Gernot Ziegler

GPU Data Structures for Graphics and Vision

- Ph.D. Thesis -

Dissertation zur Erlangung des Grades
Doktoringenieur (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Max-Planck-Institut für Informatik
Campus E1.4
66123 Saarbrücken
Germany

Eingereicht am / Submitted on 20. 9. 2010 in Saarbrücken.

Gernot Ziegler
MPI Informatik - D4
Campus E1.4
D-66123 Saarbrücken
Deutschland

E-mail: gziegler@mpi-inf.mpg.de

Betreuender Hochschullehrer - Supervisor

Prof. Dr. Christian Theobalt, Max-Planck-Institut für Informatik, Saarbrücken, Deutschland.

Prüfungsvorsitzender - Chairman of Examination Committee

Prof. Dr. Christoph Weidenbach, Max-Planck-Institut für Informatik, Saarbrücken, Deutschland.

Gutachter - Reviewers

Prof. Dr. Hans-Peter Seidel, Max-Planck-Institut für Informatik, Saarbrücken, Deutschland.
Prof. Dr-Ing. Marcus Magnor, TU Braunschweig, Braunschweig, Deutschland
Prof. Dr. Christian Theobalt, Max-Planck-Institut für Informatik, Saarbrücken, Deutschland.

Dekan - Dean

Prof. Dr. Holger Hermanns, Universität des Saarlandes, Saarbrücken, Deutschland.

Promovierter akademischer Mitarbeiter -
Academic Member of the Faculty having a Doctorate

Dr. Robert Herzog, Max-Planck-Institut für Informatik, Saarbrücken, Deutschland.

Datum des Kolloquiums - Date of Defense

6. Mai 2011 - May 6th, 2011

Abstract

Graphics hardware has in recent years become increasingly programmable, and its programming APIs use the stream processor model to expose massive parallelization to the programmer. Unfortunately, the inherent restrictions of the stream processor model, used by the GPU in order to maintain high performance, often pose a problem in porting CPU algorithms for both video and volume processing to graphics hardware. Serial data dependencies which accelerate CPU processing are counterproductive for the data-parallel GPU.

This thesis demonstrates new ways for tackling well-known problems of large scale video/volume analysis. In some instances, we enable processing on the restricted hardware model by re-introducing algorithms from early computer graphics research. On other occasions, we use newly discovered, hierarchical data structures to circumvent the random-access read/fixed write restriction that had previously kept sophisticated analysis algorithms from running solely on graphics hardware. For 3D processing, we apply known game graphics concepts such as mip-maps, projective texturing, and dependent texture lookups to show how video/volume processing can benefit algorithmically from being implemented in a graphics API.

The novel GPU data structures provide drastically increased processing speed, and lift processing heavy operations to real-time performance levels, paving the way for new and interactive vision/graphics applications.

Kurzfassung

Graphikhardware wurde in den letzen Jahren immer weiter programmierbar. Ihre APIs verwenden das Streamprozessor-Modell, um die massive Parallelisierung auch für den Programmierer verfügbar zu machen. Leider folgen aus dem strikten Streamprozessor-Modell, welches die GPU für ihre hohe Rechenleistung benötigt, auch Hindernisse in der Portierung von CPU-Algorithmen zur Video- und Volumenverarbeitung auf die GPU. Serielle Datenabhängigkeiten beschleunigen zwar CPU-Verarbeitung, sind aber für die daten-parallele GPU kontraproduktiv .

Diese Arbeit präsentiert neue Herangehensweisen für bekannte Probleme der Video- und Volumensverarbeitung. Teilweise wird die Verarbeitung mit Hilfe von modifizierten Algorithmen aus der frühen Computergraphik-Forschung an das beschränkte Hardwaremodell angepasst. Anderswo helfen neu entdeckte, hierarchische Datenstrukturen beim Umgang mit den Schreibzugriff-Restriktionen die lange die Portierung von komplexeren Bildanalyseverfahren verhindert hatten. In der 3D-Verarbeitung nutzen wir bekannte Konzepte aus der Computerspielegraphik wie Mipmaps, projektive Texturierung, oder verkettete Texturzugriffe, und zeigen auf welche Vorteile die Video- und Volumenverarbeitung aus hardwarebeschleunigter Graphik-API-Implementation ziehen kann.

