the Ruby standard library. Effective testing continues to play an important role in removing software

[16] http://www.rubyist.net/~nobu/ruby/Ruby_Extension_Manual.html
[17] http://ruby-doc.org/core/classes/Test/Unit.html

defects (Rajendran, 2002). Ideally unit tests are automated and run after every change. This makes it possible to identify bugs at an early stageor when refactoring the code, *i.e.* when they are being introduced. For example Listing 2.24 shows a unit test for the implementation of "`Array#+`" defined in Listing 2.7.

Listing 2.24: Unit test for "`Array#+`" defined in Listing 2.7

```
require 'test/unit'
class TC_Array < Test::Unit::TestCase
  def test_plus
    assert_equal [[6, 6], [6, 6]],
                 [[1, 2], [3, 4]] + [[5, 4], [3, 2]]
  end
end
# Loaded suite array_plus
# Started
# .
# Finished in 0.002186 seconds.
#
# 1 tests, 1 assertions, 0 failures, 0 errors
```

With sufficient test coverage a test suite can become an executable specification of the behaviour of a program. For example the RSpec[18] project provides sets of tests for each version of Ruby in order to test different implementations of the Ruby VM and measure their degree of compatibility.

2.4 JIT Compilers

The Ruby programming language is an interpreted, pure object-oriented, and dynamically typed general purpose programming language (see Section 2.3). Furthermore Ruby supports closures and meta-programming. Also Ruby has a straightforward API for writing extensions. Finally Ruby currently is on place 11 of the Tiobe Programming Community Index[19]. However in order for developers of machine vision software to take advantage of the productivity gains offered by Ruby, it is necessary to address the performance issue (see Figure 2.4 on page 19).

2.4.1 Choosing a JIT Compiler

Since Ruby supports meta-programming, a JIT compiler in general is indispensable for the perfomant execution of Ruby programs. Table 2.5 shows several software projects which can be used to perform JIT compilation. As one can see, the level of support varies greatly. Of the projects shown in Table 2.5 only the Low Level Virtual Machine (LLVM) project by Lattner (2002) and the RubyInline[20] approach support all the desired properties.

[18] http://rspec.info/
[19] http://www.tiobe.com/index.php/content/paperinfo/tpci/
[20] http://rubyforge.org/projects/rubyinline

Table 2.5: Just-in-time compilers

	LLVM	libJIT	RubyInline	lightning	Asmjit	Xbyak
register allocation	✔	✔	✔			
platform independence	✔	✔	✔	✔		
optimisation	✔		✔			

- **register allocation:** the JIT compiler should provide an abstract machine with an infinite number of registers
- **platform independence:** the JIT compiler should be able to generate code for at least x86, x86-64, and ARM
- **optimisation:** the JIT compiler should provide code optimisation (preferably global optimisation)

Note that Table 2.5 is not complete. However the other JIT compilers are not readily available as a free software library with a dedicated API.

The libJIT API is worth studying because it offers insights in what constitutes a JIT compiler. The Ludicrous project[21] provides a Ruby API to use the libJIT just-in-time compiler library. However the Ruby API does not support pointer operations.

After initially working with libJIT, in the end the RubyInline approach was chosen for the work of this thesis. The C language together with the Ruby extension API is a stable interface for JIT compilation. Also the GNU C compiler offers state of the art optimisation which facilitates competitive performance. The C code is generated at runtime and an ordinary AOT compiler is called to produce a dynamic-link library (DLL) which can be loaded on-the-fly.

Note that the Ricsin project (Sasada, 2009b,c) also provides a means of embedding C code into Ruby programs as well. However it requires the C code to be available AOT.

2.4.2 libJIT API

This section gives a small introduction to libJIT for the interested reader.

Listing 2.25 and Listing 2.26 show a basic array operation implemented directly in C and implemented using libJIT. The operation takes an array as argument (line 12 of Listing 2.26) and increments every element of that array by one (line 23–31). This is performed by sequentially loading each element (line 24), adding one to it (line 25), and

[21] http://rubystuff.org/ludicrous/

Listing 2.25: Array operation implemented in C

```
// g++ -o ctest ctest.c
#include <stdlib.h>
#define SIZE 1000000
unsigned char *f(unsigned char *p, unsigned char one, unsigned char *pend)
{
  for ( ; p != pend; p++ )
    *p += one;
  return p;
}
int main(void)
{
  unsigned char *data = malloc(SIZE * sizeof(unsigned char));
  f(data, 1, data + SIZE);
  free(data);
  return 0;
}
```

writing it back to memory (line 27). The array pointer is incremented (line 28–29) and a conditional branch (line 30–31) is used to continue with the next element until the whole array was processed.

The example demonstrates how the libJIT library exposes the JIT compiler functionality using the following data structures:

1. a JIT context object keeping the functions
2. functions with parameters, instructions, and return value
3. values (virtual registers) of different types
   - integers
   - floating point numbers
   - pointers
4. labels

The methods of the libJIT library expose the functionality of a complete compiler:

1. creating and destructing the context
2. defining functions and their arguments
3. adding **instructions**
   - setting virtual registers to a constant
   - loading values into a virtual register and writing values back to memory
   - logical and mathematical operations
   - control flow statements (*e.g.* conditional branching)

Listing 2.26: Array operation compiled with libJIT

```c
// gcc -o jittest jittest.c -ljit
#include <stdlib.h>
#include <jit/jit.h>
#define SIZE 1000000
int main(void)
{
  // Compile function
  jit_context_t context = jit_context_create();
  jit_context_build_start(context);
  jit_type_t params[3];
  params[0] = jit_type_void_ptr;
  params[1] = jit_type_int;
  params[2] = jit_type_void_ptr;
  jit_type_t signature =
    jit_type_create_signature(jit_abi_cdecl, jit_type_void_ptr,
                              params, 3, 1);
  jit_function_t function = jit_function_create(context, signature);
  jit_value_t p, px, one, end, eq;
  jit_label_t start = jit_label_undefined;
  p = jit_value_get_param(function, 0);
  one = jit_value_get_param(function, 1);
  end = jit_value_get_param(function, 2);
  jit_insn_label(function, &start);
  jit_value_t temp1 = jit_insn_load_relative(function, p, 0, jit_type_ubyte);
  jit_value_t temp2 = jit_insn_add(function, temp1, one);
  jit_value_t temp3 = jit_insn_convert(function, temp2, jit_type_ubyte, 0);
  jit_insn_store_relative(function, p, 0, temp3);
  jit_value_t temp4 = jit_insn_add_relative(function, p, sizeof(jit_ubyte));
  jit_insn_store(function, p, temp4);
  eq = jit_insn_lt(function, p, end);
  jit_insn_branch_if(function, eq, &start);
  jit_insn_return(function, p);
  jit_function_compile(function);
  jit_context_build_end(context);
  // Call function
  unsigned char *data = malloc(SIZE * sizeof(unsigned char));
  void *args[3];
  jit_ptr arg1 = data;
  jit_ubyte arg2 = 1;
  jit_ptr arg3 = data + SIZE;
  jit_ptr result;
  args[0] = &arg1;
  args[1] = &arg2;
  args[2] = &arg3;
  jit_function_apply(function, args, &result);
  free(data);
  // Destruct function
  jit_context_destroy(context);
  return 0;
}
```

4. creating labels to jump to
5. allocating virtual registers and constants
6. compiling and calling methods

2.5 Summary

This chapter has highlighted some basic difficulties with implementing machine vision systems. It was shown that AOT compilation of basic image processing operations is not feasible to due the large number of combinations of methods and parameter types. It was also shown that dynamic typing facilitates much more concise code than static typing.

The properties of the Ruby programming language were discussed. The Ruby programming language is an interpreted language. It is pure object-oriented and dynamically typed. It supports exception handling, mixins, and closures. Furthermore there is support for continuations and meta-programming.

Finally the choice of a JIT compiler was discussed. It was decided to use GNU C as a JIT compiler because the C language represents a stable interface. Also the GNU C compiler comes with a powerful optimiser.

"Programming languages: performance, productivity, generality - pick any two."

Mario Wolczko

"Elegance and familiarity are orthogonal."

Rich Hickey

"In 'Elephant' the programmer would write nothing about an array or database for storing reservations. Arrays of course would be necessary but the compiler would invent them."

John McCarthy

3 Handling Images in Ruby

This chapter is about the core of the work presented in this thesis. Since digital images are represented as 2D arrays, the way array operations are implemented is of great importance when developing machine vision systems. Most existing image processing libraries only support some combinations of operations and array element-types (also see Section 2.2.1). In this chapter a library for manipulating uniform arrays is introduced. The library brings together performance and productivity in an unprecedented way (also see Section 1.3).

- transposing array views (*i.e.* transposing an array "without deep copy" of elements)
- lazy evaluation of array operations (*i.e.* avoiding unnecessary memory-I/O for intermediate results)
- JIT compilation of array operations to achieve real-time performance

A set of objects is introduced to represent arrays, array views, and lazy evaluations in a modular fashion.

- Section 3.1 explains how meta-programming can be used to integrate a JIT compiler into the Ruby programming language
- Section 3.2 introduces memory objects
- Section 3.3 defines native data types based on the memory objects
- Section 3.4 presents uniform arrays based on previously introduced objects
- Section 3.5 introduces a library for describing computer vision algorithms
- Section 3.8 gives a summary of this chapter

It is demonstrated how this approach facilitates succinct implementations of machine vision algorithms.

The software was published under the name Hornetseye. The software is provided as a set of packages (see page iii).

3.1 Transparent JIT Integration

The performance of the Ruby VM is significantly lower than the performance achieved with GNU C (see Section 2.2.3). One can achieve higher performance by introducing operations based on uniform arrays with native element types into the target language.

It is desirable to introduce a JIT compiler without requiring the developer community to accept a modification of the Ruby language. In order to test the meta-programming capabilities of Ruby one can implement the method "`#method_missing`" as shown in Listing 3.1 (lines 7–13) in order to reflect on code instead of executing it. Whenever an object

Listing 3.1: Reflection using missing methods

```
1  class Const
2    attr_accessor :inspect
3    alias_method :to_s, :inspect
4    def initialize(s)
5      @inspect = s.to_s
6    end
7    def method_missing(name, *args)
8      str = "#{ self }.#{ name }"
9      unless args.empty?
10       str += "(#{args.join ', '})"
11     end
12     Const.new str
13   end
14   def coerce(y)
15     return Const.new(y), self
16   end
17 end
```

of type "`Const`" receives a message for a method that is not implemented, it invokes the method "`#method_missing`" instead (Fulton, 2006). The method creates a textual representation of the method call as shown in Listing 3.2. Listing 3.2 shows that the Ruby programming language supports changing the behaviour of unary negation (line 22), binary plus (line 24), and element access (line 26).

However Listing 3.3 shows that meta programming support in Ruby is limited (also see Section 2.3.11). For example the statement in line 34 returns "`a`" instead of "`a.or(b)`". In order to implement transparent JIT integration, it is therefore necessary to restrict the use of the programming language to a subset of Ruby with full meta programming support.

Some operations such as constructing the transpose of a 2D array merely require

Listing 3.2: Example applying reflection to simple operations

```
18 a = Const.new 'a'
19 # a
20 b = Const.new 'b'
21 # b
22 -a
23 # a.-@
24 a + b
25 # a.+(b)
26 a[2]
27 # a[](2)
28 2 * a
29 # 2.*(a)
30 2 * a + b
31 # 2.*(a).+(b)
32 2 * (a + b)
33 # 2.*(a.+(b))
```

Listing 3.3: Limitations of reflection in Ruby

```
34 a or b
35 # a
36 a < b ? a : b
37 # a
38 b = a
39 # a
40 if a > b
41   a -= b
42 end
43 # a.-(b)
44 begin
45   a += 1
46 end until a > b
47 a
48 # a.+(1)
```

the data to be interpreted differently (in this case the order of array indices needs to be swapped). This can be achieved by deferring the calculation (*i.e.* implementing support for *lazy evaluation* (Abelson et al., 1996)). It turns out that this also makes it possible to avoid writing and reading intermediate results and to build tensor expressions.

3.2 Malloc Objects

In order to convert between native representation and Ruby values, the existing methods "`Array#pack`"[1] and "`String#unpack`"[2] of the Ruby core library are used. Listing 3.4 shows how Ruby values get converted to a Ruby string with a native representation of the values (line 3) and back (line 5). The methods "`Array#pack`" and "`String#unpack`" re-

Listing 3.4: Converting arrays to binary data and back

```
1 array = [0x52, 0x75, 0x62, 0x79]
2 # [82, 117, 98, 121]
3 string = array.pack 'c' * 4
4 # "Ruby"
5 array = string.unpack 'c' * 4
6 # [82, 117, 98, 121]
```

quire a template string as parameter which specifies the native data type(s) (see Table 3.1).

Table 3.1: Directives for conversion to/from native representation

Directive	Native Data Type
C	Unsigned byte
c	Byte
d	Double-precision float
f	Single-precision float
I	Unsigned integer
i	Integer
Q	Unsigned long
q	long
S	Unsigned short
s	Short

Since Ruby strings do not support pointer operations or multiple objects viewing the same memory location, "`Malloc`"-objects were introduced (as part of the work presented in this thesis). "`Malloc`" objects allow one to see a chunk of memory as an array of cubby-holes, each containing an 8-bit character (similar as "`vector-ref`" in Scheme (Abelson et al., 1996)). Listing 3.5 shows an example using "`Malloc`" objects. The method

[1] http://www.ruby-doc.org/ruby-1.9/classes/Array.html#M000766
[2] http://www.ruby-doc.org/ruby-1.9/classes/String.html#M000659

Listing 3.5: Manipulating raw data with Malloc objects

```
require 'malloc'
include Hornetseye
m = Malloc.new 4
# Malloc(4)
m.write 'abcd'
n = m + 2
# Malloc(2)
n.read 2
# "cd"
```

"Malloc#+" is used to create a new object viewing a memory location with the given offset (also see Figure 3.1 for a graphical illustration). An explanation of important methods

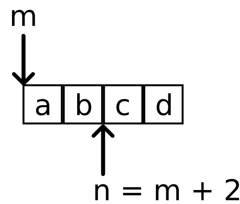


Figure 3.1: Pointer operations (compare Listing 3.5)

is given in Table 3.2.

Table 3.2: Methods for raw memory manipulation

Method	Description
"Malloc.new"	Allocate memory
"Malloc#read"	Read data from memory and return as string
"Malloc#write"	Write string data to memory
"Malloc#+"	Operation for doing pointer arithmetic

3.3 Basic Types

As shown in Section 2.2.3, in general an implementation using native data types can be optimised much better than an implementation using Ruby data types. In order to facilitate implementation of real-time machine vision software, representations of native types starting with booleans are introduced. Note that the type system is not part of the Ruby programming language or the Ruby core library. The type system was developed as part of the work presented in this thesis.

3.3.1 Booleans

In order to annotate a boolean value with type information, one can define a wrapper class for representing native booleans in Ruby (similar to "boxing" in Java[3]). Listing 3.6 shows the application of the class "`BOOL`". The methods "`[]`" and "`[]=`" are used for getting

Listing 3.6: Boxing booleans

```
b = BOOL.new
# BOOL(false)
b.class
# BOOL
b[]
# false
b[] = true
# true
b
# BOOL(true)
```

(line 5) and setting (line 7) the boolean value encapsulated by an object of this class. The method "`BOOL#inspect`" is used to define the textual representation for the output of Interactive Ruby.

Listing 3.7: Constructor short cut

```
b = BOOL true
# BOOL(true)
```

Although it is considered bad style to give Ruby methods a name starting with a capital letter, the language does not forbid this. This is sometimes used to define a short cut to a constructor such as for "`BOOL.new`" which then can be used as shown in Listing 3.7.

In formal language one would define types and variables as shown in Equation 3.1 using the boolean set $\mathbb{B} := \{false, true\}$.

$$b \in \mathbb{B}, b = true \tag{3.1}$$

3.3.2 Integers

One can introduce classes to represent native integers in a similar fashion. However note that there are different types of native integers. That is, current central processing units (CPUs) usually support 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers. Using template classes one can achieve the behaviour shown in Listing 3.8. By specifying the number of bits and the signed-ness one can instantiate an integer classes (*e.g.* lines 1 and 3). See Figure 3.2 for a corresponding visual representation. Furthermore one can define type names and constructor shortcuts for certain integer types (*e.g.* lines 5 and 7).

[3] http://eclipse.org/aspectj/doc/released/adk15notebook/autoboxing.html

Listing 3.8: Template classes for integer types

```
INT 16, UNSIGNED
# USINT
INT(16, UNSIGNED).new 65535
# USINT(65535)
u = USINT.new 65535
# USINT(65535)
u = USINT 65535
# USINT(65535)
```

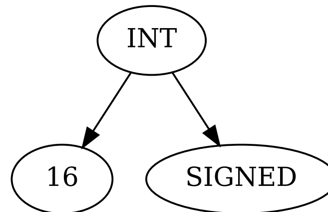


Figure 3.2: Abstract data type for 16-bit unsigned integer

In formal language usually only integers ($\mathbb{Z}$) and non-negative integers ($\mathbb{N}_0$) are distinguished. See Equation 3.2 and Equation 3.3 for examples.

$$i \in \mathbb{Z}, i = -123 \tag{3.2}$$

$$u \in \mathbb{N}_0, u = 234 \tag{3.3}$$

3.3.3 Floating-Point Numbers

Listing 3.9: Boxing floating point numbers

```
FLOAT SINGLE
# SFLOAT
FLOAT DOUBLE
# DFLOAT
FLOAT(SINGLE).new
# SFLOAT(0.0)
DFLOAT.new
# DFLOAT(0.0)
f = DFLOAT Math::PI
# DFLOAT(3.14159265358979)
```

Classes for representing single- and double-precision floating point numbers are implemented in a similar fashion as the integer classes. The behaviour of the implementation is shown in Listing 3.9. See Figure 3.3 for a visual representation of the floating point data types.

The computer's representation of a double-precision floating point number consists of 64 bits. That is, the hardware can represent 2^{64} different numbers. In mathematics the set

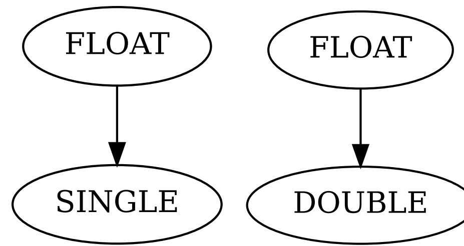


Figure 3.3: Abstract data types for single-precision and double-precision floating point numbers

of real numbers is uncountable and represented by the symbol $\mathbb{R}$ (*e.g.* see Equation 3.4).

$$\pi \in \mathbb{R}, \pi = 3.1415926\ldots \tag{3.4}$$

3.3.4 Composite Numbers

Listing 3.10 demonstrates composite numbers such as "`RGB`" (for representing red, green, blue triplets) in Ruby. Note that the red, green, blue (RGB) class supports left-handed

Listing 3.10: Composite numbers

```
c = RGB 4, 5, 6
# RGB(4,5,6)
c - 3
# RGB(1,2,3)
7 - c
# RGB(3,2,1)
```

(line 5) and right-handed (line 3) subtraction of a scalar. Right-handed subtraction of a scalar is handled by the method "`RGB#-`". In order to handle left-handed subtraction of a scalar, one has to implement the method "`RGB#coerce`". This method gets called by "`Fixnum#-`" when it encounters an unknown data type. All numeric data types of the Ruby core library handle left-handed operations with an unknown data type in this way. This makes it possible to define data types at runtime and integrate them without having to change the implementation of existing data types. Examples of data types which are not part of the Ruby core are matrices and vectors.

Composite numbers are wrapped in a similar way as scalar numbers. Listing 3.11 gives a few examples using the resulting API. Figure 3.4 illustrates how the data types are composed.

3.3.5 Pointers

Furthermore a pointer type is introduced. For example see Figure 3.5 showing how a pointer to a double precision floating point number is composed. The pointer encap-

Listing 3.11: Boxing composite numbers

```
c = UBYTERGB RGB(1, 2, 3)
# UBYTERGB(RGB(1,2,3))
c = RGB(INT(16, UNSIGNED)).new RGB(1, 2, 3)
# USINTRGB(RGB(1,2,3))
c = RGB(FLOAT(SINGLE)).new RGB(0.1, 0.2, 0.3)
# SFLOATRGB(RGB(0.1,0.2,0.3))
```

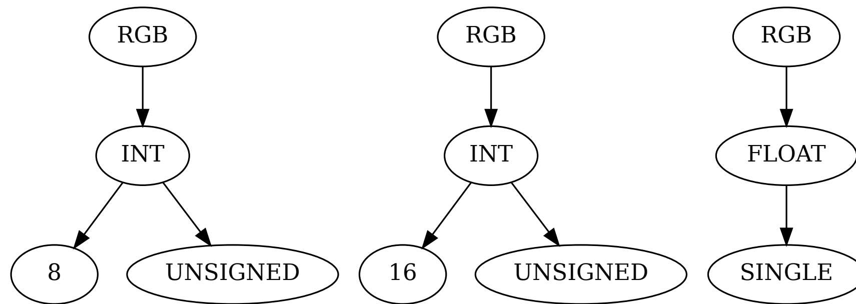


Figure 3.4: Composite types for unsigned byte RGB, unsigned short int RGB, and single-precision floating point RGB values

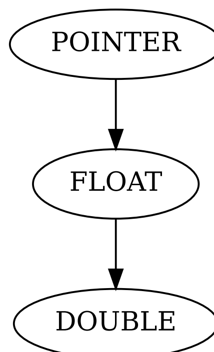


Figure 3.5: Abstract data type for pointer to double precision floating point number

sulates a "Malloc"-object. One can see a pointer object as a delayed fetch operation. That is, the pointer object supports lazy evaluation of fetch operations. In Chapter 3.2 it was shown how the methods "Array#pack" and "String#unpack" convert between a Ruby value and a string with the native representation. Listing 3.12 shows how the methods "Pointer_#store" (line 3) and "Pointer_#demand" (lines 9 and 11) can be used to write/read boxed values to/from memory. That is, the pointer object handles reading and writing native values using the type information stored in its class. The method "Pointer_#lookup" (line 5) facilitates pointer operations in Ruby which are used as a basis for defining arrays later on.

Listing 3.12: Pointer objects

```
1  p0 = Pointer(DFLOAT).new Malloc.new(16)
2  # *(DFLOAT)(Malloc(16))
3  p0.store DFLOAT(Math::PI)
4  # DFLOAT(3.14159265358979)
5  p1 = p0.lookup INT(1), INT(1)
6  # *(DFLOAT)(Malloc(8))
7  p1.store DFLOAT(Math::E)
8  # DFLOAT(2.71828182845905)
9  p0.demand
10 # DFLOAT(3.14159265358979)
11 p1.demand
12 # DFLOAT(2.71828182845905)
```

3.3.6 Ruby Objects

Listing 3.13: Boxing arbitrary Ruby objects

```
OBJECT 'Hello!'
# OBJECT("Hello!")
OBJECT 2 ** 127 - 1
# OBJECT(170141183460469231731687303715884105727)
```

In order to integrate the system of native types with the existing Ruby type system, a means of specifying non-native types is introduced. The class "OBJECT" is used to encapsulate Ruby objects which do not have a native representation as shown in Listing 3.13. Note that in practise one needs to define a class based on Ruby arrays which supports reading/writing values and pointer operations in a similar way to "Malloc" but providing this functionality for Ruby objects.

3.4 Uniform Arrays

A uniform array can be seen as a special case of a function accepting one or more indices as arguments and returning an array element as result. For example Equation 3.5 shows a

function returning the square of a value[4].

$$f:\begin{cases}\{1,2,3\} & \rightarrow & \mathbb{Z}\\ x & \mapsto & x^2\end{cases} \tag{3.5}$$

Since the function is defined on a finite set (*i.e.* the set $\{1,2,3\}$), it can instead be defined using the function values $f(1)$, $f(2)$, and $f(3)$. That is, f can also be represented as an array, vector, or sequence (*e.g.* the vector $f' = (1\ 4\ 9)^\top$ with $f'_1 = 1$, $f'_2 = 4$, and $f'_3 = 9$). Using the insight that matrices, arrays, vectors, and sequences are essentially different notations for functions, one can develop a representation which unifies these concepts.

This section shows how variables, lambda terms, lookup objects, and the basic types introduced in the previous section can be used as building blocks to construct multi-dimensional arrays.

3.4.1 Variable Substitution

Listing 3.14: Variable objects and substitution

```
v1 = Variable.new INT
# Variable(INT)
v2 = Variable.new INT
# Variable(INT)
v2.subst v1 => INT(7)
# Variable(INT)
v2.subst v2 => INT(7)
# INT(7)
INT(5).subst v1 => INT(7)
# INT(5)
```

Listing 3.14 shows application of the variable class. The method "`Variable#subst`" accepts a Ruby hash as argument (lines 5 and 7). It returns either a replacement for the object (line 8) or the object itself (line 6). By furthermore defining methods "`INT_#subst`", "`FLOAT_#subst`", "`BOOL_#subst`", ... and have them return "`self`", one can achieve the behaviour shown in Listing 3.14 where applying a substitution to a scalar value has no effect (line 9). In formal language the substitutions shown in Listing 3.14 is specified using square brackets as shown in Equation 3.6 (Church, 1951; McCarthy, 1960; Barendregt and Barendsen, 1984).

$$\begin{aligned} v_2[v_1 := 7] &\equiv v_1\\ v_2[v_2 := 7] &\equiv 7\\ 5[v_1 := 7] &\equiv 5\end{aligned} \tag{3.6}$$

[4]here the formal notation of the form $f:\begin{cases}X & \rightarrow & Y\\ x & \mapsto & f(x)\end{cases}$ according to Heuser (1991) is used with X and Y being the domain and codomain of f

3.4.2 Lambda Terms

In λ-calculus the basic operations are *abstraction* and *application* (Church, 1951; McCarthy, 1960; Barendregt and Barendsen, 1984). The concept of *application* is also known as β-reduction. For example Equation 3.7 shows the square function $x \to x * x$ written in lambda notation. A dot is used to separate the bound variable x from the term $x * x$.

$$\lambda x.x * x \tag{3.7}$$

Equation 3.8 is an example of β-reduction. Here the square function is applied to the number 3. The bound variable x is replaced with the value 3.

$$(\lambda x.x * x)3 \to_\beta (x * x)[x := 3] \to 3 * 3 \to 3 \tag{3.8}$$

A "**Lambda**" class was implemented in order to facilitate manipulation of lambda expressions in Ruby. Listing 3.15 shows application of the class "**Lambda**". The identity

Listing 3.15: Lambda abstraction and application

```
v = Variable.new INT
# Variable(INT)
l = Lambda.new v, v
# Lambda(Variable(INT),Variable(INT))
l.element INT(7)
# INT(7)
```

function is used as an example to demonstrate abstraction and application. The variable "v" becomes a bound variable in the λ-term "l" (line 3). The same example written using the formal system of λ-calculus is shown in Equation 3.9.

$$(\lambda x.x)7 \to_\beta x[x := 7] \equiv 7 \tag{3.9}$$

3.4.3 Lookup Objects

In principle one can view a one-dimensional (1D) array as a function accepting one argument. That is, given a (typed) memory pointer p and a stride s one can define an arbitrary array a as shown in Equation 3.10.

$$a : \begin{cases} \{0, 1, \ldots, n-1\} & \to & \mathbb{R} \\ i & \mapsto & fetch_{\mathbb{R}}(p + i * s) \end{cases} \tag{3.10}$$

The function $fetch_{\mathbb{R}}$ is a pristine array representing physical memory (*e.g.* memory of camera holding image data). Note that the array index is limited to a certain range, *i.e.* $i \in \{0, 1, \ldots, n-1\}$.

Listing 3.16 shows how an array is constructed using a pointer, a lookup object, and

Listing 3.16: Implementing arrays as lazy lookup

```
1  v = Variable.new INDEX(INT(5))
2  # Variable(INDEX(INT(5)))
3  p = Pointer(DFLOAT).new Malloc.new(40)
4  a = Lambda.new v, Lookup.new(p, v, INT(1))
5  a[1] = 4.2
6  # 4.2
7  a[1]
8  # 4.2
```

a lambda term. Line one creates an index variable. The size of the array becomes part of the type information of the variable (*i.e.* "`INDEX(INT(5))`"). Line 3 allocates 40 bytes of memory and creates a pointer object for interpreting the memory as a sequence of double-precision floating point numbers. In line 4 the variable and the memory are used to create a lookup object. Finally the variable is bound using a lambda term.

By defining the Ruby operators "`[]`" and "`[]=`" one can implement element access. Element-access internally is implemented as beta-reduction (see Section 3.4.2). In practise the construction of arrays is hidden using the method "`Sequence.new`" as shown in Listing 3.17.

Listing 3.17: Uniform arrays

```
a = Sequence.new DFLOAT, 5
a[1] = 4.2
# 4.2
a[1]
# 4.2
```

3.4.4 Multi-Dimensional Arrays

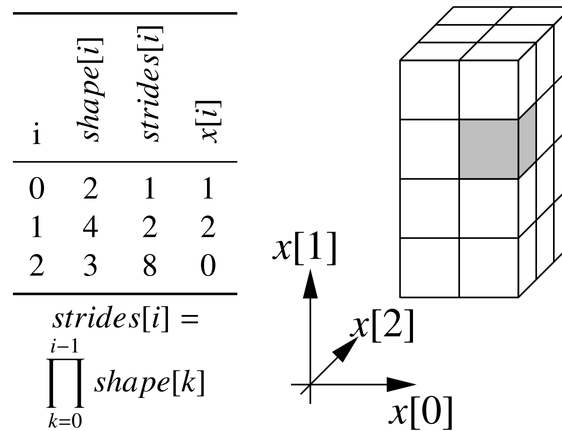
One can treat a multi-dimensional array as a function accepting *multiple* indices as arguments. Equation 3.11 shows the definition of an arbitrary 3D array m using a memory pointer p and the strides s_0, s_1, and s_2.

$$m : \begin{cases} \{0,1,\ldots,n_2-1\} \times \ldots \times \{0,1,\ldots,n_0-1\} & \rightarrow \mathbb{R} \\ (x_0\ x_1\ x_2) & \mapsto fetch_{\mathbb{R}}(p + s_2\,x_2 + s_1\,x_1 + s_0\,x_0) \end{cases} \tag{3.11}$$

Internally the multi-dimensional array is represented by recursively nesting "`Lambda`" and "`Lookup`" objects. By extending the methods "`[]`" and "`[]=`" in a suitable manner and by introducing the method "`MultiArray.new`", one can achieve the behaviour shown in Listing 3.18. The statement "`MultiArray.new UBYTE, 2, 4, 3`" in line 1 allocates a 3D array with $2 \times 4 \times 3$ elements. Figure 3.6 illustrates the shape and the strides of the 3D array. The corresponding formal notation is given in Equation 3.12.

Listing 3.18: Multi-dimensional uniform arrays

```
1  m = MultiArray.new UBYTE, 2, 4, 3
2  m[1, 2, 0] = 3
3  # 3
4  m[1, 2, 0]
5  # 3
6  m[0][2][1]
7  # 3
8  m[0][2][1] = 5
9  # 5
10 m[1, 2, 0]
11 # 5
```



i	$shape[i]$	$strides[i]$	$x[i]$
0	2	1	1
1	4	2	2
2	3	8	0

$strides[i] = \prod_{k=0}^{i-1} shape[k]$

Figure 3.6: Shape and strides for a 3D array

$$m : \begin{cases} \{0,1\} \times \{0,1,2,3\} \times \{0,1,2\} & \rightarrow & \{0,1,\dots,255\} \\ (x_0\ x_1\ x_2) & \mapsto & fetch_{\{0,1,\dots,255\}}(p + s_2\,x_2 + s_1\,x_1 + s_0\,x_0) \end{cases} \tag{3.12}$$

Note that the methods "`[]`" and "`[]=`" are defined in a fashion which makes it possible to specify either the major index first (*e.g.* "`m[0][2][1]`" in line 6 of Listing 3.18) or the minor index first (*e.g.* "`m[1,2,0]`" in line 4 of Listing 3.18). It is also possible to access the same element using hybrid notation (*e.g.* "`m[2,0][1]`"). This makes it possible to interpret the 3D array of scalars as a 2D array m' of 2D vectors or as a 2D array m'' array of 1D functions. See Equation 3.13 for the corresponding formal domain specifications (the actual array definitions are omitted here for brevity).

$$\begin{aligned} m' &: \{0,1,2,3\} \times \{0,1,2\} \rightarrow \{0,1,\dots,255\}^2 \\ m'' &: \{0,1,2,3\} \times \{0,1,2\} \rightarrow (\{0,1\} \rightarrow \{0,1,\dots,255\}) \end{aligned} \tag{3.13}$$

3.4.5 Array Views

In the Ruby language the declaration "`1 .. 5`" is a short cut for "`Range.new 1, 5, false`". Ranges are similar to intervals in mathematics, *e.g.* $[1,5]$ in this case[5]. On the other hand "`1 ... 6`" is a short cut for "`Range.new 1, 6, true`" which is a right-open interval. The mathematical notation is $[1,6)$ in that case[6]. The method "`[]`" (and also the method "`[]=`") can be extended so that they accept "`Range`" objects.

Listing 3.19 shows some examples where sub-arrays of a 2D array are specified using ranges. Note that while the vertical bar "`|`" is used to represent the absolute value in mathematical notation, in Ruby syntax it is used to denote the parameters of a block (see Table 2.4 for more detail). Note that the order of the dimensions of the resulting array is

Listing 3.19: Array views

```
m = lazy(5, 4) { |i,j| i + j * 5 }
# MultiArray(INT,2):
# [ [ 0, 1, 2, 3, 4 ],
#   [ 5, 6, 7, 8, 9 ],
#   [ 10, 11, 12, 13, 14 ],
#   [ 15, 16, 17, 18, 19 ] ]
u = m[1 .. 2]
# MultiArray(INT,2):
# [ [ 5, 10 ],
#   [ 6, 11 ],
#   [ 7, 12 ],
#   [ 8, 13 ],
#   [ 9, 14 ] ]
v = u[1 .. 3]
# MultiArray(INT,2):
# [ [ 6, 7, 8 ],
#   [ 11, 12, 13 ] ]
v = m[1 .. 3, 1 .. 2]
# MultiArray(INT,2):
# [ [ 6, 7, 8 ],
#   [ 11, 12, 13 ] ]
v = m[1 .. 2][1 .. 3]
# MultiArray(INT,2):
# [ [ 6, 7, 8 ],
#   [ 11, 12, 13 ] ]
```

cycled n times where n is the number of ranges specified. For example after declaring "`m`" in line 1, the values $5, 6, 7, \ldots$ appear in one row of the array. However extracting the sub array "`u`" (line 7) changes the order of the array indices so that the same elements now appear in one column.

Figure 3.7 illustrates the advantage of this semantics: It allows one to specify the major indices first by invoking the operation "`[]`" twice and specifying the ranges separately (*e.g.* "`m[1 .. 2][1 .. 3]`"). But it also allows one to specify the minor indices first by invoking the operation "`[]`" only once (*e.g.* "`m[1 .. 3, 1 .. 2]`").

[5] $[a,b]$ denotes the closed interval from a to b. That is, $[a,b] := \{x \in \mathbb{R} | a \le x \le b\}$

[6] $[a,b)$ denotes the right-open interval from a to b. That is, $[a,b) := \{x \in \mathbb{R} | a \le x < b\}$

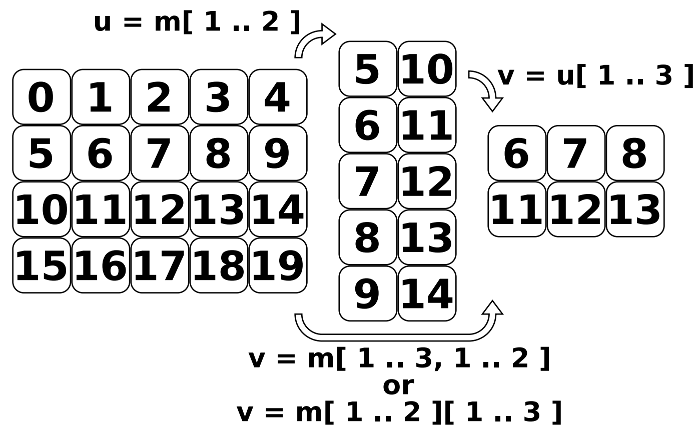


Figure 3.7: Extracting array views of a 2D array

The equivalent formal notation for the definition of the sub arrays v is shown in Equation 3.14.

$$v : \begin{cases} \{0,1,2\} \times \{0,1\} & \rightarrow & \mathbb{Z} \\ (x_0\ x_1) & \mapsto & m(x_0+1, x_1+1) \end{cases} \tag{3.14}$$

3.5 Operations

Table 3.3 gives an overview of generic array operations which can be found in computer vision algorithms. "a" and "b" are parameters, and "r" is the result. Each of "r", "a", and "b" denotes a scalar or an array depending on the operation. In some cases a function "f" is involved. "i" and "j" are array indices. Note that this set of array operations is not complete since it does not cover operations involving multi-dimensional arrays (*e.g.* matrix-vector multiplication). Also it is not minimal since an array-scalar binary function can be created using a constant array and an array-array binary function.

3.5.1 Constant Arrays

As shown in Section 3.4.3, one can see arrays as functions where the indices are the arguments of that function. A 1D constant array *carr* where the value of each element is 0 becomes a constant function as shown in Equation 3.15.

$$carr := \lambda i.0 \tag{3.15}$$

Lines 4, 7, and 11 of Listing 3.20 show different ways of using the lambda class introduced in Chapter 3.4.2 to create a function returning 0. Using closures it is possible to define the method "`lazy`". The method accepts the shape of the array as argument and a closure for generating the array elements. For example one can construct a constant array using the statement "`lazy(5) { 0 }`". Note that it is possible to define multi-dimensional arrays by nesting calls to "`lazy`".

Table 3.3: Generic set of array operations

	operation	array index
read element	r =a(b)	-
read sub-array	r(i) =a(i+b)	i
constant array	r(i) =a	i
index array	r(i) =i	i
unary function	r(i) =f(a(i))	i
scalar-array binary function	r(i) =f(a,b(i))	i
array-scalar binary function	r(i) =f(a(i),b)	i
array-array binary function	r(i) =f(a(i),b(i))	i
accumulate	r =f(r,a(i))	i
warp/mask	r(i) =a(b(i))	i
unmask	r(b(i))=a(i)	i
downsampling	r(i) =a(b*i)	i
upsampling	r(b*i) =a(i)	i
integral	r(i) =r(i-1)+a(i)	i
map	r(i) =b(a(i))	i
histogram	r(a(i))=r(a(i))+1	i
weighted hist.	r(a(i))=r(a(i))+b(i)	i
convolution	r(i) =r(i)+a(i-j)*b(j)	i,j

3.5.2 Index Arrays

In practise index arrays can be useful (*e.g.* for computing warp fields). In the 1D case the simplest index array is the identity function *id* shown in Equation 3.16.

$$id := \lambda i.i \tag{3.16}$$

Listing 3.21 gives a few examples of index arrays and how they can be constructed using the "`lazy`" method. The index arrays "`x`" and "`y`" (lines 1 and 6) are useful for computing warp fields in practise. The class method "`indgen`" is a short cut for creating a multi-dimensional index array.

3.5.3 Type Matching

Type matching is introduced in order to conveniently convert Ruby values to native values. Figure 3.8 illustrates how a Ruby value is passed on to the "`Match#fit`" method of each native type class:

- "`Sequence_#fit`" rejects the value because it is not an array
- "`RGB_#fit`" rejects the value because it is not an RGB value
- "`FLOAT_#fit`" rejects the value because it is not a floating point number

Listing 3.20: Constant arrays

```
Lambda.new Variable.new(INDEX(INT(5))), UBYTE.new(0)
# Sequence(UBYTE):
# [ 0, 0, 0, 0, 0 ]
lazy(5) { 0 }
# Sequence(UBYTE):
# [ 0, 0, 0, 0, 0 ]
lazy(3, 2) { 0 }
# MultiArray(UBYTE,2):
# [ [ 0, 0, 0 ],
#   [ 0, 0, 0 ] ]
lazy(2) { lazy(3) { 0 } }
# MultiArray(UBYTE,2):
# [ [ 0, 0, 0 ],
#   [ 0, 0, 0 ] ]
```

Listing 3.21: Index arrays

```
x = lazy(3, 3) { |i,j| i }
# MultiArray(INT,2):
# [ [ 0, 1, 2 ],
#   [ 0, 1, 2 ],
#   [ 0, 1, 2 ] ]
y = lazy(3, 3) { |i,j| j }
# MultiArray(INT,2):
# [ [ 0, 0, 0 ],
#   [ 1, 1, 1 ],
#   [ 2, 2, 2 ] ]
idx = lazy(3, 3) { |i,j| i + j * 3 }
# MultiArray(INT,2):
# [ [ 0, 1, 2 ],
#   [ 3, 4, 5 ],
#   [ 6, 7, 8 ] ]
MultiArray(INT,2).indgen 3, 3
# MultiArray(INT,2):
# [ [ 0, 1, 2 ],
#   [ 3, 4, 5 ],
#   [ 6, 7, 8 ] ]
```

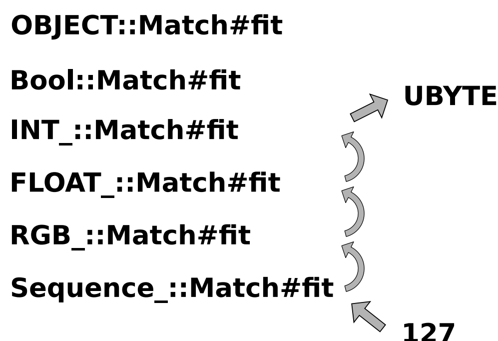


Figure 3.8: Type matching

- "`INT_#fit`" accepts the value because it is an integer between -2^{63} and $2^{64}-1$. The method returns "`UBYTE`" because this is the smallest native integer sufficient to hold the value "`127`"

That is, the method "`Match#fit`" determines whether the corresponding native type is the best fit for the given Ruby value. Listing 3.22 shows how this simplifies declarations of native arrays.

Listing 3.22: Type matching

```
Sequence[1, 2, 3]
# Sequence(UBYTE):
# [ 1, 2, 3 ]
Sequence[1, 2, -1]
# Sequence(BYTE):
# [ 1, 2, -1 ]
Sequence[1.5, RGB(1, 2, 3)]
# Sequence(DFLOATRGB):
# [ RGB(1.5,1.5,1.5), RGB(1.0,2.0,3.0) ]
MultiArray[[1, 2], [3, 4]]
# MultiArray(UBYTE,2):
# [ [ 1, 2 ],
#   [ 3, 4 ] ]
```

3.5.4 Element-Wise Unary Operations

Listing 3.23 shows element-wise operations in Ruby. The method "`Array#collect`" accepts a closure as argument which computes $2\,x+1$ for each element x of the argument in this example. In order to facilitate more concise code, one can define methods which

Listing 3.23: Element-wise unary operations using "`Array#collect`"

```
a = [1, 2, 3]
a.collect { |x| x * 2 + 1 }
# [3, 5, 7]
```

make it possible to use a shorter notation as shown in Listing 3.24. The problem however is that eager evaluation will create intermediate results. For example evaluation of the statement "`a * 2 + 1`" will create the intermediate result "`a * 2`" and then add "`1`" to each element. That is, an array to store the intermediate result is allocated and the values are written to it only to be read back again immediately. If one were to use lazy evaluation this would not happen (see Figure 3.9). That is, in order to have concise code as well as performant code, it is important to facilitate lazy evaluation.

Listing 3.25 shows how *lazy* unary operations can be represented internally using objects (*e.g.* of type "`ElementWise(proc { |x| -x }, :-@, proc { |t| t })`"). That is, , here the unary operation is characterised using following information

Listing 3.24: Short notation for element-wise operations

```
class Array
  def *(scalar)
    collect { |x| x * scalar }
  end
  def +(scalar)
    collect { |x| x + scalar }
  end
end
# Example use
a = [1, 2, 3]
a * 2 + 1
# [3, 5, 7]
```

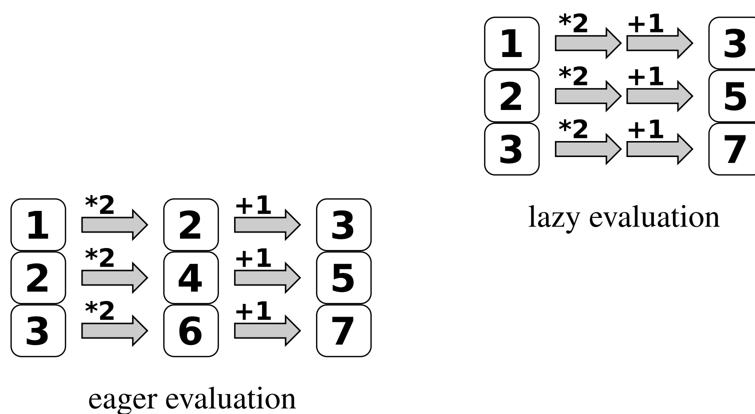


Figure 3.9: Avoiding intermediate results by using lazy evaluation

Listing 3.25: Internal representation of unary operations

```
1  s = Sequence(INT)[1, 2, 3]
2  # Sequence(INT):
3  # [ 1, 2, 3 ]
4  ElementWise(proc { |x| -x }, :-@, proc { |t| t.contiguous }).new s
5  # Sequence(INT):
6  # [ -1, -2, -3 ]
7  lazy { -s }
8  # Sequence(INT):
9  # [ -1, -2, -3 ]
10 lazy { |i| -s[i] }
11 # Sequence(INT):
12 # [ -1, -2, -3 ]
13 -s
14 # Sequence(INT):
15 # [ -1, -2, -3 ]
16 (-Sequence[:a, 1, :b])[1]
17 # NoMethodError: undefined method ‘-@’ for :a:Symbol
18 lazy { -Sequence[:a, 1, :b] }[1]
19 # -1
```

1. "`proc { |x| -x }`": A closure (see Section 2.3.8 with the operation to apply to each element
2. "`:-@`": A unique symbol to identify the operation (for debugging purposes and to compute a cache key)
3. "`proc { |t| t }`": A closure for computing the data type of the result

This representation is preserved when doing lazy evaluation (*e.g.* when executing the statement "`lazy { -s }`"). When doing eager evaluation (the default) the operation will be applied to each element and the result is stored in a new array (*e.g.* when executing the statement "`-s`"). The difference between lazy and eager operations becomes visible when using operations in combination with element access. Line 16 Listing 3.25 shows that eager evaluation of "`-Sequence[:a, 1, :b]`" will throw an exception since "`:a`" is a symbol and does not support negation. However it is possible to perform the operation lazily and extract the second element of the result without computing the other elements as shown in line 18.

Equation 3.17 shows how one can use formal language to describe the application of a unary operator to an index array as an example.

$$-(\lambda i.i) = \lambda i. -i \tag{3.17}$$

Note that in practise it is not necessary to normalise the expression. For example it is not a problem if "`lazy { |i| -s[i] }`" and "`lazy { -s }`" have different lazy representations.

3.5.5 Element-Wise Binary Operations

Listing 3.26: Element-wise binary operations using "`Array#collect`" and "`Array#zip`"

```
a = [1, 2, 3]
# [1, 2, 3]
b = [-1, 1, 3]
# [-1, 1, 3]
a.zip b
# [[1, -1], [2, 1], [3, 3]]
a.zip(b).collect { |x,y| x + y }
# [0, 3, 6]
```

Listing 3.26 shows how one can perform element-wise addition of two arrays in Ruby. The method "`Array#zip`"[7] merges elements of both arrays. Afterwards "`Array#collect`" performs element-wise addition using the array of pairs as input.

Binary operations with support for *lazy* evaluation are introduced in a similar fashion as unary operations. That is, they are internally represented as objects (*e.g.* of type

[7] http://www.ruby-doc.org/core/classes/Array.html#M002198

"`ElementWise(proc { |x,y| x + y }, :+, proc { |t,u| t.coercion u })`") and the representation is only preserved in lazy mode. In practise binary operations occur as array-array-, array-scalar-, and scalar-array-operations as one can see in Listing 3.27. In

Listing 3.27: Internal representation of binary operations

```
s = Sequence(INT)[1, 2, 3]
# Sequence(INT):
# [ 1, 2, 3 ]
ElementWise(proc { |x,y| x + y }, :+, proc { |t,u| t.coercion u }).
  new s, UBYTERGB(RGB(1, 2, 3))
# Sequence(INTRGB):
# [RGB(2,3,4), RGB(3,4,5), RGB(4,5,6)]
s + RGB(1, 2, 3)
# Sequence(INTRGB):
# [ RGB(2,3,4), RGB(3,4,5), RGB(4,5,6) ]
RGB(1, 2, 3) + s
# Sequence(INTRGB):
# [ RGB(2,3,4), RGB(3,4,5), RGB(4,5,6) ]
s + s
# Sequence(INT):
# [ 2, 4, 6 ]
```

a similar fashion as in the case of unary operations, the binary operation is characterised by the following information

1. "`proc { |x,y| x + y }`": A closure with the operation to apply to each pair of elements
2. "`:+`": A unique symbol to identify the operation (for debugging purposes and to compute a cache key)
3. "`proc { |t,u| t.coercion u }`": A closure for deriving the data type of the result

Equation 3.18 show examples of formal representation of array-scalar, scalar-array, and array-array binary operations.

$$
\begin{aligned}
(\lambda i.f(i)) + c &\equiv \lambda i.f(i) + c \\
c + (\lambda i.f(i)) &\equiv \lambda i.c + f(i) \\
(\lambda i.f(i)) + (\lambda i.g(i)) &\equiv \lambda i.f(i) + g(i)
\end{aligned}
\tag{3.18}
$$

Furthermore there are unary and binary methods. For example "`Math.sqrt(x)`" will compute the square root of x, "`Math.atan2(y, x)`" will compute the polar angle of $(x\ y)^\top$. However methods and operators are just different forms of notation for element-wise operations. That is, they are handled the same way as element-wise operations.

3.5.6 LUTs and Warps

3.5.6.1 LUTs

Like any other array, one can understand a lookup table (LUT) as a function. Element-wise lookup can be understood as element-wise application of that function. See Listing 3.28 for example.

Listing 3.28: Element-wise application of a LUT

```
MultiArray[1, 2, 0].lut Sequence[:a, :b, :c]
# Sequence(OBJECT):
# [ :b, :c, :a ]
```

In general it is desirable to support multi-dimensional LUTs, *i.e.* using vectors with multiple dimensions as lookup index. Furthermore it is possible to have LUTs with multi-dimensional arrays as elements. Equation 3.19 shows application of the LUT l to the array a.

$$b(\vec{x})(\vec{y}) = l(a(\vec{x}))(\vec{y}) \tag{3.19}$$

The array a is interpreted as an n_1-dimensional array with n_2-dimensional vectors as elements. The n_2-dimensional vectors are used for a lookup with l. See Equation 3.20 for the types of a, l, and the result b.

$$a : \mathbb{N}_0{}^{n_1} \to \mathbb{N}_0{}^{n_2},\ l : \mathbb{N}_0{}^{n_2} \to (\mathbb{N}_0{}^{n_3} \to K),\ b : \mathbb{N}_0{}^{n_1} \to (\mathbb{N}_0{}^{n_3} \to K) \tag{3.20}$$

Listing 3.29 shows how to use a LUT for computing a pseudo colour image. First

Listing 3.29: Creating a pseudo colour image

```
1  class Numeric
2    def clip(range)
3      [[self, range.begin].max, range.end].min
4    end
5  end
6  colours = Sequence.ubytergb 256
7  for i in 0...256
8    hue = 240 - i * 240.0 / 256.0
9    colours[i] =
10     RGB(((hue - 180).abs -  60).clip(0...60) * 0xFF / 60.0,
11         (120 - (hue - 120).abs).clip(0...60) * 0xFF / 60.0,
12         (120 - (hue - 240).abs).clip(0...60) * 0xFF / 60.0)
13 end
14 MultiArray.load_ubyte(ARGV[0]).lut(colours).save_ubytergb ARGV[1]
```

the "`Numeric`" class is extended with a method for clipping numbers to a certain range (lines 1–5). Then a colour palette with 256 elements is allocated (line 6) and populated with values (lines 7–13). Figure 3.10 shows a plot of the colour channels of the resulting palette. Finally an image is loaded, element-wise lookup is performed, and the resulting

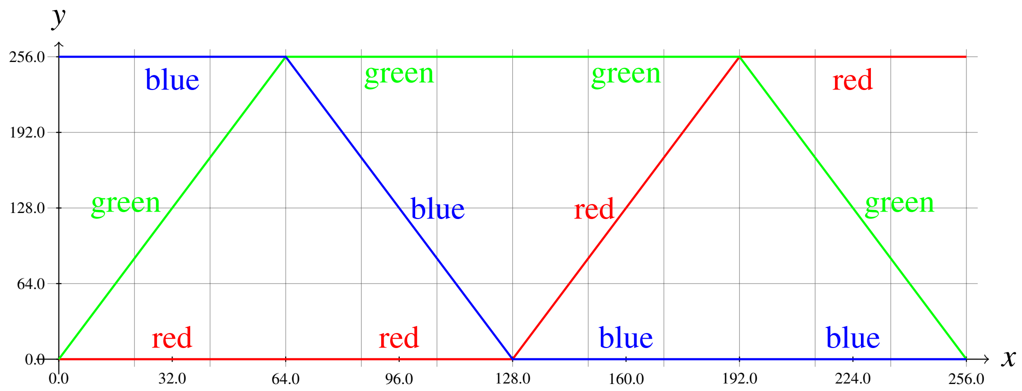
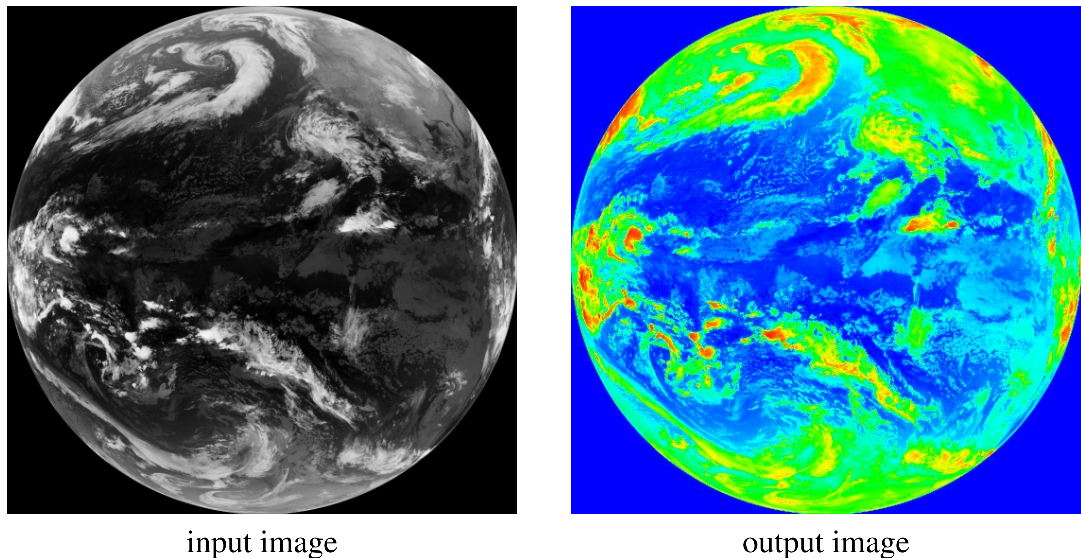


Figure 3.10: Pseudo colour palette

image is written to disk (line 14). Figure 3.11 shows a thermal image used as input and the corresponding pseudo colour image created with afore mentioned program.



input image | output image

Figure 3.11: Thermal image displayed with pseudo colours (source: NASA Visible Earth)

Listing 3.29 shows how the concepts introduced in previous chapters apply to implementation of image processing algorithms. The implementation of the pseudo colour visualisation is both concise and real-time capable.

3.5.6.2 Warps

An image warp is essentially the same as element-wise lookup. That is, the image is used as a LUT and the input data is a 2D array of warp vectors. Listing 3.30 shows how one can convert a topographical image from equirectangular to azimuthal projection. Note that in this example the components of the warp vectors are two separate arrays (*i.e.* "`angle`" in line 6 and "`radius`" in line 5). The array class of the Ruby language was extended so that the expression "`[angle, radius].lut(img)`" (line 7) can be used to specify the arrays

Listing 3.30: Warp from equirectangular to azimuthal projection

```
img = MultiArray.load_ubytergb 'world.jpg'
w, h = *img.shape
c = 0.5 * (h - 1)
x, y = lazy(h, h) { |i,j| i - c }, lazy(h, h) { |i,j| j - c }
radius = lazy { Math.hypot(x, y).to_int }
angle = lazy { ((Math.atan2(x, y) / Math::PI + 1) * w / 2).to_int }
[angle, radius].lut(img).save_ubytergb 'polar.jpg'
```

holding the components of the warp vectors.

Figure 3.12 illustrates how the components of the warp vectors are constructed. The

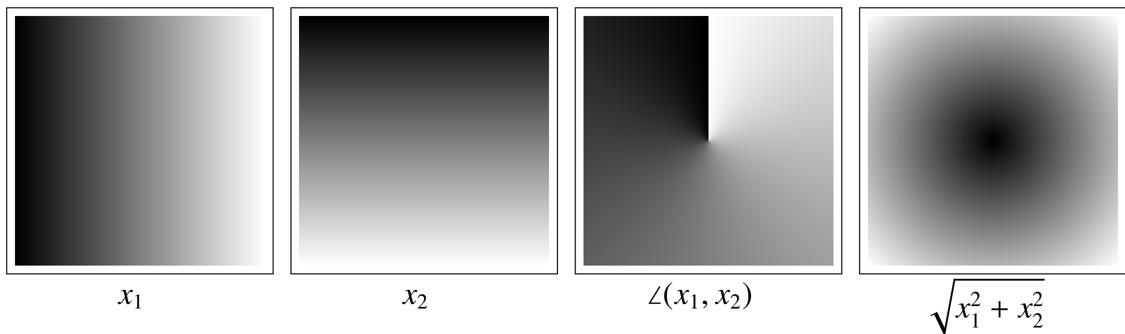


Figure 3.12: Visualised components of warp vectors

visualisation was inspired by Baker and Matthew (2004). Arrays with x_1 and x_2 values are used to construct arrays with the angle and radius of the azimuthal projection. The result of applying the warp is shown in Figure 3.13.

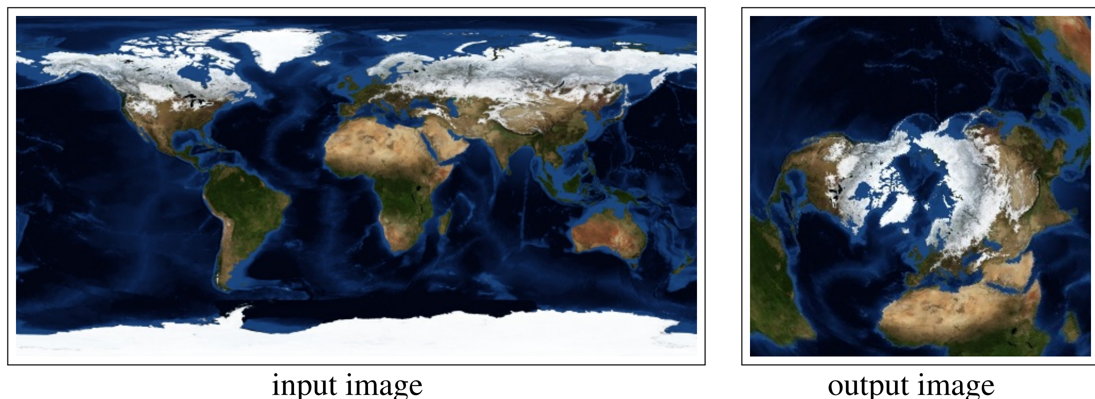


Figure 3.13: Warping a satellite image (source: NASA Visible Earth)

3.5.7 Injections

Apart from element-wise functions (*e.g.* LUTs) one frequently encounters the fold-left (*foldl*) operation in functional programming. The operation is also known as the *reduce* part of Google's MapReduce implementation (Lämmel, 2008). Given a binary function

$A \to B \to B$ and a value of type B, the $foldl$ operation yields a function taking an array as argument which does cumulative application of the binary function (Hutton, 1999) (see Equation 3.21).

$$\begin{aligned} &(A \to B \to B) \to B \to ((\mathbb{N}_0 \to A) \to B) \\ &foldl(f, v)([x_1, x_2, x_3, \ldots]) = f(\ldots f(f(f(v, x1), x2), x3), \ldots) \end{aligned} \tag{3.21}$$

Note that $foldl$ is left-associative. When applying the fold operation to non-associative operations (*e.g.* division or subtraction) it is important to distinguish between left-associative ($foldl$) and right-associative folding (fold-right ($foldr$)). In Ruby the $foldl$ operation is known as *injection*. Listing 3.31 shows two ways of specifying an injection for computing the product of all elements of an array. The equivalent formal expression

Listing 3.31: Left-associative fold operation in Ruby

```
[2, 3, 5, 7, 11].inject(1) { |a,b| a * b }
# 2310
[2, 3, 5, 7, 11].inject 1, :*
# 2310
```

using the product symbol is shown in Equation 3.22.

$$\prod_{i \in \{1,2,\ldots,5\}} s_i \text{ where } s := (2\ 3\ 5\ 7\ 11) \tag{3.22}$$

Listing 3.32 shows how an injection can be represented internally using an object of class "`Inject`". The closure (or nameless function) "`proc { |a,b| a * b }`" is con-

Listing 3.32: Internal representation of injections

```
1  s = Sequence[2, 3, 5, 7, 11]
2  # Sequence(UBYTE):
3  # [ 2, 3, 5, 7, 11 ]
4  v = Variable.new INDEX(s.size)
5  # Variable(INDEX(INT(5)))
6  v1 = Variable.new INT
7  # Variable(INT)
8  v2 = Variable.new INT
9  # Variable(INT)
10 block = proc { |a,b| a * b }.call v1, v2
11 # *(Variable(INT),Variable(INT))
12 inject = Inject.new s.element(v), v, INT(1), block, v1, v2
13 # INT(2310)
14 Sequence[2, 3, 5, 7, 11].inject(1) { |a,b| a.to_int * b }
15 # 2310
```

verted to an object by passing it "`Variable`" objects as parameters (line 10). The end of the listing shows how the operation might be invoked in practise (line 14).

The image processing operations "`min`", "`max`", and "`range`" can be implemented using $foldl$ (see Listing 3.33). Injection can be implemented recursively. The injection

Listing 3.33: Various cumulative operations based on injections

```
m = MultiArray[[2, 3, 5], [7, 11, 13]]
# MultiArray(UBYTE,2):
# [ [ 2, 3, 5 ],
#   [ 7, 11, 13 ] ]
m.min
# 2
m.max
# 13
m.range
# 2..13
```

is applied to each dimension separately. For example Figure 3.14 shows the recursive computation of the sum of an array. Other associative operations such as computing the

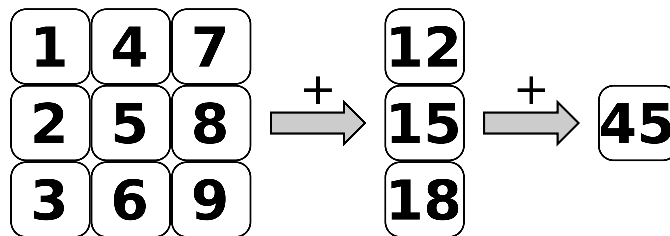


Figure 3.14: Recursive implementation of injection (here: sum)

maximum, minimum, or the product can be computed in the same recursive manner as well.

Using Ruby closures and "`Variable`" objects it is possible to develop a concise notation for sums. For example 1D and 2D sums as shown in Equation 3.23.

$$
\begin{aligned}
u_i &:= \sum_{j=1}^{2} m_{ji} \\
v &:= \sum_{j=1}^{2}\sum_{i=1}^{3} m_{ji},\ m \in \mathbb{Z}^{3\times 2}
\end{aligned}
\tag{3.23}
$$

The corresponding Ruby code to compute u and v is shown in Listing 3.34. Note that

Listing 3.34: Concise notation for sums of elements

```
m = MultiArray[[2, 3, 5], [7, 11, 13]]
# MultiArray(UBYTE,2):
# [ [ 2, 3, 5 ],
#   [ 7, 11, 13 ] ]
u = sum { |i| m[i] }
# Sequence(UBYTE):
# [ 9, 14, 18 ]
v = sum { |i,j| m[i,j] }
# 41
```

Listing 3.35: Tensor operations in Ruby (equivalent to Listing 2.6)

```
a = MultiArray.dfloat 3, 3
b = MultiArray.dfloat 3, 3
r = eager { |i,k| sum { |j| a[i,j] * b[j,k] } }
```

it is not even necessary to specify the ranges for the index variable because they can be inferred from the element access.

3.5.8 Tensor Operations

The "`lazy`" method and the "`sum`" operator (see Chapter 3.5.1 and Chapter 3.5.7) in combination with unary and binary element-wise operations (see Chapter 3.5.4 and Chapter 3.5.5) are sufficiently generic to implement tensor operations. Listing 2.6 shows a tensor operation implemented using the FTensor C++ library. The equivalent Ruby implementation is shown in Listing 3.35. The "`eager`" performs the operation lazily the same way as the "`lazy`" method does. However it forces the result to be computed and stored in a new array.

3.5.9 Argmax and Argmin

The definitions of the operations argmax and argmin are given in Equation 3.24 and Equation 3.25

$$\underset{\vec{x}}{\operatorname{argmin}}\,(f(\vec{x})) := \{\vec{x}\,\big|\,\forall\vec{x}' : f(\vec{x}) \leq f(\vec{x}')\} \tag{3.24}$$

$$\underset{\vec{x}}{\operatorname{argmax}}\,(f(\vec{x})) := \{\vec{x}\,\big|\,\forall\vec{x}' : f(\vec{x}) \geq f(\vec{x}')\} \tag{3.25}$$

where $\vec{x}$ is any coordinate in the n-dimensional input array f. The operations return the argument for which the function attains the maximum or the minimum value. Listing 3.36 shows three operations:

1. 1D argmax for locating the maximum of each row (line 10)
2. Instruction for extracting the maxmimum of each row (line 10)
3. 2D argmax for locating the maximum (line 13)

The listing demonstrates how to construct a warp for extracting the maximum of each row of the input array. Note that the argument operation returns an array (or several arrays) if the input array has more dimensions than the argument.

The argument functions are implemented recursively using warps as illustrated in Figure 3.15. First the argument maximum of each row is located. The locations are used as coordinates for a warp to select the maximum of each row. Using the warped array, the

Listing 3.36: Argument maximum

```
m = MultiArray[[1, 2, 3], [4, 5, 9], [7, 8, 3], [7, 6, 4]]
# MultiArray(UBYTE,2):
# [ [ 1, 2, 3 ],
#   [ 4, 5, 9 ],
#   [ 7, 8, 3 ],
#   [ 7, 6, 4 ] ]
argmax { |i| m.unroll[i] }
# [Sequence(INT):
# [ 2, 2, 1, 0 ]]
m.warp argmax { |i| m.unroll[i] }.first, lazy(4) { |i| i }
# Sequence(UBYTE):
# [ 3, 9, 8, 7 ]
argmax { |i,j| m[i,j] }
# [2, 1]
```

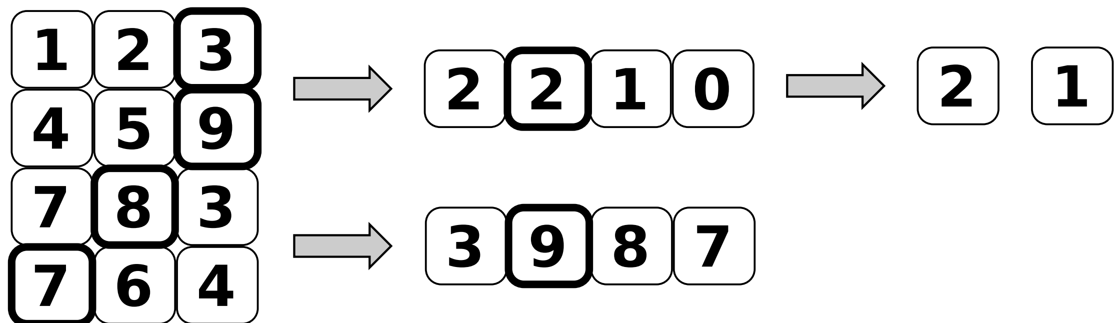


Figure 3.15: Recursive implementation of argument maximum

column of the maximum can be determined. The column in turn is used to determine the row of the maximum by selecting the appropriate element from the array of argument maxima determined earlier.

3.5.10 Convolution

Convolution filters are commonly used to define image features. Equation 3.26 gives the definition of a 1D convolution (discrete case).

$$c_i := \sum_k a_k\, b_{o+k-i},\ \text{where } o = \left\lceil \frac{n}{2} \right\rceil \tag{3.26}$$

In the continuous case a convolution integral as shown in Equation 3.27 is used.

$$(a \otimes b)(\vec{x}) := \int a(\vec{x})\, b(\vec{z} - \vec{x})\, \mathrm{d}\vec{z} \tag{3.27}$$

Once can perform a 1D convolution by first computing a product table and then computing diagonal sums of it as shown in Figure 3.16. Furthermore one can see the compu-

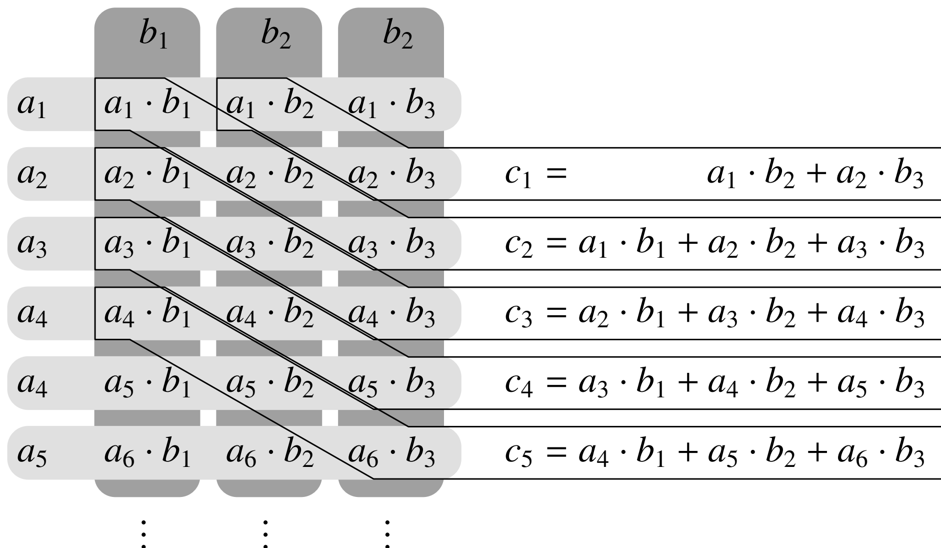


Figure 3.16: Diagonal injection

tation of diagonal sums as a special case of diagonal injection. Listing 3.37 shows how this is done in Ruby. The internal representation of diagonal injections is similar to the one of ordinary injections (see Chapter 3.5.7). Variables are used to convert the closure (declared in line 7) to an object (line 27). The diagonal injection is constructed in line 29. The array index of the result is bound using a lambda expression (line 33). In practise diagonal injections and convolutions are implemented as methods. The end of the listing shows how they can be invoked (lines 36 and 39).

Two-dimensional convolutions can be calculated using the same concept. In this case a four-dimensional product table and two sequential diagonal injections are used. See

Listing 3.37: One-dimensional convolutions in Ruby

```
a = Sequence[0, 1, 0, 0, 0, 2, 0, 0]
# Sequence(UBYTE):
# [ 0, 1, 0, 0, 0, 2, 0, 0 ]
b = Sequence[1, 2, 3]
# Sequence(UBYTE):
# [ 1, 2, 3 ]
product = a.table(b) { |x,y| x * y }
# MultiArray(UBYTE,2):
# [ [ 0, 0, 0 ],
#   [ 1, 2, 3 ],
#   [ 0, 0, 0 ],
#   [ 0, 0, 0 ],
#   [ 0, 0, 0 ],
#   [ 2, 4, 6 ],
#   [ 0, 0, 0 ],
#   [ 0, 0, 0 ] ]
i = Variable.new Hornetseye::INDEX(nil)
# Variable(INDEX(INT(nil)))
j = Variable.new Hornetseye::INDEX(nil)
# Variable(INDEX(INT(nil)))
k = Variable.new Hornetseye::INDEX(nil)
# Variable(INDEX(INT(nil)))
v1 = Variable.new product.typecode
# Variable(UBYTE)
v2 = Variable.new product.typecode
# Variable(UBYTE)
block = proc { |x,y| x + y }.call v1, v2
# +(Variable(UBYTE),Variable(UBYTE))
term = Diagonal.new product.element(j).element(k), i, j, k, nil, block, v1, v2
# ...
i.size = j.size
# INT(8)
c = Lambda.new i, term
# Sequence(UBYTE):
# [ 1, 2, 3, 0, 2, 4, 6, 0 ]
c = lazy { |j,i| a[i] * b[j] }.diagonal { |x,y| x + y }
# Sequence(UBYTE):
# [ 1, 2, 3, 0, 2, 4, 6, 0 ]
c = a.convolve b
# Sequence(UBYTE):
# [ 1, 2, 3, 0, 2, 4, 6, 0 ]
```

Listing 3.38: Two-dimensional convolutions in Ruby

```
m = MultiArray.int(6, 4).fill!; m[1, 1] = 1; m[4, 2] = 2; m
# MultiArray(INT,2):
# [ [ 0, 0, 0, 0, 0, 0 ],
#   [ 0, 1, 0, 0, 0, 0 ],
#   [ 0, 0, 0, 0, 2, 0 ],
#   [ 0, 0, 0, 0, 0, 0 ] ]
f = MultiArray(INT,2).indgen 3, 3
# MultiArray(INT,2):
# [ [ 0, 1, 2 ],
#   [ 3, 4, 5 ],
#   [ 6, 7, 8 ] ]
m.convolve f
# MultiArray(INT,2):
# [ [ 0, 1, 2, 0, 0, 0 ],
#   [ 3, 4, 5, 0, 2, 4 ],
#   [ 6, 7, 8, 6, 8, 10 ],
#   [ 0, 0, 0, 12, 14, 16 ] ]
```

Listing 3.38 for a demonstration of 2D convolutions. An array "m" (line 1) and a filter "f" (line 7) are declared. The array is convolved with the filter in line 12.

Figure 3.17 shows a moving average filter applied to an image. This filter operation

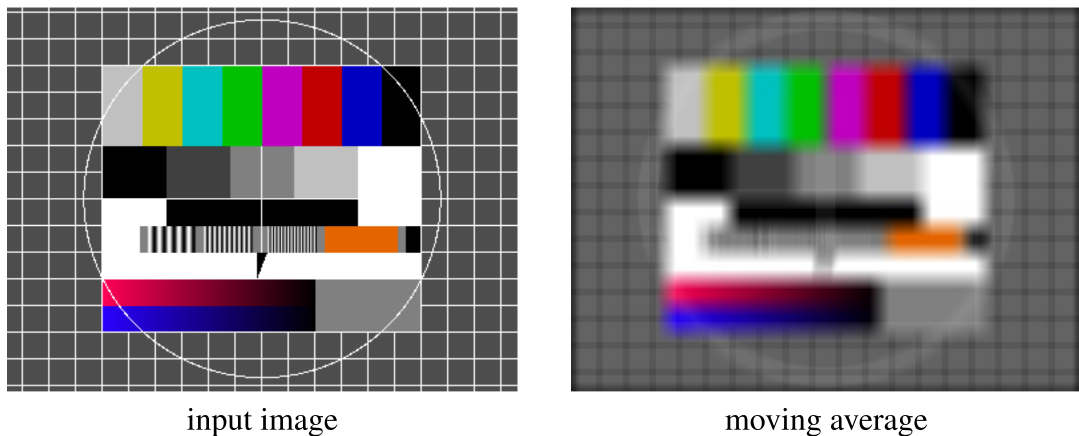

input image
moving average

Figure 3.17: Applying a moving average filter to an image

can be implemented using convolutions as shown in Listing 3.39 (gamma 1.0 was assumed). The 2D moving average filter is a separable filter, *i.e.* it can be separated into two consecutive 1D convolutions, which is computationally more efficient.

3.5.11 Integral

An integral image (or a summed area table) is an array with the sum of all elements with indices lower or equal to the current set of indices. For example Equation 3.28 shows the

Listing 3.39: Moving average filter implemented using convolutions

```
n = 15
ma_x = lazy(n, 1) { 1 }
ma_y = lazy(1, n) { 1 }
(MultiArray.load_ubytergb(ARGV[0]).to_usintrgb.
 convolve(ma_x).convolve(ma_y) / n ** 2).save_ubytergb ARGV[1]
```

definition of a 1D integral array and Equation 3.29 the definition of a 2D integral array.

$$b_i = \sum_{k=0}^{i} a_k \tag{3.28}$$

$$b_{j,i} = \sum_{l=0}^{j} \sum_{k=0}^{i} a_{l,k} \tag{3.29}$$

Integral images can be used to quickly compute the sum of elements in a rectangular region of the input data. If the sum of elements for many rectangles is required it can be computationally more efficient to make use of an integral image (*e.g.* as in the real-time face detection algorithm by Viola and Jones (2001)). In practise integral arrays are computed iteratively. The 1D case is shown in Equation 3.30.

$$\begin{aligned} b_0 &= a_0 \\ b_i &= b_{i-1} + a_i \end{aligned} \tag{3.30}$$

The 1D algorithm can be used recursively to compute multi-dimensional integral arrays. The 2D case is shown in Equation 3.31.

$$\begin{aligned} b_{0,i} &= \sum_{k=0}^{i} a_{0,k} \\ b_{j,i} &= b_{j-1,i} + \sum_{k=0}^{i} a_{j,k} \end{aligned} \tag{3.31}$$

Figure 3.18 gives an example of recursion for computing a 2D integral image. First each row is integrated and then integration is performed on each column of the resulting array.

Note that lazy computation of integral images is inefficient since the computation of an element can depend on the values of all other elements in the worst case. Therefore integral images are always computed eagerly and the result is stored in memory.

Listing 3.40 uses an integral image to apply a moving average filter to an image (assuming gamma 1.0). Note that the image boundaries are omitted here for simplicity so that the output image is smaller than the input image (see Figure 3.19).

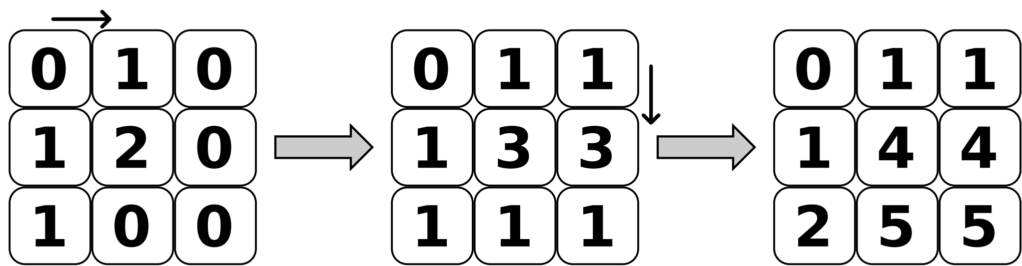


Figure 3.18: Recursive implementation of integral image

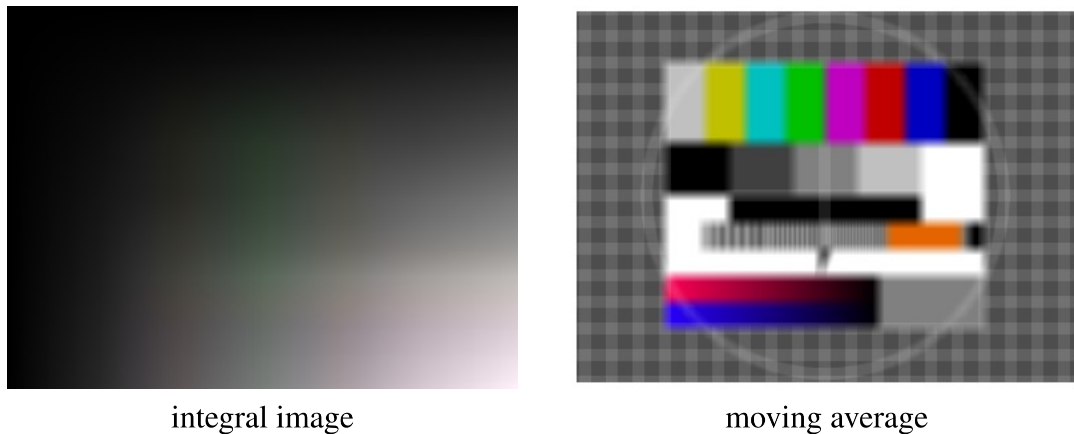


Figure 3.19: Computing a moving average filter using an integral image

Listing 3.40: Moving average filter implemented using an integral image

```
img = MultiArray.load_ubytergb ARGV[0]
w, h = *img.shape; n = 9
int = img.to_intrgb.integral
a = int[n ...     w, n ...     h]
b = int[0 ... w - n, n ...     h]
c = int[n ...     w, 0 ... h - n]
d = int[0 ... w - n, 0 ... h - n]
((a - b - c + d) / n ** 2).save_ubytergb ARGV[1]
```

Listing 3.41: Conditional selection as element-wise operation

```
class Node
  def sgn
    (self<0).conditional -1, (self>0).conditional(1,0)
  end
end
Sequence[-3,-2,-1,0,1,2,3].sgn
```

Listing 3.42: Injection with a conditional

```
Sequence[false, true, true, false].inject(0) { |a,b| a + b.conditional(1, 0) }
# 2
```

3.5.12 Masking/Unmasking

Piecewise functions such as the signum function shown in Equation 3.32 can be implemented using the element-wise conditional function as shown in Listing 3.41.

$$sgn(x) := \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.32}$$

The conditional function simply is a ternary element-wise operation.

However it is not possible to combine a conditional operation with an injection in an efficient manner, *e.g.* when computing the number of elements for which a condition is true (see Equation 3.33).

$$\sum_{x \in M} 1 \text{ where } M \subset \mathbb{Z} \tag{3.33}$$

When using the conditional function as shown in Listing 3.42, the number of additions will be 4. An efficient implementation would only require 1 or 2 additions.

Listing 3.43 shows an implementation using a masking operation to perform an injection on a subset of an array. A constant array is masked (line 2) and the sum of elements is computed (line 5). This implementation only needs 1 addition. Masking is especially useful when doing complex operations on a small subset of an array.

Listing 3.44 shows an example using a masking and an unmasking operation. A mask to select elements greater than zero is created (line 4). Then the operation 6 divided by

Listing 3.43: Injection on a subset of an array

```
1 s = Sequence[false, true, true, false]
2 m = lazy(s.size) { 1 }.mask s
3 # Sequence(UBYTE):
4 # [ 1, 1 ]
5 m.inject { |a,b| a + b }
6 # 2
```

Listing 3.44: Element-wise operation on a subset of an array

```
s = Sequence[0, 1, 2, 3]
# Sequence(UBYTE):
# [ 0, 1, 2, 3 ]
m = s > 0
# Sequence(BOOL):
# [ false, true, true, true ]
(6 / s.mask(m)).unmask m
# Sequence(UBYTE):
# [ 0, 6, 3, 2 ]
```

Listing 3.45: Two-dimensional histogram

```
MultiArray[[0, 0], [2, 1], [1, 1]].histogram 3
# Sequence(UINT):
# [ 2, 3, 1 ]
MultiArray[[0, 0], [2, 1], [1, 1]].histogram 3, 3
# MultiArray(UINT,2):
# [ [ 1, 0, 0 ],
#   [ 0, 1, 1 ],
#   [ 0, 0, 0 ] ]
```

x is performed on the masked array and the result is "unmasked" (line 7). That is, the implementation performs an element-wise operation on a subset of an array.

Note that the design of the Intel Larrabee architecture for parallel processing also includes masking and unmasking operations. They are called "vcompress" and "vexpand" (Abrash, 2009).

3.5.13 Histograms

A histogram is a record of the number of pixels in an image or a region that fall into particular quantization buckets in some colour space (Forsyth and Ponce, 2003). Equation 3.34 shows the definition of the histogram of a.

$$h(\vec{y}) = \mathcal{H}\{a\}(\vec{y}) = \sum_{\vec{x}} \begin{cases} 1 & a(\vec{x}) = \vec{y} \\ 0 & \text{otherwise} \end{cases} \quad \text{where } a : \mathbb{N}_0{}^{n_1} \to \mathbb{N}_0{}^{n_2},\ h : \mathbb{N}_0{}^{n_2} \to \mathbb{N}_0 \qquad (3.34)$$

In a similar way as with integral images, the computational complexity of computing a single element is the same as the complexity of computing all elements of a histogram. Therefore histograms are computed eagerly as well.

Listing 3.45 shows a Ruby program which first computes a 1D histogram (with three elements) of the values of a 2D 2×3 array (line 1). Listing 3.45 furthermore demonstrates that the 2D array given as input can be interpreted as a 1D array of 2D vectors resulting in a 2D (3 × 3) histogram (line 4).

Listing Listing 3.46 shows computation of a colour (here RGB) histogram. The expression "`reference >> 2`" (left-shift by two) divides the red (R), green (G), and blue

Listing 3.46: Histogram segmentation
```
reference = MultiArray.load_ubytergb 'neon.png'
hist = (reference >> 2).histogram(64, 64, 64).convolve lazy(5, 5, 5) { 1 }
img = MultiArray.load_ubytergb 'neontetra.jpg'
seg = (img >> 2).lut(hist).convolve lazy(9, 9) { 1 }
(img * seg).normalise.save_ubytergb 'fish.jpg'
```

(B) values by 4 resulting in 64 quantisation steps for each channel (line 2). The 64×64×64 reference histogram is convolved with a 3D moving average to reduce noise (line 2). The reference histogram then is used as a LUT with another image (line 4). Finally a moving average filter is applied to the result (line 4). Some example data is given in Figure 3.20. The input image is a picture of a tropical aquarium with Neon Tetra fish. The histogram of

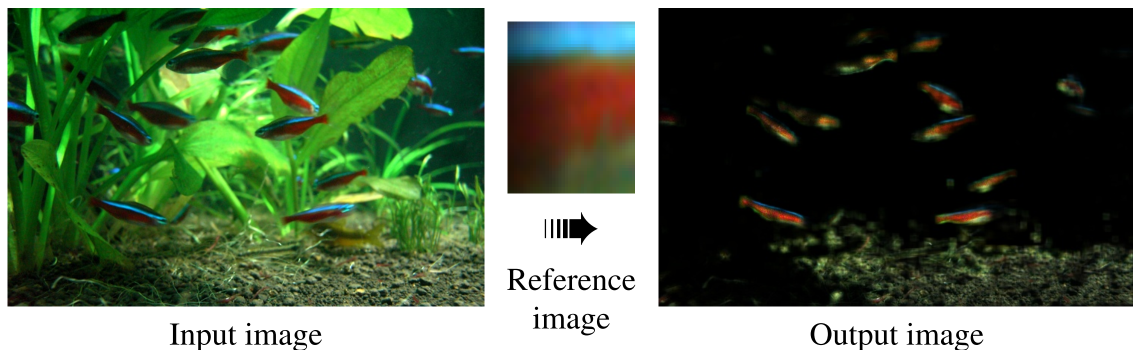


Figure 3.20: Histogram segmentation example

the reference image is used as a LUT. The result is used to create an output image which highlights the areas which have similar colours as found in the reference image. Although the floor of the aquarium is not fully discarded, the algorithm is able to highlight most of the fish visible in the image.

The histograms of the red, green, and blue colour channels of the reference image are shown in Figure 3.21, *i.e.* three 1D histogram. The histogram segmentation example however uses a 3D histogram which is more difficult to visualise. See Figure 3.22 for a visualisation of the 3D histogram using a visual representation inspired by Barthel (2006) (the visualisation was created using the POV-Ray ray tracer (POVRay, 2005)). Note that in practise it is often preferable to use a colour space which is independent of the luminosity (*e.g.* hue, saturation, and value (HSV)).

3.6 JIT Compiler

In order to achieve real-time performance, each array operation is converted to a C method on-the-fly. The C method is compiled to a Ruby extension (*i.e.* a shared object (SO) file or a DLL). This library is loaded dynamically and the method is registered with the Ruby VM. Then the method is called with the appropriate parameters (also see Chapter 2.3.13).

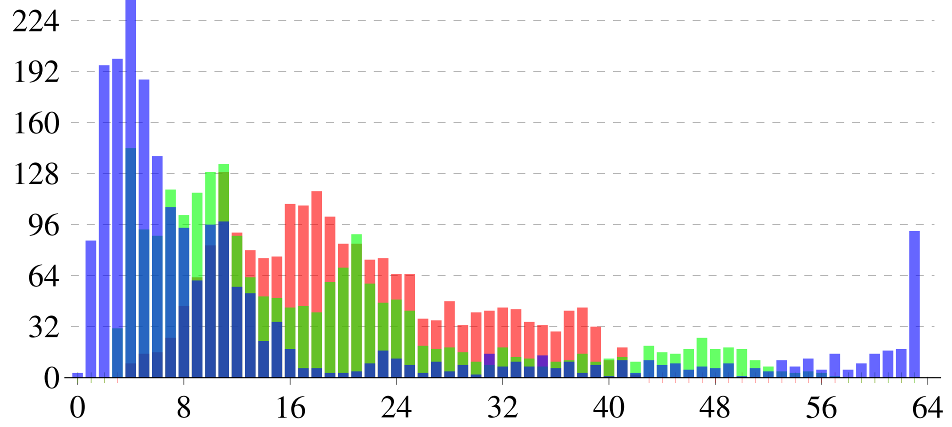


Figure 3.21: Histograms of the red, green, and blue colour channel of the reference image

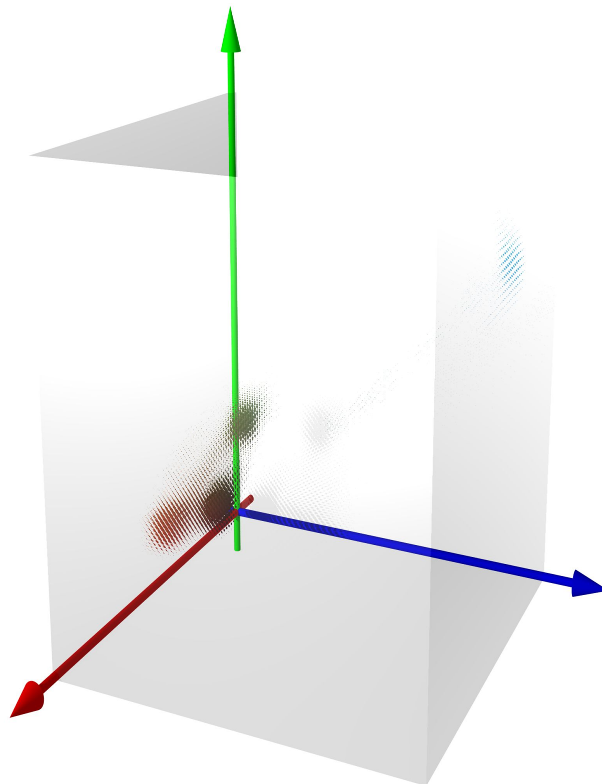


Figure 3.22: 3D histogram

Listing 3.47: Lazy negation of integer

```
1 a = INT 5
2 # INT(5)
3 b = ElementWise(proc { |x| -x }, :-@).new a
4 # Continued in Listing 3.48 ...
```

Listing 3.48: Stripping values from an expression

```
5 # ...  continuing from Listing 3.47
6 r = Pointer(INT).new
7 term = Store.new r, b
8 variables, values, skeleton = term.strip
9 # [[Variable(*(INT)), Variable(INT)], [*(INT)(Malloc(4)), INT(5)],
10 #  Store(Variable(*(INT)),-@(Variable(INT)))]
11 types = variables.collect { |var| var.meta }
12 # [*(INT), INT]
13 # Continued in Listing 3.49 ...
```

The following sections explain how the C code is generated.

3.6.1 Stripping Terms

Listing 3.47 shows a simple negation of an integer value. A lazy negation is instantiated explicitly to avoid immediate evaluation in Ruby. The equivalent in formal notation is shown in Equation 3.35.

$$a = -5, b = -a \tag{3.35}$$

In practise it is necessary to compute the values of every expression at some point and store the value(s) in memory. In this case it can be done by introducing the operation $store_{\mathbb{Z}}$ which has a side-effect on the memory location pointed to by r as shown in Equation 3.36.

$$store_{\mathbb{Z}}(r, -a) \tag{3.36}$$

3.6.2 Compilation and Caching

In order to generate the corresponding C code, the expression of Listing 3.47 is stripped as shown in Listing 3.48 in line 4. The stripping operations results in variables, the corresponding values, and a skeleton of the expression. To avoid repeated compilation of the same expression, a unique descriptor of the expression will be used as a method names. Listing 3.49 shows how the method name is computed. The hash table "`labels`" with labels for the variables is created (line 2) in order to distinguish expressions where only the order of variables is different. This hash table is used to generate a unique descriptor for the stripped expression (line 4). Since method names in C cannot contain special characters, a translation table is used to replace special characters with alphanumerical ones (lines 6–7).

Listing 3.49: Generating a unique descriptor

```
14 # ...  continuing from Listing 3.48
15 labels = Hash[*variables.zip((0 ... variables.size).to_a).flatten]
16 # {Variable(*(INT))=>0, Variable(INT)=>1}
17 descriptor = skeleton.descriptor labels
18 # "Store(Variable0(*(INT)),-@(Variable1(INT)))"
19 method_name = ('_' + descriptor).tr('(),+\-*/%.@?~&|^<=>',
20                                      '0123\456789ABCDEFGH')
21 # "_Store0Variable0050INT112490Variable10INT111"
22 # Continued in Listing 3.50 ...
```

Listing 3.50: Using special objects to generate C code

```
23 # ...  continuing from Listing 3.49
24 c = GCCContext.new 'extension'
25 f = GCCFunction.new c, method_name, *types
26 subst = Hash[*variables.zip(f.params).flatten]
27 # {Variable(*(INT))=>*(INT)(param0), Variable(INT)=>INT(param1)}
28 skeleton.subst(subst).demand
29 f.compile
30 # Continued in Listing 3.52 ...
```

The variables of the stripped expression are substituted with parameter objects (*i.e.* "`param0`", "`param1`", ...) as shown in line 28 of Listing 3.50. This expression is re-evaluated (by calling "`#demand`") in order to generate C code (line 28). In this example the code for a function to compute $-a$ is generated. This is achieved using operator overloading. That is, re-evaluating generates code instead of performing the actual computation. Listing 3.51 shows the C method generated by above example. In addition code for registering the method and for converting the arguments is generated. The code is not shown here. See Listing 2.22 for a complete listing of a Ruby extension.