Die präsentierten GPU-Datenstrukturen bieten drastisch schnellere Verarbeitung und heben rechenintensive Operationen auf Echtzeit-Niveau. Damit werden neue, interaktive Bildverarbeitungs- und Graphik-Anwendungen möglich.

Summary

This doctoral thesis details how well-known tasks in large scale video/volume analysis, previously too complex for graphics hardware, can now be implemented using our new algorithmic concepts and data structures.

The content begins with an introduction to graphics hardware and the graphics processing unit (GPU). We explain how graphics hardware works internally, how it differs from the CPU, and how its high performance is achieved. Afterwards we elaborate how the high performance design choices, namely parallelization and massive data processing, impose several restrictions on the programming model. Finally, we describe how general purpose computing on the GPU (GPGPU) emerged from the increasing flexibility of the graphics hardware, as well as which hardware/driver features are pivotal for GPGPU applications.

Chapter 3 first explains the use of graphics hardware in Free Viewpoint Video (FVV) processing, which requires a merger of computer graphics and image processing to generate novel viewpoints from multi-view video streams. We describe how graphics hardware can not only assist in FVV playback but also in the compression of multi view video.

In Chapter 4, we introduce how the camera/projector dualism can be implemented on graphics hardware. We demonstrate its usefulness in interactive camera calibration via proxy geometry, explain how depth projection can be easier represented in the OpenGL graphics API, and provide several approaches to fast reconstruction of novel views from depth video.

Chapter 5 is dedicated to hierarchical image processing, which fits graphics hardware very well due to its regular structure. We explain how depth reconstruction via a plane-sweep approach benefits from graphics hardware acceleration. Then, the chapter continues on feature clustering based on reduction, which we use in the real-time detection of connected feature regions in images. The last section addresses how histograms can be computed in a data-parallel fashion, and how camera lens compensation can be accelerated via rapid histogram analysis of feature contour angles.

Chapter 6 introduces a hierarchical data structure for GPU-based data compaction, the HistoPyramid, originally only a byproduct of fast histogram computation. We demonstrate how a traversal of HistoPyramids can circumvent many architectural restrictions of graphics hardware in the compaction of large data arrays to few selected input elements. The chapter continues with first applications: Real-time point cloud extraction and Streamline visualization for volume data, and concludes with future applications of the discovered data structure and algorithms.

Chapter 7 demonstrates how HistoPyramids even prove useful in data expansion, a concept for replication and modification of input elements while writing an output array. After explaining the necessary modifications, we present two key applications: Real-time iso-surface extraction from volume data via the marching cubes algorithm, and Light Wavefront Simulation via adaptive ODEs.

Chapter 8 introduces QuadPyramids, a descendant of HistoPyramids, which is used to cluster common regions in input data. With the help of these data structures, compact quadtrees can be extracted directly into graphics memory. We show first results from real-time video analysis, sketch relevant applications, and discuss how different methods of feature clustering affect the outcome.

Chapter 9 presents the QuadPyramid´s logical extension to 3D, the OcPyramid. Beyond the promotion of the original algorithm to three dimensions, we explain how algorithmic extensions enable pointer octree generation on graphics hardware, and can thus contribute to a renaissance for the use of octrees in real-time computer graphics.

Chapter 10 summarizes the contributions of the presented research and concludes the thesis with an outlook on the future development of GPU algorithms in the wider context of hardware evolution.