The compiled code becomes a class method of the class "`GCCCache`". Listing 3.52 shows how the method is called. The arguments are extracted from the values (line 32). The C method is called and it writes the result to the specified memory location (line 34). Note that the code is sufficiently generic to handle cases where the result of the computation is a composite number or an array.

Listing 3.51: The resulting C code to Listing 3.50

```
VALUE _Store0Variable0050INT112490Variable10INT111(unsigned char *param0,
                                                    int param1)
{
  int v01;
  int v02;
  v01 = param1;
  v02 = -(v01);
  *(int *)(param0 + 0) = v02;
}
```

Listing 3.52: Calling the compiled method

```
31 # ...  continuing from Listing 3.50
32 args = values.collect { |arg| arg.values }.flatten
33 # [Malloc(4), 5]
34 GCCCache.send method_name, *args
35 r
36 # INT(-5)
```

3.7 Unit Testing

The implementations of the various array operations presented in this chapter and the JIT compiler are tested using unit testing (also see Section 2.3.14.)

Scalar operations can be seen as units which can be tested individually. Listing 3.53 shows a few tests for operations on boxed integers (introduced in Section 3.3.2).

Listing 3.53: Some unit tests for integers

```
require 'test/unit'
require 'multiarray'
class TC_Int < Test::Unit::TestCase
  I = Hornetseye::INT
  def I(*args)
    Hornetseye::INT *args
  end
  def test_int_inspect
    assert_equal 'INT', I.inspect
  end
  def test_typecode
    assert_equal I, I.new.typecode
  end
  def test_shape
    assert_equal [], I.new.shape
  end
  def test_inspect
    assert_equal 'INT(42)', I(42).inspect
  end
  def test_plus
    assert_equal I(3 + 5), I(3) + I(5)
  end
end
# Loaded suite irb
# Started
# .....
# Finished in 0.001675 seconds.
#
# 5 tests, 5 assertions, 0 failures, 0 errors
```

Array operations are tested in a similar fashion. Listing 3.54 shows a few tests for array operations. Technically speaking these are functional tests, since the array operations are not tested separately from the scalar operations.

Listing 3.54: Tests for array operations

```
require 'test/unit'
require 'multiarray'
class TC_Int < Test::Unit::TestCase
  O = Hornetseye::OBJECT
  I = Hornetseye::INT
  C = Hornetseye::INTRGB
  M = Hornetseye::MultiArray
  def S(*args)
    Hornetseye::Sequence *args
  end
  def M(*args)
    Hornetseye::MultiArray *args
  end
  def test_multiarray_inspect
    assert_equal 'MultiArray(OBJECT,2)', M(O,2).inspect
    assert_equal 'MultiArray(OBJECT,2)', S(S(O)).inspect
  end
  def test_int_inspect
    assert_equal 'INT', I.inspect
  end
  def test_typecode
    assert_equal O, M(O, 2).new(3, 2).typecode
    assert_equal I, M(I, 2).new(3, 2).typecode
    assert_equal C, M(C, 2).new(3, 2).typecode
  end
  def test_shape
    assert_equal [3, 2], M(O, 2).new(3, 2).shape
  end
  def test_inspect
    assert_equal "MultiArray(OBJECT,2):\n[[:a, 2, 3],\n  [4, 5, 6]]",
                 M[[:a, 2, 3], [4, 5, 6]].inspect
  end
  def test_plus
    assert_equal M[[2, 3, 5], [3, 5, 7]],
                 M[[1, 2, 4], [2, 4, 6]] + 1
    assert_equal M[[2, 3, 5], [3, 5, 7]],
                 1 + M[[1, 2, 4], [2, 4, 6]]
    assert_equal M[[-1, 2, 3], [4, 5, 6]],
                 M[[-3, 2, 1], [8, 6, 4]] +
                 M[[2, 0, 2], [-4, -1, 2]]
  end
  def test_dilate
    assert_equal [[1, 1, 0], [1, 1, 0], [0, 0, 0]],
                 M[[1, 0, 0], [0, 0, 0], [0, 0, -1]].dilate.to_a
  end
end
# Loaded suite irb
# Started
# .......
# Finished in 2.230944 seconds.
# 7 tests, 12 assertions, 0 failures, 0 errors
```

3.8 Summary

In this chapter a Ruby library for implementing machine vision algorithms was developed. A set of native basic types was introduced. It was shown that multi-dimensional, uniform arrays can be represented as lazy pointer operations. That is, multi-dimensional arrays are just a special case of functions. Finally a generic set of operations on these data types was introduced. In some cases it was shown, how this array operations relate to image processing.

Seldon: "I have said, and I say again, that Trantor will lie in ruins within the next three centuries."
Advocate: "You do not consider your statement a disloyal one?"
Seldon: "No, sir. Scientific truth is beyond loyalty and disloyalty."
Advocate: "You are sure that your statement represents scientific truth?"
Seldon: "I am."
Advocate: "On what basis?"
Seldon: "On the basis of the mathematics of psychohistory."
Advocate: "Can you prove that this mathematics is valid?"
Seldon: "Only to another mathematician."

Isaac Asimov - The Foundation Trilogy

4 Input/Output

Apart from the array operations introduced in Chapter 3, implementation of machine vision systems requires input and output of images. Figure 4.1 gives an overview of the I/O integration implemented in context of this thesis. There are image sources (cameras and files) and image sinks (displays and files). Furthermore there are other popular free software libraries which were integrated in order to take advantage of the functionality offered by them (Fourier transforms) or in order to facilitate projects requiring integration of other competing projects (OpenCV, NArray).

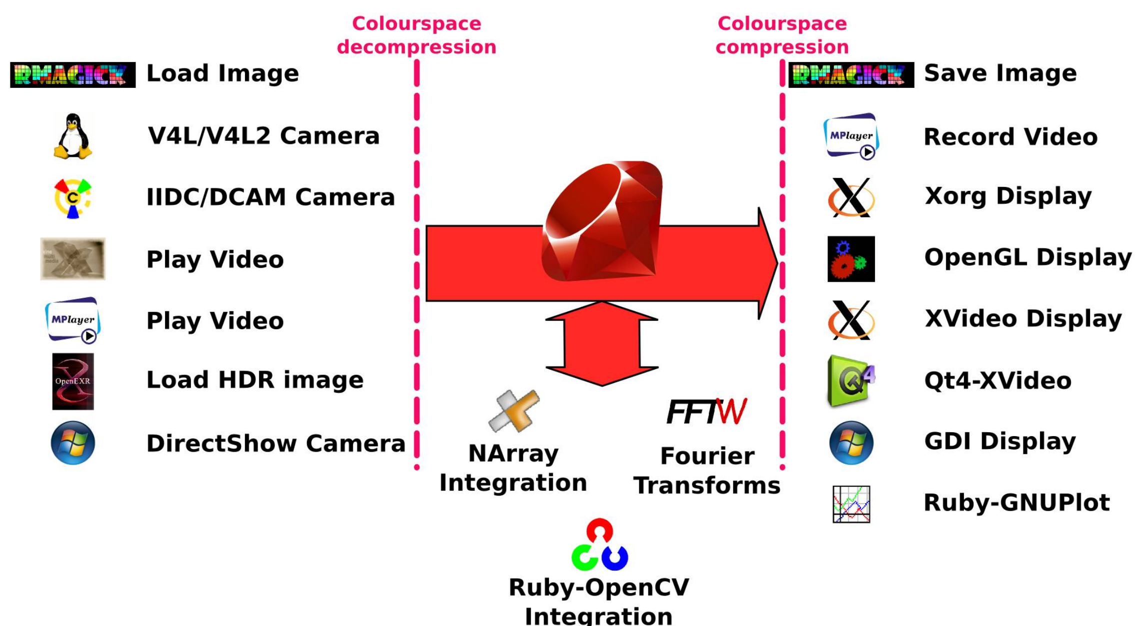


Figure 4.1: Input/output integration

This chapter introduces various I/O libraries and how they interoperate with the array operations introduced in the previous section.

- Section 4.1 explains different colour spaces commonly used by I/O devices
- Section 4.2 presents the interface to the RMagick library for loading and saving LDR images
- Section 4.3 shows the integration of the OpenEXR library for loading and saving HDR images
- Section 4.4 covers the different issues one encounters with encoding and decoding video files
- Section 4.5 points out how Ruby closures can be used to provide a powerful and concise API for accessing cameras
- Section 4.6 shows how Ruby closures can be used to implement a concise API for displaying videos
- The integration of an RGB and depth (RGBD) sensor is presented in Section 4.7
- Section 4.8 is about the integration with the Qt4-QtRuby library for developing GUIs
- Section 4.9 gives a summary of this chapter

4.1 Colour Spaces

4.1.1 sRGB

There are different representations of images (see Table 4.1). In practise images are discrete, finite, quantised, grey scale or colour functions. The image is acquired by a capturing device with a limited number of photosensitive cells. On the other hand for purposes of theoretical signal processing it can be beneficial to represent images as continuous, infinite, high dynamic range, colour functions. For example when dealing with convolutions (see Section 3.5.10) using a finite domain would require a formal treatment of the image boundaries.

There is no upper limit for $g(\vec{x})$ which expresses the fact that there is no upper limit for luminosity (see Reinhard et al., 2006 for a detailed introduction to high dynamic range imaging). In practise however, most capture and display devices have a limited and quantised codomain (typically it is $\{0, 1, \ldots, 255\}$ or $\{0, 1, \ldots, 255\}^3$).

Humans have trichromatic vision. There are different colour spaces for representing trichromatic images. Usually the standard RGB colour space (sRGB) is used with primaries defined in terms of the CIE 1391 primaries (Smith and Guild, 1931). These are not as generally believed the sensitivity curves of the human photosensitive cones, which have maxima at 445 nm, 535 nm, and 570 nm (Dröscher, 1975) (furthermore the photo

Table 4.1: Different types of images

discrete, finite, quantised, grey scale function	g:!$\{0,1,\dots,w-1\}\times\{0,1,\dots,h-1\}\to\{0,1,\dots,255\}$
discrete, finite, high dynamic range, grey scale function	$g:\{0,1,\dots,w-1\}\times\{0,1,\dots,h-1\}\to\mathbb{R}$
discrete, infinite, high dynamic range, grey scale function	$g:\mathbb{Z}^2\to\mathbb{R}$
continuous, infinite, high dynamic range, grey scale function	$g:\mathbb{R}^2\to\mathbb{R}$
discrete, finite, quantised, colour function	$\vec{c}:\{0,1,\dots,w-1\}\times\{0,1,\dots,h-1\}\to\{0,1,\dots,255\}^3$
discrete, finite, high dynamic range, colour function	$\vec{c}:\{0,1,\dots,w-1\}\times\{0,1,\dots,h-1\}\to\mathbb{R}^3$
discrete, infinite, high dynamic range, colour function	$\vec{c}:\mathbb{Z}^2\to\mathbb{R}^3$
continuous, infinite, high dynamic range, colour function	$\vec{c}:\mathbb{R}^2\to\mathbb{R}^3$

receptors for black-and-white vision at night have their sensitivity maximum at 507 nm). However as long as the sensitivity curves of the three colour channels of the camera are more or less accurate linear combinations of the sensitivity curves of the human visual cortex, it is possible to accurately reproduce the visual impression perceived by the human visual cortex.

It is worth mentioning that the relation between the radiant intensity (or luminosity) and the luma value is non-linear. The sRGB standard closely models the behaviour of a cathode ray tube (CRT) monitor with gamma of 2.2 while avoiding a slope of zero at the origin for practical reasons. Given sRGB values in $[0,1]$ the corresponding linear intensity values can be obtained using Equation 4.1 (where C is one of R, G, or B) (Stokes et al., 1996).

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{sRGB}}}{12.92}, & C_{\text{sRGB}} \leq 0.04045 \\ \left(\frac{C_{\text{sRGB}}+0.055}{1.055}\right)^{2.4}, & C_{\text{sRGB}} > 0.04045 \end{cases} \tag{4.1}$$

Although for performance reasons it is omitted in the work presented here, strictly speaking one has to take the sRGB definition into account when processing images. For example many web browsers (and even image processing programs) implicitly assume a gamma of 1.0 when scaling images which can lead to significant errors (Brasseur, 2007).

4.1.2 YC_bC_r

Many cameras make use of compressed colour spaces. The YC_bC_r (or YUV) colour space separates luma- and colour-information as shown in Equation 4.2 (R, G, and B represent

the red, green, and blue channel of an image) (Wilson, 2007).

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.500 \\ 0.500 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \tag{4.2}$$

The YC_bC_r colour space actually is the digital version (with values in $\{0, 1, \ldots, 255\}$) of the YP_bP_r colour space (see Equation 4.3).

$$\begin{pmatrix} Y \\ P_b \\ P_r \end{pmatrix} = \begin{pmatrix} K_r\,R + (1 - K_r - K_b)\,G + K_b\,B \\ \frac{1}{2}\frac{B - Y}{1 - K_b} \\ \frac{1}{2}\frac{R - Y}{1 - K_r} \end{pmatrix}, \text{ where } \begin{matrix} R, G, B \in [0, 1] \\ Y \in [0, 1] \\ P_b, P_r \in [-0.5, 0.5] \end{matrix} \tag{4.3}$$

The values K_r and K_b are the estimated sensitivities of the human photo receptors for black-and-white vision to the colours red and blue. Here (*i.e.* in Equation 4.2) the definitions $K_r = 0.299$ and $K_b = 0.114$ of the Joint Photographic Experts Group (JPEG) standard (Hamilton, 1992) are used (*e.g.* see Figure 4.2).

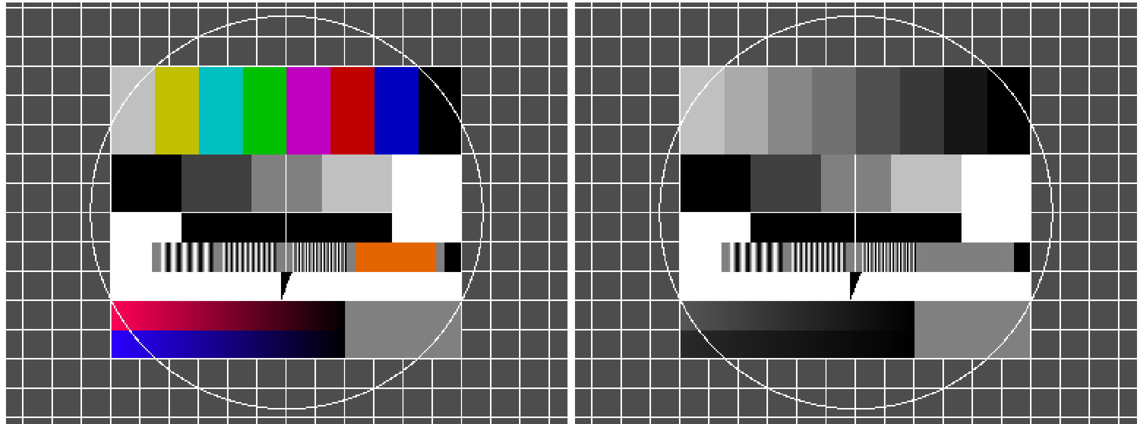
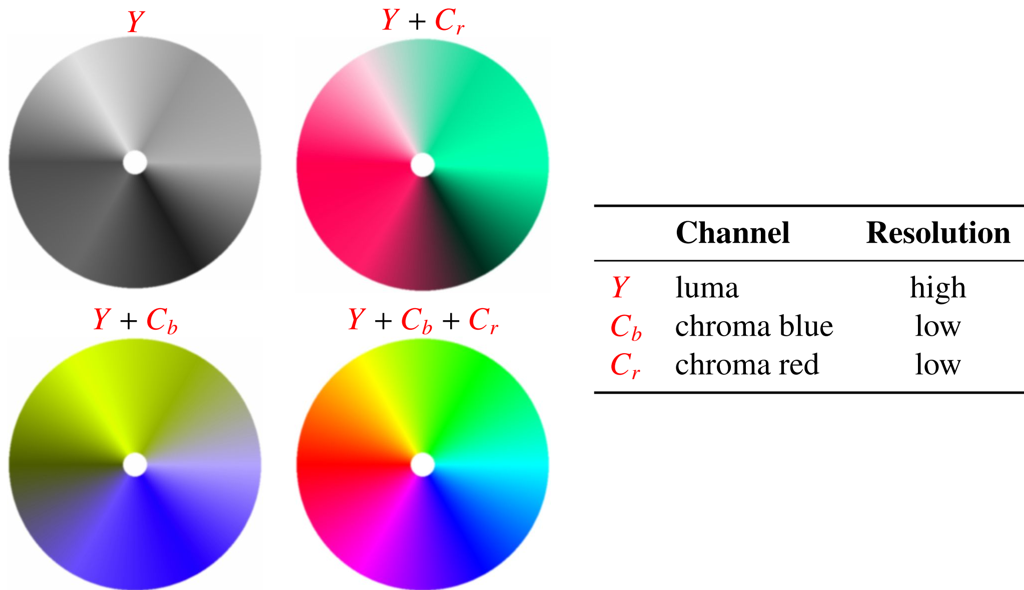


Figure 4.2: Colour image and corresponding grey scale image according to sensitivities of the human eye

Figure 4.3 gives a visual explanation of colour space compression. While the luma (Y) channel is provided in high resolution, the chroma channels chroma blue (C_b) and chroma red (C_r) are sampled with a lower resolution. Note that the chroma channels cannot be visualised separately. In fact chroma values can represent negative colour offsets.

Y, C_b, and C_r are also known as Y, U, and V. In practise there are various ways of ordering, sub sampling, and aligning the channels Wilson (2007). Popular pixel formats are

- **YV12:** The format comprises an $n \times m$ Y plane followed by $\frac{n}{2} \times \frac{m}{2}$ chroma red (V) and chroma blue (U) planes (see Figure 4.4). The lines of each plane are 8-byte memory-aligned.



	Channel	Resolution
Y	luma	high
C_b	chroma blue	low
C_r	chroma red	low

Figure 4.3: Colour space conversions (Wilson, 2007)

- **I420:** Same as YV12 but the V plane follows the U plane.
- **YUY2:** This is a packed pixel format. The V and U component are down sampled only in one direction. The component packing order is Y_0, U_0, Y_1, V_0 (see Figure 4.5). The lines are 8-byte memory aligned.
- **UYVY:** Same as UYVY but with different component packing order (see Figure 4.6).

$Y_{0,0}$	$Y_{1,0}$	$\cdots$	$Y_{n-2,0}$	$Y_{n-1,0}$
$Y_{0,1}$	$Y_{1,1}$	$\cdots$	$Y_{n-2,1}$	$Y_{n-1,1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$Y_{0,m-2}$	$Y_{1,m-2}$	$\cdots$	$Y_{n-2,m-2}$	$Y_{n-1,m-2}$
$Y_{0,m-1}$	$Y_{1,m-1}$	$\cdots$	$Y_{n-2,m-1}$	$Y_{n-1,m-1}$

$V_{0,0}$	$\cdots$	$V_{n/2-1,0}$
$\vdots$	$\ddots$	$\vdots$
$V_{0,m/2-1}$	$\cdots$	$V_{n/2-1,m/2-1}$

$U_{0,0}$	$\cdots$	$U_{n/2-1,0}$
$\vdots$	$\ddots$	$\vdots$
$U_{0,m/2-1}$	$\cdots$	$U_{n/2-1,m/2-1}$

Figure 4.4: YV12 colour space (Wilson, 2007)

In order to control colour space conversions in Ruby, compressed frame data is exposed in Ruby using parametrised classes as shown in Listing 4.1. The actual conversions are performed using the FFmpeg rescaling library (libswscale). One can see that a conversion round trip from unsigned byte RGB to YV12 and back affects the values. Fig-

$Y_{0,0}$	$U_{0,0}$	$Y_{1,0}$	$V_{0,0}$	$Y_{2,0}$	$U_{1,0}$	$Y_{3,0}$	$V_{1,0}$	$\cdots$
$Y_{0,1}$	$U_{0,1}$	$Y_{1,1}$	$V_{0,1}$	$Y_{2,1}$	$U_{1,1}$	$Y_{3,1}$	$V_{1,1}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 4.5: YUY2 colour space (Wilson, 2007)

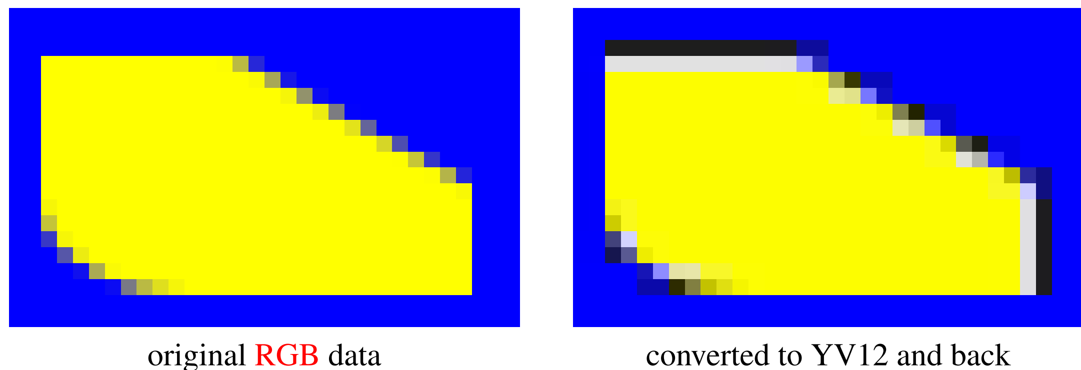
$U_{0,0}$	$Y_{0,0}$	$V_{0,0}$	$Y_{1,0}$	$U_{1,0}$	$Y_{2,0}$	$V_{1,0}$	$Y_{3,0}$	$\cdots$
$U_{0,1}$	$Y_{0,1}$	$V_{0,1}$	$Y_{1,1}$	$U_{1,1}$	$Y_{2,1}$	$V_{1,1}$	$Y_{3,1}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 4.6: UYVY colour space (Wilson, 2007)

Listing 4.1: Handling compressed colour spaces

```
img = MultiArray.load_ubytergb 'test.jpg'
# MultiArray(UBYTERGB,2):
# [ [ RGB(35,38,45), RGB(45,48,55), RGB(46,50,59), RGB(46,50,59), .... ],
#   [ RGB(46,49,56), RGB(55,58,65), RGB(56,59,68), RGB(57,60,67), .... ],
#   [ RGB(46,51,57), RGB(57,60,67), RGB(57,60,67), RGB(58,61,68), .... ],
#   [ RGB(46,51,57), RGB(58,61,68), RGB(58,61,68), RGB(58,61,68), .... ],
#   [ RGB(47,50,59), RGB(58,61,70), RGB(59,62,69), RGB(59,62,69), .... ],
#   ....
frame = img.to_yv12
# Frame(YV12,320,240)(0x042caf2a)
frame.to_ubytergb
# MultiArray(UBYTERGB,2):
# [ [ RGB(33,38,45), RGB(42,47,54), RGB(45,49,59), RGB(45,49,59), .... ],
#   [ RGB(44,48,55), RGB(53,58,65), RGB(54,59,68), RGB(54,59,68), .... ],
#   [ RGB(45,49,54), RGB(54,59,63), RGB(54,59,66), RGB(55,60,67), .... ],
#   [ RGB(45,49,54), RGB(55,60,65), RGB(55,60,67), RGB(55,60,67), .... ],
#   [ RGB(45,49,56), RGB(55,60,67), RGB(56,61,68), RGB(56,61,68), .... ],
#   ....
```

ure 4.7 shows how YV12 colour space compression leads to compression artefacts when the edges do not align favourably with the lower resolution of the U and V channels.



original RGB data — converted to YV12 and back

Figure 4.7: Artefacts caused by colour space compression

4.2 Image Files

The RMagick[1] Ruby-extension allows one to use the powerful ImageMagick/Magick++[2] library in Ruby for loading and saving images. The ImageMagick library supports a large

[1] http://rmagick.rubyforge.org/
[2] http://www.imagemagick.org/Magick++/

number of file formats. Frequently used file formats are

- Bitmap Image File (BMP) format
- Graphics Interchange Format (GIF)
- Portable Network Graphics (PNG) format
- portable pixmap (PPM) format (with variations portable graymap (PGM) and portable bitmap (PBM))
- Joint Photographic Experts Group (JPEG) format
- Tagged Image File Format (TIFF)
- Digital Imaging and Communications in Medicine (DICOM)

In general one needs a format for lossy compression (*e.g.* JPEG) and a format for lossless compression (*e.g.* PNG) when using 8-bit grey scale or 24-bit colour images (*e.g.* see Figure 4.8). Listing 4.2 shows how the functionality of the RMagick Ruby-extension was in-

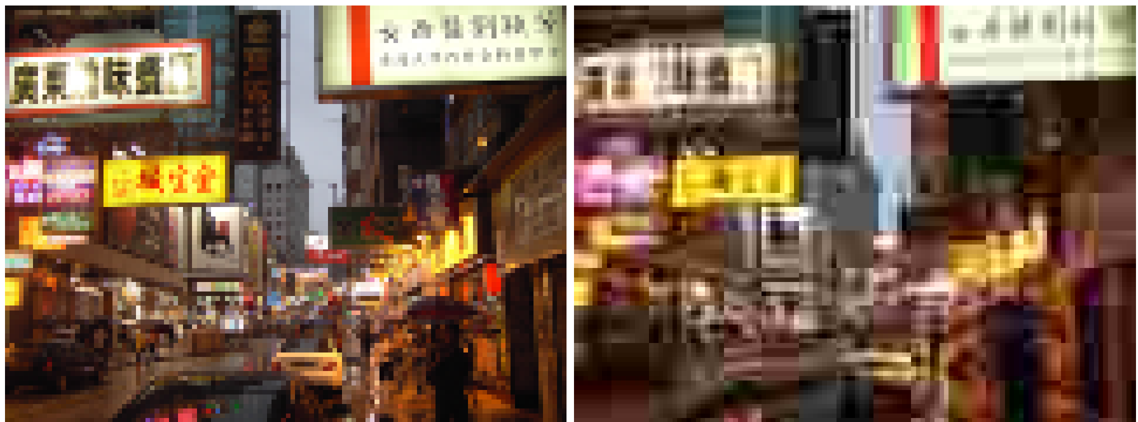


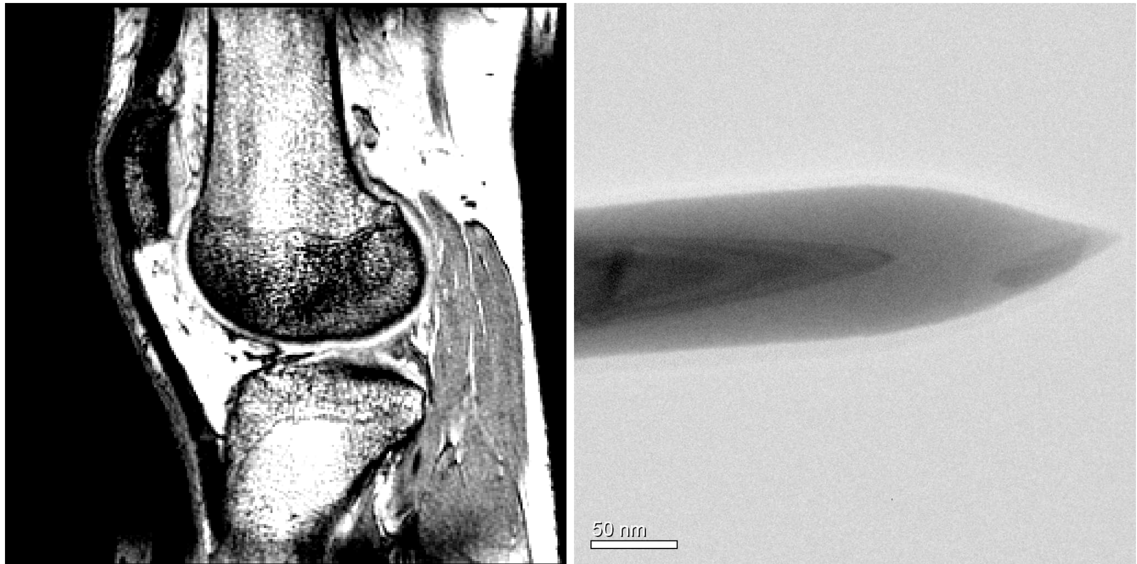
Figure 4.8: Low resolution colour image using lossless PNG and (extremely) lossy JPEG compression

tegrated. The method "`MultiArray.load_ubytergb`" uses the RMagick library to load an image and convert it to unsigned byte (8 bit) RGB data. The resulting "`Malloc`" object (see Chapter 3.2) is used to construct a uniform array of RGB values. The "`#save_ubytergb`" method provides saving unsigned byte RGB data.

Medical image processing software frequently uses the DICOM format for storing 16-bit radiology images and in material science TIFF is a common format for exchanging 16-bit electron images. Figure 4.9 shows examples of images in medical science and material science. Here they are shown using 8-bit quantisation only since most computer displays do not support more than 2^8 grey levels. To support saving and loading of grey scale and colour images of different depth the methods shown in Table 4.2 were implemented (using the RMagick library).

Listing 4.2: Loading and saving images

```
img = MultiArray.load_ubytergb 'lossless.png'
# MultiArray(UBYTERGB,2):
# [ [ RGB(18,14,13), RGB(21,15,15), RGB(24,17,14), RGB(27,19,14), .... ],
#   [ RGB(34,20,14), RGB(36,18,14), RGB(78,51,34), RGB(88,63,43), .... ],
#   [ RGB(33,24,18), RGB(37,23,18), RGB(69,44,23), RGB(71,40,30), .... ],
#   [ RGB(29,21,16), RGB(41,27,18), RGB(58,29,18), RGB(52,30,19), .... ],
#   [ RGB(17,11,13), RGB(37,16,17), RGB(59,24,18), RGB(43,22,17), .... ],
#   [ RGB(19,12,15), RGB(40,15,18), RGB(57,22,19), RGB(13,11,12), .... ],
#   [ RGB(20,13,15), RGB(35,15,15), RGB(44,19,17), RGB(17,12,12), .... ],
#   [ RGB(17,12,14), RGB(46,17,19), RGB(63,19,20), RGB(70,22,22), .... ],
#   [ RGB(23,13,16), RGB(64,24,21), RGB(71,31,23), RGB(79,36,26), .... ],
#   [ RGB(35,25,27), RGB(60,40,35), RGB(68,45,35), RGB(83,66,54), .... ],
#   ....
img.save_ubytergb 'lossy.jpg'
#   ....
```



MR scan of knee (source: Sébastien Barré's DICOM collection)

TEM image of tungsten tip (courtesy of Sheffield University Nanorobotics Research Group)

Figure 4.9: Examples of images in medical science and material science

Table 4.2: Methods for loading and saving images

(integer) type	loading	saving
8 bit monochrome	"`MultiArray.load_ubyte`"	"`Node#save_ubyte`"
16 bit monochrome	"`MultiArray.load_usint`"	"`Node#save_usint`"
32 bit monochrome	"`MultiArray.load_uint`"	"`Node#save_uint`"
8 bit RGB	"`MultiArray.load_ubytergb`"	"`Node#save_ubytergb`"
16 bit RGB	"`MultiArray.load_usintrgb`"	"`Node#save_usintrgb`"
32 bit RGB	"`MultiArray.load_uintrgb`"	"`Node#save_uintrgb`"

In some cases it might be desirable to let the file format determine the image type (bit depth, grey-scale/colour) and vice versa instead of forcing it. For this case methods named "`MultiArray.load_magick`" and "`Node#save_magick`" were implemented which make use of dynamic typing available in the Ruby language. In fact this two methods are the basis for the other methods for loading and saving images.

Note that dynamic typing has a compelling advantage in the case of loading images. C/C++ libraries such as OpenCV and ImageMagick cannot make use of the (static) type system of the programming language to handle image types because in general it is not desirable having to specify the image type before loading the image.

4.3 HDR Image Files

Theoretically there is no upper limit for the radiant intensity. Therefore in many situation the linear quantisation (low dynamic range) does not allow for an optimal trade-off between quantisation noise and measurement range. The human eye uses photo receptors with different sensitivities and a non-linear response to address this problem.

It is possible to acquire HDR images using LDR devices with the help of exposure bracketing. That is, a series of pictures with different exposures is acquired and fused using an algorithm. HDR images are usually represented using arrays of floating point numbers. A popular format for HDR images is the OpenEXR[3] format by Industrial Light & Magic (Kainz and Bogard, 2009). The format uses 16 bit (half precision) floating point numbers.

Tone mapping is the digital analogy to the traditional technique of dodging and burning. Tone mapping maps a HDR image to an LDR image by locally adapting the luminosity of the image. Figure 4.10 illustrates the complete process of HDR imaging using a consumer camera. First a set of images with different exposures is acquired. If the camera has shifted, the images need to be aligned using feature matching (here the panorama stitching software Hugin[4] was used). The images are fused to an HDR image. For display on a low dynamic range device, the image is tone mapped (here the tone mapping software QtPfsGui[5] was used). There are different algorithms for tone mapping. Figure 4.10 shows a result obtained using the algorithm by Fattal et al. (2002).

To support saving and loading of grey scale and colour images and convert them to floating point arrays of different depth the methods shown in Table 4.3 were implemented (using the OpenEXR library). Listing 4.3 shows how the methods might be used in practise.

Similar as in Chapter 4.2 the methods "`MultiArray.load_openexr`" as well as the method "`Node#save_openexr`" let the file format determine the image type.

[3] http://www.openexr.com/
[4] see http://hugin.sourceforge.net/
[5] see http://qtpfsgui.sourceforge.net/

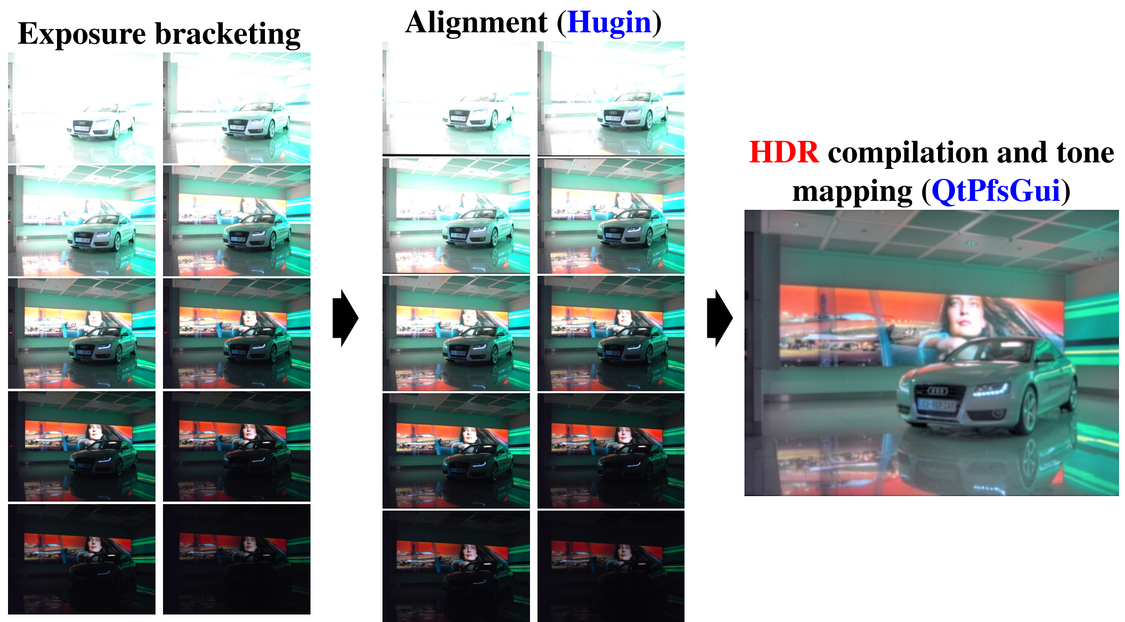


Figure 4.10: Bracketing, alignment, and tone mapping

Table 4.3: Methods for loading and saving images

(floating point) type	loading	saving
32 bit monochr.	"`MultiArray.load_sfloat`"	"`Node#save_sfloat`"
64 bit monochr.	"`MultiArray.load_dfloat`"	"`Node#save_dfloat`"
32 bit RGB	"`MultiArray.load_sfloatrgb`"	"`Node#save_sfloatrgb`"
64 bit RGB	"`MultiArray.load_dfloatrgb`"	"`Node#save_dfloatrgb`"

Listing 4.3: Converting an HDR image to an LDR image (no tone mapping)

```
img = MultiArray.load_dfloatrgb 'hdr.exr'
# ...
(img * 1000).round / 1000
# MultiArray(DFLOATRGB,2):
# [ [ RGB(0.249,0.283,0.363), RGB(0.294,0.324,0.422), .... ],
#   [ RGB(0.293,0.329,0.412), RGB(0.339,0.376,0.478), .... ],
#   [ RGB(0.289,0.337,0.429), RGB(0.345,0.389,0.486), .... ],
#   [ RGB(0.298,0.336,0.437), RGB(0.357,0.393,0.506), .... ],
#   [ RGB(0.307,0.331,0.435), RGB(0.372,0.39,0.513), .... ],
#   [ RGB(0.317,0.338,0.478), RGB(0.369,0.397,0.529), .... ],
#   [ RGB(0.324,0.34,0.514), RGB(0.362,0.4,0.548), .... ],
#   [ RGB(0.318,0.338,0.458), RGB(0.359,0.394,0.537), .... ],
#   [ RGB(0.321,0.339,0.473), RGB(0.353,0.403,0.554), .... ],
#   [ RGB(0.309,0.35,0.45), RGB(0.363,0.412,0.528), .... ],
#   ....
peak = img.max
# RGB(71.2704849243164,56.86302947998047,81.38863372802734)
factor = 255 / [peak.r, peak.g, peak.b].max
# 3.133115624622988
(img * factor).save_ubytergb 'ldr.png'
# ...
```

4.4 Video Files

File formats generally use algorithms such as Huffmann coding for lossless compression. Image file formats furthermore exploit spatial similarity. For example the JPEG format uses a block-wise discrete cosine transform followed by application of a custom quantisation matrix. Most video codecs additionally exploit temporal similarity by replacing full frames with motion vector fields and motion-compensated difference pictures. The occasional full frame is included to reduce the computational cost of randomly accessing a frame in the video.

For reading and writing video files the FFmpeg[6] library was integrated. Many video formats consist of a container format which usually offers a video stream and potentially an audio stream. Many container formats support several video and audio codecs. The video and audio codec determine the format of the video and audio stream. Popular file formats and codecs are

- **popular container formats**
  - Audio Video Interleave (AVI)
  - Advanced Systems Format (ASF)
  - Flash Video (FLV)
  - Apple Quicktime Movie (MOV)
  - MPEG standard version 4 (MPEG-4)
  - Xiph.Org container format (Ogg)
- **popular video codecs**
  - On2 Truemotion VP6 codec (VP6)
  - MPEG-4 AVC standard (H.264)
  - Windows Media Video (WMV)
  - Xiph.Org video codec (Theora)
- **popular audio codecs**
  - Advanced Audio Coding (AAC)
  - MPEG Audio Layer 3 (MP3)
  - Xiph.Org audio codec (Vorbis)
  - Windows Media Audio (WMA)

[6] http://www.ffmpeg.org/

Listing 4.4: Reading a video file

```
input = AVInput.new 'test.avi'
input.frame_rate
# (15/1)
input.sample_rate
# 44100
input.pos = 60
input.read_video
# Frame(YV12,320,240)(0x04abdce6)
input.read_audio
# MultiArray(SINT,2):
# [ [ 0, 0 ],
#   [ 0, 0 ],
#   [ 0, 0 ],
#   [ 0, 0 ],
#   [ 0, 0 ],
#   ....
input.video_pos
# (184/3)
input.audio_pos
# (122671/2000)
```

Figure 4.11 shows the coarse architecture of the FFmpeg decoder. The demuxer of the container format decodes the file and generates audio and video packets. The video decoder accepts video packets and decodes video frames. The audio decoder accepts audio packets and returns audio frames. The video frames are usually given as YV12 data (see Chapter 4.1.2). Typically the video is given with a frame rate of 25 frames/second. Note that video data might have a pixel aspect ratio other than $1 : 1$ and it can be interlaced. The audio frames are given as 16-bit signed integer arrays (for stereo audio the values come in pairs of two). A common sampling rate is 44.1 kHz. The audio and video frames

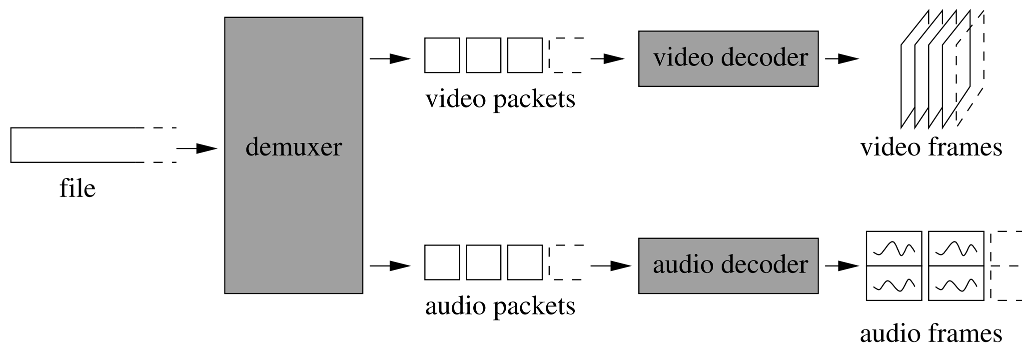


Figure 4.11: Decoding videos using FFmpeg

are tagged with a presentation time stamp. The time stamps are required to synchronize the audio and video frames properly. Another reason for the existence of time stamps are video formats with support for variable frame rate.

Listing 4.4 demonstrates the behaviour of the class "**AVInput**" which was developed to make the FFmpeg decoder accessible in Ruby. A video file is opened (line 1) and

the decoding of the first 60 seconds is skipped (line 6). A video frame (320×240 YV12 data) and an audio frame (1152 16-bit stereo samples) are decoded (lines 7 and 9). At the end the time stamps of the previously decoded frames are retrieved (lines 17 and 19). One can see that the time stamps of audio and video frames do not necessarily coincide ($\frac{184}{3} = 61.\bar{3}$, $\frac{122671}{2000} = 61.3355$).

The FFmpeg library also supports video and audio encoding. One can see in Figure 4.12 that the encoder's data flow is symmetric to the decoder's data flow. Instead of video and audio decoders there are video and audio encoders. The demuxer is replaced with a muxer.

The class "`AVOutput`" was implemented to expose the encoding functionality of the FFmpeg library in Ruby. Listing 4.5 shows a small program creating an MPEG-4 file with a static picture (loaded in line 5) as video and a 400 Hz sine wave (created in lines 13–15) as audio. The interface is minimalistic and does not give access to the various

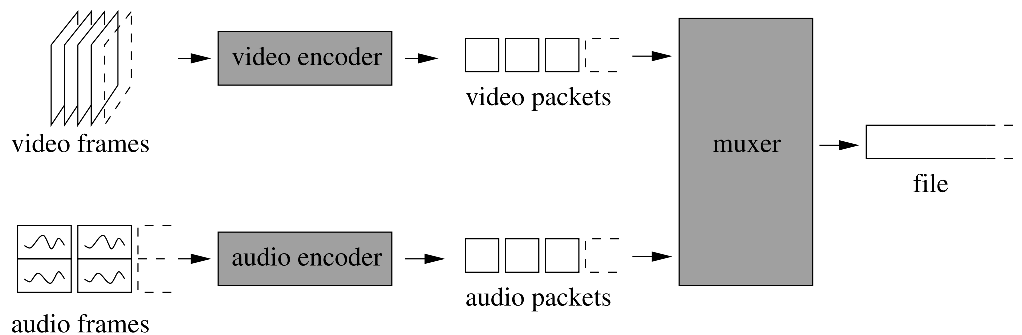


Figure 4.12: Encoding videos using FFmpeg

parameters of the encoding algorithms. When encoding a video file one has to merely specify the following basic properties (see lines 23–25 of Listing 4.5)

- **video bit rate:** the approximate number of bits per second for encoding the video stream
- **width:** the width of each video frame
- **height:** the height of each video frame
- **frame rate:** the number of video frames per second
- **pixel aspect ratio:** the ratio of pixel width to pixel height
- **video codec:** the video codec to use
- **audio bit rate:** the approximate number of bits per second for encoding the audio stream
- **audio sampling rate:** the number of audio samples per second

Listing 4.5: Writing a video file

```
VIDEO_BITRATE = 500_000; AUDIO_BITRATE = 30_000
DURATION      =       3; RATE          = 44_100
FPS           =      25; CHANNELS      =      2
ASPECT        = 1.quo 1; LEN           =    110
img = MultiArray.load_ubytergb 'test.png'
# MultiArray(UBYTERGB,2):
# [ [ RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), .... ],
#   [ RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), .... ],
#   [ RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), .... ],
#   [ RGB(255,255,255), RGB(255,255,255), RGB(255,255,255), .... ],
#   [ RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), RGB(77,77,77), .... ],
#   ....
wave = lazy(CHANNELS, LEN) do |j,i|
  Math.sin(i * 2 * Math::PI / LEN) * 0x7FFF
end.to_sint
# MultiArray(SINT,2):
# [ [ 0, 0 ],
#   [ 1870, 1870 ],
#   [ 3735, 3735 ],
#   [ 5587, 5587 ],
#   [ 7421, 7421 ],
#   ....
output = AVOutput.new 'test.mp4', VIDEO_BITRATE, img.width, img.height,
                      FPS, ASPECT, AVOutput::CODEC_ID_MPEG4, true,
                      AUDIO_BITRATE, RATE, CHANNELS, AVOutput::CODEC_ID_MP3
(RATE * DURATION / LEN).times do
  output.write_audio wave
end
(FPS * DURATION).times do
  output.write_video img
end
```

- **audio channels:** the number of audio channels (*e.g.* two for stereo audio)
- **audio codec:** the audio codec to use

Note that the program shown in Listing 4.5 encodes all the audio frames (lines 26–28) before starting to encode the video frames (lines 29–31). When creating large files this is not good practise because the muxer needs to interleave video and audio packets. That is, the muxer would be forced to allocate a lot of memory for queueing the audio frames.

4.5 Camera Input

Camera input is a prime example for the benefit of closures. Initialising a camera (*e.g.* Logitech Quickcam Pro 9000 shown in Figure 4.13) requires the calling program to choose a video mode (*i.e.* width, height, and colour space). However the supported

Figure 4.13: Logitech Quickcam Pro 9000 (a USB webcam)

video modes can only be requested *after* opening the camera device. For this reasons most APIs either allow the calling program to handle a camera device which is not fully initialised yet, or the calling program has to specify a preferred video mode which might not be supported by the camera.

Using closures however it is possible to involve the calling program during initialisation in an elegant way as demonstrated in Listing 4.6. The constructor "`V4L2Input.new`" opens the specified device (here "`'/dev/video0'`"). The supported video modes are requested, compiled to a list, and passed to the closure as a parameter. In this example the closure prints the list to the terminal and the user is prompted to select one.

Listing 4.7 gives a more minimalistic example. Here it is assumed that the camera supports the specified video mode (800 × 600, YUY2 colour space). The closure ignores the list of modes and just returns the desired resolution. If the camera does not support the desired video mode, the initialisation will fail.

Listing 4.6: Opening a V4L2 device and negotiating a video mode

```
camera = V4L2Input.new '/dev/video0' do |modes|
  modes.each_with_index { |mode,i| puts "#{i + 1}: #{mode}" }
  modes[STDIN.readline.to_i - 1]
end
# 1: Frame(YUY2,160,120)
# 2: Frame(YUY2,176,144)
# 3: Frame(YUY2,320,240)
# 4: Frame(YUY2,352,288)
# 5: Frame(YUY2,640,480)
# 6: Frame(YUY2,800,600)
# 7: Frame(YUY2,960,720)
# 8: Frame(YUY2,1600,1200)
# $ 6
# #<Hornetseye::V4L2Input:0x993af7c>
camera.read
# Frame(YUY2,800,600)(0x049c3d40)
```

Listing 4.7: Opening a V4L2 device and selecting a fixed mode

```
camera = V4L2Input.new('/dev/video0') { Frame(YUY2, 800, 600) }
# #<Hornetseye::V4L2Input:0x8ca9ee8>
camera.read
# Frame(YUY2,800,600)(0x046283d4)
```

4.6 Image Display

When developing computer vision algorithms it is important to be able to visually inspect processed or annotated images on the desktop. Figure 4.14 shows the structure of a standard X Window desktop. The X library allows a program to access multiple displays and open multiple windows on each of them. The window manager software draws the title bar and the window boundary. The program is only responsible for drawing the content of each window. When the program is idle it can query the X Server to get notified when an event occurs (*e.g.* a window close button was pressed, a key was pressed, or an area of a window needs repainting).

Functionality for displaying a single image was exposed in Ruby using the "`#show`" method as demonstrated in Listing 4.8. The method opens a window showing the image and it returns control to the calling program when the window is closed (or [Esc] or [Space] was pressed).

Apart from displaying single images it is also important to be able to display videos. The OpenCV computer vision library offers functionality for displaying videos as shown in Listing 4.9. The program reads frames from a video file, converts them to grey scale,

Listing 4.8: Loading and displaying an image

```
MultiArray.load_ubytergb('test.png').show
```

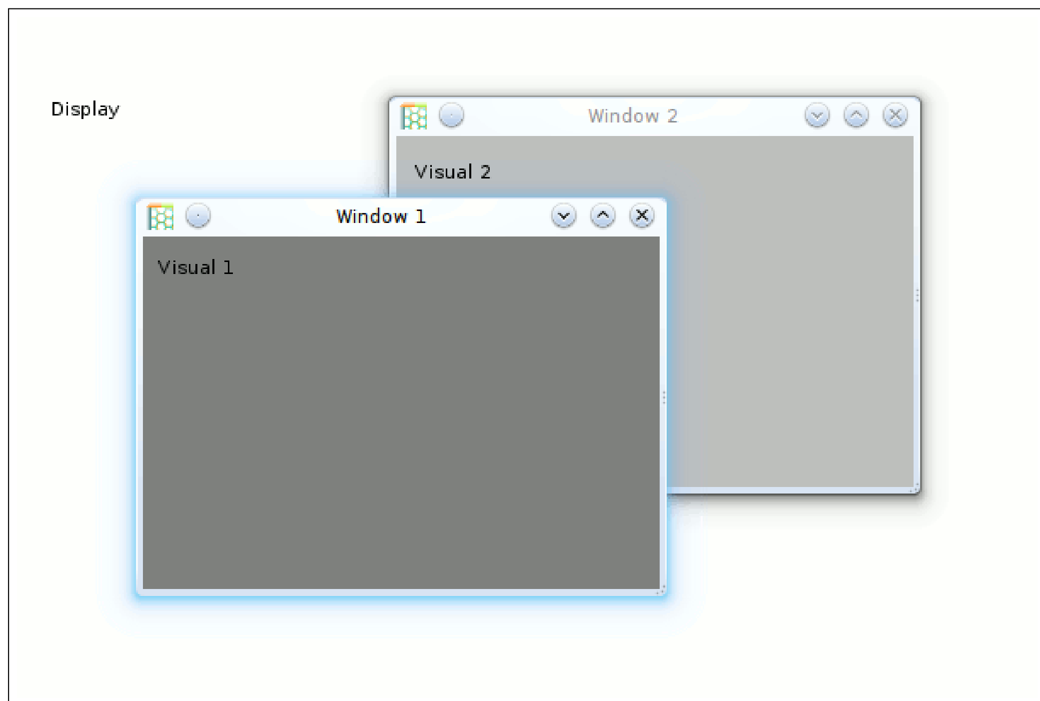


Figure 4.14: Display, windows, and visuals on a standard X Window desktop

Listing 4.9: Displaying a video using Python and OpenCV

```
import sys
from opencv import cv
from opencv import highgui
highgui.cvNamedWindow('test video')
capture = highgui.cvCreateFileCapture('test.avi')
while 1:
  frame = highgui.cvQueryFrame(capture)
  gray = cv.cvCreateImage(cv.cvSize(frame.width, frame.height), 8, 1)
  cv.cvCvtColor(frame, gray, cv.CV_BGR2GRAY)
  highgui.cvShowImage('test video', gray)
  if highgui.cvWaitKey(5) > 0:
    break
```

and then displays them in a window.

Listing 4.10 shows an equivalent program implemented using Ruby and the Hornetseye Ruby extension presented in this thesis. For comparison with Listing 4.9 the complete code with loading of libraries and name space handling is given. The Ruby programming language has mature support for closures (see Chapter 2.3.8). This makes it possible to implement a method such as "`X11Display.show`" which accepts a closure returning successive video frames for display. Only the custom part of the display loop needs to be specified by the calling program. Thus the calling program is much more concise and the overall redundancy is less.

Listing 4.11 is another example emphasizing the importance of closures. It shows a minimalistic video player (no sound, assuming pixel aspect ratio of 1 : 1, no handling of variable frame rate). Without closures the code for synchronising video display with the

Listing 4.10: Displaying a video using Ruby and Hornetseye

```
require 'rubygems'
require 'hornetseye_ffmpeg'
require 'hornetseye_xorg'
include Hornetseye
video = AVInput.new 'test.avi'
X11Display.show(:title => 'test video') do
  video.read.to_ubyte
end
```

Listing 4.11: Minimalistic video player

```
video = AVInput.new 'test.avi'
X11Display.show(:frame_rate => video.frame_rate) { video.read }
```

clock would have to be part of the calling program. The program shown in Listing 4.11 does not do any signal processing. However it provides the video I/O necessary to implement a machine vision algorithm with visualisation. That is, the code of the minimalistic video player represents a lower bound for the most concise implementation of a machine vision algorithm. Therefore it is worthwhile to minimise its size.

When visualising real-time machine vision algorithms, the time for displaying the results can exceed the time of the algorithms involved. In order to address this problem one can use 2D hardware acceleration. Most graphic cards provide hardware accelerated display (the XVideo extension of the X Window System) for a single visual. The acceleration typically requires the image to be compressed as YV12 data. Listing 4.12 shows how Ruby optional parameters are used to expose that functionality in Ruby. Note that adding sound I/O to this program results in a video player which compares with professional video player software in terms of performance (see Appendix A.7.2).

4.7 RGBD Sensor

RGBD sensors provide a depth channel in addition to the RGB channels. A recent example is the Primesense sensor (which is part of the Microsoft Kinect device). The sensor uses an Infrared (IR) laser to project a pattern. The IR camera takes a picture of the pattern and the device uses correlation techniques to estimate the disparity and the depth (Freedman et al., 2010). A separate RGB camera is used to acquire optical images. Figure 4.15 shows a depth image and optical image acquired with the device. The image pair was acquired using the program shown in Listing 4.13. The Ruby bindings are based on

Listing 4.12: Hardware accelerated video output

```
video = AVInput.new 'test.avi'
X11Display.show(:frame_rate => video.frame_rate,
                :output => XVideoOutput) { video.read }
```

Figure 4.15: RGB- and depth-image acquired with an RGBD sensor

Listing 4.13: Ruby example for accessing a Kinect sensor

```
class Numeric
  def clip(range)
    [[self, range.begin].max, range.end].min
  end
end
colours = Sequence.ubytergb 256
for i in 0 ... 256
  hue = 240 - i * 240.0 / 256.0
  colours[i] =
    RGB(((hue - 180).abs -  60).clip(0 ...60) * 0xFF / 60.0,
        (120 - (hue - 120).abs).clip(0 ...60) * 0xFF / 60.0,
        (120 - (hue - 240).abs).clip(0 ...60) * 0xFF / 60.0)
end
input = KinectInput.new
img = MultiArray.ubytergb 1280, 480
X11Display.show do
  img[   0 ...  640, 0 ... 480] = input.read_video
  img[640 ... 1280, 0 ... 480] = (input.read_depth >> 2).clip.lut colours
  img
end
```

the libfreenect library[7]. The 11-bit depth image is shifted right by 2 bits. The resulting 9-bit depth image is clipped to 8-bit values and converted to a pseudo colour image. Note that the depth image and the RGB image are not aligned properly. That is, it is necessary to calibrate the sensor and rectify the data if depth- and RGB-images are to be used in conjunction. Another problem is that the depth- and RGB-images are always taken at different times.

4.8 GUI Integration

When developing a GUI to parametrise and run computer vision algorithms it usually requires video display as part of the interface. For the work presented in this thesis the Qt4 library (Gurtovoy and Abrahams, 2009) was used. The Qt4 library has become the tool of choice for developing cross-platform GUIs (Blanchette and Summerfield, 2008). Qt4 is a C++ library. However Richard Dale's qt4-qtruby[8] extension facilitates development of GUIs in Ruby using the work flow shown in Figure 4.16. For displaying videos in a Qt4

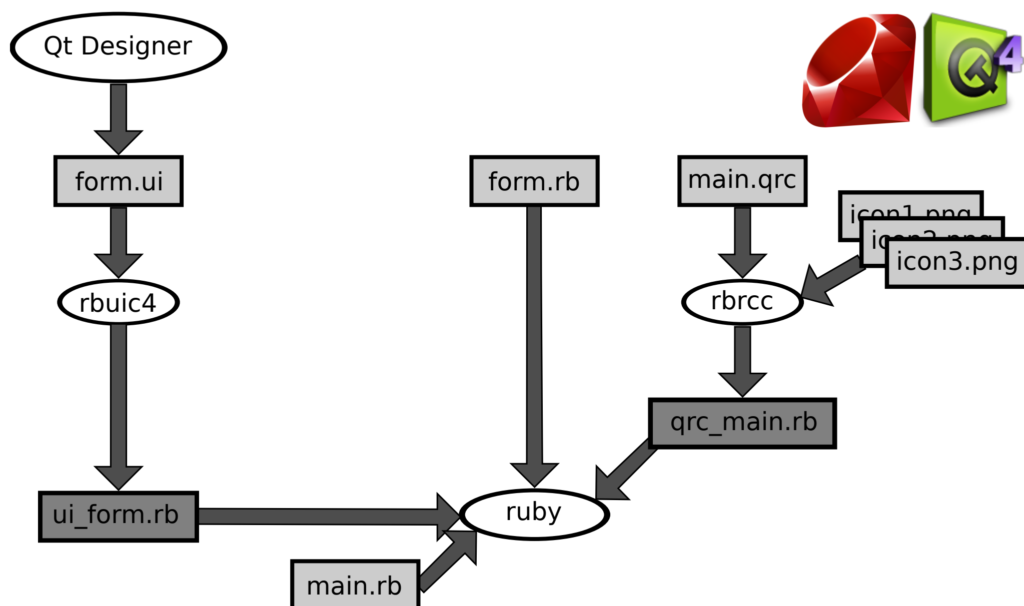


Figure 4.16: Work flow for creating Qt4 user interfaces

GUI, a widget for 2D hardware accelerated display (XVideo) was developed. The widget makes it possible to display videos with high frame rate as part of a GUI. Figure 4.17 shows a minimalistic video player window consisting of a XVideo widget and a slider.

[7] http://openkinect.org/
[8] http://rubyforge.org/projects/korundum/

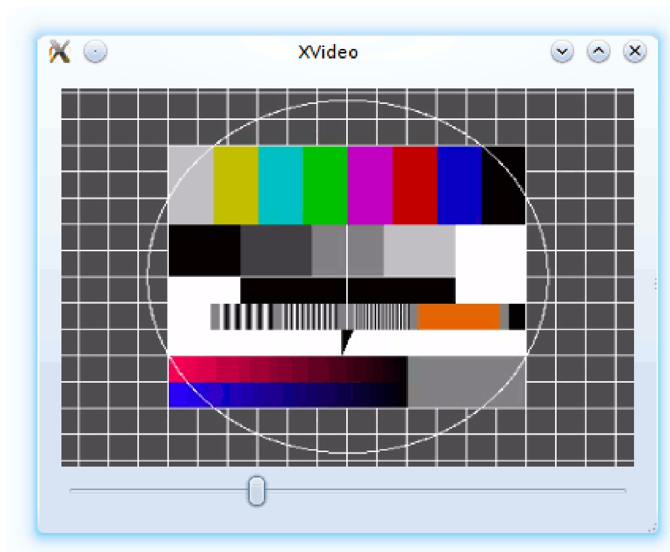


Figure 4.17: XVideo widget embedded in a GUI

4.9 Summary

In this chapter it was shown how various I/O facilities can be integrated into the Ruby language in order to acquire and output or store images. In order to get per-pixel access to the data, it is necessary to understand and manage colour space transformations. It was pointed out that most devices and file formats provide a limited quantisation range. However there are specific file formats (and even devices) for handling HDR data.

Ruby closures where used to implement an API which supports negotiation of a camera resolution when opening a camera. Ruby closures also proved to be useful for defining a concise API for displaying videos. Furthermore the power of Ruby as a glue language comes into play when integrating the different APIs for the GUI, the video I/O, and image processing.

5 Machine Vision

"An older and by now well-accepted idea is that of the stored-program computer. In such a computer the program and the data reside in the same memory; that is, the program is itself data which can be manipulated as any other data by the processor. It is this idea which allows the implementation of such powerful and incestuous software as program editors, compilers, interpreters, linking loaders, debugging systems, etc.
One of the great failings of most high-level languages is that they have abandoned this idea. It is extremely difficult, for example, for a PL/I (PASCAL, FORTRAN, COBOL ...) program to manipulate PL/I (PASCAL, FORTRAN, COBOL ...) programs."

Guy Steele and Gerald Sussman (1979)

"I want computers to be my servants, not my masters. Thus, I'd like to give them orders quickly. A good servant should do a lot of work with a short order."

Yukihiro Matsumoto

This chapter shows how the I/O integration and the array operations introduced in previous chapters facilitate concise implementation of machine vision algorithms.

- Section 5.1 shows how various preprocessing algorithms can be implemented using the array operations introduced previously in this thesis
- Section 5.2 illustrates that the array operations are sufficiently generic to create concise implementations of various corner and edge detectors
- Section 5.3 shows how feature locations and descriptors can be developed using basic array operations such as masks and warps
- Section 5.4 gives a summary of this chapter

5.1 Preprocessing

This section will demonstrate how the array operations presented in previous sections can be used to implement basic image processing operations.

5.1.1 Normalisation and Clipping

In general displaying an image with an LDR display can lead to numerical overflow as shown in Figure 5.1. This is because graphic cards typically accept 8-bit integers for each colour channel. In general it is therefore necessary to either normalise or clip the values before displaying them. The definitions for normalisation and clipping of grey

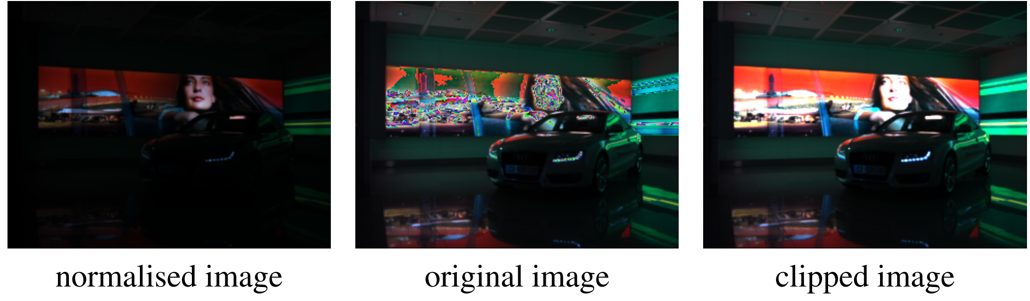


Figure 5.1: Normalisation and clipping of RGB values

scale images are given in Equation 5.1 and Equation 5.2.

$$\mathcal{N}\{g\}(\vec{x}) := \left(g(\vec{x}) - \min_{\vec{v}}(g(\vec{v}))\right)\frac{255-0}{\max_{\vec{v}}(g(\vec{v})) - \min_{\vec{v}}(g(\vec{v}))} + 0 \tag{5.1}$$

$$C\{g\}(\vec{x}) := \begin{cases} 0 & g(\vec{x}) < 0 \\ 255 & g(\vec{x}) \geq 255 \\ g(\vec{x}) & \text{otherwise} \end{cases} \tag{5.2}$$

The operations can be extended to colour images analogously.

5.1.2 Morphology

5.1.2.1 Erosion and Dilation

General erosion and dilation in grey scale morphology can be defined according to Haralick et al. (1987). The definition of grey scale erosion and dilation are shown in Equation 5.3 and Equation 5.4.

$$(f \ominus k)(\vec{x}) := \min_{\vec{z}\in K,\ \vec{x}-\vec{z}\in F} \{f(\vec{x}-\vec{z}) - k(\vec{z})\} \text{ where } f : F \to E \text{ and } k : K \to E \tag{5.3}$$

$$(f \oplus k)(\vec{x}) := \max_{\vec{z}\in K,\ \vec{x}-\vec{z}\in F} \{f(\vec{x}-\vec{z}) + k(\vec{z})\} \text{ where } f : F \to E \text{ and } k : K \to E \tag{5.4}$$

Here only the special case of a flat, block-shaped structuring element is considered (see Equation 5.5).

$$k : \begin{cases} \{-1, 0, 1\}^n & \to & \mathbb{R} \\ \vec{z} & \mapsto & 0 \end{cases} \tag{5.5}$$

The dilation of a 1D array can be performed by creating a sum table followed by a diagonal injection taking the maximum as shown in Listing 5.1. Multi-dimensional erosion and dilation can be implemented using the same principle (see Figure 5.2 for dilation and erosion with a 3×3 structuring element applied to a 2D image). Note that the implementation of grey scale morphology is almost identical to the implementation of convolutions which was shown in Section 3.5.10. Instead of a product table, a sum table is used (line 7 of Listing 5.1). Instead of using the diagonal injection to compute the sum,

Listing 5.1: Implementing dilation using diagonal injection

```
f = Sequence[0, 0, 1, 0, 0, 0, 1, 1, 0]
# Sequence(UBYTE):
# [ 0, 0, 1, 0, 0, 0, 1, 1, 0 ]
k = Sequence[0, 0, 0]
# Sequence(UBYTE):
# [ 0, 0, 0 ]
sum = f.table(k) { |x,y| x + y }
# MultiArray(UBYTE,2):
# [ [ 0, 0, 0 ],
#   [ 0, 0, 0 ],
#   [ 1, 1, 1 ],
#   [ 0, 0, 0 ],
#   [ 0, 0, 0 ],
#   [ 0, 0, 0 ],
#   [ 1, 1, 1 ],
#   [ 1, 1, 1 ],
#   [ 0, 0, 0 ] ]
sum.diagonal { |x,y| x.major y }
# Sequence(UBYTE):
# [ 0, 1, 1, 1, 0, 1, 1, 1, 1 ]
```

input image　　eroded image　　dilated image

Figure 5.2: Grey scale erosion and dilation

Listing 5.2: Implementing a structuring element using convolution and a lookup table

```
img = MultiArray.load_ubyte('morph.png') >= 0x80
# MultiArray(BOOL,2):
# [ [ false, false, false, false, false, false, false, false, ... ],
#   [ false, false, false, false, false, false, false, false, ... ],
#   ....
filter = lazy(3, 3) { |i,j| 1 << (i + j * 3) }.to_usint
# MultiArray(USINT,2):
# [ [ 1, 2, 4 ],
#   [ 8, 16, 32 ],
#   [ 64, 128, 256 ] ]
cross = MultiArray[[false, true, false],
                   [true, true, true],
                   [false, true, false] ]
# MultiArray(BOOL,2):
# [ [ false, true, false ],
#   [ true, true, true ],
#   [ false, true, false ] ]
bit_mask = cross.conditional(filter, 0).inject { |a,b| a | b }
# 186
lut = finalise(0x200) { |i| (i & bit_mask).ne 0 }
# Sequence(BOOL):
# [false, false, true, true, false, false, true, true, true, true, ....]
img.to_ubyte.convolve(filter).lut(lut).show
```

the maximum is computed (line 18).

5.1.2.2 Binary Morphology

Listing 5.2 shows how one can implement binary dilation with a non-trivial structuring element using convolution and a lookup table according to Gerritsen and Verbeek (1984). The binary image (line 1) is convolved (line 27) with a filter (line 6) so that the bits of each pixel of the result are a copy of the local 3×3 region of the binary input image. Since there are only $2^{3 \cdot 3} = 512$ possible values, one can use a lookup table (line 20) to implement any morphological operation. Figure 5.3 shows the result of Listing 5.2 which implements binary dilation with a cross-shaped structuring element (line 11 of Listing 5.2).

input image   dilated image

Figure 5.3: Binary dilation according to Listing 5.2

Listing 5.3: Otsu thresholding

```ruby
class Node
  def otsu(hist_size = 256)
    hist = histogram hist_size
    idx = lazy(hist_size) { |i| i }
    w1 = hist.integral
    w2 = w1[w1.size - 1] - w1
    s1 = (hist * idx).integral
    s2 = to_int.sum - s1
    m1 = w1 > 0
    u1 = m1.conditional s1.to_sfloat / w1, 0
    m2 = w2 > 0
    u2 = m2.conditional s2.to_sfloat / w2, 0
    between_variance = (u1 - u2) ** 2 * w1 * w2
    self > argmax { |i| between_variance[i] }.first
  end
end
if ARGV.size != 2
  raise "Syntax: otsu.rb <input image> <output image>"
end
img = MultiArray.load_ubyte ARGV[0]
img.otsu.conditional(255, 0).show
```

5.1.3 Otsu Thresholding

Otsu thresholding refers to Otsu's algorithm for choosing a threshold for binarising an image (Otsu, 1979). A general thresholding operation is shown in Equation 5.6.

$$b(\vec{x}) = \begin{cases} 0 & g(\vec{x}) < t \\ 1 & \text{otherwise} \end{cases} \tag{5.6}$$

Given a threshold t and an image g the binary image b is obtained by element-wise application of a thresholding operation. Otsu's method chooses the threshold so that the in-class variance of the binarisation is minimal. This is equivalent to maximising the between-class variance (Otsu, 1979). Listing 5.3 shows that, using the argmax operation, Otsu thresholding can be implemented in 21 lines of code.

5.1.4 Gamma Correction

As explained in Section 4.1.1 the response of an sRGB display is non-linear. Figure 5.4 shows two grey scale gradients. The top bar has linear values in memory and the bottom

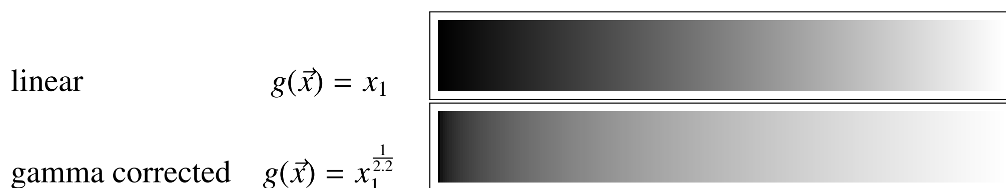


Figure 5.4: Gamma correction

Listing 5.4: Generating a gamma corrected gradient

```
finalise(256, 32) { |i,j| (i / 255.0) ** (1.0 / 2.2) * 255.0 }.show
```

bar has gamma corrected values ($\gamma = 2.2$). That is, only the bottom bar displays a linear slope of intensities on a standard display.

However gamma correction is merely an element-wise binary operation (see Section 3.5.5). Listing 5.4 shows how the gamma corrected bar of Figure 5.4 generated using the element-wise operation "**".

5.1.5 Convolution Filters

Convolution filters (or linear shift-invariant filters) are based on the convolution which was introduced in Section 3.5.10.

5.1.5.1 Gaussian Blur

The Gaussian blur filter is an infinite impulse response (IIR) filter. That is, in practise the filter can only be approximated. Equation 5.7 gives the definition of the one-dimensional (1D) Gauss curve (Forsyth and Ponce, 2003).

$$\phi_\sigma(x) := \frac{1}{\sqrt{2\pi}\,|\sigma|} e^{-\dfrac{x^2}{2\sigma^2}} \tag{5.7}$$

The 1D Gauss filter (for $\sigma = 1$) is shown in Figure 5.5. A boundary for the approximation error of the convolution integral (or sum) can be determined using the error function. That is, given a maximum error ϵ, the minimum filter size $2\,s$ can be determined using Equation 5.8.

$$\begin{aligned}
0 \overset{(*)}{\leq} (g \otimes \phi_\sigma)(x) - \int_{x-s}^{x+s} g(x)\,\phi_\sigma(z-x)\,\mathrm{d}z &= \int_{-\infty}^{x-s} g(x)\,\phi_\sigma(z-x)\,\mathrm{d}z + \int_{x+s}^{\infty} g(x)\,\phi_\sigma(z-x)\,\mathrm{d}z \\
&\overset{(*)}{\leq} 2\,G \int_{x+s}^{\infty} \phi_\sigma(z-x)\,\mathrm{d}z \\
&= G\left(1 - \operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)\right) \overset{!}{\leq} \epsilon \\
&(*) \text{ using } \forall x \in \mathbb{R} : 0 \leq g(x) \leq G
\end{aligned} \tag{5.8}$$

Note that for $0 \leq g(x) \leq G$ the convolution integral over the range $[-s, +s]$ will always underestimate the value of the IIR filter. For example applying the Gaussian filter to a constant function $g(x) = g_0$ should have no effect but the approximation will be $g_0 \cdot$

$\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)$ which is always below g_0. Therefore it is better to apply a scaling factor resulting in the unbiased approximation shown in Equation 5.9.

$$(g \otimes \phi_\sigma)(x) \approx \frac{1}{\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)} \int_{x-s}^{x+s} g(x)\,\phi_\sigma(z-x)\,\mathrm{d}z \tag{5.9}$$

The error boundaries for the approximation are given in Equation 5.10.

$$\underbrace{G\left(1-\frac{1}{\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)}\right)}_{\approx -\epsilon' \geq -\epsilon} \leq (g\otimes\phi_\sigma)(x) - \frac{1}{\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)} \int_{x-s}^{x+s} g(x)\,\phi_\sigma(z-x)\,\mathrm{d}z \leq \underbrace{G\left(1-\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)\right)}_{\leq \epsilon} \tag{5.10}$$

Note that the lower boundary in Equation 5.10 is below $-\epsilon$. However since $\lim_{x\to\infty} \operatorname{erf}(x) = 1$, the lower boundary can be approximated by $\epsilon' \leq \epsilon$.

The corresponding discrete filter can be generated using the integral of the Gaussian as shown in Equation 5.11.

$$\phi_{\sigma,s}(i) := \frac{1}{\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)} \int_{i-1/2}^{i+1/2} \phi_\sigma(z)\mathrm{d}z = \frac{\operatorname{erf}\left(\frac{i+1/2}{\sqrt{2}\,|\sigma|}\right) - \operatorname{erf}\left(\frac{i-1/2}{\sqrt{2}\,|\sigma|}\right)}{2\,\operatorname{erf}\left(\frac{s}{\sqrt{2}\,|\sigma|}\right)} \tag{5.11}$$

In practise σ and a desired error boundary ϵ are given. Starting with a filter size $s = 1/2$, the filter size s is increased by 1 until the approximation has the desired accuracy. For example if $s = 5/2$ would be sufficient, the filter would have 5 elements ($i \in \{-2, -1, 0, 1, 2\}$) as shown in Figure 5.5.

Gaussian filters for higher dimensions are essentially separable filters where each component is equal to the 1D Gaussian filter. See Equation 5.12 for the 2D Gauss filter for example (Hinderer, 1993).

$$\varphi_\sigma : \begin{cases} \mathbb{R}^2 & \to & \mathbb{R} \\ \vec{x} & \mapsto & \frac{1}{2\,\pi\,\sigma^2} e^{-\frac{|\vec{x}|^2}{2\,\sigma^2}} \end{cases} \tag{5.12}$$

The 2D Gauss filter (for $\sigma = 1$) is shown in Figure 5.6. Since the 2D filter is separable, the required filter size can be chosen by simply requiring a maximum error of $\epsilon/2$ for each component so that the overall error boundary is ϵ. Figure 5.7 shows application of a Gaussian filter with $\sigma = 3$ and $\epsilon = 1/256$ applied to a colour image.

5.1.5.2 Gaussian Gradient

The Gauss gradient filter is an IIR filter as well. It is commonly used to locate steps or edges in a signal. Equation 5.13 shows the Gauss gradient filter which is simply the

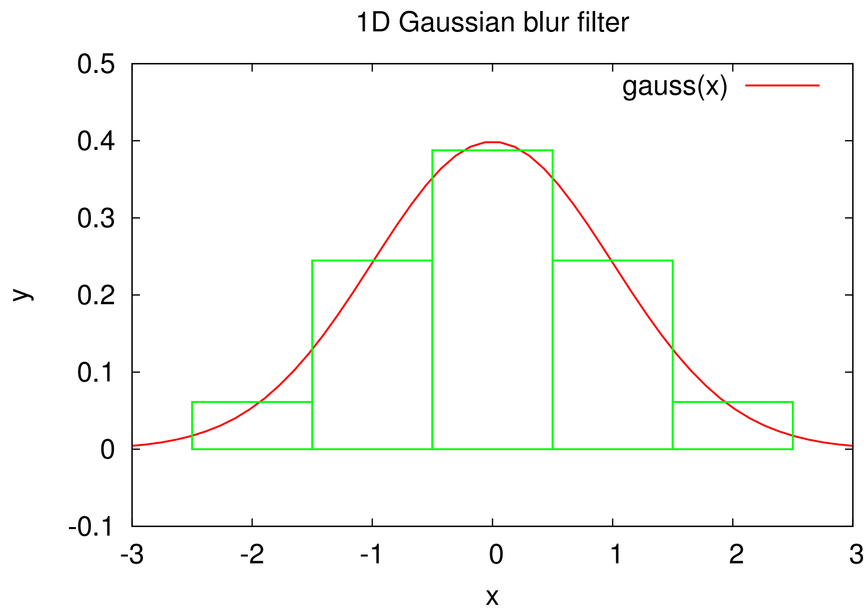


Figure 5.5: 1D Gaussian blur filter

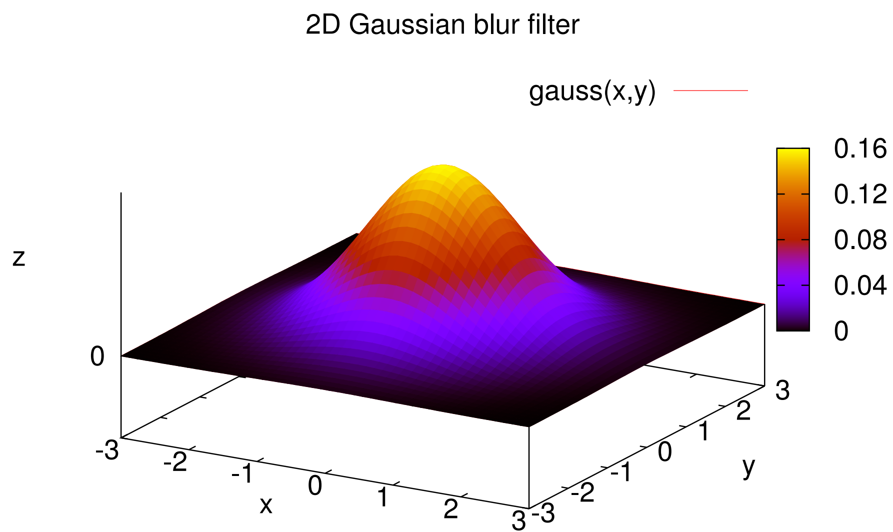


Figure 5.6: 2D Gaussian blur filter

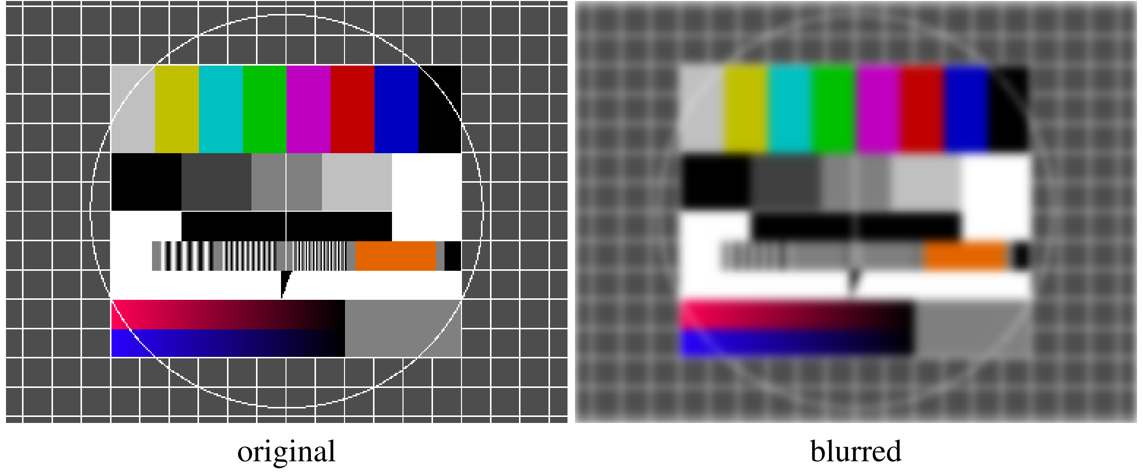


Figure 5.7: Gaussian blur ($\sigma = 3$, $\epsilon = 1/256$) applied to a colour image

derivative of the Gauss filter shown earlier in Equation 5.7.

$$\phi'_\sigma(x) = -\frac{x}{\sqrt{2\,\pi}\,|\sigma|^3}\,e^{-\dfrac{x^2}{2\,\sigma^2}} \tag{5.13}$$

Figure 5.8 shows the 1D Gauss gradient filter for $\sigma = 1$. The minimal filter size for a given error bound can be determined in a similar fashion as was shown for the Gauss filter in Section 5.1.5.1.

Error boundaries for the Gauss gradient filter are estimated in a similar fashion as for the Gauss blur filter. However here absolute integrals are used as shown in Equation 5.14 since the Gauss gradient filter has negative values.

$$\begin{aligned} \left|(g \otimes \phi'_\sigma)(x) - \int_{x-s}^{x+s} g(x)\,\phi'_\sigma(z-x)\,\mathrm{d}z\right| &\le \int_{-\infty}^{x-s} \left|g(x)\,\phi'_\sigma(z-x)\right|\mathrm{d}z + \int_{x+s}^{\infty} \left|g(x)\,\phi'_\sigma(z-x)\right|\mathrm{d}z \\ &\overset{(*)}{\le} 2\,G \int_{x+s}^{\infty} \left|\phi'_\sigma(z-x)\right|\mathrm{d}z \\ &= 2\,G\,\phi_\sigma(s) \overset{!}{\le} \epsilon \\ &(*) \text{ using } \forall x \in \mathbb{R} : 0 \le g(x) \le G \end{aligned} \tag{5.14}$$

The corresponding discrete filter can be generated using the Gaussian as shown in Equation 5.15.

$$\phi'_{\sigma,s}(i) := \frac{1}{C}\int_{i-1/2}^{i+1/2} \phi'_\sigma(z)\mathrm{d}z = \frac{\phi_\sigma(i+1/2) - \phi_\sigma(i-1/2)}{C} \tag{5.15}$$

A normalisation constant C is used so that the convolution of a linear function $f(i) = a\,i + b$ with the Gauss gradient filter will result in the constant function $f'(i) = a$. That is, C is

chosen so that $\sum_i i\,\phi_{\sigma,s}(i) = 1$. Figure 5.8 shows the result for a filter with 5 elements ($s = 5/2$).

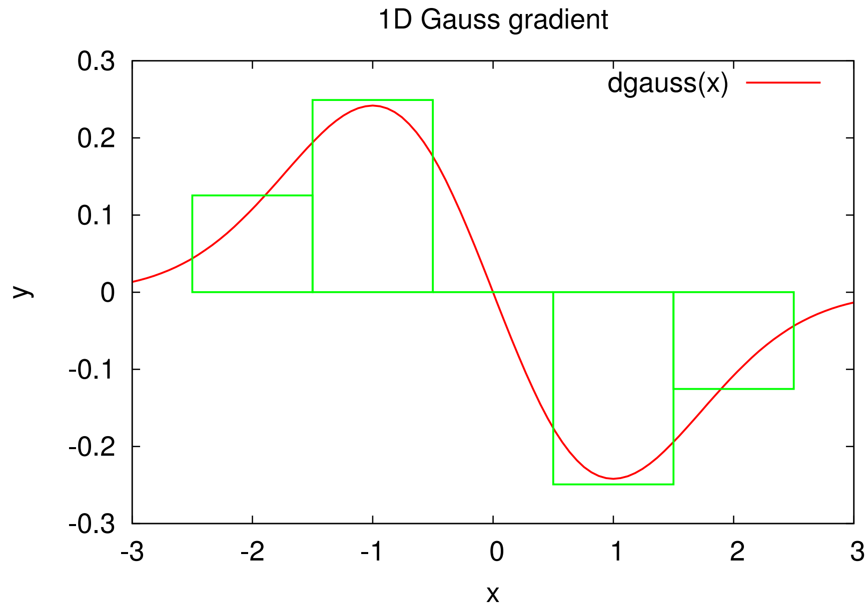


Figure 5.8: 1D Gauss gradient filter

The Gauss gradient filter for a higher dimension is the derivative of the corresponding Gauss filter. For example the Gauss gradient filter for the 2D case is shown in Equation 5.16.

$$\varphi'_\sigma : \begin{cases} \mathbb{R}^2 & \rightarrow & \mathbb{R}^2 \\ \vec{x} & \mapsto & -\dfrac{\vec{x}}{2\,\pi\,\sigma^4}\,e^{-\dfrac{|\vec{x}|^2}{2\,\sigma^2}} \end{cases} \tag{5.16}$$

Note that the derivative has two components as shown in Figure 5.9. Figure 5.10 shows

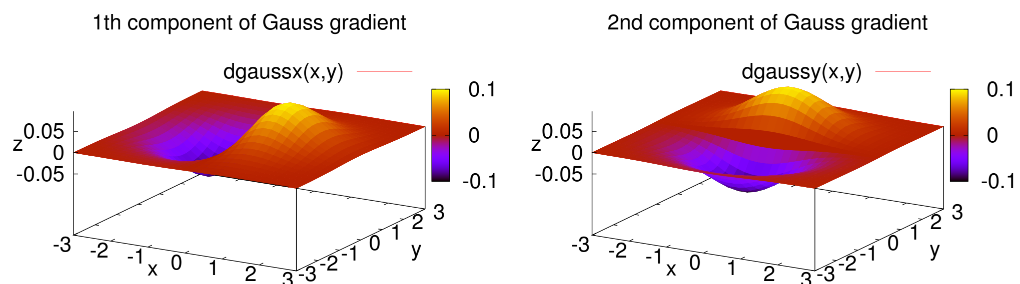


Figure 5.9: 2D Gauss gradient filter

application of a Gauss gradient filter with $\sigma = 3$ and $\epsilon = 1/256$ applied to a colour image. Note that the images of the gradient components are normalised. It is not possible to display negative values.

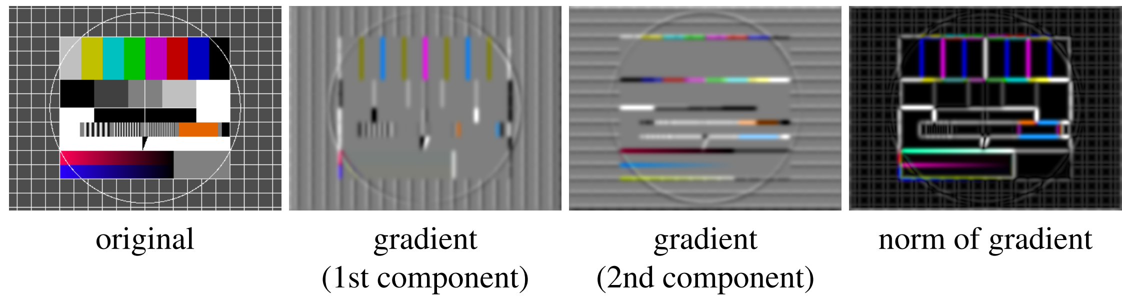


Figure 5.10: Gauss gradient filter ($\sigma = 3$, $\epsilon = 1/256$) applied to a colour image

5.1.6 Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm to compute the values of the discrete Fourier transform (DFT) by using the divide-and-conquer principle. While a naive implementation of the DFT is of complexity $O(n^2)$ (with n being the number of samples), FFT is of complexity $O(n\ \log(n))$. Here the FFTW version 3 (FFTW3) library was used. The FFTW3 library provides an optimiser which selects a combination of different decomposition methods to facilitate the recursion (Frigo and Johnson, 2005). The FFTW3 library implements Fourier transforms of multidimensional arrays. It also implements Fourier transforms of real data exploiting the fact that the Fourier transform of real data is symmetric complex data. In contrast many FFT implementations only implement the radix-2 Cooley-Tukey algorithm (Cooley and Tukey, 1965) which requires the array dimensions to be a power of two or or they implement the mixed-radix Cooley-Tukey algorithm with a worst case complexity of $O(n^2)$ (*i.e.* for n being a prime number).

Equation 5.17 shows the definition of the 1D Fourier transform.

$$\mathcal{F}\{f\}(\omega) := \int_{-\infty}^{\infty} f(x)\, e^{-2\pi x \omega} \mathrm{d}x \tag{5.17}$$

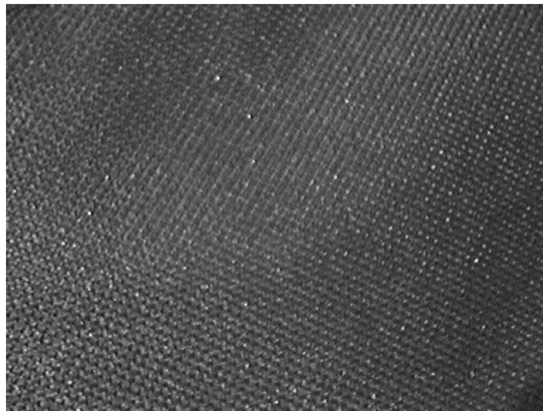
The power spectrum of a signal can be estimated by computing the Fourier transform of the autocorrelated signal (see Equation 5.18).

$$\mathcal{F}^{*}\{f\}\mathcal{F}\{f\} \tag{5.18}$$

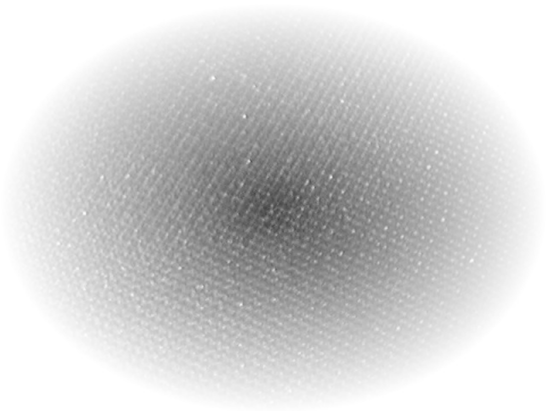
In practise the signal always is finite. In order to implement a consistent estimator for the spectrum of a signal, it is necessary to apply a window function (*e.g.* a triangular window as shown in lines 2–9 of Listing 5.5) and to use zero padding (lines 10–14 of Listing 5.5) in order to avoid cyclical convolution. Figure 5.11 shows the optical image of a piece of Nylon fabric and the steps to obtain a spectral estimate of the signal. Since the pattern is self-similar, it shows pronounced peaks in the spectral estimate.

Listing 5.5: Estimating the spectrum of a 2D signal

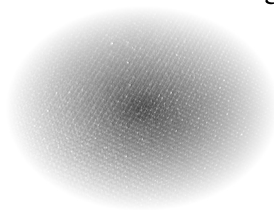
```
class Node
  def window
    finalise do |i,j|
      x = ((i + 0.5 - 0.5 * width ) / (0.5 * width )).abs
      y = ((j + 0.5 - 0.5 * height) / (0.5 * height)).abs
      w = (1 - Math.sqrt(x ** 2 + y ** 2)).major 0.0
      self[i,j] * w
    end
  end
  def zeropad
    retval = MultiArray.new(typecode, 2 * width, 2 * height).fill!
    retval[0 ... width, 0 ... height] = self
    retval
  end
  def spectrum
    fft = window.zeropad.fft
    (fft.conj * fft).real
  end
end
img = MultiArray.load_ubyte 'test.png'
(img.spectrum ** 0.1).normalise(255 .. 0).show
```



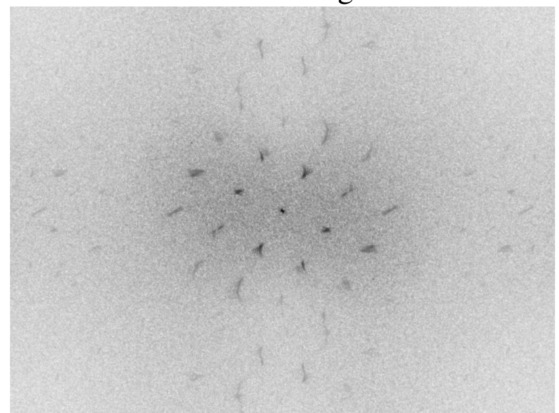
original



windowing



zero padding



spectral estimate

Figure 5.11: Spectral image of a piece of fabric

Listing 5.6: Roberts' Cross edge-detector

```
class Node
  def roberts
    r1 = MultiArray(SINT,2)[[-1,  0], [0,  1]]
    r2 = MultiArray(SINT,2)[[ 0, -1], [1,  0]]
    convolve(r1).abs + convolve(r2).abs
  end
end
img = MultiArray.load_ubyte 'test.png'
img.roberts.normalise(255 .. 0).show
```

5.2 Feature Locations

Many algorithms for object recognition and tracking are feature-based in order to make a real-time implementation possible. The input image is reduced to a set of feature locations and descriptors. The low-level operations presented in Section 3.5 can be used to create concise implementations of feature extraction algorithms.