Contents

1. Overview.....15

1.1. Motivation and Overview.....15

1.2. Supervised student theses.....17

1.3. Publications.....17

2. The GPU - An Introduction.....18

2.1. The History of Graphics Hardware.....18

2.2. General Purpose Computing on the GPU.....21

2.2.1. Early Studies 21

2.2.2. Data-Parallel Architecture.....22

2.2.3. GPU vs. CPU.....24

2.2.4. Related Architectures.....25

2.2.5. GPU Programming Peculiarities.....26

2.2.5.1.Available Programming Models.....26

2.2.5.2.GPGPU Computation in OpenGL 26

2.2.5.3.Fixed Thread Write Location.....27

2.2.5.4.Mapping.....27

2.2.5.5.Reduction.....28

2.2.5.6.Bus Transfers.....28

2.2.5.7.Immediate Output Readback.....28

2.2.5.8.Random Access Writes.....28

2.3. Conclusion.....29

3. Free Viewpoint Video Compression.....30

3.1. Introduction.....30

3.2. Related work.....31

3.3. MVV Texture Generation.....32

3.4. MVV Texture Compression.....33

3.4.1. 2D-Diff / Wavelet Compression 33

3.4.2. 4D-SPIHT Wavelet compression.....35

3.5. Visibility Mask Reconstruction.....37

3.6. Conclusion....37
3.7. Future work....38
4. Color and Depth Reprojection....39
4.1. Interactive Multi-Camera Calibration....39
4.2. Depth Reprojection....42
4.2.1. Related work....43
4.2.2. Camera Representation....43
4.2.3. Data Storage: GeoCast....45
4.2.4. Basic Implementation....47
4.2.5. Depth Sweep Accumulation....49
4.2.6. CSG-assisted Novel Viewpoint Generation....49
4.2.7. Future Work....52
4.3. Conclusion....52
5. Hierarchical GPU Image processing....53
5.1. Depth Reconstruction via Plane Sweep....53
5.1.1. Related work....55
5.1.2. Mipmap-Based Plane Sweep....55
5.1.2.1.Results....58
5.1.3. Multi-View Depth Reconstruction....59
5.1.3.1.Result....61
5.1.3.2.Remaining issues....62
5.1.4. Coarse-To-Fine Plane Sweep Reconstruction....63
5.1.4.1.Results....66
5.1.5. Future work....67
5.2. Hierarchical Feature Clustering....68
5.2.1. Feature Detection....70
5.2.2. Reduction....70
5.2.3. Results....72
5.2.4. Conclusion and Future Work....74
5.3. Histogram Buildup via Reduction....75
5.3.1. Lens Compensation....76
5.4. Conclusion....79

6. GPU Data Compaction.....81

6.1. Basic Problem Setting.....81

6.2. Overview.....83

6.3. Related Work.....84

6.4. Classification.....85

6.5. HistoPyramid Buildup.....86

6.6. HistoPyramid Traversal.....86

6.7. Discussion.....88

6.7.1. Traversal Order.....88

6.7.2. Caching Considerations.....88

6.7.3. Complexity.....89

6.7.4. Variants.....89

6.7.5. Memory Requirements.....89

6.7.6. CPU-GPU bus bottlenecks.....89

6.8. Implemented Applications.....90

6.8.1. Point Cloud extraction.....91

6.8.2. Vector Field Contours.....94

6.8.2.1.Related work.....94

6.8.2.2.Overview.....95

6.8.3. Results.....97

6.9. Other Applications.....99

6.9.1. Sparse Matrix Creation.....99

6.9.2. Level-Set Techniques.....99

6.10. Conclusions and Future Work.....99

7. GPU Data Expansion.....101

7.1. Basic Concepts.....101

7.2. Light Wavefront Simulation.....104

7.2.1. System Overview.....105

7.2.2. Background.....105

7.2.3. Scene Representation.....106

7.2.4. Light Simulator.....107

7.2.4.1.Overview.....107

7.2.4.2.Wavefront Setup.....107

7.2.4.3.Wavefront Propagation....108
7.2.4.4.Wavefront Refinement....110
7.2.4.5.Wavefront Voxelization....113
7.2.5. View Renderer....113
7.2.6. Results....114
7.2.6.1.Light wavefront simulation....114
7.2.6.2.View renderer....115
7.2.7. Future Work....115
7.3. HistoPyramid Marching Cubes....116
7.3.1. Background....116
7.3.2. Previous and Related Work....118
7.3.3. HistoPyramids....119
7.3.3.1.Construction....119
7.3.3.2.Traversal....120
7.3.4. Marching Cubes....121
7.3.5. Marching Cubes in Stream and HistoPyramid-Processing....123
7.3.6. Implementation Details....124
7.3.7. CUDA Implementation....125
7.3.8. Results....126
7.3.9. Future Work....129
7.4. Conclusion....129
8. QuadPyramids....131
8.1. Overview....131
8.2. Related Work....132
8.3. Classifier....133
8.4. QuadPyramid Builder....134
8.5. QuadList Extraction....134
8.6. Algorithmic Variants....136
8.6.1. Feature Clustering Variant....136
8.6.2. Edge Thresholding Variant....137
8.7. Applications....138
8.7.1. Feature Clustering in Computer Vision....138
8.7.2. Rendering on Demand....138
8.7.3. QuadTrees in Data Compression....138

8.7.4. Mesh Reconstruction from Depth Maps 138
8.7.5. Light-weight Preclustering of Linearly Behaving Regions....139
8.8. Results....139
8.9. Conclusion....141
9. OcPyramids....142
9.1. Related Work....142
9.1.1. Tutorial Application....144
9.2. Overview....144
9.3. Classifier....145
9.4. Leaf Lists....146
9.4.1. Leaf OcPyramid Buildup....146
9.4.2. Leaf List Creation....147
9.5. Node Lists....148
9.5.1. Node/Leaf OcPyramid Buildup....148
9.5.1.1.In-Pyramid Allocation....149
9.5.2. Node List Creation....149
9.5.2.1.Co-traversal / Path Sharing for Node Children....150
9.5.2.2.Summary of Node Output....151
9.6. Implementation Remarks....152
9.6.1. Algorithmic Complexity....152
9.7. Octree Queries with Node/Leaf Lists....152
9.7.1. Point Query....153
9.7.2. Ray Intersection....153
9.8. Algorithmic Variants....154
9.8.1. Octree Capping....154
9.8.2. Node OcPyramids....154
9.8.3. Base Level Upshift....155
9.8.4. Feature Clustering....155
9.8.5. Tiling/Partial Processing....155
9.8.6. Hi-Res Processing (Sub-OcPyramids, Hierarchical Processing)....155
9.8.7. Large Volume Extension 156
9.9. Results....156
9.9.1. Voxelization 156
9.9.2. Solid vs. Empty Space Clustering....157

9.9.3. Performance....158
9.9.4. Dynamic Octree Generation....160
9.10. Applications....161
9.10.1. Ocxelization....161
9.10.2. Volume compression....161
9.10.3. Volume analysis....161
9.11. Conclusions and Outlook....162
10. Discussion and Conclusions....163
10.1. GPU Techniques for Image Processing....163
10.2. General Data Processing....164
10.3. Innovation from Programming Model Restrictions....164
10.4. Relevance of Programming Model....165
10.5. OpenGL, CUDA and OpenCL....166
10.6. Future....166
11. Bibliography....167

Preface

"First we thought the PC was a calculator. Then we found out how to turn numbers into letters with ASCII — and we thought it was a typewriter. Then we discovered graphics, and we thought it was a television." — Douglas Adams (1952—2001).

Graphics. Computer graphics. Ever since science-fiction embraced computer-generated images, I have not been able to let my eyes off of it. Tron, the first CG-based movie from 1982, was a revelation – Ars Electronica Festival´s showing of "Luxor Junior" just as well. From these days on, I have associated advanced computer graphics with science-fiction – and I wanted to see all of it come real.

But many times after my initial enthusiasm, when I saw something in the real world that actually *did* look like in the movies, I quickly noticed: Oh. This result actually takes *hours* to calculate. That was not what I wanted. I needed things to be tangible. Interactive. Just like in the movies...

When graphics hardware surpassed the computing power of the main CPU, I saw my chance had come: I could help transforming all of these cool, but slow, algorithms into what they should be: *Science-fiction come true.*

This document summarizes my research contributions over the last four years.

The thesis is roughly chronological, with occasional changes in the timeline to assist the underlying reasoning and to serialize parallel research efforts[1]. Since the main topic of this thesis is the usage of graphics hardware in computer vision, 3D-graphics and general purpose computing, I have toned down other aspects of the included publications, and let the reader explore the corresponding articles for details not related to graphics hardware.