5.2.1 Edge Detectors

5.2.1.1 Roberts' Cross Edge-Detector

The Roberts' Cross edge-detector is based on a pair of 2×2 filter kernels (Fisher et al., 2003). The kernels $\mathcal{R}_1$ and $\mathcal{R}_2$ are shown in Equation 5.19.

$$\mathcal{R}_1 = \begin{pmatrix} -1 & 0 \\ 0 & +1 \end{pmatrix} \text{ and } \mathcal{R}_2 = \begin{pmatrix} 0 & -1 \\ +1 & 0 \end{pmatrix} \tag{5.19}$$

These kernels are designed to respond maximally to edges running at 45° to the pixel grid, one kernel for each of the two perpendicular orientations (Fisher et al., 2003). For computational efficiency one can use the Manhattan norm in order to estimate the edge strength (see Equation 5.20).

$$|(g \otimes \mathcal{R}_1)(\vec{x})| + |(g \otimes \mathcal{R}_2)(\vec{x})| \tag{5.20}$$

Listing 5.6 shows how one can use open classes in Ruby to extend the "`Node`" class with a method for computing Roberts' Cross edge strength. Figure 5.12 shows the Roberts' Cross edge-detector applied to an example image.

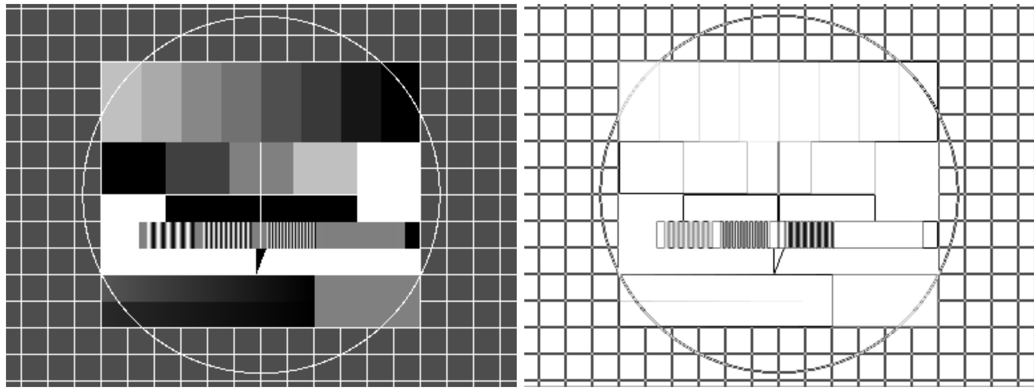


Figure 5.12: Example image and corresponding Roberts' Cross edges

Listing 5.7: Sobel edge-detector

```
class Node
  def sobel
    s1 = MultiArray(SINT,2)[[1, 0, -1], [2, 0, -2], [ 1,  0, -1]]
    s2 = MultiArray(SINT,2)[[1, 2,  1], [0, 0,  0], [-1, -2, -1]]
    Math.sqrt convolve(s1) ** 2 + convolve(s2) ** 2
  end
end
img = MultiArray.load_ubyte 'test.png'
img.sobel.normalise(255 .. 0).show
```

5.2.1.2 Sobel Edge-Detector

The Sobel edge-detector uses two 3×3 filter kernels (Sobel and Feldman, 1968). The kernels $\mathcal{S}_1$ and $\mathcal{S}_2$ are shown in Equation 5.21.

$$\mathcal{S}_1 = \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \text{ and } \mathcal{S}_2 = \frac{1}{4}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{5.21}$$

One can compute the norm of the resulting gradient vector as shown in Equation 5.22.

$$\sqrt{(g \otimes \mathcal{S}_1)^2(\vec{x}) + (g \otimes \mathcal{S}_2)^2(\vec{x})} \tag{5.22}$$

The corresponding Ruby code using open classes is shown in Listing 5.7. Figure 5.13 shows the Sobel edge-detector applied to an example image.

Note that the Prewitt edge detector is based on a similar set of filters. The Prewitt filter kernels $\mathcal{P}_1$ and $\mathcal{P}_2$ are shown in Equation 5.23.

$$\mathcal{P}_1 = \frac{1}{3}\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \text{ and } \mathcal{P}_2 = \frac{1}{3}\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \tag{5.23}$$

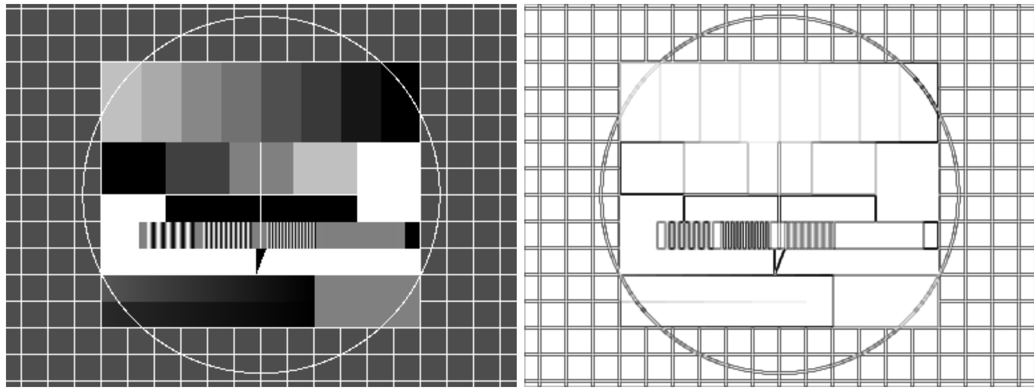


Figure 5.13: Example image and corresponding Sobel edges

Listing 5.8: Non-maxima suppression for edges

```
THRESHOLD = 3.0
SIGMA = 1.4
PI = Math::PI
input = V4L2Input.new
w, h = input.width, input.height
X11Display.show do
  img = input.read_ubyte
  x, y = img.gauss_gradient(SIGMA, 0), img.gauss_gradient(SIGMA, 1)
  norm, angle = Math.hypot(x, y), Math.atan2(y, x)
  orientation = (((2 - 1.0 / 8) * PI + angle) * (4 / PI)).to_ubyte % 4
  idx, idy = lazy(w, h) { |i,j| i }, lazy(w, h) { |i,j| j }
  dx = orientation.lut Sequence[-1,  0,  1,  1]
  dy = orientation.lut Sequence[-1, -1, -1,  0]
  edges = norm >= norm.warp(idx + dx, idy + dy).
    major(norm.warp(idx - dx, idy - dy)).major(THRESHOLD)
  edges.conditional RGB(255, 0, 0), img
end
```

5.2.1.3 Non-Maxima Suppression for Edges

Edge points are usually defined as locations where the gradient magnitude reaches a local maximum in the direction of the gradient vector (Canny, 1986). That is, the gradient norm of each pixel is only to be compared with neighbouring values perpendicular to the edge.

Non-maxima suppression for edges can be implemented using warps as shown in Listing 5.8. The gradient norm of each pixel (line 9) is compared with the gradient norm of two neighbouring pixel (line 14–15). The locations of the two neighbouring pixel depends on the orientation of the gradient as shown in Table 5.1.

The result of applying the algorithm to an image is shown in Figure 5.14.

5.2.2 Corner Detectors

5.2.2.1 Corner Strength by Yang *et al.*

Yang et al. (1996) shows a corner detector which uses the local distribution of gradients

Table 5.1: Non-maxima suppression for edges depending on the orientation

orientation	0	1	2	3
locations for comparison				

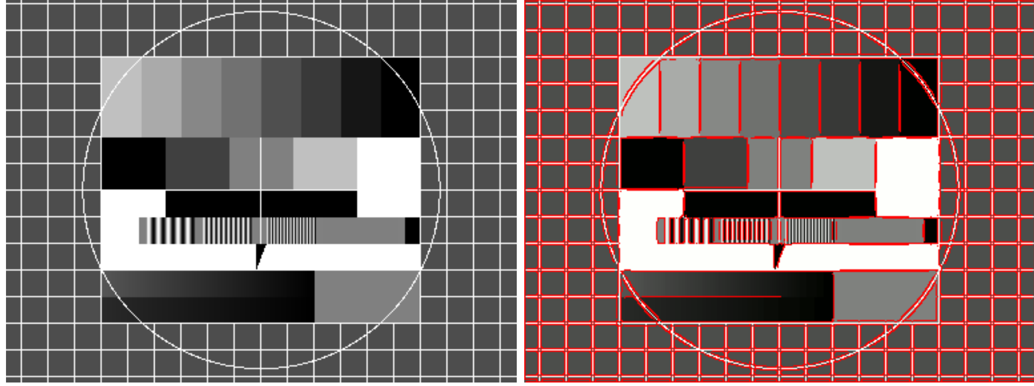


Figure 5.14: Non-maxima suppression for edges

to detect corners. An anisotropic measure is defined which is high if the covariance of the gradient vectors in the local area Ω around $\vec{x}$ is low. The values of the 2×2 structure tensor $\mathcal{S}$ are shown in Equation 5.24.

$$\mathcal{S}\{g\}(\vec{x}) := \begin{pmatrix} \iint_\Omega (\frac{\delta g}{\delta x_1})^2 \mathrm{d}x_1 \mathrm{d}x_2 & \iint_\Omega (\frac{\delta g}{\delta x_1})(\frac{\delta g}{\delta x_2}) \mathrm{d}x_1 \mathrm{d}x_2 \\ \iint_\Omega (\frac{\delta g}{\delta x_1})(\frac{\delta g}{\delta x_2}) \mathrm{d}x_1 \mathrm{d}x_2 & \iint_\Omega (\frac{\delta g}{\delta x_2})^2 \mathrm{d}x_1 \mathrm{d}x_2 \end{pmatrix} \tag{5.24}$$

The elements of the structure tensor are used to define an anisotropic measure (for a continuous, infinite image g) as shown in Equation 5.25.

$$\mathrm{YBFU}\{g\}(\vec{x}) := \frac{\left(\iint_\Omega (\frac{\delta g}{\delta x_1})^2 - (\frac{\delta g}{\delta x_2})^2 \mathrm{d}x_1 \mathrm{d}x_2\right)^2 + \left(\iint_\Omega 2(\frac{\delta g}{\delta x_1})(\frac{\delta g}{\delta x_2}) \mathrm{d}x_1 \mathrm{d}x_2\right)^2}{\left(\sigma^2 + \iint_\Omega (\frac{\delta g}{\delta x_1})^2 + (\frac{\delta g}{\delta x_2})^2 \mathrm{d}x_1 \mathrm{d}x_2\right)^2} \tag{5.25}$$

This heuristic function emphasises regions which have large gradient vectors as well as high anisotropy.

Listing 5.9 show the corresponding Ruby code. The Gaussian gradient is used to estimate the local gradient of the image (lines 6 and 7). Instead of a local area Ω, a Gaussian blur is used to perform a weighted sum in order to compute the local structure tensor $\mathcal{S}$ (lines 8–10). Figure 5.15 shows a result obtained with this technique.

5.2.2.2 Shi-Tomasi Corner Detector

Shi and Tomasi (1994) introduced a corner-detector based on the eigenvalues of the struc-

Listing 5.9: Yang *et al.* corner detector

```
1  GRAD_SIGMA = 2.0
2  COV_SIGMA = 1.0
3  NOISE = 1.0
4  EXP = 0.5
5  img = MultiArray.load_ubyte 'test.png'
6  x = img.gauss_gradient GRAD_SIGMA, 0
7  y = img.gauss_gradient GRAD_SIGMA, 1
8  a = (x ** 2).gauss_blur COV_SIGMA
9  b = (y ** 2).gauss_blur COV_SIGMA
10 c = (x * y ).gauss_blur COV_SIGMA
11 g = ((a - b) ** 2 + (2 * c) ** 2) / (a + b + NOISE ** 2) ** 2
12 result = g.normalise(1.0 .. 0.0) ** EXP * (x ** 2 + y ** 2)
13 result.normalise(0xFF .. 0).show
```

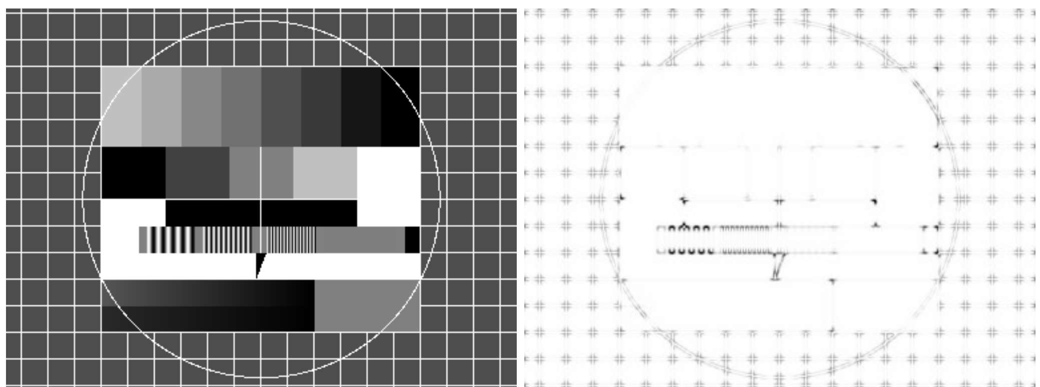


Figure 5.15: Corner detection by Yang *et al.*

Listing 5.10: Shi-Tomasi corner detector

```
GRAD_SIGMA = 1
COV_SIGMA = 1
img = MultiArray.load_ubyte 'test.png'
x = img.gauss_gradient GRAD_SIGMA, 0
y = img.gauss_gradient GRAD_SIGMA, 1
a = (x ** 2).gauss_blur COV_SIGMA
b = (y ** 2).gauss_blur COV_SIGMA
c = (x * y ).gauss_blur COV_SIGMA
tr = a + b
det = a * b - c * c
# "major" is needed to deal with numerical errors.
dissqrt = Math.sqrt((tr * tr - det * 4).major(0.0))
# Take smallest eigenvalue. Eigenvalues are "0.5 * (tr +- dissqrt)"
result = 0.5 * (tr - dissqrt)
result.normalise(0xFF .. 0).show
```

ture tensor of the gradient vectors in a local region. The heuristic function chosen here is simply the smallest eigenvalue of the structure tensor $\mathcal{S}$ (see Equation 5.26).

$$\mathrm{ST}\{g\}(\vec{x}) := \min(\lambda_1, \lambda_2) \text{ where } \exists \Lambda \in \mathbb{R}^{2\times 2} : \mathcal{S}\{g\}(\vec{x}) = \Lambda^\top \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \Lambda \wedge \Lambda^\top \Lambda = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{5.26}$$

This corner-detector was developed with the motivation to find features which are suitable for tracking. Tomasi and Kanade (1992) demonstrates that the corner-detector indeed is suitable to serve as a basis for a stable tracking algorithm. The corner detector also was used to estimate stereo disparity (Lucas and Kanade, 1981).

Listing 5.10 shows a implementation of the Shi-Tomasi corner detector in Ruby. The result of applying the Shi-Tomasi corner detector to an image is shown in Figure 5.16.

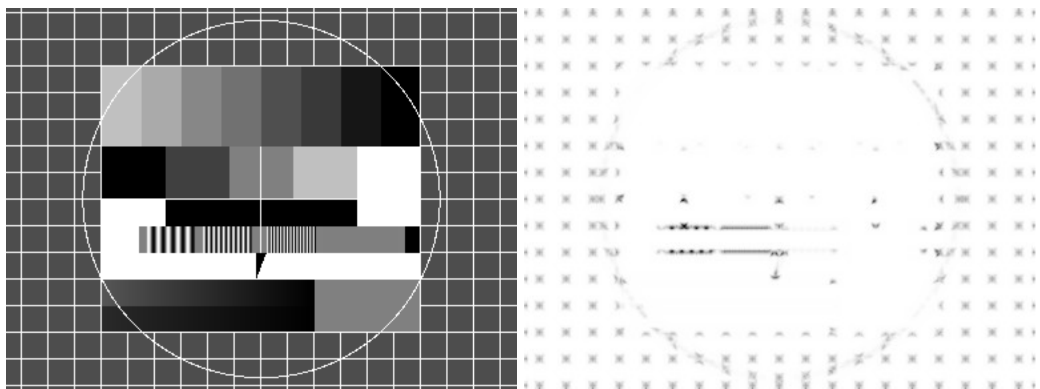


Figure 5.16: Shi-Tomasi corner-detector

5.2.2.3 Harris-Stephens Corner- and Edge-Detector

Harris and Stephens, 1988 (also see Derpanis (2004)) have developed a filter which can detect corners as well as edges. Similar to the approach by Shi and Tomasi a measure

based on the covariance of the gradient vectors is used. A heuristic function is chosen which has a large positive value where there is a high anisotropy (*i.e.* a corner). If there are large gradient vectors with high covariance the function has a high negative value (*i.e.* an edge). The heuristic measure uses a constant k as shown in Equation 5.27.

$$\mathrm{HS}_k\{g\}(\vec{x}) := \lambda_1\,\lambda_2 - k\,(\lambda_1 + \lambda_2)^2 \text{ where } \lambda_1 \text{ and } \lambda_2 \text{ defined as in Equation 5.26} \quad (5.27)$$

Figure 5.17 shows how the response function behaves for different values of λ_1 and λ_2. The implementation of the Harris-Stephens corner and edge detector is given in List-

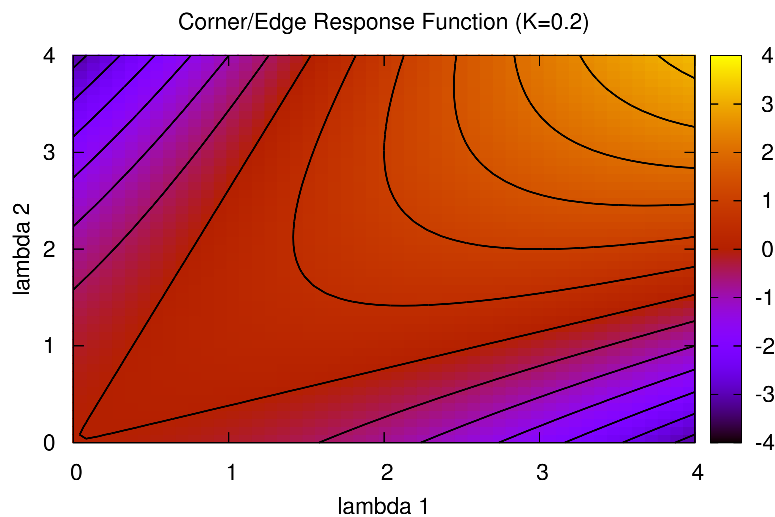


Figure 5.17: Harris-Stephens response function

ing 5.11. Note that the filter was implemented as a method by extending the class "`Node`". The result of applying the filter (with $k = 0.05$) to an image is shown in Figure 5.18.

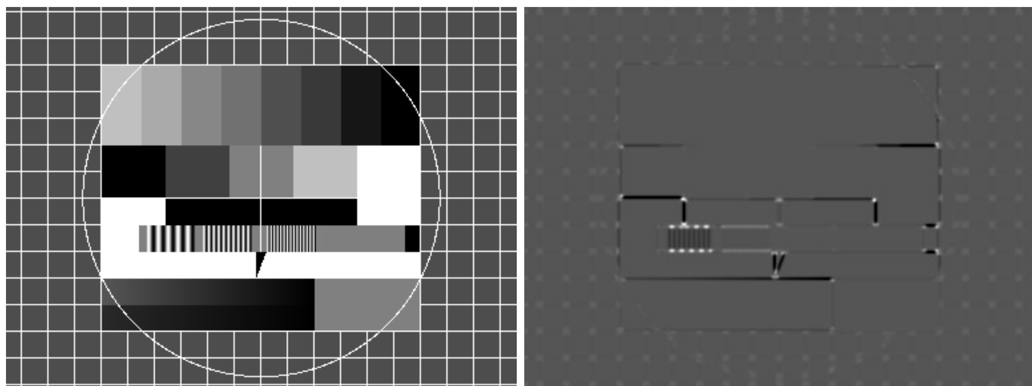


Figure 5.18: Harris-Stephens corner- and edge-detector (negative values (edges) are black and positive values (corners) are white)

Listing 5.11: Harris-Stephens corner and edge detector

```
class Node
  def features(grad_sigma, cov_sigma, k)
    x = gauss_gradient grad_sigma, 0
    y = gauss_gradient grad_sigma, 1
    a = (x ** 2).gauss_blur cov_sigma
    b = (y ** 2).gauss_blur cov_sigma
    c = (x * y ).gauss_blur cov_sigma
    trace = a + b
    determinant = a * b - c * c
    determinant - k * trace ** 2
  end
end
GRAD_SIGMA = 1
COV_SIGMA = 1
K = 0.05
img = MultiArray.load_ubyte 'test.png'
img.features(GRAD_SIGMA, COV_SIGMA, K).normalise.show
```

5.2.2.4 Non-Maxima Suppression for Corners

Corner locations are local maxima of the corner image. A local maximum can be determined by comparison of the corner image with the grey scale dilation as introduced in Section 5.1.2.1. Listing 5.12 shows an implementation of the Harris-Stephens corner- and edge-detector followed by non-maxima suppression for corners (*i.e.* "`Node#maxima`"). The result of applying the algorithm to an image is shown in Figure 5.19.

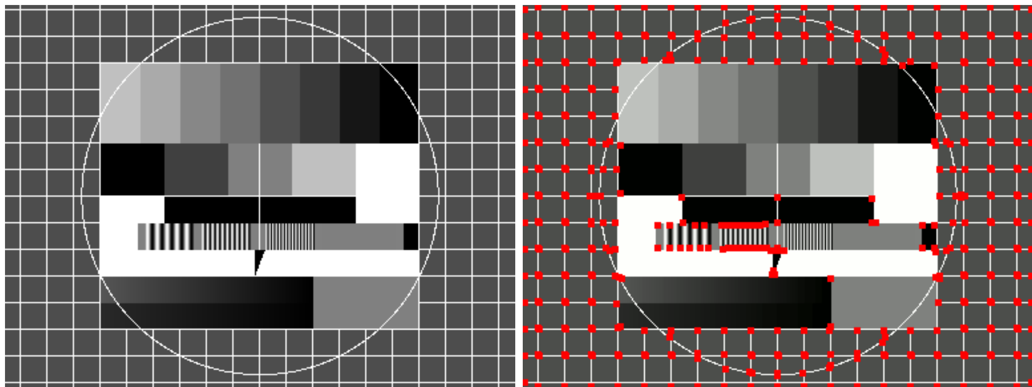


Figure 5.19: Non-maxima suppression for corners

5.3 Feature Descriptors

5.3.1 Restricting Feature Density

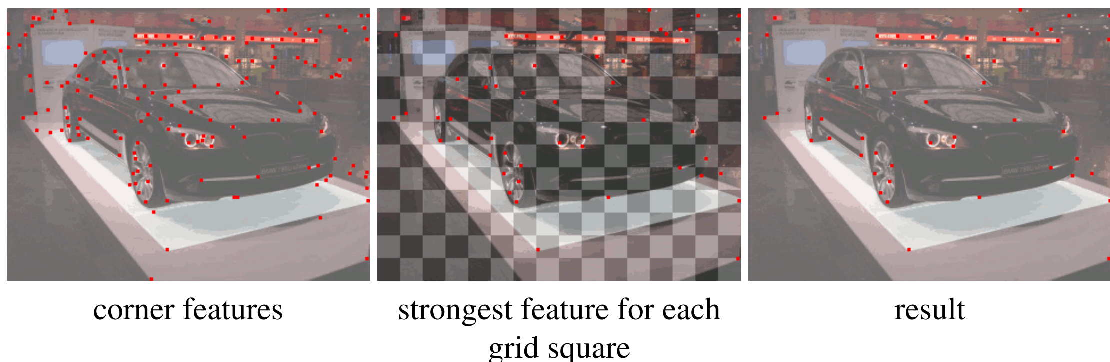
In Section 5.2.2.3 it was shown how to obtain a feature image using the Harris-Stephens corner-detector. The individual corners can be extracted using non-maxima suppression for corners (as shown in Section 5.2.2.4). Usually a certain threshold is used to ignore

Listing 5.12: Non-maxima suppression for corners

```
class Node
  def maxima(threshold)
    (self >= threshold * max).and eq(dilate)
  end
  def features(sigma, k)
    gx, gy = gauss_gradient(sigma, 0), gauss_gradient(sigma, 1)
    cov = [gx ** 2, gy ** 2, gx * gy]
    a, b, c = cov.collect { |arr| arr.gauss_blur sigma }
    trace = a + b
    determinant = a * b - c ** 2
    determinant - k * trace ** 2
  end
end
THRESHOLD = 0.05
SIGMA = 1.0
K = 0.1
img = MultiArray.load_ubyte 'test.png'
features = img.features SIGMA, K
mask = features.maxima THRESHOLD
mask.dilate(5).conditional(RGB(255, 0, 0), img).show
```

weak features. However in order to be able to track motions in every part of the image, it is desirable to have a constant feature density. Bouget (also see OpenCV source code) computes the distance of each new feature to every already accepted feature. The feature is rejected if the minimum distance is below a certain threshold.

Figure 5.20 illustrates a different approach which is based on array operations. A warp



corner features | strongest feature for each grid square | result

Figure 5.20: Restricting feature density

is used to transform the feature image into a 3D array of blocks. Afterwards the strongest feature in each block can be determined using the “**argmax**” method as shown below.

```
maxima = argmax { |i,j| lazy { |k| features.warp(*warp)[i,j,k] } }
```

This enforces an upper limit on the feature density. The complete source code is shown in Section A.7.6.

Listing 5.13: Extracting local texture patches

```
class Node
  def maxima(threshold)
    (self >= threshold * max).and eq(dilate)
  end
  def features(sigma, k)
    gx, gy = gauss_gradient(sigma, 0), gauss_gradient(sigma, 1)
    cov = [gx ** 2, gy ** 2, gx * gy]
    a, b, c = cov.collect { |arr| arr.gauss_blur sigma }
    trace = a + b
    determinant = a * b - c ** 2
    determinant - k * trace ** 2
  end
  def descriptors(img, r)
    x = lazy(*shape) { |i,j| i }.mask self
    y = lazy(*shape) { |i,j| j }.mask self
    dx = lazy(2 * r + 1, 2 * r + 1) { |i,j| i - r }
    dy = lazy(2 * r + 1, 2 * r + 1) { |i,j| j - r }
    warp_x = lazy { |i,j,k| x[k] + dx[i,j] }
    warp_y = lazy { |i,j,k| y[k] + dy[i,j] }
    img.warp warp_x, warp_y
  end
end
THRESHOLD = 0.17
SIGMA = 1.0
K = 0.1
R = 4
img = MultiArray.load_ubytergb 'test.png'
features = img.to_sfloat.features SIGMA, K
mask = features.maxima THRESHOLD
descriptors = mask.descriptors img, R
```

5.3.2 Local Texture Patches

Feature matching algorithms usually use descriptors for feature matching. Here the feature descriptors are based on local image patches around each corner feature. The method "**Node#descriptors**" in Listing 5.13 creates a 3D field of 2D warp vectors for extracting the descriptors. The components of the vectors are stored in the variables "**warp_x**" and "**warp_y**". By applying the warp to the image one can obtain a stack of local image patches (*i.e.* a 3D array) in an efficient manner. See Figure 5.21 for an illustration.

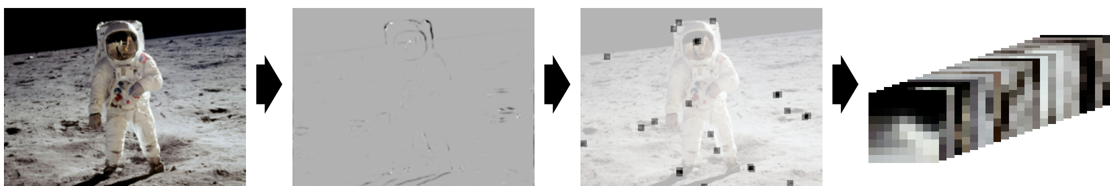


Figure 5.21: Computing feature locations and descriptors

Listing 5.14: SVD matching

```
# ...
measure = proximity * similarity
t, d, ut = *measure.svd
e = lazy(*d.shape) { |i,j| i.eq(j).conditional 1, 0 }
s = t.x(e).x(ut)
max_col = argmax { |j| lazy { |i| s[j, i] } }.first
max_row = argmax { |j| lazy { |i| s[i, j] } }.first
mask_col = [lazy(s.shape[0]) { |i| i }, max_row].histogram(*s.shape) > 0
mask_row = [max_col, lazy(s.shape[1]) { |i| i }].histogram(*s.shape) > 0
q = mask_col.and(mask_row).and measure >= CUTOFF
# ...
```

5.3.3 SVD Matching

Pilu (1997) presents a simple algorithm for feature matching based on the singular value decomposition (SVD). A combined measure of the feature proximity and similarity id computed for every possible pair of features (I_i, I_j) with descriptors A and B (see Equation 5.28).

$$g_{j,i} := e^{-\frac{(c_{j,i}-1)^2}{2\gamma^2}} \cdot e^{-\frac{r_{j,i}^2}{2\sigma^2}} \text{ where } c_{j,i} := \frac{\sum_{u,v}(A_{uv} - \bar{A})(B_{uv} - \bar{B})}{\sum_{u,v}(A_{uv} - \bar{A})^2 \sum_{u,v}(B_{uv} - \bar{B})^2} \text{ and } r_{i,j} := \left|I_i - I_j\right| \tag{5.28}$$

Equation 5.29 shows the SVD of the resulting matrix $\mathcal{G}$ (Pilu, 1997).

$$\mathcal{G} := \mathcal{T}\,\mathcal{D}\,\mathcal{U} \text{ where } \mathcal{D} = \operatorname{diag}(d_1, d_2, \ldots) \tag{5.29}$$

Equation 5.30 shows how the resulting matrices $\mathcal{T}$ and $\mathcal{U}$ are multiplied with the rectangular identity matrix $\mathcal{E}$ (Pilu, 1997).

$$\mathcal{P} := \mathcal{T}\,\mathcal{E}\,\mathcal{U} \tag{5.30}$$

A pair of features (I_j, I_i) is regarded a match if and only if $p_{j,i}$ is both the greatest element in its row and the greatest element in its column (Pilu, 1997).

Listing 5.14 shows the corresponding implementation. The "`argmax`" function is used to locate the maximum in each row and column of the correspondence matrix. The histogram operation is used to create two masks. The two masks are combined to detect elements in the matrix which are the greatest element in its row as well as in its column. Additionally a threshold is introduced in order to discard matches with low correspondence (see Section A.7.7 for complete code listing). Figure 5.22 shows an example of the algorithm in action.

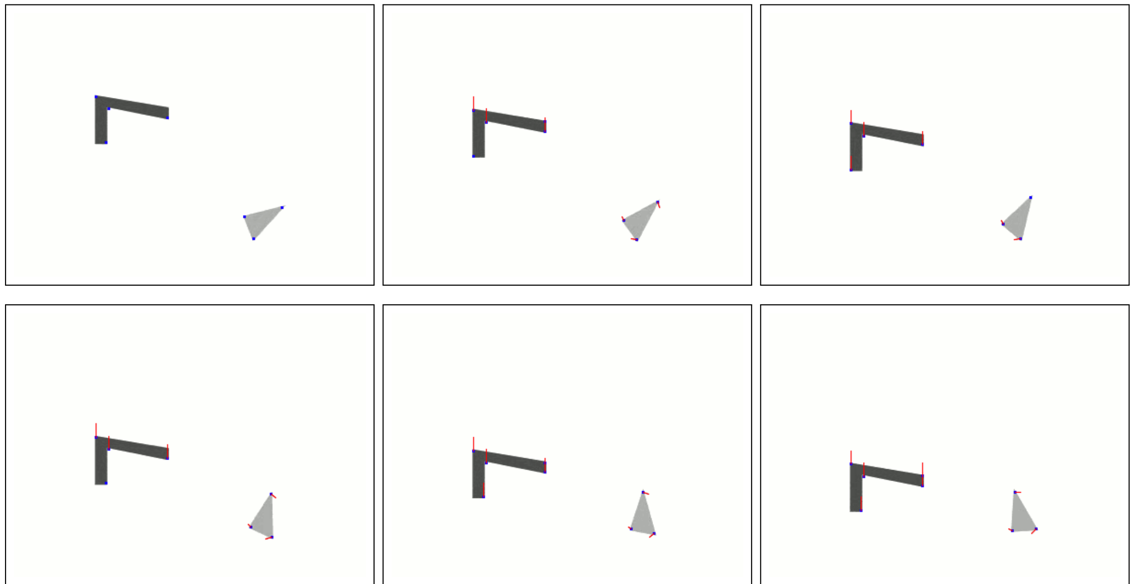


Figure 5.22: SVD tracking

5.4 Summary

This chapter has shown how many preprocessing operations commonly found in the area of machine vision can be implemented using the library introduced in Chapter 3. The concise code also made the similarities of different popular corner detectors visible. Finally it was shown how warps can be used to extract feature descriptors from selected regions of the input image.

Evaluation

"Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best."

Paul Graham

"We haven't found intelligent life so far. Some people believe it has yet to occur on earth."

Stephen Hawking

"If you're using early-binding languages as most people do, rather than late-binding languages, then you really start getting locked in to stuff that you've already done."

Alan Kay

This chapter shows how the concepts introduced in previous chapters can be used to create concise implementations of real-time machine vision systems.

- Section 6.1 describes the FOSS packages developed as part of this thesis
- Section 6.2 presents concise implementations of several computer vision applications
- Section 6.3 provides a performance analysis and a comparison with equivalent C implementations
- Section 6.5 gives a summary of this chapter

6.1 Software Modules

The software developed as part of this thesis was made available as free software and released on Github[1] and on Rubygems[2]. It is packaged as follows.

- The package malloc defines the "**`Malloc`**" class (also see Section 3.2). "**`Malloc`**" objects are used to introduce pointer operations to the Ruby language. Pointer objects are the simplest possible interface for exchanging image- and video-data. Instead of requiring the different I/O-packages to use certain static types for exchanging information, each I/O-package comes with dynamic types providing the meta-information.

[1] http://github.com/
[2] http://rubygems.org/

- The multiarray extension provides the image processing operations presented in Chapter 3. The package uses GNU Compiler Collection (GCC) as a JIT compiler in order to achieve high performance.
- Conversion of compressed video frames is handled by the hornetseye-frame Ruby extension. The Ruby extension makes use of the FFmpeg rescaling library.
- The following packages implement various I/O-interfaces.
  - The hornetseye-xorg package can be used to display images and videos using the X Window System.
  - hornetseye-rmagick provides saving and loading of image files using the RMagick library.
  - hornetseye-ffmpeg provides saving and loading of video files using the FFmpeg library.
  - hornetseye-alsa provides capture and playback of audio using ALSA.
  - hornetseye-fftw3 provides Fast Fourier Transforms using the FFTW3 library.
  - hornetseye-v4l2 provides live camera images using V4L2.
  - hornetseye-dc1394 provides live camera images using DC1394-compatible Firewire cameras.
  - hornetseye-kinect provides capture of depth images using the Microsoft Kinect.
  - hornetseye-openexr provides loading and saving of HDR images with the OpenEXR library.
- The following packages provide integration with other libraries.
  - hornetseye-opencv provides integration with the Ruby bindings of the OpenCV computer vision library.
  - hornetseye-narray provides conversions between "`NArray`" objects and the arrays of the Hornetseye library.
  - hornetseye-linalg provides conversions between the "`DMatrix`" objects of the linalg library (LAPACK bindings for Ruby) and the arrays of the Hornetseye library.
  - hornetseye-qt4 provides a Qt4 widget for hardware accelerated video output in a Qt4 GUI.

6.2 Assessment of Functionality

6.2.1 Fast Normalised Cross-Correlation

One can use correlation methods in order to detect a 2D template in an image if only translations are involved (*i.e.* no scale or rotation). Lewis (1995) shows how to compute the correlation coefficients γ shown in Equation 6.1 for every possible shift $\vec{u}$.

$$\gamma(\vec{u}) = \frac{\sum_{\vec{x}\in T(\vec{u})} [g(\vec{x}) - \bar{g}(\vec{u})]\,[t(\vec{x}-\vec{u}) - \bar{t}]}{\sqrt{\sum_{\vec{x}\in T(\vec{u})} [f(\vec{x}) - \bar{f}(\vec{u})]^2 \;\sum_{\vec{x}\in T(\vec{u})} [t(\vec{x}-\vec{u}) - \bar{t}]^2}} \tag{6.1}$$

Here g is the input image, $\bar{g}$ is the average of a template-sized area of the input image, t is the template, and $\bar{t}$ is the average value of the template. The integration area $T(\vec{u})$ ensures that $t(\vec{x}-\vec{u})$ is defined for every $\vec{x} \in T(\vec{u})$.

Using $t'(\vec{x}) := t(\vec{x}) - \bar{t}$ one can simplify the numerator of the correlation coefficient as shown in Equation 6.2

$$\overset{\text{num}}{\gamma}(\vec{u}) = \sum_{\vec{x}\in T(\vec{u})} g(\vec{x})\,t'(\vec{x}-\vec{u}) - \bar{g}(\vec{u}) \underbrace{\sum_{\vec{x}\in T(\vec{u})} t'(\vec{x}-\vec{u})}_{=0} \tag{6.2}$$

The numerator requires correlation of the input image with the template which (for larger template sizes) is most efficiently done in the Fourier domain.

The second part of the denominator (see Equation 6.3) simply is the variance of the template. The first part of the denominator can be computed using integral images (Lewis, 1995) (also see Section 3.5.11).

$$\overset{\text{den}}{\gamma^2}(\vec{u}) = \sum_{\vec{x}\in T(\vec{u}} [f^2(\vec{x}) - \bar{f}^2(\vec{u})] \sum_{\vec{x}\in T(\vec{u}} [t(\vec{x}) - \bar{t}]^2 \tag{6.3}$$

The implementation of the normalised cross-correlation is given in Appendix A.7.3.

Figure 6.1 shows how the normalised cross-correlation can be used to successfully locate a template in a test image.

6.2.2 Lucas-Kanade Tracker

The warps introduced in Section 3.5.6.2 can be used to create a concise implementation of a Lucas-Kanade tracker (Baker and Matthew, 2004; Wedekind et al., b; Wedekind, 2008). The (inverse compositional) Lucas-Kanade tracker works by comparing the warped input image with a template. Equation 6.4 shows the warp formula for a 2D isometry with

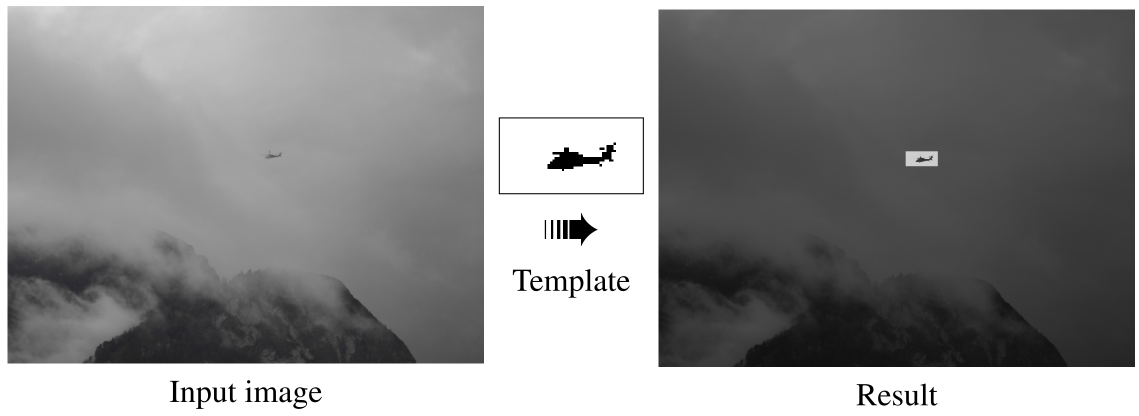


Figure 6.1: Normalised cross-correlation example

translation parameters p_1 and p_2 and rotation parameter p_3.

$$W_{\vec{p}}(\vec{x}) = \begin{pmatrix} x\cos(p_3) - y\sin(p_3) + p_1 \\ x\sin(p_3) + y\cos(p_3) + p_2 \end{pmatrix} \tag{6.4}$$

The vector $\vec{p}$ with the transformation parameters of the warp is updated iteratively by adding an optimal offset $\widehat{\Delta\vec{p}}$ in order to minimise the difference between the template and the warped image (see Equation 6.5 and Figure 6.2).

$$\widehat{\Delta\vec{p}} = \underset{\Delta\vec{p}}{\operatorname{argmin}} \sum_{\vec{x}} \left[t(\vec{x}) - g(W^{-1}_{\vec{p}+\Delta\vec{p}}(\vec{x})) \right]^2 \tag{6.5}$$

The inverse compositional algorithm makes use of the fact that the warp for the changed

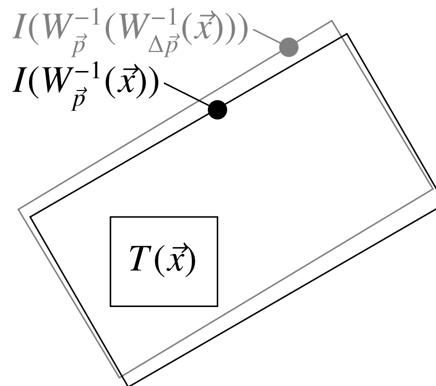


Figure 6.2: Comparison of template and warped image

vector of parameters $\vec{p} + \Delta\vec{p}$ can be approximated by concatenation as shown in Equation 6.6.

$$W_{\vec{p}+\Delta\vec{p}}(\vec{x}) \approx W_{\vec{p}}(W_{\Delta\vec{p}}(\vec{x})) \tag{6.6}$$

The difference between the template and the warped image in Equation 6.5 can be simplified using Equation 6.6 and by substituting $\vec{x} = W_{\Delta\vec{p}}(\vec{x}')$ (see Equation 6.7).

$$t(\vec{x}) - g(W^{-1}_{\vec{p}+\Delta\vec{p}}(\vec{x})) \overset{(6.6)}{=} t(\vec{x}) - g\Big(W^{-1}_{\vec{p}}(W^{-1}_{\Delta\vec{p}}(\vec{x}))\Big) = t(W_{\Delta\vec{p}}(\vec{x}')) - g(W^{-1}_{\vec{p}}(\vec{x}')) \qquad (6.7)$$

Near $\vec{p} = \vec{0}$ the warped template $t(W_{\Delta\vec{p}}(\vec{x}))$ can be approximated with a first order Taylor expansion as shown in Equation 6.8.

$$t(W_{\Delta\vec{p}}(\vec{x})) \approx t(\vec{x}) + \left(\frac{\delta t}{\delta\vec{x}}(\vec{x})\right)^\top \left(\frac{\delta W_{\vec{p}}}{\delta\vec{p}}\bigg|_{\vec{p}=\vec{0}}(\vec{x})\right)\Delta\vec{p} \qquad (6.8)$$

Using Equation 6.7 and the approximation of Equation 6.8 one can reformulate Equation 6.5 as a linear least squares problem (see Equation 6.9).

$$\begin{gathered}
\widehat{\Delta\vec{p}} \approx \underset{\Delta\vec{p}}{\operatorname{argmin}}(||\mathcal{H}\,\Delta\vec{p} + \vec{b}||) = (\mathcal{H}^\top\mathcal{H})^{-1}\mathcal{H}^\top\vec{b} \\
\text{where } \mathcal{H} = \begin{pmatrix} h_{1,1} & h_{1,2} & \cdots \\ h_{2,1} & h_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \text{ and } \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \end{pmatrix} \\
\text{with } h_{i,j} = \left(\frac{\delta t}{\delta\vec{x}}(\vec{x}_i)\right)^\top \cdot \left(\frac{\delta W_{\vec{p}}}{\delta p_j}\bigg|_{\vec{p}=\vec{0}}(\vec{x}_i)\right) \text{ and } b_i = t(\vec{x}_i) - g(W^{-1}_{\vec{p}}(\vec{x}_i))
\end{gathered} \qquad (6.9)$$

In practise it is usually necessary to update $\vec{p}$ a couple of times since Equation 6.9 merely gives an approximation for $\widehat{\Delta\vec{p}}$.

Listing 6.1 shows a Ruby implementation of the (inverse compositional) Lucas-Kanade tracker (using a Gauss gradient with $\sigma = 2.5$ and using two iterations for each video frame). The tracking template is captured (line 30) from the first frame (line 26) of the input video (line 13) using the initial vector of transformation parameters (line 14). Note that the Gauss gradient of the template is computed from a bigger area of the input image in order to avoid boundary effects (see Figure 6.3). Another noteworthy implementation

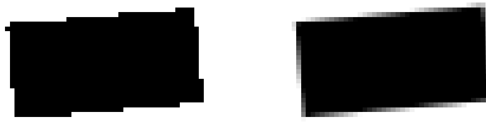
Figure 6.3: Gradient boundaries of template

detail is the interpolated warp (see Figure 6.4). It is mandatory to implement interpolation since the Lucas-Kanade tracker might not converge on a stable solution otherwise.

Figure 6.5 shows every 10th frame of a test video. The Lucas-Kanade tracks the polygon, which undergoes shifts and rotations, successfully.

Listing 6.1: Lucas-Kanade tracker

```
1  class Node
2    def warp_clipped_interpolate(x, y)
3      x0, y0 = x.floor.to_int, y.floor.to_int
4      x1, y1 = x0 + 1, y0 + 1
5      fx1, fy1 = x - x0, y - y0
6      fx0, fy0 = x1 - x, y1 - y
7      return warp(x0, y0) * fx0 * fy0 + warp(x1, y0) * fx1 * fy0 +
8             warp(x0, y1) * fx0 * fy1 + warp(x1, y1) * fx1 * fy1
9    end
10 end
11 SIGMA, N = 2.5, 2
12 W, H = 94, 65
13 input = AVInput.new 'test.avi'
14 p = Vector[80.0, 35.0, 0.0]
15 def model(p, x, y)
16   cw, sw = Math::cos(p[2]), Math::sin(p[2])
17   Vector[x * cw - y * sw + p[0], x * sw + y * cw + p[1]]
18 end
19 def derivative(x, y)
20   Matrix[[1, 0], [0, 1], [-y, x]]
21 end
22 def compose(p, d)
23   cw, sw = Math::cos(p[2]), Math::sin(p[2])
24   p + Matrix[[cw, -sw, 0], [sw,  cw, 0], [0, 0, 1]] * d
25 end
26 img = input.read_ubyte
27 b = (Array.gauss_gradient_filter(SIGMA).size - 1) / 2
28 x = lazy(W + 2 * b, H + 2 * b) { |i,j| i - b }
29 y = lazy(W + 2 * b, H + 2 * b) { |i,j| j - b }
30 tpl = img.warp_clipped_interpolate *model(p, x, y)
31 gx = tpl.gauss_gradient SIGMA, 0
32 gy = tpl.gauss_gradient SIGMA, 1
33 tpl, gx, gy, x, y = *[tpl, gx, gy, x, y].
34                     collect { |arr| arr[b...(W+b), b...(H+b)] }
35 c = derivative(x, y) * Vector[gx, gy]
36 hs = (c * c.covector).collect { |e| e.sum }
37 hsinv = hs.inverse
38 X11Display.show do
39   img = input.read_ubyte
40   for i in 0...N
41     diff = tpl - img.warp_clipped_interpolate(*model(p, x, y))
42     s = c.collect { |e| (e * diff).sum }
43     p = compose(p, hsinv * s)
44   end
45   gc = Magick::Draw.new
46   gc.fill_opacity 0
47   gc.stroke('red').stroke_width 1
48   gc.line *(model(p, 0, 0).to_a + model(p, W, 0).to_a)
49   gc.line *(model(p, 0, H).to_a + model(p, W, H).to_a)
50   gc.line *(model(p, 0, 0).to_a + model(p, 0, H).to_a)
51   gc.line *(model(p, W, 0).to_a + model(p, W, H).to_a)
52   gc.circle *(model(p, 0, 0).to_a + model(p, 3, 0).to_a)
53   result = img.to_ubytergb.to_magick
54   gc.draw result
55   result.to_ubytergb
56 end
```



without interpolation   with interpolation

Figure 6.4: Warp without and with interpolation

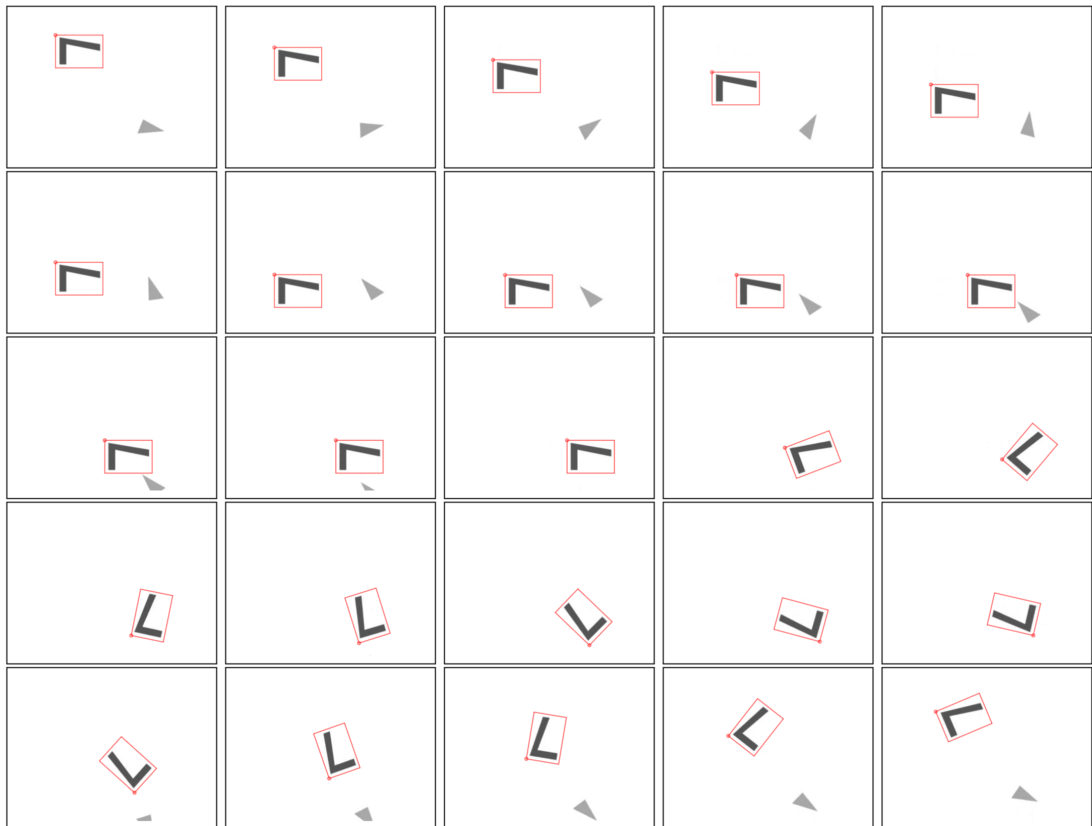


Figure 6.5: Example of Lucas-Kanade tracker in action

Listing 6.2: Hough transform to locate lines

```
A_RANGE = 0 .. 179
THRESHOLD = 0x7F
img = MultiArray.load_ubyte 'test.png'
diag = Math.sqrt(img.width ** 2 + img.height ** 2)
d_range = -diag.to_i.succ.div(2) ... diag.to_i.succ.div(2)
binary = img <= THRESHOLD
x = lazy(*img.shape) { |i,j| i - img.width  / 2 }.mask binary
y = lazy(*img.shape) { |i,j| j - img.height / 2 }.mask binary
angle = lazy(A_RANGE.end + 1) { |i| Math::PI * i / A_RANGE.end }
dist = lazy(d_range.end + 1 - d_range.begin) { |i| i + d_range.begin }
cos, sin = lazy { |i| Math.cos(angle[i]) }, lazy { |i| Math.sin(angle[i]) }
a = lazy(angle.size, x.size) { |i,j| i }
d = lazy { |i,j| (x[j] * cos[i] + y[j] * sin[i] - d_range.begin).to_int }
histogram = [a, d].histogram A_RANGE.end + 1, d_range.end + 1 - d_range.begin
(histogram ** 0.5).normalise(255 .. 0).show
```

6.2.3 Hough Transform

The Hough transform (Duda and Hart, 1972) can be used to efficiently detect shapes in images if the parameter space has two dimensions or less (*e.g.* lines or circles with a fixed radius). If the parameter space has more than two dimensions (*e.g.* circles with unknown radius), the Hough transform is usually complemented with other algorithms in order to keep the computational cost manageable (*e.g.* using the gradient direction to reduce the number of votes (Borovička, 2003)).

The lazy operations introduced in Section 3.4 facilitate concise and flexible implementations of the Hough transform. Listing 6.2 shows the implementation of a Hough transform to locate lines. The application of the Hough transform is shown in Figure 6.6.

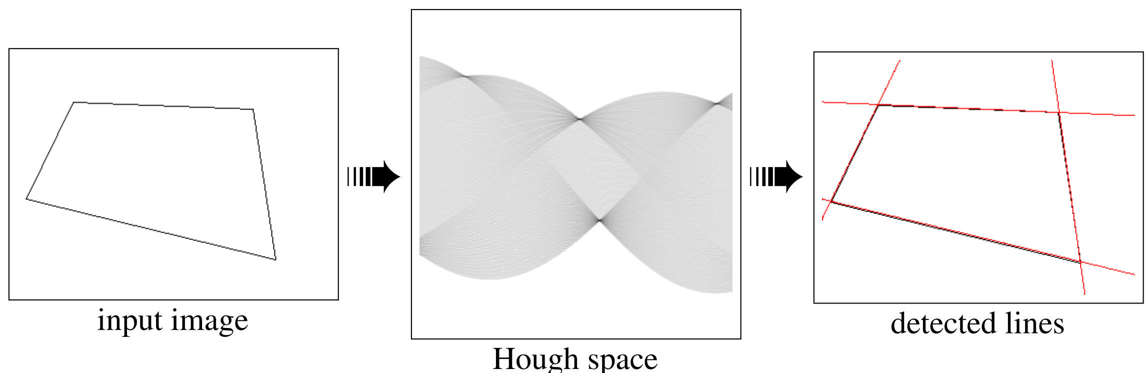


Figure 6.6: Line detection with the Hough transform

For every point in the input image, the Hough transform accumulates votes for the parameters of all possible lines passing through that point (Duda and Hart, 1972). The peaks in the Hough transform correspond to detected lines in the input image (see detected lines in Figure 6.6). The peaks can be located using thresholding followed by non-maxima sup-

pression. Note that the Hough transform in Listing 6.2 is implemented using a histogram operation (line 14). The votes for different combinations of distance ("d") and angle ("a") are generated using lazy tensor expressions (lines 13 and 12). A formal representation of the Hough transform is given in Equation 6.10.

$$h(a, d) = \sum_{i,j} \delta(a - \theta_j)\, \delta(d - \lfloor x_{i,1} \ \cos(\theta_j) + x_{i,2} \ \sin(\theta_j) \rfloor) \tag{6.10}$$

The implementation shown in Listing 6.2 is not optimal though. It uses a masking operation to create the intermediate arrays "x" and "y" with the coordinates of all black points of the input image.

6.2.4 Microscopy Software

Using the GUI integration shown in Section 4.8 one can develop sophisticated Qt4 GUIs involving video I/O. Figure 6.7 shows a dialog for configuring different object recognition and tracking algorithms. The figure shows the software being tested on an artificial test

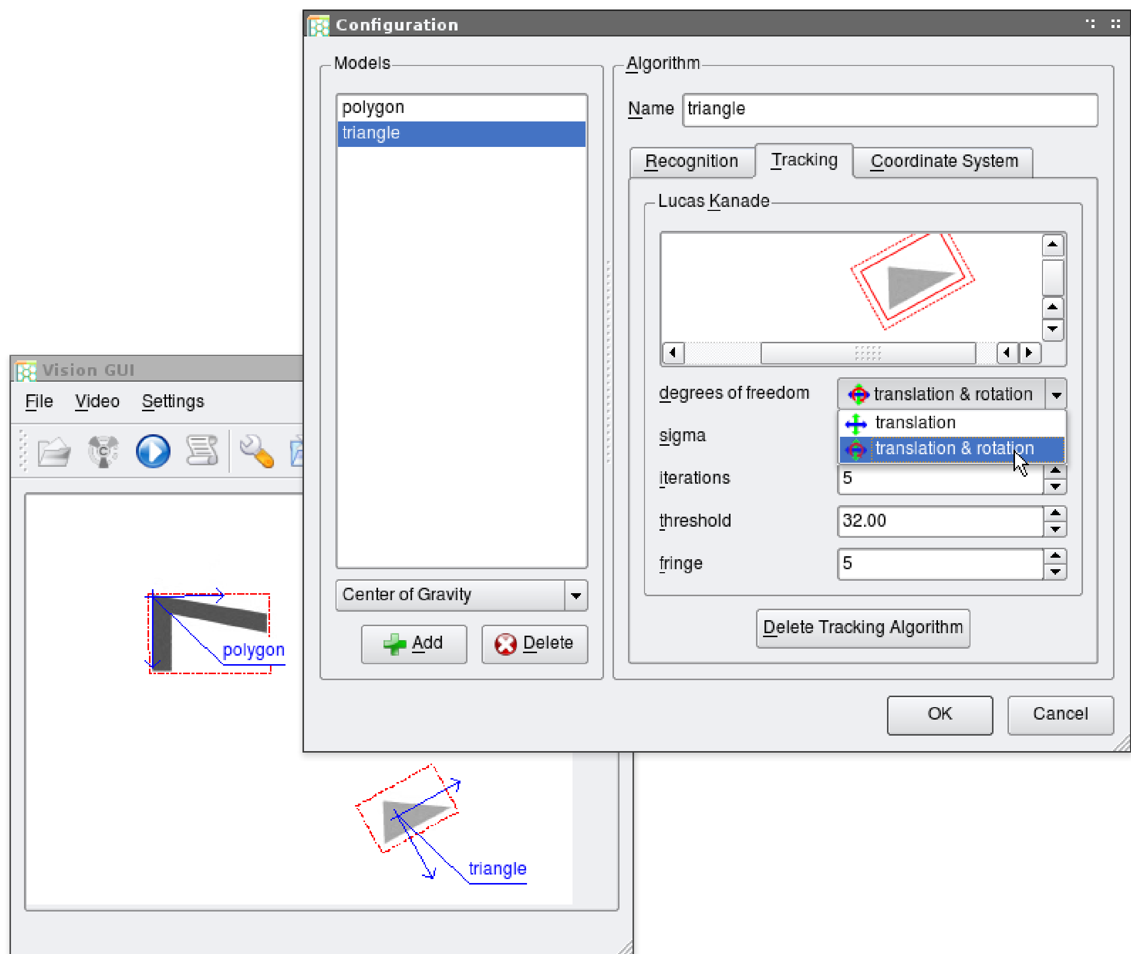
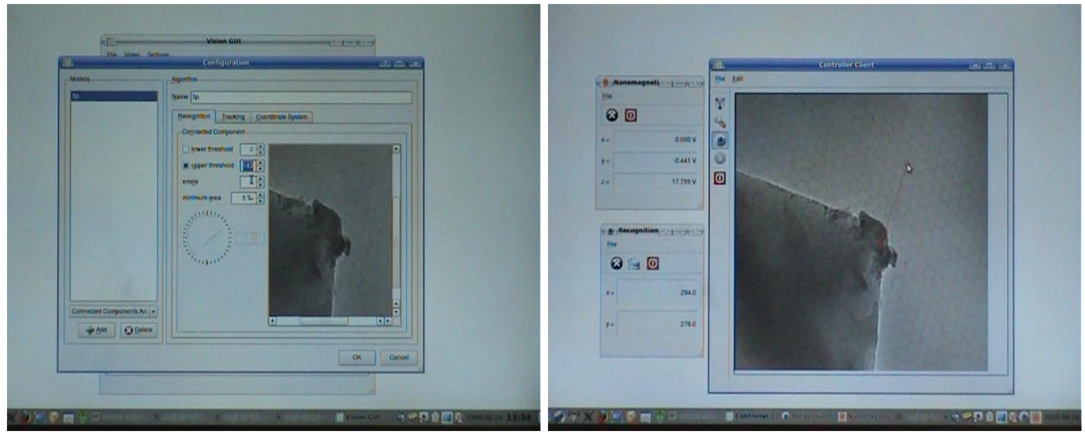


Figure 6.7: Configuration GUI

video.



Configuration   Closed-loop control

Figure 6.8: Closed-loop control of a nano manipulator in a TEM

The software was used to demonstrate closed-loop control of a nano manipulator in a transmission electron microscopy (TEM) as shown in Figure 6.8 (Lockwood et al., 2010). The software was also used to measure the hysteresis of Piezo drives.

6.2.5 Depth from Focus

Depth from focus is a computational method to estimate a 3D profile using a focus stack of images (Wedekind, 2002). A local sharpness measure is defined to estimate which image is nearer to the focal plane. A possible sharpness measure is the (square of the) Sobel gradient norm shown in Equation 6.11

$$s_z(\vec{x}) = (g_z \otimes \mathcal{S}_1)^2(\vec{x}) + (g_z \otimes \mathcal{S}_2)^2(\vec{x}) \tag{6.11}$$

The depth map or 3D profile of the object is obtained by determining the depth at which the local sharpness reaches its maximum.

$$dm(\vec{x}) = \underset{z}{\operatorname{argmax}}\, s_z(\vec{x}) \tag{6.12}$$

$$dv(\vec{x}) = g_{dm(\vec{x})}(\vec{x}) \tag{6.13}$$

Listing 6.3 shows an implementation of Depth from Focus. The Sobel gradient magnitude of the focus stack is used as a sharpness measure (lines 29–30). An image with extended depth of field is created (line 26 and 33). Furthermore a height field is generated (line 25 and 32). The implementation makes use of the “trollop” library to provide a command line interface (lines 1–20).

Figure 6.9 shows every 10th image of a focus stack. The series of images was taken using an optical microscope and it shows small glass fibres[3]. Figure 6.10 shows the result

[3] glass fibres courtesy of BMRC, Sheffield Hallam University

Listing 6.3: Implementation of Depth from Focus

```
opts = Trollop::options do
  banner <<EOS
Generate height field and deep view from focus stack.
Usage:
        ./depthfromfocus.rb [options] <file names>+

where [options] are:
EOS
  opt :sigma, 'Sigma for Gaussian blur (1/pixelsize)', :default => 2.5
  opt :field, 'Output PGM file name for height field', :type => String
  opt :view, 'Output PPM file name for deep view', :type => String
end
sigma = opts[ :sigma ]
Trollop::die :sigma, 'must be greater than zero' unless sigma > 0
field_file = opts[ :field ]
Trollop::die :field, 'is required' unless field_file
view_file = opts[ :view ]
Trollop::die :view, 'is required' unless view_file
stack_file = ARGV
Trollop::die 'Cannot handle more than 255 files' if stack_file.size > 255
field, view, max_sharpness = nil, nil, nil
stack_file.each_with_index do |f_name,i|
  img = MultiArray.load_ubytergb f_name
  unless field
    field = MultiArray.ubyte( *img.shape ).fill!
    view = MultiArray.ubytergb( *img.shape ).fill!
    max_sharpness = MultiArray.dfloat( *img.shape ).fill!
  end
  sharpness = ( img.sobel( 0 ) ** 2 + img.sobel( 1 ) ** 2 ).
    to_dfloat.gauss_blur sigma
  mask = sharpness > max_sharpness
  field = mask.conditional i, field
  view = mask.conditional img, view
  max_sharpness = mask.conditional sharpness, max_sharpness
end
field.save_ubyte field_file
view.save_ubytergb view_file
```

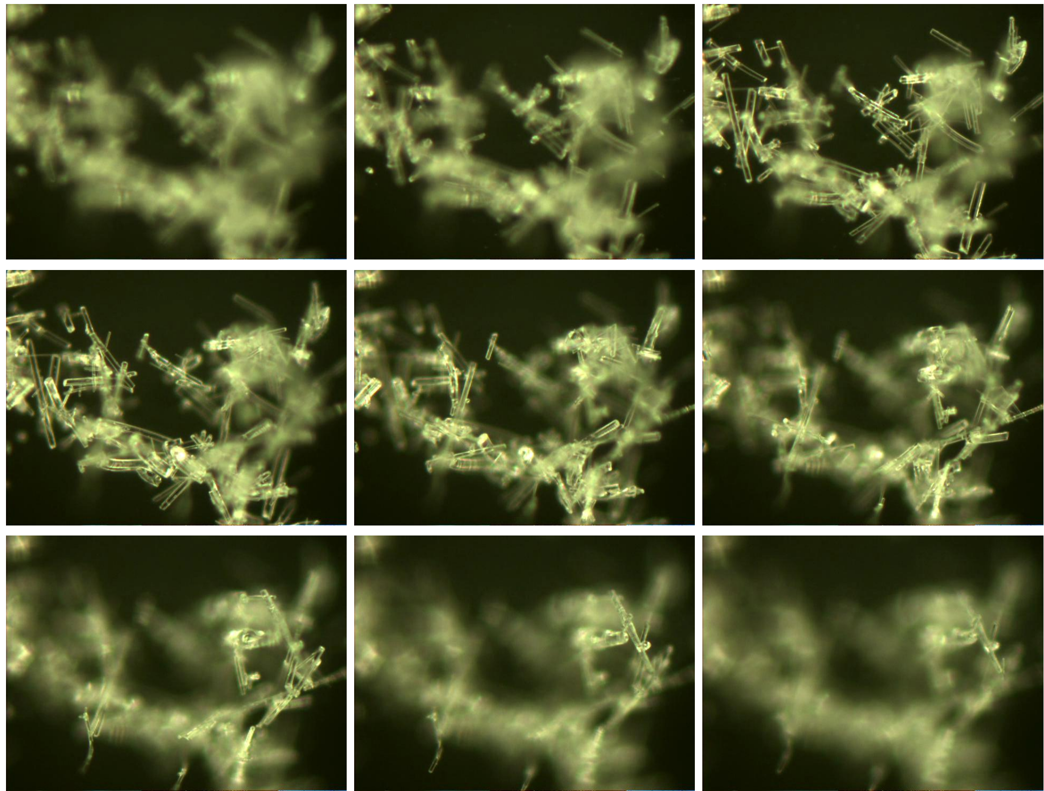
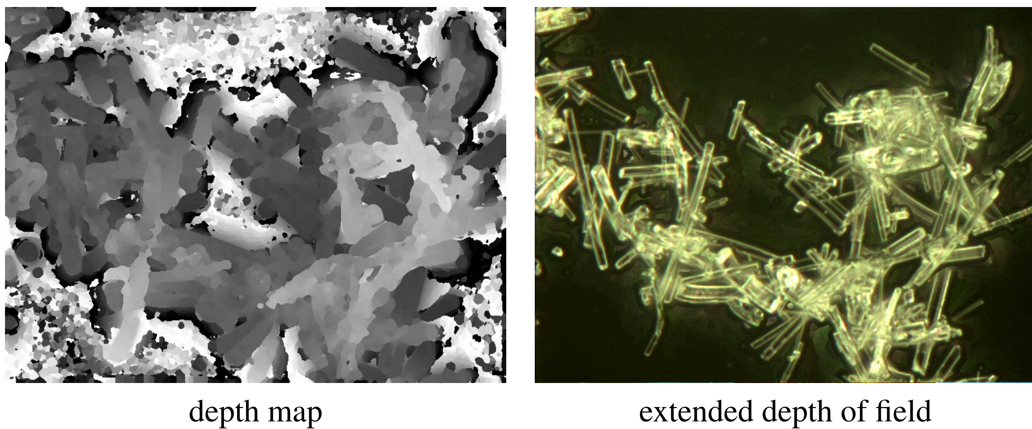


Figure 6.9: Part of focus stack showing glass fibres

obtained with the Depth from Focus method.



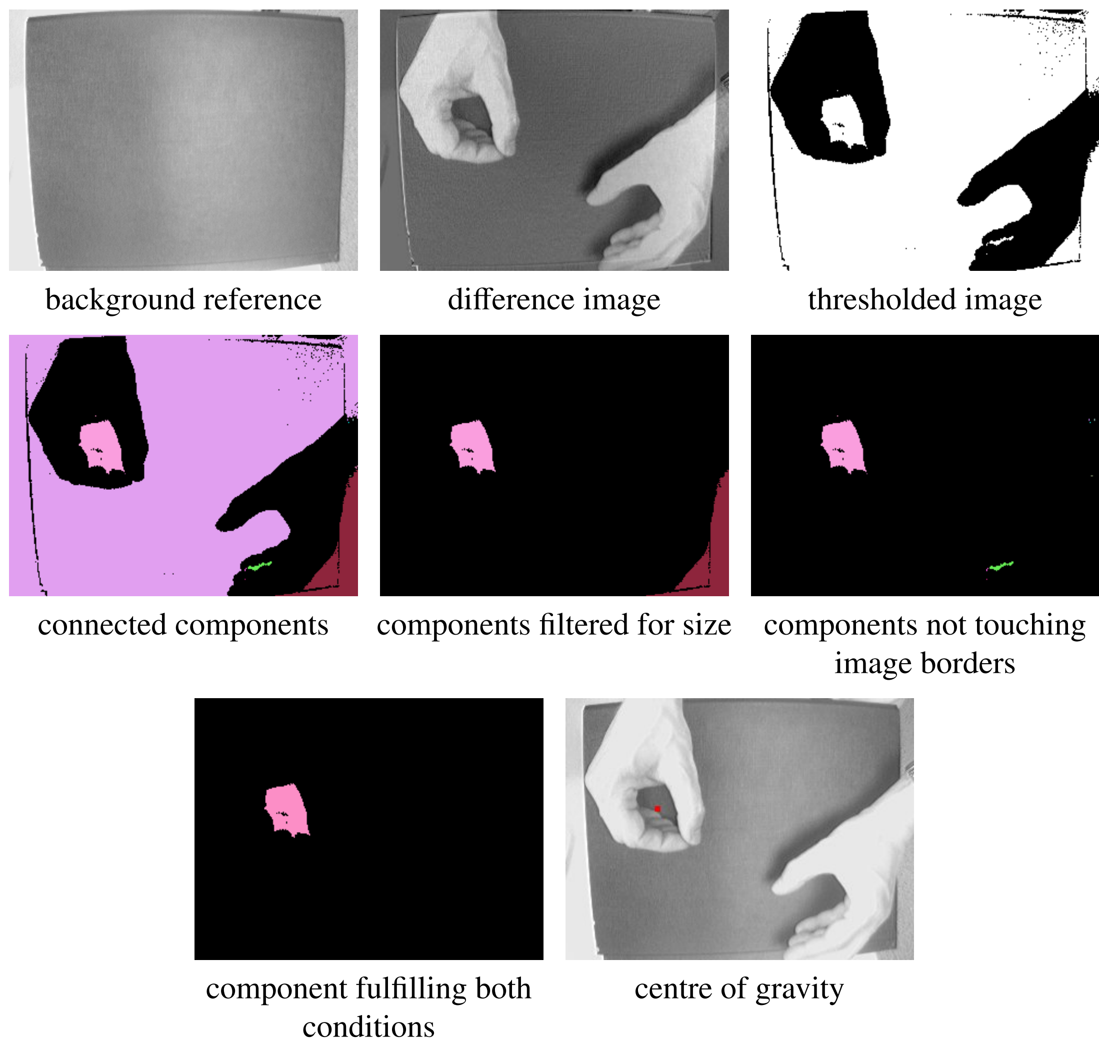
depth map | extended depth of field

Figure 6.10: Results of Depth from Focus

6.2.6 Gesture-based Mouse Control

Wilson (2006) shows how one can use basic image processing operations to implement a computer vision system to control a mouse cursor with gestures. The algorithm can detect pinching gestures where touching of the thumb and forefinger creates a disjoint region where the background is visible. The concept is illustrated in Figure 6.11.

1. acquire image
2. subtract background reference
3. apply a threshold
4. label connected components
5. suppress components which are too small or too large
6. suppress components connected to image borders
7. determine centre of gravity if exactly one component remains



background reference; difference image; thresholded image; connected components; components filtered for size; components not touching image borders; component fulfilling both conditions; centre of gravity

Figure 6.11: Human computer interface for controlling a mouse cursor

The Ruby source code of this algorithm is shown in Listing 6.4. The size of each component is computed by taking a histogram of the label image (line 13). Components touching the image borders are found out by taking a weighted histogram (line 15) where the image border pixels have non-zero weights (lines 6–7). "`mask_area`" and

Listing 6.4: Human computer interface for controlling the mouse cursor

```
THRESHOLD = 15
RANGE = 0.001 .. 0.25
input = V4L2Input.new
background = input.read_sint
shape = background.shape
border = MultiArray.int(*shape).fill! 1
border[1 ... shape[0] - 1, 1 ... shape[1] - 1] = 0
X11Display.show do
  img = input.read_ubyte
  binary = img - background <= THRESHOLD
  components = binary.components
  n = components.max + 1
  area = components.histogram n
  mask_area = area.between? RANGE.min * img.size, RANGE.max * img.size
  mask_border = components.histogram(n, :weight => border).eq 0
  mask = mask_area.and(mask_border).to_ubyte
  target = components.lut(mask.integral * mask)
  if target.max == 1
    sum = target.sum.to_f
    x = lazy(*shape) { |i,j| i }.mask(target.to_bool).sum / sum
    y = lazy(*shape) { |i,j| j }.mask(target.to_bool).sum / sum
    puts "#{x} #{y}"
  end
  img
end
```

"`mask_border`" are 1D arrays with boolean values indicating for each component whether it is to be suppressed or not. "`mask`" is a 1D array of integers where "`1`" indicates a component which is fulfilling all conditions (line 16). The components are re-labelled by using "`mask.integral * mask`" as a lookup table (line 17). For example Listing 6.5 shows a case where "`mask`" has 3 non-zero values. By multiplying the integral array with the array, one can obtain a "`1D`" lookup table which assigns a running index to accepted components. All rejected components get mapped to zero.

The coordinates obtained with Listing 6.4 can be used to create mouse-motion and

Listing 6.5: Lookup table for re-labelling

```
mask
# Sequence(UBYTE):
# [ 0, 0, 1, 0, 0, 0, 1, 1, 0 ]
mask.integral
# Sequence(UBYTE):
# [ 0, 0, 1, 1, 1, 1, 2, 3, 3 ]
lut = mask.integral * mask
# Sequence(UBYTE):
# [ 0, 0, 1, 0, 0, 0, 2, 3, 0 ]
MultiArray[[0, 0, 2, 0, 0], [3, 0, 0, 0, 6]].lut lut
# MultiArray(UBYTE,2):
# [ [ 0, 0, 1, 0, 0 ],
#   [ 0, 0, 0, 0, 2 ] ]
```

mouse-button events (Wilson, 2006). It is also possible to extend the concept so that two hands can be used to create mouse-scroll events. Listing 6.4 demonstrates that using the Ruby-extension presented in this thesis it requires only a short amount of time to put an idea into practise.

6.2.7 Slide Presenter

Figure 6.12 illustrates the idea for a software for changing slides. After a certain time

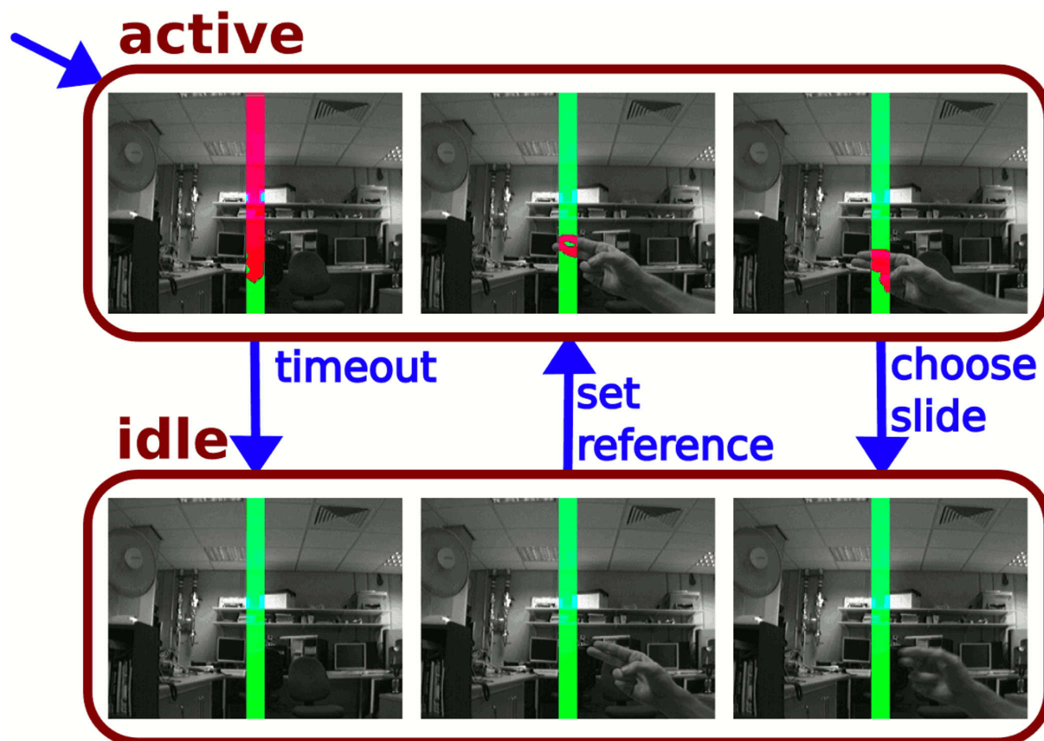


Figure 6.12: Software for vision-based changing of slides

a reference image is acquired. If the user's hand reaches into the centre of the image, the blurred difference image is thresholded and the centre of gravity of the resulting binary image is used as a 1D sensor for controlling the presentation of slides. Listing 6.6 implements the basic concept.

The 1D sensor input is used as follows. A quick downward or upward motion is used to display the next or previous slide (see Figure 6.13). A slow gesture is used to display a menu and select a particular slide (see Figure 6.14). The system was used to give a presentation at a conference (Wedekind, 2009)[4]. Note that the sensor value might change significantly when the hand is removed. To make recognition more robust, the system picks the slide which was selected for the longest period of time in such cases. This can be achieved using weighted histograms.

[4]See http://www.wedesoft.demon.co.uk/rubyconf09video.html for a video of the talk

Listing 6.6: Vision-based changing of slides

```
input = DC1394Input.new
w, h, o = 20, input.height, input.width / 2 - 10
box = [ o ... o + w, 0 ... h ]
bg = input.read_ubyte[*box]
t = Time.new.to_f
X11Display.show do
  img = input.read_ubytergb
  if Time.new.to_f > t + 10
    bg = img[*box].to_ubyte
    t = Time.new.to_f
  end
  slice = (img[*box].to_sint - bg).gauss_blur(2).abs >= 12
  n = slice.to_ubyte.sum
  if n > 20
    y = lazy(w, h) { |i,j| j }.mask(slice).sum / n
    puts y
  else
    t = Time.new.to_f
    puts ’----’
  end
  img[*box].r = slice.to_ubyte * 255
  img[*box].g = slice.not.to_ubyte * 255
  img
end
```

Figure 6.13: Quick gesture for displaying the next slide

Figure 6.14: Slow gesture for choosing a slide from a menu

As one can see in Listing 6.6, the input class "`DC1394Input`" was used instead of "`V4L2Input`". This input class is for accessing DC1394-compatible Firewire cameras (see Figure 6.15). The camera has the advantage that it supports a fixed exposure setting

Figure 6.15: Unibrain Fire-I (a DC1394-compatible Firewire camera)

which is important when using difference images.

6.2.8 Camera Calibration

Camera calibration usually is done by taking images of a chequer board of known size. A corner detector then detects the coordinates of the projected corners. The corners are labelled so that for each corner of the chequer board the corresponding corner in the image is known. Using a pinhole camera model (see Appendix A.3), camera calibration then becomes an optimisation problem (Ballard and Brown, 1982; Zhang, 2000; Faucher, 2006). Camera calibration is a requirement for 3D reconstruction with lasers (Lim, 2009)

and real-time visual SLAM (Davison et al., 2007; Pupilli, 2006) for example. In general some means of initial calibration or self-calibration (Mendonça and Cipolla, 1999) is required in order to solve non-trivial 3D machine vision problems.

6.2.8.1 Corners of Calibration Grid

Many algorithms for detecting the corners of a calibration grid are not fully automated. For example to most popular calibration toolbox requires the user to manually select the four outer corners of the calibration grid (Bouguet, 2010). Other calibration software such as the implementation of the OpenCV library uses a sophisticated custom algorithm which makes use of basic operations such as thresholding, connected components, convex hull, and nearest neighbours (see OpenCV source code[5]).

Here an elegant algorithm is presented which is based on standard image processing operations. A planar homography is used to establish the order of the corners. The algorithm is illustrated in Figure 6.16 and Figure 6.17.

Listing 6.7 shows the implementation of the algorithm including visualisation. The implementation makes use of other algorithms presented earlier in this thesis. The implementation makes use of the Ruby matrix library and the LAPACK bindings for Ruby[6] (the code for integrating Ruby matrices and LAPACK matrices is not shown). The algorithm consists of the following steps:

1. Apply Otsu Thresholding to input image (line 8).
2. Take difference of dilated and eroded image to get edge regions (line 9).
3. Label connected components (line 10).
4. Compute corners of input image (*e.g.* Harris-Stephens corners as shown in Section 5.2.2.3) and use non-maxima suppression (lines 7 and 8).
5. Count corners in each component (line 11, implementation of "`have`" is not shown)
6. Look for a component which contains exactly 40 corners (line 11, implementation of "`have`" is not shown).
7. Get largest component of inverse of grid (i.e. the surroundings) (line 15, implementation of "`largest`" is not shown).
8. Grow that component and find all corners on it (i.e. corners on the boundary of the grid) (lines 15–16).
9. Find centre of gravity of all corners and compute vectors from centre to each boundary corner (lines 14 and 17).

[5] https://sourceforge.net/projects/opencvlibrary/
[6] http://rubyforge.org/projects/linalg/

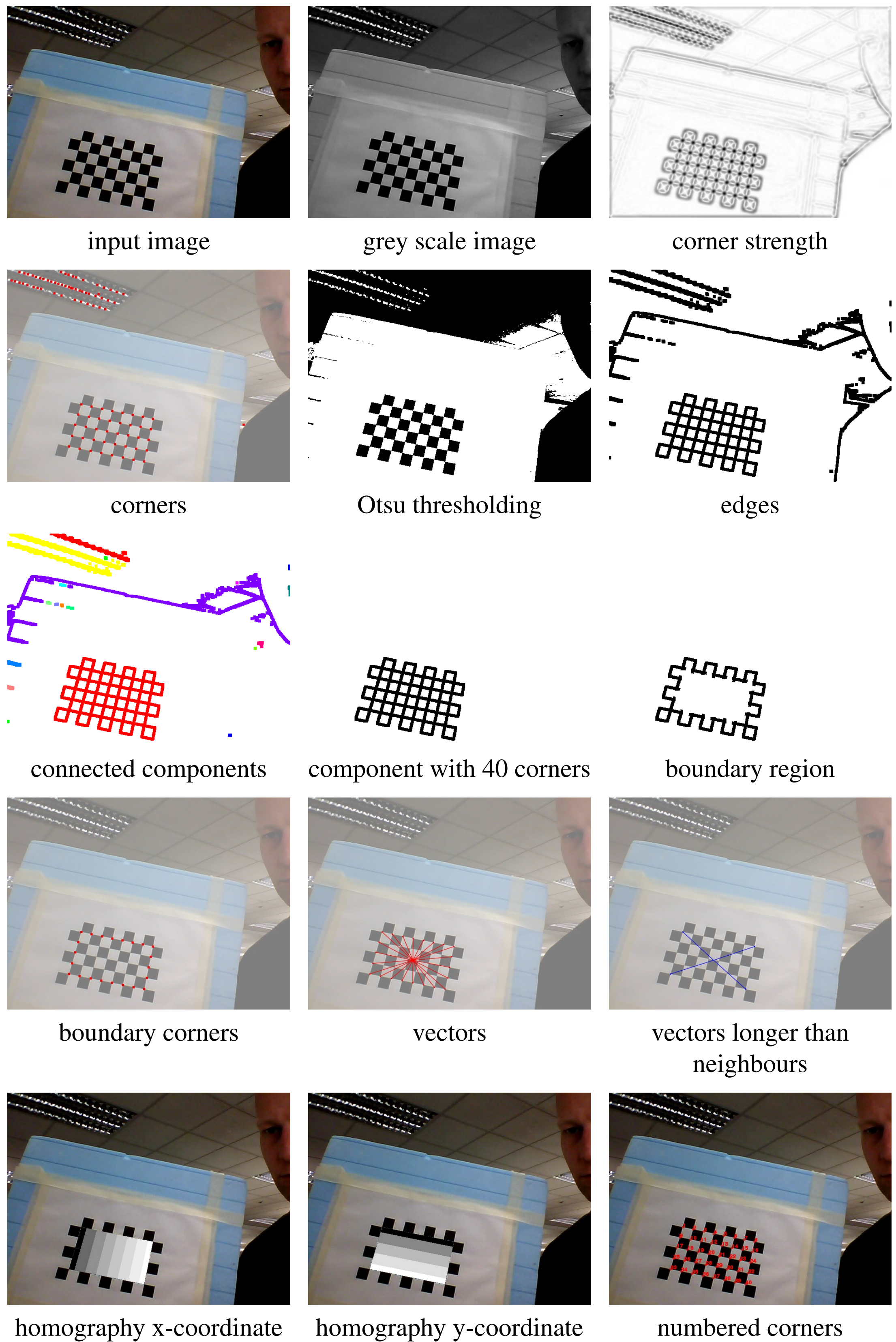


Figure 6.16: Custom algorithm for labelling the corners of a calibration grid

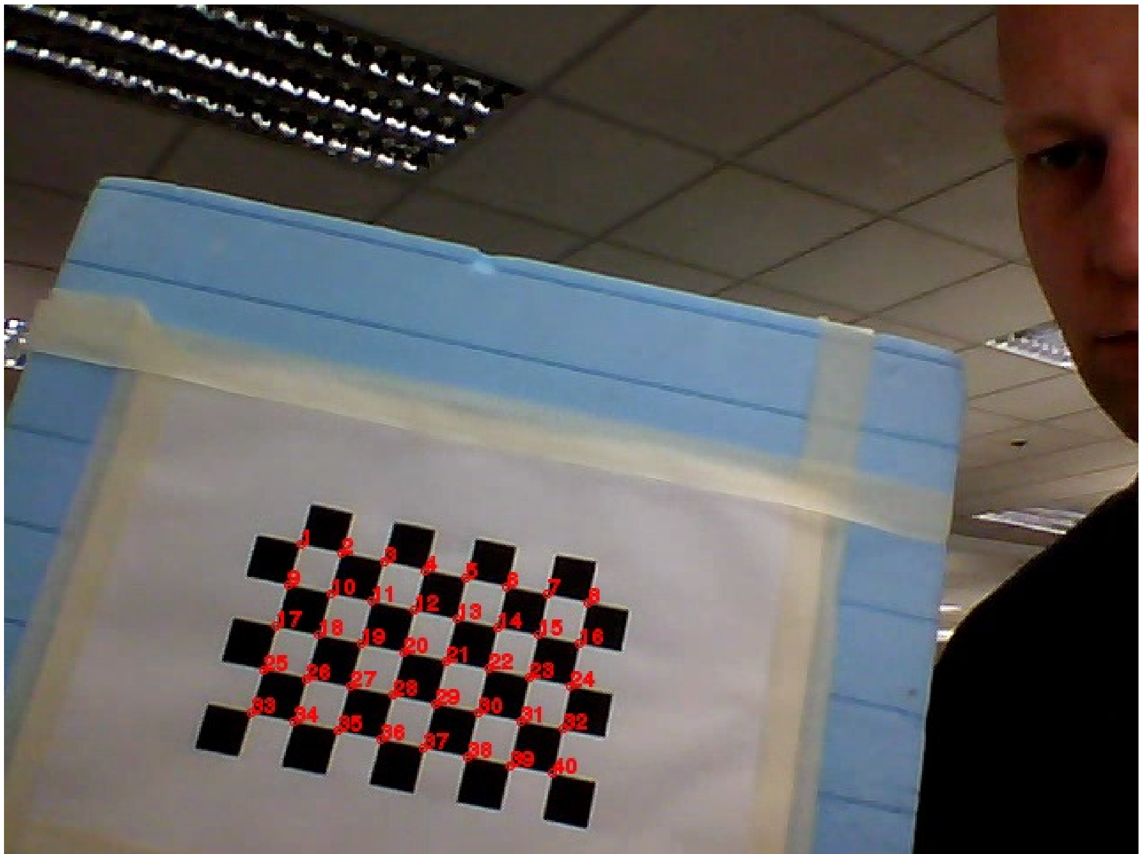


Figure 6.17: Result of labelling the corners

Listing 6.7: Custom algorithm for labelling the corners of a calibration grid

```
CORNERS = 0.3; W, H = 8, 5; N = W * H; GRID, BOUNDARY = 7, 19
input = V4L2Input.new
coords = finalise(input.width, input.height) { |i,j| i + Complex::I * j }
pattern = Sequence[*(([1] + [0] * (W - 2) + [1] + [0] * (H - 2)) * 2)]
X11Display.show do
  img          = input.read_ubytergb;      grey = img.to_ubyte
  corner_image = grey.corners;             abs  = corner_image.abs
  corners      = abs.nms CORNERS * abs.max; otsu = grey.otsu
  edges = otsu.dilate(GRID).and otsu.not.dilate(GRID)
  components = edges.components
  grid = components.have N, corners
  result = img
  if grid
    centre = coords.mask(grid.and(corners)).sum / N.to_f
    boundary = grid.not.components.largest.dilate BOUNDARY
    outer = grid.and(boundary).and corners
    vectors = (coords.mask(outer) - centre).to_a.sort_by { |c| c.arg }
    if vectors.size == pattern.size
      mask = Sequence[*(vectors * 2)].shift(vectors.size / 2).abs.nms(0.0)
      mask[0] = mask[mask.size-1] = false
      conv = lazy(mask.size) { |i| i }.
        mask(mask.to_ubyte.convolve(pattern.flip(0)).eq(4))
      if conv.size > 0
        offset = conv[0] - (pattern.size - 1) / 2
        r = Sequence[*vectors].shift(-offset)[0 ... vectors.size].
              mask(pattern) + centre
        m = Sequence[Complex(-0.5 * W,-0.5 * H), Complex(0.5 * W, -0.5 * H),
                     Complex(0.5 * W, 0.5 * H), Complex(-0.5 * W, 0.5 * H)]
        constraints = []
        for i in 0 ... 4 do
          constraints.push [m[i].real, m[i].imag, 1.0, 0.0, 0.0, 0.0,
            -r[i].real * m[i].real, -r[i].real * m[i].imag, -r[i].real]
          constraints.push [0.0, 0.0, 0.0, m[i].real, m[i].imag, 1.0,
            -r[i].imag * m[i].real, -r[i].imag * m[i].imag, -r[i].imag]
        end
        h = Matrix[*constraints].svd[2].row(8).reshape(3, 3).inv
        v = h.inv * Vector[coords.real, coords.imag, 1.0]
        points = coords.mask grid.and(corners) +
          Complex(input.width/2, input.height/2)
        sorted = (0 ... N).
          zip((v[0] / v[2]).warp(points.real, points.imag).to_a,
              (v[1] / v[2]).warp(points.real, points.imag).to_a).
          sort_by { |a,b,c| [(c - H2).round, (b - W2).round] }.
          collect { |a,b,c| a }
        result = (v[0] / v[2]).between?(-0.5 * W, 0.5 * W).and((v[1] / v[2]).
          between?(-0.5 * H, 0.5 * H)).conditional img * RGB(0, 1, 0), img
        gc = Magick::Draw.new
        gc.fill_opacity(0).stroke('red').stroke_width 1
        sorted.each_with_index do |j,i|
          gc.circle points[j].real, points[j].imag,
                    points[j].real + 2, points[j].imag
          gc.text points[j].real, points[j].imag, "#{i+1}"
        end
        result = result.to_magick; gc.draw result; result = result.to_ubytergb
      end
    end
  end
  result
end
```

10. Sort boundary corners by angle of those vectors (line 17).

11. Use non-maxima suppression on list of length of vectors to get the 4 "corner corners" (convexity) (lines 19–26).

12. Use the locations of the 4 "corner corners" to compute a planar homography (see Appendix A.4) mapping the image coordinates of the 8 times 5 grid to the ranges 0..7 and 0..4 respectively (lines 27–36).

13. Use the homography to transform the 40 corners and round the coordinates (lines 37–42).

14. Order the points using the rounded coordinates (line 43).

Once the corners of the calibration grid shown in a camera image have been identified, the correspondences can be used to establish a more accurate 2D homography (see Appendix A.4).

6.2.8.2 Camera Intrinsic Matrix

Zhang (2000) describes a method for determining the parameters of a linear camera model for calibration which includes displacement of the chip (principal point), skewness, and non-square pixel size. Furthermore a non-linear method for determining radial distortion is discussed. The method requires at least four pictures of the calibration grid being in different positions.

However in many cases one can assume that there is no radial distortion, skewness, displacement of the chip and that the pixel are square-shaped. In that case it is only necessary to determine the ratio of focal length to pixel size $f/\Delta s$.

Let $\mathcal{H}$ be the planar homography (multiplied with an unknown factor) which was determined according to Appendix A.4. Let $\mathcal{A}$ be the intrinsic and $\mathcal{R}'$ the extrinsic camera matrix (see Equation 6.14 and Equation A.11).

$$\mathcal{H} = \lambda\,\mathcal{A}\,\mathcal{R}' \Leftrightarrow \lambda\,\mathcal{R}' = \mathcal{A}^{-1}\,\mathcal{H} \tag{6.14}$$

The rotational part of $\mathcal{R}'$ is an isometry as shown in Equation 6.15.

$$\mathcal{R}'^{\top}\,\mathcal{R}' \overset{!}{=} \begin{pmatrix} 1 & 0 & * \\ 0 & 1 & * \\ * & * & * \end{pmatrix} \tag{6.15}$$

Using Equation 6.14 and Equation 6.15 and by decomposing $\mathcal{H}$ as shown in Equa-

tion 6.16, one obtains Equation 6.17 (Zhang, 2000).

$$\begin{pmatrix}\vec{h}_1 & \vec{h}_2 & \vec{h}_3\end{pmatrix} := \mathcal{H} \text{ where } \vec{h}_i = \begin{pmatrix}h_{1,i}\\ h_{2,i}\\ h_{3,i}\end{pmatrix} \tag{6.16}$$

$$\vec{h}_1^\top \mathcal{A}^{-\top} \mathcal{A}^{-1} \vec{h}_2 \stackrel{!}{=} 0 \text{ and } \vec{h}_1^\top \mathcal{A}^{-\top} \mathcal{A}^{-1} \vec{h}_1 \stackrel{!}{=} \vec{h}_2^\top \mathcal{A}^{-\top} \mathcal{A}^{-1} \vec{h}_2 \tag{6.17}$$

If $f/\Delta s$ is the only intrinsic camera parameter, the camera intrinsic matrix is a diagonal matrix as shown in Equation 6.18.

$$\mathcal{A} = \begin{pmatrix} f/\Delta s & 0 & 0\\ 0 & f/\Delta s & 0\\ 0 & 0 & 1\end{pmatrix} \tag{6.18}$$

In this case Equation 6.17 can be reformulated as shown in Equation 6.19.

$$\begin{aligned}(h_{1,1}\,h_{1,2} + h_{2,1}\,h_{2,2})(\Delta s/f)^2 + h_{3,1}\,h_{3,2} \stackrel{!}{=} 0 \text{ and}\\ (h_{1,1}^2 + h_{2,1}^2 - h_{1,2}^2 - h_{2,2}^2)(\Delta s/f)^2 + h_{3,1}^2 - h_{3,2}^2 \stackrel{!}{=} 0\end{aligned} \tag{6.19}$$

This is a overdetermined equation system which does not have a solution in general. Therefore the least squares algorithm (see Appendix A.2) is used in order to find a value for $(\Delta s/f)^2$ which minimises the left-hand terms shown in Equation 6.19. Furthermore the least squares estimation is performed for a set of frames in order to get a more robust estimate for $f/\Delta s$.

The horizontal camera resolution w (*e.g.* 640 pixel) together with the ratio $f/\Delta s$ can be used to determine the camera's horizontal angle of view α.

$$\alpha = \arctan \frac{w\,\Delta s}{2\,f} \tag{6.20}$$

The complete source code of the camera calibration is shown in Section A.7.4.

6.2.8.3 3D Pose of Calibration Grid

If the camera intrinsic matrix $\mathcal{A}$ is known, it is possible to derive the 3D pose of the calibration grid from the planar homography $\mathcal{H}$. Using Equation 6.14 one can determine the camera extrinsic matrix $\mathcal{R}'$ up to a scale factor. However since $\mathcal{R}$ is an isometry, $|\vec{r}_1| \stackrel{!}{=} 1$ and $|\vec{r}_2| \stackrel{!}{=} 1$ must hold. That is, one can estimate the scaling factor λ according to Equation 6.21.

$$\lambda = \frac{t_3}{2\,|t_3|}\left(|\vec{r}_1| + |\vec{r}_2|\right) \tag{6.21}$$

When the calibration grid is detected, it must be in front of the camera. The factor $t/|t_3|$ ensures that $\lambda\, t_3 > 0$

Equation A.11 shows that $\mathcal{R}'$ is composed out of two rotational vectors ($\vec{r}_1$ and $\vec{r}_2$) and a translational vector $\vec{t}$. The composition is shown in Equation 6.22.

$$\begin{pmatrix}\vec{r}_1 & \vec{r}_2 & \vec{t}\end{pmatrix} := \mathcal{R}' \text{ where } \vec{r}_1 = \begin{pmatrix} r_{1,1} \\ r_{2,1} \\ r_{3,1} \end{pmatrix}, \vec{r}_2 = \begin{pmatrix} r_{1,2} \\ r_{2,2} \\ r_{3,2} \end{pmatrix}, \text{ and } \vec{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \tag{6.22}$$

The third vector $\vec{r}_3$ of the 3D rotation matrix must be orthogonal to $\vec{r}_1$ and $\vec{r}_2$ and can be determined using the vector cross-product as shown in Equation 6.23.

$$\vec{r}_3 = \vec{r}_1 \times \vec{r}_2 \tag{6.23}$$

Due to noise the resulting matrix $Q = \begin{pmatrix}\vec{r}_1 & \vec{r}_2 & \vec{r}_3\end{pmatrix}$ usually will not be an isometry. However Zhang (2000) shows that the nearest isometry (nearest in terms of Frobenius norm) can be determined using the SVD of Q (see Equation 6.24).

$$\mathcal{R} = \mathcal{U}\,\mathcal{V}^\top \text{ where } U\,\Sigma\,V^\top = Q \text{ is SVD of } Q = \begin{pmatrix}\vec{r}_1 & \vec{r}_2 & \vec{r}_3\end{pmatrix} \tag{6.24}$$

An example sequence is shown in Figure 6.18 (the visualisations were created using the POV-Ray ray tracer (POVRay, 2005)). Note that sometimes the colours of the chequer board do not match because the detection of the calibration grid is based on corners and there is a 180° rotational ambiguity. Otherwise the alignment of the model and the camera image is good. This means that the perfect pinhole camera model is sufficiently accurate for the camera in use (Feiya Technology built-in camera sensor).

6.2.9 Augmented Reality

Kato and Billinghurst (1999) first published the ARToolKit[7] augmented reality tool kit. Inspection of the source code reveals that the markers are located by an algorithm which locates rectangular markers by tracking contours and recording contours which have 4 abrupt orientation changes (*i.e.* 4 corners).

Figure 6.19 illustrates a different custom algorithm for recognising a rectangular marker. The algorithm can be implemented using the basic image processing operations presented in this thesis. The algorithm works as follows:

1. acquire image
2. apply threshold
3. do connected component labelling

[7] also see http://www.hitl.washington.edu/artoolkit/

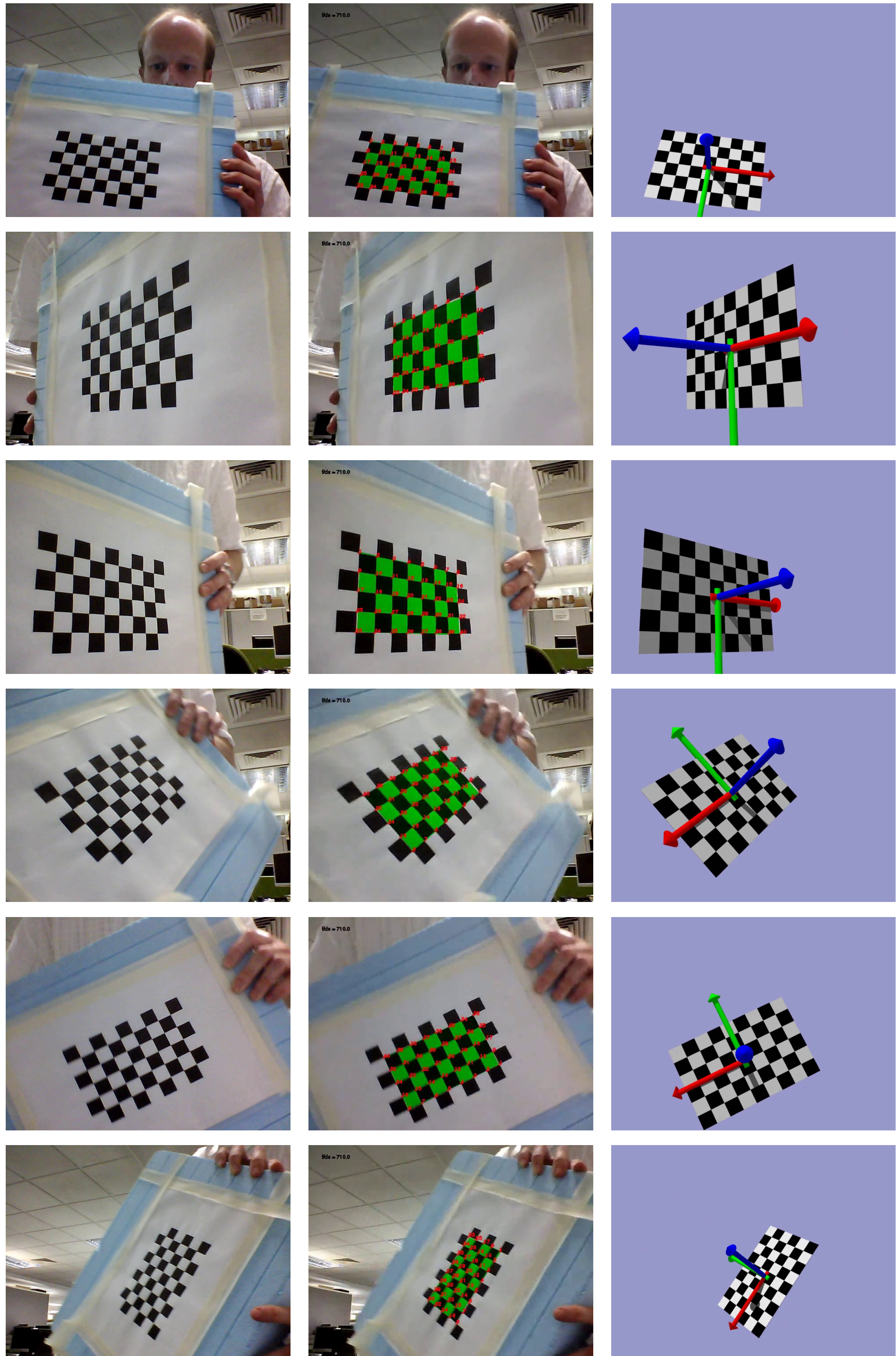


Figure 6.18: Estimating the pose of the calibration grid

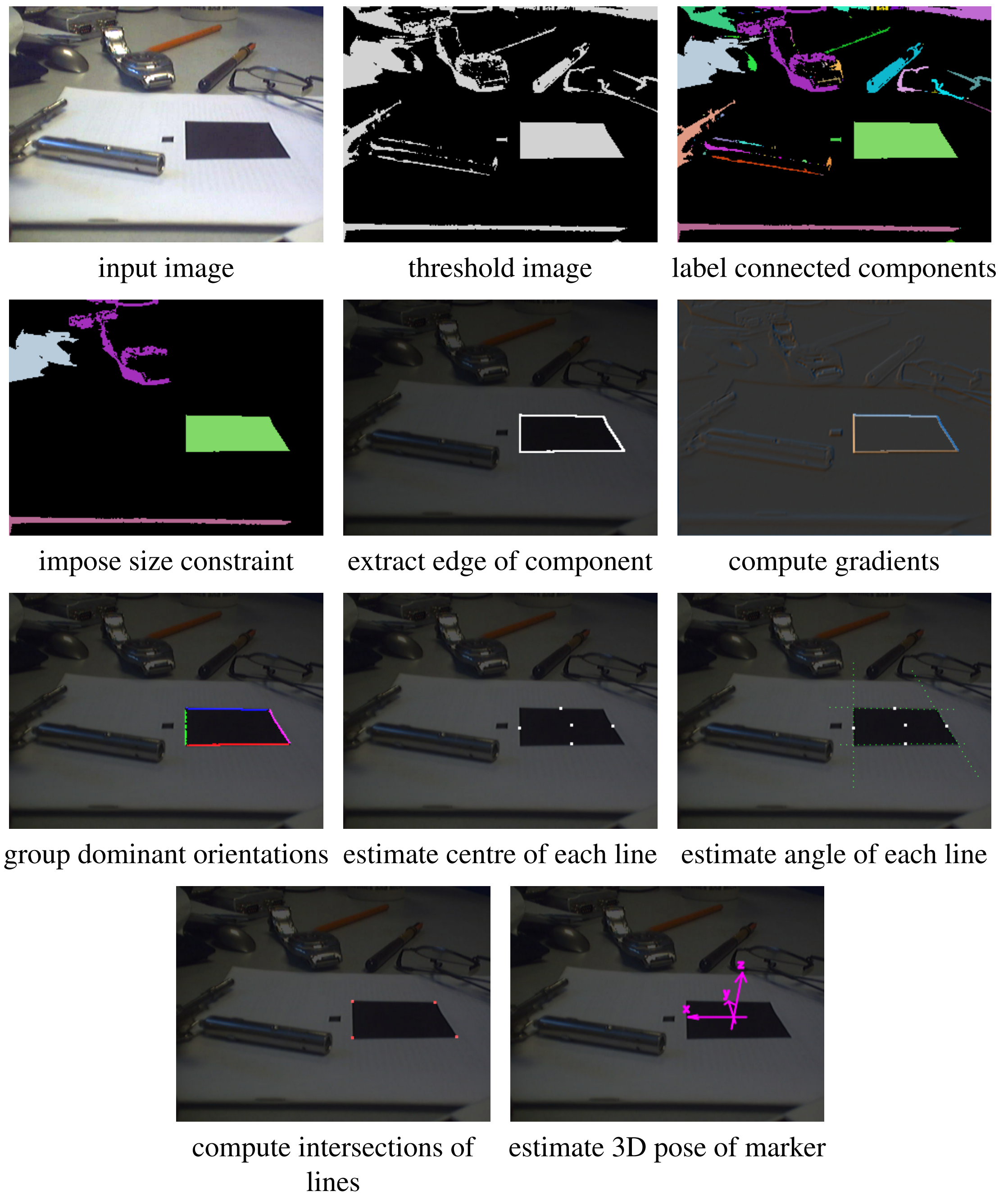


Figure 6.19: Custom algorithm for estimating the 3D pose of a marker

4. suppress components which are too small or too large
5. for each remaining component:
    (a) extract edge of component
    (b) compute gradients
    (c) group dominant orientations
    (d) if there are four dominant orientations:
        i. estimate centre of each line
        ii. estimate angle of each line
        iii. compute intersections of lines
        iv. estimate 3D pose of marker

Appendix A.7.5 shows that it only requires 44 lines of code to locate the four corners of the marker. The homography then can be estimated as shown in Section 6.2.8.

Figure 6.20 shows the augmented reality demonstration based on the approach presented in this section. Note that there are 4 possible solutions for the 3D pose of the rectangular marker since the marker is self-similar. A small dot is used to resolve this ambiguity. The correct 3D pose is chosen by determining the brightness of the image in each of the 4 possible locations and choosing the pose with the lowest one.

6.3 Performance

6.3.1 Comparison with NArray and C++

Figure 6.21 shows different operations and the time required for performing them 1000 times with Hornetseye (Ruby 1.9.2 and GCC 4.4.3), NArray (Ruby 1.9.2 and GCC 4.1.3), and a naive C++ implementation (G++ 4.4.3). The tests were performed on an Intel™ Celeron™ 2.20GHz processor. The arrays "m" and "n" are single-precision floating point arrays with 500×500 and 100×100 elements.

The results show that Hornetseye takes about four times as much processing time as the pure C++ implementation. The fact that NArray is almost as fast as the C++ implementation shows that the overhead incurred by the Ruby VM can be negligible.

Figure 6.22 shows the time required for running the operation "`m + 1`" for arrays of different size. One can see that there are steps in the processing time at 4 MByte and 8 MByte. This is probably because of the mark-and-sweep garbage collector running more often when larger return values need to be allocated (the steps disappear if the statement "`GC.start`" is used to force a run of the garbage collector after each array operation).

Another problem is that the processing time for the current implementation of lazy expressions does not scale linearly with the size of the expression (see Figure 6.23). The

Figure 6.20: Augmented reality demonstration

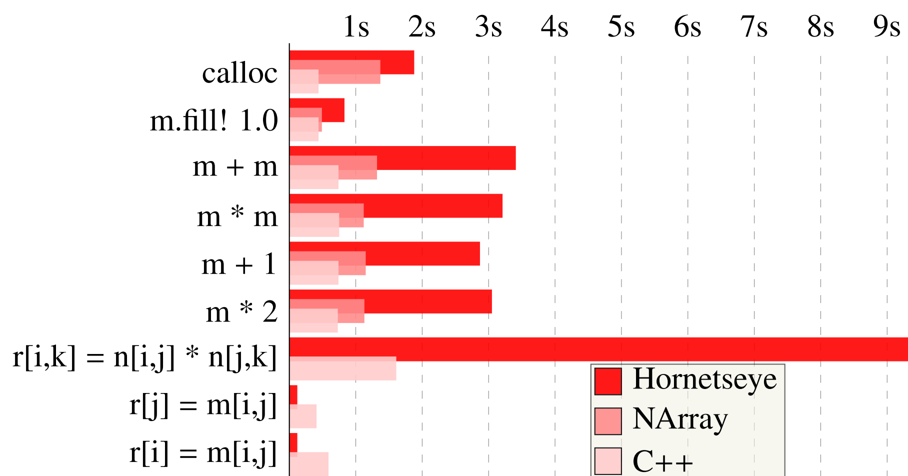


Figure 6.21: Performance comparison of different array operations

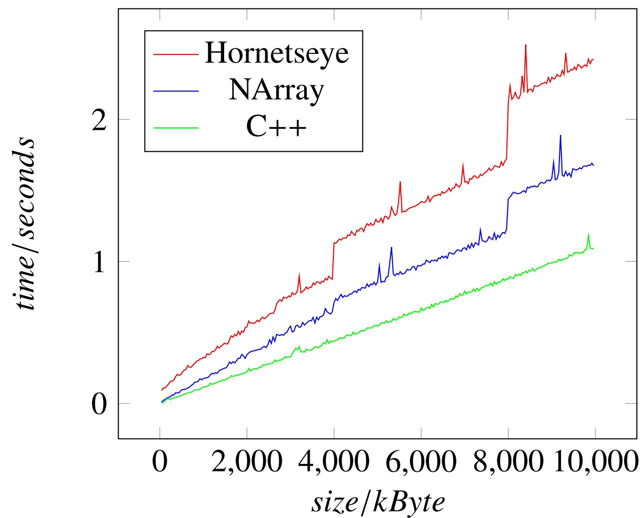


Figure 6.22: Processing time of running "`m + 1`" one-hundred times for different array sizes

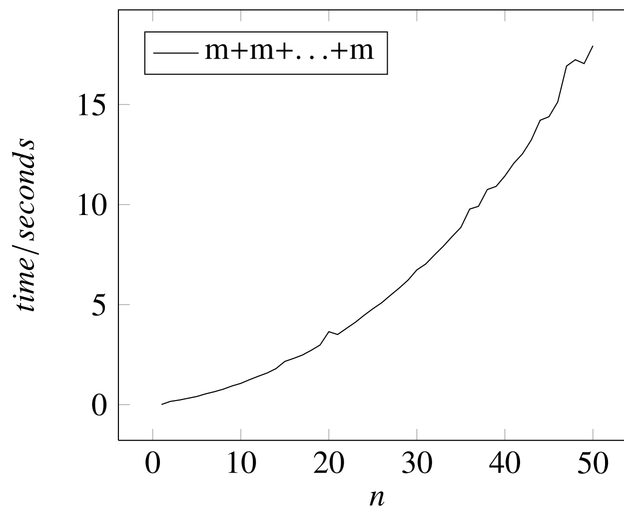


Figure 6.23: Processing time increasing with length of expression

reason is that the tree is traversed multiple times when an expression is composed. That is, the complexity is $O(n^2)$ where n is the length of the expression.

6.3.2 Breakdown of Processing Time

Table 6.1 shows time measurements of the processing time for performing element-wise negation for an array with one million elements a thousand times. The measurements where obtained by manually optimising the generated code and in other cases by removing parts of the program. The values obtained can be used to get a more detailed understanding of where the processing time is spent.

Figure 6.24 shows a breakdown of the processing time for the element-wise negation. The processing time is broken down into the following parts

Table 6.1: Processing times measured for tasks related to computing "`-s`" for an array

program	**processing time**	
current implementation	2.581641	s
manually optimised Ruby code	1.937845	s
manually optimised Ruby and C code	1.913475	s
Ruby memory allocation only	0.708125	s
C only	1.065510	s
C allocation only	0.000184	s

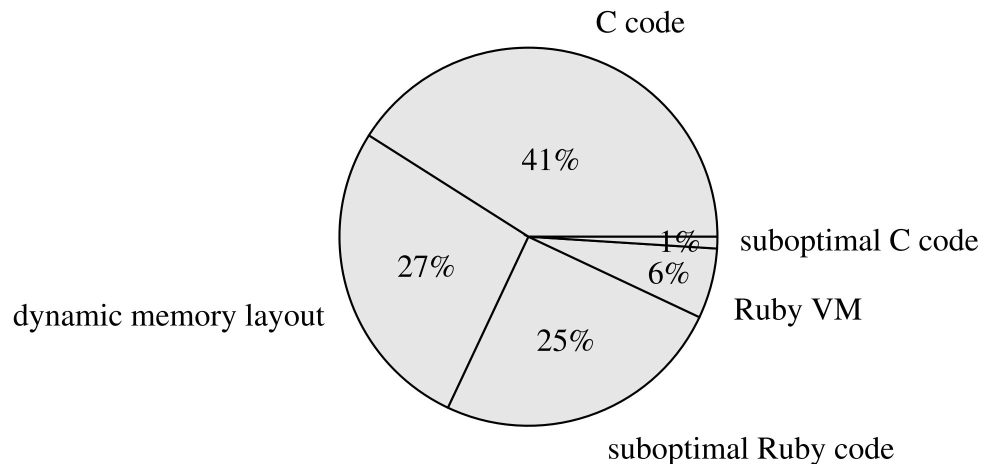


Figure 6.24: Breakdown of processing time for computing "`-s`" where "`s`" is an array with one million elements

- **C code**: time for doing the actual array operation in C
- **dynamic memory layout**: cost of using a dynamic memory layout instead of a static memory layout
- **suboptimal Ruby code**: optimisation potential in the calling Ruby program
- **Ruby VM**: estimated lower bound for overhead of Ruby program
- **suboptimal C code**: optimisation potential in the generated C code

One can see that most of the optimisation potential is in using a static memory layout (such as commonly done in Fortran and C implementations). For example Tanaka (2011) provides method calls for specifying in-place operations. However this requires manual optimisation by the developer.

Finally computing the method name and extracting the parameters as shown in Section 3.6.2 incurs a large overhead as well. Unfortunately meta-programming in Ruby is not sufficiently powerful to replace the operation with a method call to the compiled

Listing 6.8: Webcam viewer implemented using Python and OpenCV

```
import sys
from opencv import cv
from opencv import highgui
highgui.cvNamedWindow('Camera')
capture = highgui.cvCreateCameraCapture(-1)
while 1:
  frame = highgui.cvQueryFrame(capture)
  gray = cv.cvCreateImage(cv.cvSize(frame.width, frame.height), 8, 1)
  cv.cvCvtColor(frame, gray, cv.CV_BGR2GRAY)
  highgui.cvShowImage('Camera', gray)
  if highgui.cvWaitKey(5) > 0:
    break
```

Listing 6.9: Webcam viewer implemented using Ruby and Hornetseye

```
require 'hornetseye_v4l2'
require 'hornetseye_xorg'
include Hornetseye
capture = V4L2Input.new
X11Display.show(:title => 'Camera') { capture.read.to_ubyte }
```

code. A workaround would be to use compact representations for arrays and to use meta-programming in order to generate efficient Ruby code for calling the compiled C methods. However this requires sophisticated meta-programming and it would make maintaining the code more difficult.

6.4 Code Size

6.4.1 Code Size of Programs

Listing 6.8 shows the implementation of a webcam viewer using Python and OpenCV. The equivalent implementation using Ruby and Hornetseye is shown in Listing 6.9. One can see that the Ruby implementation is much shorter. One can also see that the semantics of the Ruby implementation is simpler. The OpenCV implementation requires the developer to write code for allocating memory for the result of the conversion to grey scale. Also the code for displaying images in a loop is more verbose because Python does not have support for closures.

A more sophisticated example is the Sobel gradient viewer. Listing 6.10 shows the Python/OpenCV implementation and Listing 6.11 shows the corresponding implementation using Ruby/Hornetseye. One can see that the OpenCV code is much more verbose. The readability of the code suffers because the code is cluttered with instructions for allocating memory for the return values (note that the static memory layout leads to performance improvements though (see Section 6.3.2)).

Listing 6.10: Sobel gradient viewer implemented using Python and OpenCV

```
import sys
from opencv import cv
from opencv import highgui
highgui.cvNamedWindow('Sobel Gradient')
capture = highgui.cvCreateCameraCapture(-1)
while 1:
  frame = highgui.cvQueryFrame(capture)
  gray = cv.cvCreateImage(cv.cvSize(frame.width, frame.height), 8, 1)
  cv.cvCvtColor(frame, gray, cv.CV_BGR2GRAY)
  sobel_x = cv.cvCreateMat(gray.height, gray.width, cv.CV_16S)
  cv.cvSobel(gray, sobel_x, 1, 0)
  sobel_y = cv.cvCreateMat(gray.height, gray.width, cv.CV_16S)
  cv.cvSobel(gray, sobel_y, 0, 1)
  square = cv.cvCreateMat(gray.height, gray.width, cv.CV_32F)
  cv.cvConvert(sobel_x * sobel_x + sobel_y * sobel_y, square)
  magnitude = cv.cvCreateMat(gray.height, gray.width, cv.CV_32F)
  cv.cvPow(square, magnitude, 0.5)
  dest = cv.cvCreateImage(cv.cvSize(frame.width, frame.height), 8, 1)
  cv.cvNormalize(magnitude, dest, 0, 255, cv.CV_MINMAX);
  highgui.cvShowImage('Sobel Gradient', dest)
  if highgui.cvWaitKey(5) > 0:
    break
```

Listing 6.11: Sobel gradient viewer implemented using Ruby and Hornetseye

```
require 'hornetseye_v4l2'
require 'hornetseye_xorg'
include Hornetseye
capture = V4L2Input.new
X11Display.show :title => 'Sobel Gradient' do
  img = capture.read.to_ubyte
  sobel_x, sobel_y = img.sobel(0).to_int, img.sobel(1).to_int
  Math.sqrt(sobel_x * sobel_x + sobel_y * sobel_y).normalise 0 .. 255
end
```

Table 6.2: Size of OpenCV code for filtering images

	lines	words	characters
cvcorner.cpp	664	2438	22743
cvderiv.cpp	831	3424	34024
cvfilter.cpp	2666	11235	100672
cvsmooth.cpp	1099	3917	34098
total	5260	21014	191537

Table 6.3: Size of Hornetseye code for all array operations

	lines	words	characters
total	5747	14721	132979

6.4.2 Code Size of Library

Implementing image processing operations is not well supported by the C/C++ language as shown in Section 2.2.1. This makes implementation of basic machine vision functionality more labour-intensive; a fact which is also reflected in the size of the library code. Table 6.2 shows that the OpenCV-1.0.0 library contains about 5400 lines of code and (excluding headers and comments) for implementing 2D filter operations for various integer and floating-point element types.

Table 6.3 shows the code size of *all* array operations of the Hornetseye library. This includes the source code for all type definitions and array operations presented in this thesis as well as the JIT-compiler. Also the operations are not limited to two or three dimensions. That is, implementing, maintaining, and extending a machine vision library is much less time-consuming when using a dynamically typed language.

6.5 Summary

In this chapter it was shown how the Ruby library developed in this thesis can be used to develop concise implementations of machine vision systems. For example the code for the Lucas-Kanade tracker (see Listing 6.1) is of similar size as the corresponding mathematical formalism. It is also worth noting that the lazy operations of the Ruby library facilitate a generic API for the Hough transform (*e.g.* see Listing 6.2).

The performance of the library was compared with the NArray Ruby extension and it was compared with an equivalent C++ implementation. It was shown that the Hornetseye library is about 4 times slower than an equivalent static C++ implementation. It was also shown that most of the performance loss is caused by the dynamic memory layout.

Further optimisation potential is in the code for calling compiled methods.

A webcam viewer and a Sobel gradient viewer each were implemented first using Python and OpenCV and then using Ruby and Hornetseye. In each case the implementation using Ruby+Hornetseye requires half as many lines of code as the Python+OpenCV implementation. Also the semantics of the Ruby implementation is much more concise. Furthermore the implementations of the libraries themselves were compared. It was clearly demonstrated that the library developed in this thesis allows for much higher productivity.

"I figure that since proprietary software developers use copyright to stop us from sharing, we cooperators can use copyright to give other cooperators an advantage of their own: they can use our code."

Richard Stallman

"Programs are not models of a part of reality. They are, when executed, a part of reality."

Klaus Ostermann

"Talk is cheap. Show me the code."

Linus Torvalds

7 Conclusions & Future Work

In this final chapter the results of the research in efficient implementation of machine vision algorithms are discussed. At the end of the thesis, future work is suggested.

7.1 Conclusions

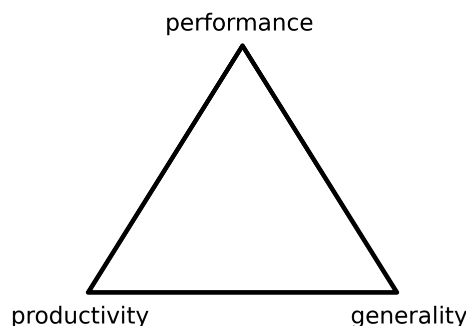


Figure 7.1: The main requirements when designing a programming language or system (Wolczko, 2011)

As Figure 7.1 illustrates, the fundamental problem of designing a programming language or system is to bring together *performance*, *productivity*, and *generality*. The work presented in this thesis is in that spirit. That is, it is about bringing together performance and productivity in an unprecedented way. Although this thesis is about machine vision, the results could be applied to numerical processing in general.

Existing free and open source software (FOSS) for machine vision is predominantly implemented in C/C++. Albeit the performance of machine code generated by C/C++ compilers is high, the static type system of the C++ language makes it exceedingly difficult to provide a complete and coherent basis for developing machine vision software.

It is hard to support all possible combinations of operations and native data types in a statically typed language. Therefore most libraries implemented in such a programming language either only support some combinations (*e.g.* OpenCV and NArray) or they always default back to a few selected native types (*e.g.* Lush and NumPy). In contrast Ruby already comes with a set of numeric data types which can be combined seamlessly.

The contribution of this thesis is a machine vision system which brings together performance and productivity in an unprecedented way. To achieve this, 16 Ruby extensions where implemented. In terms of **generality** the Ruby extensions provide

- extensive I/O integration for image- and video-data
- generic array operations for uniform multi-dimensional arrays
  - a set of objects to represent arrays, array views, and lazy evaluations in a modular fashion
  - optimal type coercions for all combinations of operations and data types

To address the **performance**, it was shown how the evaluation of expressions in Ruby can be changed so that it takes advantage of a JIT compiler . Since Ruby programs are not readily available as data the way they are in Lisp or Racket, it was necessary to use Ruby objects to represent various operations. The implementation presented in this thesis is about 4 times slower than the compiled code of an equivalent C implementation. That is, the system can be used for prototyping of real-time systems in industrial automation and in some cases the performance is sufficient to implement the actual system. It was shown that the main bottlenecks are the dynamic memory layout (*i.e.* less utilisation of the CPU cache) and the overhead of the calling Ruby code.

The programming language facilitates concise and flexible implementations which means that developers can achieve high **productivity**. It was demonstrated how the library introduced in this thesis can be used to implement machine vision algorithms. Concise implementations of various computer vision algorithms were presented in order to demonstrate the productivity of the system. Note that the concise implementations make several formal identities visible:

- Section 5.2.2 demonstrates the commonalities of different corner detectors
- In Section 6.2.3 it was shown how the Hough transform can be implemented using lazy operations and a histogram operation
- Sum, product, minimum, and maximum all can be defined using injections (introduced in Section 3.5.7)
- Section 3.5.6 shows that warps and lookup-tables are formally identical
- The convolutions in Section 3.5.10 and the morphological operations shown in Section 5.1.2.1 are both implemented using diagonal injections (also see Figure 3.16)

It was shown beyond proof-of-concept that a dynamically typed language can be used to overcome the limitations of current machine vision systems.

7.2 Future Work

Although the Ruby programming language was used, the field of machine vision could greatly benefit from *any* programming language which has equal or stronger support for meta-programming (*e.g.* Racket (former PLT Scheme)). Computer programs implemented in a language such as LISP, Racket, or Clojure are specified using s-expressions. That is, the program is itself data which can be manipulated by another part of the program. This facilitates implementation of optimisation algorithms which would be hard to do in currently popular programming languages. For example generating optimal code for a convolution with a certain filter and a certain number of dimensions. Another example would be to implement a garbage collector which could imitate the static memory layout for better performance of the resulting program.

Recently GPGPUs have become popular for doing parallel computing. Using meta-programming it is possible to avoid implementing large amounts of hardware-dependent code to access APIs such as OpenCL[1] in order to utilise GPGPUs. Future work could be to transparently integrate GPGPU computation the same way a JIT compiler can be integrated as shown in this thesis. The difference would be that the JIT compiler would generate calls to the OpenCL API instead of generating C code. Note that data transfers rates from main memory to the GPU and back are low on current architectures. That is, it is important to compile large expressions and avoid round-trips to the main memory. Lazy operations as implemented in this thesis could be used to address this problem.

The thesis was mostly focused on a more rigorous formal understanding of basic image processing operations and feature extraction. Future should address the problem of implementing more complex algorithms such as FFT or RANSAC in a similar fashion in order to build complex machine vision systems for tasks such as panorama stitching or SLAM. Developing a more rigorous formal understanding of existing machine vision algorithms might help to discover more commonalities. By understanding how different algorithms are related, one can avoid redundant work when developing machine vision systems. Furthermore by aligning formal descriptions and actual implementation with each other, the work flow becomes more efficient. That is, there should be no need to first specify a prototype system in an abstract language, and then have it specified again in the programming language used to implement the actual system.

Figure 7.2 explains why a popular programming language can remain popular for a long time even if it is obviously deficient. Developers tend to choose a programming languages with a familiar syntax. That means their choice is biased toward one of the popular programming languages or a language which looks similar. But if many deve-

[1] http://www.khronos.org/opencl/

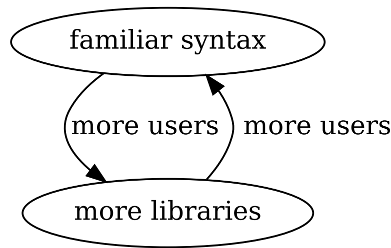


Figure 7.2: Vicious cycle leading to programming languages becoming entrenched

lopers choose a particular programming language, they will write many programming libraries for it which in turn attracts more users. That is, the individual choice of programming language is forced by the availability of libraries available for that language.

It is important to be aware of the factors influencing once's choice when selecting a programming language for developing a machine vision system. A programming language offering superior abstractions makes it possible to develop a more rigorous understanding of existing machine vision algorithms and capture it for future work. Not only will it make the library more generic and powerful but it will also make it easier for users of future programming languages to reuse the code.

S-expression syntax is more verbose than Ruby syntax for small programs. For example the Racket equivalent to the Ruby term "`2 + 3`" is "`(+ 2 3)`". Ruby could be seen as a programming language which offers less meta-programming than Racket but provides a more readable syntax. Future research could be into developing programming languages which have a regular structure and are open to modification like Racket and are readable like Ruby. This work could take inspiration from the way the numerical libraries of Ruby are implemented.

```
v=0000;eval$s=%q~d=%!^Lcf<LK8,                  _@7gj*LJ=c5nM)Tp1g0%Xv.,S[<>YoP
4ZojjV)O>qIH1/n[|2yE[>:ieC       "%.#%  :::##"       97N-A&Kj_K_><wS5rtWk@*a+Y5
yH?b[F^e7C/56j|pmRe+:)B     "##%      ::##########"     O98(Zh)’Iof*nm.,$C5Nyt=
PPu01Avw^<IiQ=5$’D-y?    "##:         ###############"    g6‘YT+qLw9k^ch|K’),tc
6ygIL8xI#LNz3v}T=4W    "#            #.   .####:#######"    lL27FZ0ij)7TQCI)P7u
}RT5-iJbbG5P-DHB<.   "              ##### # :###########"    R,YvZ_rnv6ky-G+4U’
$*are@b4U351Q-ug5   "              #######################"   00x8RR%‘Om7VDp4M5
PFixrPvl&<p[]1IJ   "              ############:####  %#####"   EGgDt8Lm#;bc4zS^
y]0‘_PstfUxOC(q   "              .#############:##%   .##  ."   /,}.YOIFj(k&q_V
zcaAi?]^lCVYp!;  " %%            .################.    #.   "  ;s="v=%04o;ev"%
(;v=(v-($*+[45,  ":####:          :##############%       :   "  ])[n=0].to_i;)%
360)+"al$s=%q#{  "%######.              #########            "  ;;"%c"%126+$s<<
126}";d.gsub!(/  "##########.           #######%             "  |\s|".*"/,"");;
require"zlib"||  "###########           :######.             "  ;d=d.unpack"C*"
d.map{|c|n=(n||  ":#########:           .######: .           "  )*90+(c-2)%91};
e=["%x"%n].pack   " :#######%           :###### #:          "   &&"H*";e=Zlib::
Inflate.inflate(   "  ######%           .####% ::          "   &&e).unpack("b*"
)[0];22.times{|y|   "  ####%             %###             "   ;w=(Math.sqrt(1-(
(y*2.0-21)/22)**(;   " .###:             .#%             "   ;2))*23).floor;(w*
2-1).times{|x|u=(e+    " %##                           "    )[y*z=360,z]*2;u=u[
90*x/w+v+90,90/w];s[(    " #.                        "    ;y*80)+120-w+x]=(""<<
32<<".:%#")[4*u.count((     " .                   "     ;"0"))/u.size]}};;puts\
s+";_ The Qlobe#{" "*18+ (       "#  :#######"       ;"Copyright(C).Yusuke End\
oh, 2010")}";exit~;_ The Qlobe                  Copyright(C).Yusuke Endoh, 2010
```

Yusuke Endoh - Ruby quine with rotating globe

Appendix

A.1 Connascence

"Connascence" is a term introduced by Weirich (2005, 2009). It is defined as follows

> Connascence occurs between two software components when ...
>
> - It is possible to postulate that some change in one component requires a change in the other component to preserve overall correctness.
> - It is possible to postulate some change that require both components to change together to preserve overall correctness.

Weirich (2005) gives examples of different types of connascence ordered by increasing degree of connascence

- Connascence of Name (static)
- Connascence of Type (static)
- Connascence of Meaning (static)
- Connascence of Algorithm (static)
- Connascence of Position (static)
- Connascence of Execution (dynamic)
- Connascence of Timing (dynamic)
- Connascence of Value (dynamic)
- Connascence of Identity (dynamic)

As a general rule it is desirable to reduce the degree of connascence in a system. Weirich (2009) uses several examples to illustrate how a connascence of high degree can be converted to a connascence of lower degree.

A.2 Linear Least Squares

The Gauss-Markov theorem states that given a regression model with uncorrelated zero-mean errors of equal variance, the linear least squares method is the best linear unbiased estimator. Any linear least squares problem can be formulated using a design matrix $\mathcal{H}$ and an observation vector $\vec{b}$ as shown in Equation A.1.

$$\mathcal{H}\,\vec{x} = \vec{b} + \vec{\epsilon} \tag{A.1}$$

The popular LAPACK library provides single- and double-precision solvers for linear systems. The LAPACK methods *sgels*[1] and *dgels*[2] are solvers for overdetermined as well as under determined linear systems. Here only the case where the equation system is overdetermined is discussed. In the overdetermined case a solution with minimal $|\vec{\epsilon}|$ is desired. That is, the term shown in Equation A.2 is to be minimised.

$$J(\vec{x}) = \left|\vec{\epsilon}\right|^2 = \vec{\epsilon}^\top\,\vec{\epsilon} = (\mathcal{H}\vec{x} - \vec{b})^\top\,(\mathcal{H}\vec{x} - \vec{b}) \tag{A.2}$$

The minimum can be determined using the necessary condition $\left.\frac{\delta J(\vec{x})}{\delta \vec{x}}\right|_{\widehat{\vec{x}}} \overset{!}{=} 0$ as shown in Equation A.3.

$$\left.\frac{\delta J(\vec{x})}{\delta \vec{x}}\right|_{\widehat{\vec{x}}} = \left.\frac{\delta(\vec{x}^\top\,\mathcal{H}^\top\,\mathcal{H}\,\vec{x} - 2\,\vec{x}^\top\,\mathcal{H}^\top\,\vec{b} + \vec{b}^\top\,\vec{b})}{\delta \vec{x}}\right|_{\widehat{\vec{x}}} = 2\,\mathcal{H}^\top\,\mathcal{H}\,\widehat{\vec{x}} - 2\,\mathcal{H}^\top\,\vec{b} \tag{A.3}$$

The linear least square estimate is obtained by solving for $\widehat{\vec{x}}$.

$$\widehat{\vec{x}} = (\mathcal{H}^\top\mathcal{H})^{-1}\mathcal{H}^\top\vec{b} \tag{A.4}$$

A.3 Pinhole Camera Model

Figure A.1 shows the pinhole camera model (Forsyth and Ponce, 2003). The object at position $\vec{x} = (x_1\ x_2\ x_3)^\top$ is projected onto the screen of the pinhole camera where it appears at the position $\vec{x}'$. The projection is a simple linear relation as shown in Equation A.5 and

[1] http://www.netlib.org/lapack/single/sgels.f
[2] http://www.netlib.org/lapack/double/dgels.f

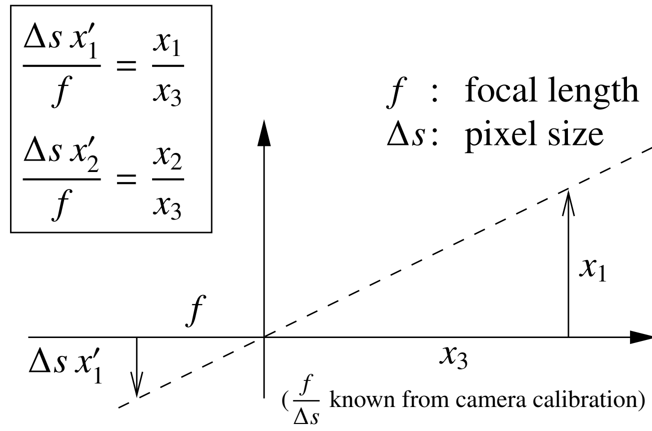


Figure A.1: Pinhole camera model

Equation A.6.

$$\frac{\Delta\, x_1'}{f} = \frac{x_1}{x_3} \tag{A.5}$$

$$\frac{\Delta\, x_2'}{f} = \frac{x_2}{x_3} \tag{A.6}$$

Δs is the size of a camera pixel and f is the focal length of the camera. That is, calibration of an ideal pinhole camera merely requires determining the ratio $f/\Delta s$.

A.4 Planar Homography

Equation A.5 and Equation A.6 can be represented using 2D homogeneous coordinates by introducing the unknown variable λ and an additional constraint for it as shown in Equation A.7 (Zhang, 2000).

$$\begin{aligned}
x_3\, x_1' &= \frac{f}{\Delta s}\, x_1 \\
x_3\, x_2' &= \frac{f}{\Delta s}\, x_2 \\
\Leftrightarrow \exists \lambda \in \mathbb{R}/\{0\} :\; & \begin{aligned} \lambda\, x_1' &= \frac{f}{\Delta s}\, x_1 \\ \lambda\, x_2' &= \frac{f}{\Delta s}\, x_2 \\ \lambda &= x_3 \end{aligned} \\
\Leftrightarrow \exists \lambda \in \mathbb{R}/\{0\} :\; & \lambda \begin{pmatrix} x_1' \\ x_2' \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} f/\Delta s & 0 & 0 \\ 0 & f/\Delta s & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{intrinsic parameters}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}
\end{aligned} \tag{A.7}$$

The parameters characterising the camera are also known as the *intrinsic parameters* (Zhang, 2000).

With homogeneous coordinates introducing an additional rotation and translation is straightforward. Equation A.8 shows the modified homogeneous equation.

$$
\begin{aligned}
&\exists\lambda \in \mathbb{R}/\{0\} :\ \lambda \begin{pmatrix} x'_1 \\ x'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} f/\Delta s & 0 & 0 \\ 0 & f/\Delta s & 0 \\ 0 & 0 & 1 \end{pmatrix} \overbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}}^{=:\mathcal{R}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \\
\Leftrightarrow &\exists\lambda \in \mathbb{R}/\{0\} :\ \lambda \begin{pmatrix} x'_1 \\ x'_2 \\ 1 \end{pmatrix} = \begin{pmatrix} f/\Delta s & 0 & 0 \\ 0 & f/\Delta s & 0 \\ 0 & 0 & 1 \end{pmatrix} \underbrace{\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix}}_{\text{extrinsic parameters}} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix}
\end{aligned}
\tag{A.8}
$$

Equation A.8 also shows how the rotation and translation can be represented with a single matrix if 3D homogeneous coordinates are used (Zhang, 2000). Rotation and translation of the camera can be different for each picture and the parameters are called *extrinsic parameters* (Zhang, 2000).

The planar calibration grid gives n correspondences of picture coordinates and 3D scene coordinates located on a plane in 3D space. That is, instead of a single pair $(\vec{x}, \vec{x}'$ there are n pairs of points as shown in Equation A.9

$$
\begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} \text{ is } \begin{pmatrix} m'_{11} \\ m'_{12} \end{pmatrix}, \begin{pmatrix} m'_{21} \\ m'_{22} \end{pmatrix}, \dots, \begin{pmatrix} m'_{n1} \\ m'_{n2} \end{pmatrix} \text{ and } \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \text{ is } \begin{pmatrix} m_{11} \\ m_{12} \\ 0 \end{pmatrix}, \begin{pmatrix} m_{21} \\ m_{22} \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} m_{n1} \\ m_{n2} \\ 0 \end{pmatrix}
\tag{A.9}
$$

Each point pair shown in Equation A.9 is inserted into Equation A.8. Equation A.8 also is modified to take into account that in reality the coordinates of the projection will be distorted by noise (ϵ_{ij}). The result is shown in Equation A.10 where $i \in \{1, 2, \dots, n\}$ (Zhang, 2000).

$$
\exists\lambda_i \in \mathbb{R}/\{0\} :\ \lambda_i \left(\begin{pmatrix} m'_{i1} \\ m'_{i2} \\ 1 \end{pmatrix} + \begin{pmatrix} \epsilon_{i1} \\ \epsilon_{i2} \\ 0 \end{pmatrix} \right) = \begin{pmatrix} f/\Delta s & 0 & 0 \\ 0 & f/\Delta s & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 0 \\ 1 \end{pmatrix}
\tag{A.10}
$$

$$
\Leftrightarrow\exists\lambda_i \in \mathbb{R}/\{0\} :\ \lambda_i \left(\begin{pmatrix} m'_{i1} \\ m'_{i2} \\ 1 \end{pmatrix} + \begin{pmatrix} \epsilon_{i1} \\ \epsilon_{i2} \\ 0 \end{pmatrix} \right) = \underbrace{\overbrace{\begin{pmatrix} f/\Delta s & 0 & 0 \\ 0 & f/\Delta s & 0 \\ 0 & 0 & 1 \end{pmatrix}}^{=:\mathcal{A}} \overbrace{\begin{pmatrix} r_{11} & r_{12} & t_1 \\ r_{21} & r_{22} & t_2 \\ r_{31} & r_{32} & t_3 \end{pmatrix}}^{=:\mathcal{R}'}}_{=:\mathcal{H}} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 1 \end{pmatrix}
\tag{A.11}
$$

The planar homography $\mathcal{H}$ is the product of the unknown camera intrinsic matrix $\mathcal{A}$ and the unknown extrinsic camera matrix $\mathcal{R}'$ as shown in Equation A.11. The equation is

reformulated as shown in Equation A.12.

$$
\begin{aligned}
(A.10) \Leftrightarrow \exists \lambda_i \in \mathbb{R}/\{0\} :\ \lambda_i \left(\begin{pmatrix} m'_{i1} \\ m'_{i2} \\ 1 \end{pmatrix} + \begin{pmatrix} \epsilon_{i1} \\ \epsilon_{i2} \\ 0 \end{pmatrix} \right) &= \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{33} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 1 \end{pmatrix} \\
\Leftrightarrow \underbrace{(h_{31}\, m_{i1} + h_{32}\, m_{i2} + h_{33})}_{\lambda_i} \left(\begin{pmatrix} m'_{i1} \\ m'_{i2} \end{pmatrix} + \begin{pmatrix} \epsilon_{i1} \\ \epsilon_{i2} \end{pmatrix} \right) &= \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{33} \end{pmatrix} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 1 \end{pmatrix}
\end{aligned}
\tag{A.12}
$$

It is not possible to isolate the error ϵ_{ij} in Equation A.12. However assuming that $\lambda_1 \approx \lambda_2 \approx \ldots \approx \lambda_n$ one can introduce $\widetilde{\epsilon}_{ij}$ as shown in Equation A.13 in order to isolate the error term (Zhang, 2000).

$$
\begin{aligned}
(h_{31}\, m_{i1} + h_{32}\, m_{i2} + h_{33}) \begin{pmatrix} m'_{i1} \\ m'_{i2} \end{pmatrix} + \begin{pmatrix} \widetilde{\epsilon}_{i1} \\ \widetilde{\epsilon}_{i2} \end{pmatrix} &= \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{33} \end{pmatrix} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 1 \end{pmatrix} \\
\Leftrightarrow \begin{pmatrix} \widetilde{\epsilon}_{i1} \\ \widetilde{\epsilon}_{i2} \end{pmatrix} &= \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{33} \end{pmatrix} \begin{pmatrix} m_{i1} \\ m_{i2} \\ 1 \end{pmatrix} - (h_{31}\, m_{i1} + h_{32}\, m_{i2} + h_{33}) \begin{pmatrix} m'_{i1} \\ m'_{i2} \end{pmatrix}
\end{aligned}
\tag{A.13}
$$

The errors $\widetilde{\epsilon}_{ij}$ are assumed to be uncorrelated, have zero mean, and have equal variance. That is, the least square estimator is the best linear unbiased estimator for the camera matrix $\mathcal{H}$ (Gauss-Markov theorem).

The complete linear least squares problem can be formulated by collecting the equations in a large matrix as shown in Equation A.14 and by stacking the coefficients of the matrix $\mathcal{H}$ in the vector $\vec{h}$.

$$
\underbrace{\begin{pmatrix}
m_{11} & m_{12} & 1 & 0 & 0 & 0 & -m'_{11}\, m_{11} & -m'_{11}\, m_{12} & -m'_{11} \\
0 & 0 & 0 & m_{11} & m_{12} & 1 & -m'_{12}\, m_{11} & -m'_{12}\, m_{12} & -m'_{12} \\
m_{21} & m_{22} & 1 & 0 & 0 & 0 & -m'_{21}\, m_{21} & -m'_{21}\, m_{22} & -m'_{21} \\
0 & 0 & 0 & m_{21} & m_{22} & 1 & -m'_{22}\, m_{21} & -m'_{22}\, m_{22} & -m'_{22} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
m_{n1} & m_{n2} & 1 & 0 & 0 & 0 & -m'_{n1}\, m_{n1} & -m'_{n1}\, m_{n2} & -m'_{n1} \\
0 & 0 & 0 & m_{n1} & m_{n2} & 1 & -m'_{n2}\, m_{n1} & -m'_{n2}\, m_{n2} & -m'_{n2}
\end{pmatrix}}_{=:\mathcal{M}}
\underbrace{\begin{pmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{33} \end{pmatrix}}_{=:\vec{h}}
=
\begin{pmatrix} \widetilde{\epsilon}_{11} \\ \widetilde{\epsilon}_{12} \\ \widetilde{\epsilon}_{21} \\ \widetilde{\epsilon}_{22} \\ \vdots \\ \widetilde{\epsilon}_{n1} \\ \widetilde{\epsilon}_{n2} \end{pmatrix}
\tag{A.14}
$$

Additionally the constraint $||\vec{h}|| = \mu \neq 0$ is introduced in order to avoid the trivial solution. That is, the problem now is to find $\vec{h} \in \mathbb{R}^9$ so that $||\mathcal{M}\,\vec{h}||$ is minimal and $||\vec{h}|| = \mu$.

The solution is to perform a SVD on $\mathcal{M}$ as shown in Equation A.15.

$$
\mathcal{M} = \mathcal{U}\, \Sigma\, \mathcal{V}^* \tag{A.15}
$$

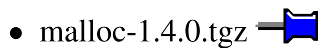
The linear least squares solution is $\vec{h} = \mu\,\vec{v}_9$ where $\vec{v}_9$ is the right-handed singular vector with the smallest singular value σ_9 (here μ is an arbitrary scale factor). The final solution for the planar homography is shown in Equation A.16.

$$\mathcal{H} = \mu \begin{pmatrix} v_{91} & v_{92} & v_{93} \\ v_{94} & v_{95} & v_{96} \\ v_{97} & v_{98} & v_{99} \end{pmatrix} \tag{A.16}$$

The planar homography $\mathcal{H}$ is sufficient to correctly map points from the plane of the calibration grid to the screen.

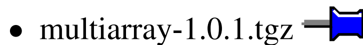
A.5 "malloc" gem

The source code of the "malloc" Ruby extension is provided as PDF attachment.

- malloc-1.4.0.tgz 

A.6 "multiarray" gem

The source code of the "multiarray" Ruby extension is provided as PDF attachment.

- multiarray-1.0.1.tgz 

A.7 Miscellaneous Sources

A.7.1 JIT Example

```
require 'rubygems'
require 'multiarray'
include Hornetseye
a = INT 5
b = ElementWise(proc { |x| -x }, :-@).new a
r = Pointer(INT).new
term = Store.new r, b
variables, values, skeleton = term.strip
types = variables.collect { |var| var.meta }
labels = Hash[*variables.zip((0 ... variables.size).to_a).flatten]
descriptor = skeleton.descriptor labels
method_name = ('_' + descriptor).tr( '(),+\-*/%.@?~&|^<=>',
                                     '0123\456789ABCDEFGH')
c = GCCContext.new 'extension'
f = GCCFunction.new c, method_name, *types
subst = Hash[*variables.zip(f.params).flatten]
skeleton.subst(subst).demand
f.compile
args = values.collect { |arg| arg.values }.flatten
GCCCache.send method_name, *args
r
```

A.7.2 Video Player

```
require 'rubygems'
require 'hornetseye_ffmpeg'
require 'hornetseye_xorg'
require 'hornetseye_alsa'
include Hornetseye
input = AVInput.new ARGV[0]
alsa = AlsaOutput.new 'default', input.sample_rate, input.channels
audio_frame = input.read_audio
X11Display.show((input.width * input.aspect_ratio).to_i, input.height,
                :title => ARGV.first, :output => XVideoOutput) do |display|
  video_frame = input.read_video
  n = alsa.avail
  while alsa.avail >= audio_frame.shape[1]
    alsa.write audio_frame
    audio_frame = input.read_audio
  end
  t = input.audio_pos - (alsa.delay + audio_frame.shape[1]).quo(alsa.rate)
  delay = [input.video_pos - t, 0].max
  display.event_loop delay
  video_frame
end
```

A.7.3 Normalised Cross-Correlation

```
class Node
  def avg
    sum / size
  end
  def sqr
    self * self
  end
  def corr(other)
    (rfft * other.rfft.conj).irfft
  end
  def zcorr(other)
    zother = MultiArray.dfloat(*shape).fill!
    zother[0 ... other.shape[0], 0 ... other.shape[1]] = other
    corr zother
  end
  def ma(*box)
    iself = MultiArray.dfloat(*shape).fill!
    iself[1 ... shape[0], 1 ... shape[1]] = self[0 ... shape[0] - 1,
                                                 0 ... shape[1] - 1]
    int = iself.integral
    int[0 ... shape[0] - box[0], 0 ... shape[1] - box[1]] +
      int[box[0] ... shape[0], box[1] ... shape[1]] -
      int[0 ... shape[0] - box[0], box[1] ... shape[1]] -
      int[box[0] ... shape[0], 0 ... shape[1] - box[1]]
  end
  def ncc(other, noise)
    box = other.shape
    zcorr(other - other.avg)[0 ... shape[0] - box[0],
                             0 ... shape[1] - box[1]] /
      Math.sqrt((sqr.ma(*other.shape) -
                 ma(*other.shape).sqr / other.size) *
                (other - other.avg).sqr.sum + noise)
```

```
  end
end
image = MultiArray.load_ubyte 'scene.jpg'
template = MultiArray.load_ubyte 'template.png'
ncc = image.to_dfloat.ncc template.to_dfloat, 0.1
shiftx, shifty = argmax { |i,j| ncc[i,j] }
result1 = image / 2
result2 = MultiArray.ubyte(*image.shape).fill!
result2[shiftx ... shiftx + template.shape[0],
        shifty ... shifty + template.shape[1]] = template / 2
(result1 + result2).show
```

A.7.4 Camera Calibration

```
require 'rubygems'
require 'matrix'
require 'linalg'
require 'hornetseye_rmagick'
require 'hornetseye_ffmpeg'
require 'hornetseye_xorg'
require 'hornetseye_v4l2'
include Linalg
include Hornetseye
class Matrix
  def to_dmatrix
    DMatrix[ *to_a ]
  end
  def svd
    to_dmatrix.svd.collect { |m| m.to_matrix }
  end
end
class Vector
  def norm
    Math.sqrt inner_product(self)
  end
  def normalise
    self * (1.0 / norm)
  end
  def reshape( *shape )
    Matrix[*MultiArray[*self].reshape(*shape).to_a]
  end
  def x( other )
    Vector[self[1] * other[2] - self[2] * other[1],
           self[2] * other[0] - self[0] * other[2],
           self[0] * other[1] - self[1] * other[0]] *
      (2.0 / (norm + other.norm))
  end
end
class DMatrix
  def to_matrix
    Matrix[ *to_a ]
  end
end
class Node
  def nms(threshold)
    self >= dilate.major(threshold)
  end
  def have(n, corners)
    hist = mask(corners).histogram max + 1
```

```
    msk = hist.eq n
    if msk.inject :or
      id = lazy(msk.size) { |i| i }.mask(msk)[0]
      eq id
    else
      nil
    end
  end
  def abs2
    real * real + imag * imag
  end
  def largest
    hist = histogram max + 1
    msk = hist.eq hist.max
    id = lazy(msk.size) { |i| i }.mask(msk)[0]
    eq id
  end
  def otsu(hist_size = 256)
    hist = histogram hist_size
    idx = lazy(hist_size) { |i| i }
    w1 = hist.integral
    w2 = w1[w1.size - 1] - w1
    s1 = (hist * idx).integral
    s2 = to_int.sum - s1
    u1 = (w1 > 0).conditional s1.to_sfloat / w1, 0
    u2 = (w2 > 0).conditional s2.to_sfloat / w2, 0
    between_variance = (u1 - u2) ** 2 * w1 * w2
    max_between_variance = between_variance.max
    self > idx.mask(between_variance >= max_between_variance)[0]
  end
end
def homography(m, ms)
  constraints = []
  m.to_a.flatten.zip(ms.to_a.flatten).each do |p,ps|
    constraints.push [p.real, p.imag, 1.0, 0.0, 0.0, 0.0,
                      -ps.real * p.real, -ps.real * p.imag, -ps.real]
    constraints.push [0.0, 0.0, 0.0, p.real, p.imag, 1.0,
                      -ps.imag * p.real, -ps.imag * p.imag, -ps.imag]
  end
  Matrix[*constraints].svd[2].row(8).reshape 3, 3
end
CORNERS = 0.3
W, H = ARGV[1].to_i, ARGV[2].to_i
W2, H2 = 0.5 * (W - 1), 0.5 * (H - 1)
N = W * H
SIZE = 21
GRID = 7
BOUNDARY = 19
SIZE2 = SIZE.div 2
f1, f2 = *(0 ... 2).collect do |k|
  finalise(SIZE,SIZE) do |i,j|
    a = Math::PI / 4.0 * k
    x = Math.cos(a) * (i - SIZE2) - Math.sin(a) * (j - SIZE2)
    y = Math.sin(a) * (i - SIZE2) + Math.cos(a) * (j - SIZE2)
    x * y * Math.exp( -(x**2+y**2) / 5.0 ** 2)
  end.normalise -1.0 / SIZE ** 2 .. 1.0 / SIZE ** 2
end
input = AVInput.new ARGV.first
width, height = input.width, input.height
coords = finalise(width, height) { |i,j| i - width / 2 + Complex::I * (j - height / 2) }
pattern = Sequence[*(([1] + [0] * (W - 2) + [1] + [0] * (H - 2)) * 2)]
```

```
o = Vector[]
d = Matrix[]
X11Display.show do
  img = input.read_ubytergb
  grey = img.to_ubyte
  corner_image = grey.convolve f1 + f2 * Complex::I
  abs2 = corner_image.abs2
  corners = abs2.nms CORNERS * abs2.max
  otsu = grey.otsu
  edges = otsu.dilate(GRID).and otsu.not.dilate(GRID)
  components = edges.components
  grid = components.have N, corners
  result = img
  if grid
    centre = coords.mask(grid.and(corners)).sum / N.to_f
    boundary = grid.not.components.largest.dilate BOUNDARY
    outer = grid.and(boundary).and corners
    vectors = (coords.mask(outer) - centre).to_a.sort_by { |c| c.arg }
    if vectors.size == pattern.size
      mask = Sequence[*(vectors * 2)].shift(vectors.size / 2).abs.nms(0.0)
      mask[0] = mask[mask.size-1] = false
      conv = lazy(mask.size) { |i| i }.mask(mask.to_ubyte.convolve(pattern.flip(0)).eq(4))
      if conv.size > 0
        offset = conv[0] - (pattern.size - 1) / 2
        m = Sequence[Complex(-W2, -H2), Complex(W2, -H2),
                     Complex(W2, H2), Complex(-W2, H2)]
        rect = Sequence[*vectors].shift(-offset)[0 ... vectors.size].mask(pattern) + centre
        h = homography m, rect
        v = h.inv * Vector[coords.real, coords.imag, 1.0]
        points = coords.mask(grid.and(corners)) + Complex(width/2, height/2)
        sorted = (0 ... N).zip((v[0] / v[2]).warp(points.real, points.imag).to_a,
                               (v[1] / v[2]).warp(points.real, points.imag).to_a).
          sort_by { |a,b,c| [(c - H2).round,(b - W2).round] }.collect { |a,b,c| a }
        m = finalise(W, H) { |i,j| i - W2 + (j - H2) * Complex::I }
        h = homography(m, sorted.collect { |j| points[j] - Complex(width/2, height/2)})
        o = Vector[*(o.to_a + [-h[2, 0] * h[2, 1], h[2, 1] ** 2 - h[2, 0] ** 2])]
        d = Matrix[*(d.to_a + [[h[0, 0] * h[0, 1] + h[1, 0] * h[1, 1]],
                   [h[0, 0] ** 2 + h[1, 0] ** 2 - h[0, 1] ** 2 - h[1, 1] ** 2]])]
        fs = 1.0 / ((d.transpose * d).inv * d.transpose * o)[0]
        if fs > 0
          f = Math.sqrt fs
          a = Matrix[[f, 0.0, 0.0], [0.0, f, 0.0], [0.0, 0.0, 1.0]]
          r1, r2, t = *proc { |r| (0 .. 2).collect { |i| r.column i } }.call(a.inv * h)
          s = (t[2] >= 0 ? 2.0 : -2.0) / (r1.norm + r2.norm)
          q = Matrix[(r1 * s).to_a, (r2 * s).to_a, (r1 * s).x(r2 * s).to_a].t
          r = proc { |u,l,vt| u * vt }.call *q.svd
          v = h.inv * Vector[coords.real, coords.imag, 1.0]
          result = (v[0] / v[2]).between?(-W2, W2).and((v[1] / v[2]).between?(-H2, H2)).
            conditional img * RGB(0, 1, 0), img
          gc = Magick::Draw.new
          gc.fill_opacity(0).stroke('red').stroke_width 1
          for i in 0 ... N
            j = sorted[i]
            gc.circle points[j].real, points[j].imag, points[j].real + 2, points[j].imag
            gc.text points[j].real, points[j].imag, "#{i+1}"
          end
          gc.stroke 'black'
          gc.text 30, 30, "f/ds = #{f}"
          result = result.to_ubytergb.to_magick
          gc.draw result
          result = result.to_ubytergb
```

```
        end
      end
    end
  end
  result
end
```

A.7.5 Recognition of a rectangular marker

```
W, H = 320, 240
THRESHOLD = 80
SIGMA = 1.5
ANGLE_BINS = 36
ORIENTATION_NOISE = 3
RANGE = 100 .. 10000
img = MultiArray.load_ubyte 'test.png'
grad_x, grad_y = img.gauss_gradient(SIGMA, 0), img.gauss_gradient(SIGMA, 1)
arg = ((Math.atan2(grad_y, grad_x) /
       Math::PI + 1) * ANGLE_BINS / 2).to_int % ANGLE_BINS
norm = Math.hypot grad_x, grad_y
components = (img <= THRESHOLD).components
n = components.max + 1
hist = components.histogram n
mask = hist.between? RANGE.min, RANGE.max
lazy(n) { |i| i }.mask(mask).to_a.each do |c|
  component = components.eq c
  edge = component.dilate.and component.erode.not
  orientations = arg.mask edge
  distribution = orientations.histogram ANGLE_BINS, :weight => norm.mask(edge)
  msk = distribution >= distribution.sum / (4 * ORIENTATION_NOISE)
  segments = msk.components
  if msk[0] and msk[msk.size - 1]
    segments = segments.eq(segments.max).conditional 1, segments
  end
  if segments.max == 4
    partitions = orientations.lut segments
    weights = partitions.histogram(5).major 1
    x = lazy(W, H) { |i,j| i + Complex::I * j }.mask edge
    centre = partitions.histogram(5, :weight => x) / weights
    diff = x - partitions.lut(centre)
    slope = Math.sqrt partitions.histogram(5, :weight => diff ** 2)
    corner = Sequence[*(0 .. 3).collect do |i|
        i1, i2 = i + 1, (i + 1) % 4 + 1
        l1, a1, l2, a2 = centre[i1], slope[i1], centre[i2], slope[i2]
        ( l1 * a1.conj * a2 - l2 * a1 * a2.conj -
          l1.conj * a1 * a2 + l2.conj * a1 * a2 ) /
          ( a1.conj * a2 - a1 * a2.conj )
    end]
    # Sequence(DCOMPLEX):
    # [ Complex(262.0, 117.7), Complex(284.2, 152.4), ... ]
    # ...
  end
end
```

The dominant gradient orientations are estimated by creating a gradient orientation histogram with 36 bins (“`distribution`”) and thresholding it (“`msk`”). The components of the resulting 1D binary array are labelled (“`segments`”). The variable “`partitions`” is a

1D array with the label of each edge pixel. The coordinates of each edge pixel are represented as complex numbers ("**x**"). The centre of each edge is determined by taking a histogram of the labels and using the pixel coordinates as weights ("**centre**").

The orientation of each edge is determined by computing a complex number representing the vector to the centre of the edge for each edge pixel ("**diff**"), squaring that number, accumulating the numbers for each edge, and taking the square root of the result for each edge ("**slope**"). Using squares of complex numbers takes care of the ambiguous representation of the edge's orientation (*i.e.* vectors with angle α and $\alpha + \pi$ can be used to represent the same edge orientation). This ensures that the vectors of an edge, which point in opposing directions, accumulate instead of cancelling each other out (the method was inspired by Bülow (1999) where a similar method is used to deal with the ambiguity of structure tensors).

The intersections of the four edges are the corners of the rectangle ("**corner**"). The corners can be used to determine a planar homography (see Appendix A.4). If the ratio of focal length to pixel size is known from camera calibration (see Section 6.2.8.2), it is also possible to estimate the 3D pose of the marker (*i.e.* separate the camera intrinsic and extrinsic parameters).

A.7.6 Constraining Feature Density

```
require 'rubygems'
require 'hornetseye_v4l2'
require 'hornetseye_xorg'
include Hornetseye
class Node
  def features(grad_sigma = 1.0, cov_sigma = 2.0, k = 0.05)
    gx, gy = gauss_gradient(grad_sigma, 0), gauss_gradient(grad_sigma, 1)
    cov = [gx ** 2, gy ** 2, gx * gy]
    a, b, c = cov.collect { |arr| arr.gauss_blur cov_sigma }
    trace = a + b
    determinant = a * b - c ** 2
    determinant - k * trace ** 2
  end
  def maxima(threshold)
    (self >= max * threshold).and eq(dilate)
  end
end
BLOCK = 20
THRESHOLD = 0.001
input = V4L2Input.new('/dev/video0') { [YUY2, 320, 240] }
w, h = input.width, input.height
m, n = w / BLOCK, h / BLOCK
x0, y0 = (w - m * BLOCK) / 2, (h - n * BLOCK) / 2
warp = [lazy(BLOCK, BLOCK, m * n) { |i,j,k| i + x0 + (k % m) * BLOCK },
        lazy(BLOCK, BLOCK, m * n) { |i,j,k| j + y0 + (k / m) * BLOCK }]
X11Display.show do
  img = input.read_ubytergb
  features = img.to_ubyte.features
  warped = features.warp *warp
  maxima = argmax { |i,j| lazy { |k| warped[i,j,k] } }
  xp = lazy { |i| x0 + maxima[0][i] + (i % m) * BLOCK }
```

```
  yp = lazy { |i| y0 + maxima[1][i] + (i / m) * BLOCK }
  mask = features.maxima(THRESHOLD).warp xp, yp
  x, y = xp.mask(mask), yp.mask(mask)
  [x, y].histogram(w, h).to_bool.dilate.conditional RGB(255, 0, 0), img
end
```

A.7.7 SVD Matching

```
require 'rubygems'
require 'hornetseye_v4l2'
require 'hornetseye_ffmpeg'
require 'hornetseye_linalg'
require 'hornetseye_xorg'
require 'hornetseye_rmagick'
require 'matrix'
include Linalg
include Hornetseye
class Node
  def features(grad_sigma = 1.0, cov_sigma = 2.0, k = 0.05)
    gx, gy = gauss_gradient(grad_sigma, 0), gauss_gradient(grad_sigma, 1)
    cov = [gx ** 2, gy ** 2, gx * gy]
    a, b, c = cov.collect { |arr| arr.gauss_blur cov_sigma }
    trace = a + b
    determinant = a * b - c ** 2
    determinant - k * trace ** 2
  end
  def maxima(threshold)
    (self >= max * threshold).and eq(dilate)
  end
  def svd
    to_dmatrix.svd.collect { |m| m.to_multiarray }
  end
  def x(other)
    (to_dmatrix * other.to_dmatrix).to_multiarray
  end
end
SIGMA = 40.0
GAMMA = 0.4
BLOCK = 20
PATCH = 5
CUTOFF = 0.8
THRESHOLD = 0.01
input = V4L2Input.new('/dev/video0') { [YUY2, 320, 240] }
w, h = input.width, input.height
m, n = w / BLOCK, h / BLOCK
x0, y0 = (w - m * BLOCK) / 2, (h - n * BLOCK) / 2
warp = [lazy(BLOCK, BLOCK, m * n) { |i,j,k| i + x0 + (k % m) * BLOCK },
        lazy(BLOCK, BLOCK, m * n) { |i,j,k| j + y0 + (k / m) * BLOCK }]
patch = [lazy(PATCH, PATCH) { |i,j| i - PATCH / 2 },
         lazy(PATCH, PATCH) { |i,j| j - PATCH / 2 }]
x_old, y_old, descriptor_old, variance_old = nil, nil, nil, nil
X11Display.show do
  img = input.read_ubytergb
  features = img.to_ubyte.features
  warped = features.warp *warp
  maxima = argmax { |i,j| lazy { |k| features.warp(*warp)[i,j,k] } }
  xp = lazy { |i| x0 + maxima[0][i] + (i % m) * BLOCK }
  yp = lazy { |i| y0 + maxima[1][i] + (i / m) * BLOCK }
  mask = features.maxima(THRESHOLD).warp xp, yp
```

```
  x, y = xp.mask(mask), yp.mask(mask)
  descriptor = img.to_ubyte.warp lazy { |i,j,k| x[k] + patch[0][i, j] },
                                 lazy { |i,j,k| y[k] + patch[1][i, j] }
  descriptor -= lazy { |k| descriptor[k].sum } / PATCH ** 2.0
  variance = finalise { |i| Math.sqrt (descriptor[i] ** 2).sum }
  result = [x, y].histogram(w, h).to_bool.dilate.conditional RGB(0, 0, 255), img
  if x_old and y_old and descriptor_old
    covariance = finalise { |i,j| (descriptor[i] * descriptor_old[j]).sum } /
                 lazy { |i,j| variance[i] * variance_old[j] }
    proximity = finalise do |i,j|
      Math.exp -((x[i] - x_old[j]) ** 2 + (y[i] - y_old[j]) ** 2) / (2 * SIGMA ** 2.0)
    end
    similarity = finalise do |i,j|
      Math.exp -(covariance[i,j] - 1) ** 2.0 / (2 * GAMMA ** 2.0)
    end
    measure = proximity * similarity
    t, d, ut = *measure.svd
    e = lazy(*d.shape) { |i,j| i.eq(j).conditional 1, 0 }
    s = t.x(e).x(ut)
    max_col = argmax { |j| lazy { |i| s[j, i] } }.first
    max_row = argmax { |j| lazy { |i| s[i, j] } }.first
    mask_col = [lazy(s.shape[0]) { |i| i }, max_row].histogram(*s.shape) > 0
    mask_row = [max_col, lazy(s.shape[1]) { |i| i }].histogram(*s.shape) > 0
    q = mask_col.and(mask_row).and measure >= CUTOFF
    gc = Magick::Draw.new
    gc.stroke 'red'
    gc.stroke_width 1
    a, b = lazy(*d.shape) { |i,j| i }.mask(q), lazy(*d.shape) { |i,j| j }.mask(q)
    a.to_a.zip(b.to_a).each do |i,j|
      gc.line x[i], y[i], x_old[j], y_old[j]
    end
    img = result.to_ubytergb.to_magick
    gc.draw img
    result = img.to_ubytergb
  end
  x_old, y_old, descriptor_old, variance_old = x, y, descriptor, variance
  result
end
```

Bibliography

Hal Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Citeseer, 1996. URL http://mitpress.mit.edu/sicp/. 40

M. Abrash. A first look at the Larrabee new instructions (LRBni). *The Dr. Dobb's Journal*, 2009. URL http://drdobbs.com/high-performance-computing/216402188. 72

I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. ROOT – a C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12): 2499–2512, 2009. URL http://www.sciencedirect.com/science/article/pii/S0010465509002550. 4

S. Baker and I. Matthew. Lucas-Kanade 20 years on: a unifying framework. *International Journal of Computer Vision*, 56(3):221–55, February 2004. URL http://www.ri.cmu.edu/projects/project_515.html. 61, 127

D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice Hall, 1982. URL http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/bandb.htm. 141

H.P. Barendregt and E. Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4(2):337–372, 1984. URL ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf. 47, 48

Kai Uwe Barthel. 3d-data representation with ImageJ. In *ImageJ User and Developer Conference 2006*, May 2006. URL http://www.f4.htw-berlin.de/~barthel/ImageJ/ImageJ3D/3D-Data%20Representation%20with%20ImageJ.pdf. 73

Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2008. ISBN 9780137143979. URL http://www.qteverywhere.com/qt/book/c-gui-programming-with-qt-4-2ndedition.pdf. 99

M. Boissenin, J. Wedekind, A. N. Selvan, B. P. Amavasai, F. Caparrelli, and J. R. Travis. Computer vision methods for optical microscopes. *Image and Vision Computing*, 25

(7):1107–16, July 2007. URL http://dx.doi.org/10.1016/j.imavis.2006.03.009. ii

Jaroslav Borovička. Circle detection using Hough transform documentation. Technical report, 2003. URL http://linux.fjfi.cvut.cz/~pinus/bristol/imageproc/hw1/report.pdf. 132

Jean-Yves Bouget. Pyramidal implementation of the Lucas Kanade feature tracker description of the algorithm. Technical report. URL http://robots.stanford.edu/cs223b04/algo_tracking.pdf. 121

Jean-Yves Bouguet. Camera calibration toolbox for Matlab, 2010. URL http://www.vision.caltech.edu/bouguetj/calib_doc/. 142

Eric Brasseur. Gamma error in picture scaling. Web site, 2007. URL http://www.4p8.com/eric.brasseur/gamma.html. 82

Thomas Bülow. *Hypercomplex Spectral Signal Representations for Image Processing and Analysis*. PhD thesis, Christian-Albrechts-Universität, Kiel, Germany, 1999. URL http://www.informatik.uni-kiel.de/uploads/tx_publication/1999_tr03.ps.gz. 174

J. Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986. URL http://www.limsi.fr/Individu/vezien/PAPIERS_ACS/canny1986.pdf. 115

Alonso Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941). 47, 48

J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. URL http://dx.doi.org/10.2307/2003354. 111

Peter Cooper. *Beginning Ruby: from novice to professional*. Apress, 2009. URL http://beginningruby.org/. 21

A. J. Davison. Real-time simultaneous localisation and mapping with a single camera. In *ICCV 2003: 9th International Conference on Computer Vision*, volume 2, pages 1403–10, Los Alamitos, CA, USA, October 2003. Dept. of Eng. Sci., Oxford Univ., UK. URL http://www.doc.ic.ac.uk/~ajd/Publications/davison_iccv2003.pdf. 10

Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. MonoSLAM: Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007. 142

Tom DeMarco and Timothy Lister. *Peopleware: productive projects and teams*. New York: Dorset House Publishing, second edition, 1987. 2

Konstantinos G. Derpanis. The Harris corner detector. Technical report, October 2004. URL http://www.cse.yorku.ca/~kosta/CompVis_Notes/harris_detector.pdf. 118

Michael Droettboom, Karl Macmillan, and Ichiro Fujinaga. The Gamera framework for building custom recognition systems. In *Proceedings of the Symposium on Document Image Understanding Technologies*, pages 275–286, 2003. URL http://gamera.informatik.hsnr.de/publications/droettboom_gamera_03.pdf. 15

V. B. Dröscher. *Magie der Sinne im Tierreich*. Dt. Taschenbuch-Verl., 1975. 81

R.O. Duda and P.E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972. ISSN 0001-0782. URL http://www.ai.sri.com/pubs/files/tn036-duda71.pdf. 132

R. Fattal, D. Lischinski, and M. Werman. Gradient domain high dynamic range compression. *ACM Transactions on Graphics*, 21(3):249–256, 2002. ISSN 0730-0301. URL http://www.cs.huji.ac.il/~danix/hdr/hdrc.pdf. 88

Julien Faucher. Camera calibration and 3-d reconstruction. Technical report, June 2006. 141

Paul Fenwick. An illustrated history of failure. Presentation at OSCON 2008, Portland, Oregon, 2008. URL http://blip.tv/file/1137169. 5

Bob Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. Roberts cross edge detector. Web page, 2003. URL http://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm. 113

David A. Forsyth and Jean Ponce. *Computer Vision: A modern Approach*. Prentice Hall, 2003. URL http://luthuli.cs.uiuc.edu/~daf/. 72, 106, 164

Barak Freedman, Alexander Shpunt, Meir Machline, and Yoel Arieli. Depth mapping using projected patterns. United States Patent, May 2010. URL http://www.freepatentsonline.com/20100118123.pdf. 97

M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–31, February 2005. URL http://fftw.org/fftw-paper-ieee.pdf. 111

Hal Fulton. *The Ruby Way*. Addison Wesley, November 2006. URL http://rubyhacker.com/. xviii, 21, 24, 26, 27, 30, 38

Frans A. Gerritsen and Piet W. Verbeek. Implementation of cellular-logic operators using 3*3 convolution and table lookup hardware. *Computer Vision, Graphics, and Image Processing*, 27(1):115–123, 1984. ISSN 0734-189X. URL http://dx.doi.org/10.1016/0734-189X(84)90086-0. 104

Paul Graham. *On Lisp*. Prentice Hall, 1994. URL http://www.paulgraham.com/onlisptext.html. 20

M. A. Greenspan, L. Shang, and P. Jasiobedzki. Efficient tracking with the Bounded Hough Transform. In *CVPR'04: Computer Vision and Pattern Recognition*, June 2004. URL http://www.ece.queensu.ca/hpages/faculty/greenspan/papers/GreShaJas04.pdf. 10

A. Gurtovoy and D. Abrahams. Qt4.6 white paper. Technical report, Nokia, 2009. URL http://qt.nokia.com/products/files/pdf/qt-4.6-whitepaper. 99

Eric Hamilton. JPEG file interchange format, 1992. URL http://www.jpeg.org/public/jfif.pdf. 83

Robert M. Haralick, Stanley R. Sternberg, and Xinhua Zhuang. Image analysis using mathematical morphology. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-9(4):532 –550, July 1987. URL http://dx.doi.org/10.1109/TPAMI.1987.4767941. 102

C. G. Harris and M. Stephens. A combined corner and edge detector. *Proceedings 4th Alvey Vision Conference*, pages 147–151, 1988. URL http://www.rose-hulman.edu/class/cs/csse461/handouts/Day26/avc-88-023.pdf. 118

H. Heuser. *Lehrbuch der Analysis, Teil 1*. Teubner, ninth edition, 1991. 47

Karl Hinderer. *Stochastik für Informatiker und Ingenieure*. Institut für Mathematische Stochastik Universität Karlsruhe, Karlsruhe, fourth edition, 1993. 107

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. URL http://www.cs.nott.ac.uk/~gmh/fold.pdf. 62

Andrew E. Johnson and Martial Hebert. Using Spin images for efficient object recognition in cluttered 3d scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(5):433–449, 1999. URL http://www.ri.cmu.edu/publication_view.html?pub_id=3598. 10

Florian Kainz and Rod Bogard. Technical introduction to OpenEXR. Technical report, February 2009. URL http://www.openexr.com/TechnicalIntroduction.pdf. 88

Hirukazu Kato and Mark Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR 99)*, page 85, October 1999. URL http://www.hitl.washington.edu/artoolkit/Papers/IWAR99.kato.pdf. 148

Y. Lamdan and H. J. Wolfson. Geometric hashing: a general and efficient model-based recognition scheme. Second International Conference on Computer Vision, pages 238–249, Tampa, FL, USA, December 1988. Courant Inst. of Math., New York Univ., NY, USA. ISBN 0 8186 0883 8. URL http://www.cs.utexas.edu/~grauman/courses/spring2007/395T/395T/papers/Lamdan88.pdf. 10

W. Landry. Implementing a high performance tensor library. *Scientific Programming*, 11(4):273–90, 2003. URL http://www.oonumerics.org/FTensor/FTensor.pdf. 18

Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. URL http://llvm.org/pubs/2002-12-LattnerMSThesis.html. 32

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278 –2324, November 1998. URL http://dx.doi.org/10.1109/5.726791. 20

J. P. Lewis. Fast template matching. In *Vision Interface*, volume 10, pages 120–123. Citeseer, 1995. URL http://scribblethink.org/Work/nvisionInterface/vi95_lewis.pdf. 10, 127

Kim Chuan Lim. *Development of 3-D Surface Data Acquisition System Using Non-Calibrated Laser Alignment Techniques*. PhD thesis, Sheffield Hallam University, September 2009. 141

A. J. Lockwood, J. Wedekind, R. S. Gay, M. S. Bobji, B. P. Amavasai, M. Howarth, G. Möbus, and B. J. Inkson. Advanced transmission electron microscope triboprobe with automated closed-loop nanopositioning. *Measurement Science and Technology*, 21(7):075901, 2010. URL http://dx.doi.org/10.1088/0957-0233/21/7/075901. ii, 134

Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 674–679, April 1981. URL http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.2019. 118

R. Lämmel. Google's MapReduce programming model - revisited. *Science of Computer Programming*, 70(1):1–30, 2008. ISSN 0167-6423. URL http://www.systems.ethz.ch/education/past-courses/hs08/map-reduce/reading/mapreduce-progmodel-scp08.pdf. 61

Robert Martin. What killed Smalltalk could kill Ruby, too. Presentation at RailsConf, 2009. URL http://www.youtube.com/watch?v=YX3iRjKj7C0. 31

Yukihiro Matsumoto. The Ruby programming language. *informIT*, June 2000. URL http://www.informit.com/articles/article.aspx?p=18225. 21

Yukihiro Matsumoto. *Ruby in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, Inc., 2002. URL http://oreilly.com/catalog/9780596002145. 21

Yukihiro Matsumoto. *Beautiful Code*, chapter 29. O'Reilly Media, Inc., 2007. URL http://oreilly.com/catalog/9780596510046. 22

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.9776. 47, 48

Paulo R. S. Mendonça and R. Cipolla. A simple technique for self-calibration. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 1. IEEE, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.5767&rep=rep1&type=pdf. 142

Joseph Mundy. Model-based computer vision. University Video Communication, 1987. URL http://www.archive.org/details/JosephMu1987. 3

J. Nakamura and M. Csikszentmihalyi. The concept of flow. *Handbook of positive psychology*, pages 89–105, 2002. URL http://myweb.stedwards.edu/michaelo/2349/paper1/ConceptOfFlow.pdf. 2

O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gaelli, and R. Wuyts. On the revival of dynamic languages. In *Software Composition, 4th International Workshop*, pages 1–13. Springer, 2005. URL http://scg.unibe.ch/archive/papers/Nier05bRevival.pdf. 1, 3, 4

T. Oliphant. *A Guide to NumPy*. Trelgol Publishing, 2006. URL http://www.tramy.us/numpybook.pdf. 17

Nobuyuki Otsu. A threshold selection method from gray-level histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1):62–66, January 1979. URL http://web.ics.purdue.edu/~kim497/ece661/OTSU_paper.pdf. 105

Q. Pan, G. Reitmayr, and T. Drummond. ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition. In *Proc. 20th British Machine Vision Conference (BMVC)*, London, September 2009. URL http://mi.eng.cam.ac.uk/~qp202/my_papers/BMVC09/BMVC09.pdf. 10

L. D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2): 12–15, February 2007. URL http://dx.doi.org/10.1109/MC.2007.53. 6

Maurizio Pilu. Uncalibrated stereo correspondence by singular value decomposition. Technical report, Hwelett Packard Lab Technical Publ Dept, 1997. URL http://www.hpl.hp.com/techreports/97/HPL-97-96.pdf. 123

M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59(3):207–32, 2004. URL http://www.cs.unc.edu/~marc/pubs/PollefeysIJCV04.pdf. 10

POVRay. Persistence of vision raytracer, 2005. URL http://www.povray.org/. 73, 148

Mark Pupilli. *Particle Filtering for Real-time Camera Localisation*. PhD thesis, Department of Computer Science, Bristol University, 2006. URL http://www.cs.bris.ac.uk/Publications/pub_master.jsp?id=2000621. 10, 142

R.Venkat Rajendran. White paper on unit testing. Technical report, February 2002. URL http://www.mobilein.com/WhitePaperonUnitTesting.pdf. 32

Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 2006. ISBN 978-0-12-585263-0. URL http://www.hdrbook.com/. 81

Brent Roman, Chris Scholin, Scott Jensen, Eugene Massion, Roman Marin III, Christina Preston, Dianne Greenfield, William Jones, and Kevin Wheeler. Controlling a robotic marine environmental sampler with the Ruby scripting language. *JALA - Journal of the Association for Laboratory Automation*, 12(1):56–61, 2007. URL http://www.jalajournal.com/article/S1535-5535(06)00349-2/abstract. 3

Koichi Sasada. Future of Ruby VM. Presentation at RubyConf 2008, Orlando, Florida, 2008. URL http://www.atdot.net/~ko1/activities/rubyconf2008_ko1.pdf. 22

Koichi Sasada. Ruby memory management hacks. Presentation at RubyConf 2009, San Francisco, California, 2009a. URL http://www.atdot.net/~ko1/activities/rubyconf2009_ko1_pub.pdf. 31

Koichi Sasada. Ricsin: RubyにCを埋め込むシステム. In 情報処理学会論文誌, volume 2, pages 13–26, 2009b. URL http://www.atdot.net/~ko1/activities/ricsin2009.pdf. 33

Koichi Sasada. Ricsin: RubyにCを埋め込むシステム. Conference presentation, 2009c. URL http://www.atdot.net/~ko1/activities/ricsin2009_pro.pdf. 33

Y. Shan, B. Matei, HS Sawhney, R. Kumar, D. Huber, and M. Hebert. Linear model hashing and batch RANSAC for rapid and accurate object recognition. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages 121–128, 2004. URL http://www.ri.cmu.edu/pub_files/pub4/shan_y_2004_1/shan_y_2004_1.pdf. 10

Jianbo Shi and Carlo Tomasi. Good features to track. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 593–600, Seattle, WA, USA, June 1994. Cornell Univ, Ithaca, NY, USA. URL http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.135.7147. 116, 118

T. Smith and J. Guild. The C.I.E. colorimetric standards and their use. *Transactions of the Optical Society*, 33(3):73, 1931. URL http://dx.doi.org/10.1088/1475-4878/33/3/301. 81

Irwin Sobel and Gary Feldman. A 3x3 isotropic gradient operator for image processing. Never published but presented at a talk at the Stanford Artificial Project, 1968. 114

Guy L Steele and Gerald J Sussman. Design of lisp-based processors, or scheme: A dielectric lisp, or finite memories considered harmful, or lambda: The ultimate opcode. Technical report, Cambridge, MA, USA, March 1979. URL http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-514.pdf. 20, 101

Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta. A standard default color space for the Internet - sRGB. Web site, 1996. URL http://www.w3.org/Graphics/Color/sRGB. 82

Masahiro Tanaka. NArray and scientific computing with Ruby. Presentation at RubyKaigi 2010, Tokyo, Japan, August 2010a. URL http://www.slideshare.net/masa16tanaka/narray-and-scientific-computing-with-ruby. 16

Masahiro Tanaka. Pwrake distributed workflow engine for e-science. Presentation at RubyConf 2010, New Orleans, Louisiana, November 2010b. URL http://www.slideshare.net/masa16tanaka/ruby-conftanaka16. 16

Masahiro Tanaka. Ruby科学データ処理ツールの開発NArrayとPwrake. Presentation, July 2011. URL http://www.slideshare.net/masa16tanaka/narray-pwrake. 154

Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby*. Pragmatic Bookshelf, first edition, 2004. 27

C. Tomasi and T. Kanade. Shape and motion from image streams under orthography: a factorization method. *International Journal of Computer Vision*, 9(2):137–54, November 1992. URL http://www-inst.cs.berkeley.edu/~cs294-6/fa06/papers/TomasiC_Shape%20and%20motion%20from%20image%20streams%20under%20orthography.pdf. 10, 118

Laurence Tratt and Roel Wuyts. Guest editors' introduction: Dynamically typed languages. *IEEE Software*, 24(5):28–30, 2007. URL http://dx.doi.org/10.1109/MS.2007.140. 4, 5, 13

M. Alex O. Vasilescu and Demetri Terzopoulos. Multilinear projection for appearance-based recognition in the tensor framework. *Computer Vision, IEEE International Conference on*, 0:1–8, 2007. URL http://www.media.mit.edu/~maov/mprojection/iccv07.pdf. 10

Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001. URL http://research.microsoft.com/en-us/um/people/viola/Pubs/Detect/violaJones_IJCV.pdf. 10, 69

J. Wedekind, B. P. Amavasai, and K. Dutton. Steerable filters generated with the hypercomplex dual-tree wavelet transform. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 1291–4, a. URL http://shura.shu.ac.uk/953/. ii

J. Wedekind, B. P. Amavasai, K. Dutton, and M. Boissenin. A machine vision extension for the Ruby programming language. In *2008 International Conference on Information and Automation (ICIA)*, pages 991–6. IEEE, b. URL http://shura.shu.ac.uk/952/. ii, xviii, 13, 127

Jan Wedekind. Fokusserien-basierte Rekonstruktion von Mikroobjekten. Master's thesis, University of Karlsruhe (TH), May 2002. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1872002. 134

Jan Wedekind. Real-time computer vision with Ruby. Presentation at OSCON 2008, Portland, Oregon, 2008. URL http://www.slideshare.net/wedesoft/oscon08-foils. ii, 127

Jan Wedekind. Computer vision using Ruby and libJIT. Presentation at RubyConf 2009, San Francisco, California, 2009. URL http://www.slideshare.net/wedesoft/rubyconf09. iii, 139

Jan Wedekind, Manuel Boissenin, Bala P. Amavasai, Fabio Caparrelli, and Jon R. Travis. Object recognition and real-time tracking in microscope imaging. Proceedings of the 2006 Irish Machine Vision and Image Processing Conference (IMVIP 2006), pages 164–171, Dublin City University, 2006. URL http://www.scribd.com/doc/71015261/Object-Recognition-and-Real-time-Tracking-in-Microscopic-Imaging. ii

Jan Wedekind, Jacques Penders, Hussein Abdul-Rahman, Martin Howarth, Ken Dutton, and Aiden Lockwood. Implementing machine vision systems with a dynamically typed language. Demonstration at ECOOP 2011, Lancaster, United Kingdom, 2011. URL http://ecoop11.comp.lancs.ac.uk/?q=content/implementing-machine-vision-systems-dynamically-typed-language. iii

Jim Weirich. Connascence and Java. Web site, March 2005. URL http://onestepback.org/articles/connascence/. 163

Jim Weirich. The building blocks of modularity. Presentation at Mountain West Ruby Conference 2009, Salt Lake City, Utah, 2009. URL http://confreaks.net/videos/77-mwrc2009-the-building-blocks-of-modularity. 13, 163, 164

Andy Wilson. Robust computer vision-based detection of pinching for one and two-handed gesture input. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 255–258. ACM, 2006. ISBN 1595933131. URL http://research.microsoft.com/en-us/um/people/awilson/publications/wilsonuist2006/UIST%202006%20TAFFI.pdf. 136, 139

Dave Wilson. Video codecs and pixel formats. Web site, 2007. URL http://www.fourcc.org/. xix, 83, 84, 85

Mario Wolczko. Past, present and future of virtual machines: A personal view. Presentation at ICOOOLPS workshop at ECOOP, 2011. xxi, 159

Mario Wolczko, Ole Agesen, and David Ungar. Towards a universal implementation substrate for object-oriented languages. *OOPSLA 99 workshop on Simplicity, Performance, and Portability in Virtual Machine Design*, 1999. URL http://labs.oracle.com/people/mario/pubs/substrate.pdf. 5

Jingyu Yan and Marc Pollefeys. Automatic kinematic chain building from feature trajectories of articulated objects. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2006*, volume 1, pages 712–719, New York, NY, United States, June 2006. Department of Computer Science, University of North

Carolina at Chapel Hill, Chapel Hill, NC 27599, Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States. URL http://www.cs.unc.edu/~marc/pubs/YanCVPR06.pdf. 10

G. Z. Yang, P. Burger, D. N. Firmin, and S. R. Underwood. Structure adaptive anisotropic image filtering. *Image and Vision Computing*, 14(2):135–45, March 1996. URL http://dx.doi.org/10.1016/0262-8856(95)01047-5. 115

Zhengyou Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000. URL http://research.microsoft.com/en-us/um/people/zhang/Papers/TR98-71.pdf. 141, 146, 147, 148, 165, 166, 167