Considerable effort has been put into improving readability over the original research publications: Some terminology has been standardized (e.g. the term *Classifier* replaces *Discriminator*, even for HistoPyramids), and diagrams have been updated to convey a more consistent visual language to the reader. It should be noted that major parts of this thesis are centered around NVIDIA-hardware, the hardware platform used in algorithm development. However, the algorithms themselves are all implemented in OpenGL and thus portable.

Naturally, insight and experience have grown after this research has been published. Additional insights have therefore been worked into the original material, and some paragraphs have been totally re-written to reflect a more recent view of the state-of-art.

Unfortunately, plain paper is not the right medium for interactive computer graphics (at least not until the introduction of electronic paper...). To counter this, references to video material are provided throughout this work, in the hope that moving imagery will improve understanding. Also, since software implementations can never be described into their last details, most of it has been published online, and we encourage the reader to conduct more experiments on his or her own.

After all, this thesis is a starting point for an exciting journey into *visual computing, in a sea of processors*. Welcome aboard.

London, March 20th, 2010

Gernot Ziegler

1 no pun intended...

1. Overview

1.1. Motivation and Overview

Graphics hardware has become increasingly flexible, and its massive computing power has made it more and more attractive for uses outside computer game graphics. While some fields like data visualization (as used e.g. in medical imaging) slowly transitioned from CPU to GPU processing and increasingly embraced new graphics hardware features, research in computer vision was often considered “too hard to port” to GPUs. Computer vision approaches were thus often completely implemented on the CPU, leaving only simple display tasks for graphics hardware. Later, hybrid attempts were made by mixing CPU and GPU processing, but often suffered from the bottleneck of databus transfers. These early findings already indicated that also the remaining CPU tasks must be ported so that GPU performance can be utilized properly.

Therefore, this thesis aims at *implementing algorithms completely on graphics hardware*, at least in as far as current GPU functionality allows.

Chapter 2 begins with an introduction to graphics hardware and the graphics processing unit (GPU). We explain how graphics hardware works internally, how it differs from the CPU, and how its high performance is achieved. Afterwards we elaborate how the high performance design choices, namely parallelization and massive data processing, impose several restrictions on the programming model. Finally, we describe how general purpose computing on the GPU (GPGPU) emerged from the increasing flexibility of the graphics hardware, as well as which hardware/driver features are pivotal for GPGPU applications.

In the first part of this doctoral thesis, we will stretch the concepts of graphics APIs and computer vision's mathematical frameworks into the realm of each other, and utilize graphics hardware for the acceleration of applications on the boundary between of computer vision and graphics.

Chapter 3 first explains the use of graphics hardware in Free Viewpoint Video (FVV) processing, which requires a merger of computer graphics and image processing to generate novel viewpoints from multi-view video streams. We describe how graphics hardware can not only assist in FVV playback, but also accelerate 3D reconstruction and actual multi-video stream compression (published in [ZLA*04] on 2D-Diff compression, and [ZLM*04] on a 4D-SPIHT compression).

In Chapter 4, we introduce the camera/projector dualism's mapping to graphics hardware. We demonstrate its usefulness in fast reconstruction of 3D views from depth video data from multiple camera views (published in an article on the data format GeoCast [ZHMS05]), and further its use in research on interactive camera calibration via proxy geometry.

Chapter 5 is dedicated to hierarchical image processing, which fits graphics hardware very well due to its regular structure. We explain how several algorithms that use depth reconstruction via a plane-sweep method benefit from graphics hardware acceleration (resulted in the diploma thesis by Heidenreich [Hei07]). After this, the chapter continues on hierarchical feature clustering, used in the detection of connected feature regions in images (resulted in the bachelor thesis by Schilz [Sch07] and Arnold [Arn07]). The last addresses how histograms can be built in a data-parallel fashion, and how camera image rectification can be accelerated via rapid classification of feature contour angles.

The second part of this doctoral thesis explains how well-known tasks from large scale volume analysis, previously too complex for graphics hardware and handled by the CPU, can now be implemented with the help of new algorithms and data structures.

Chapter 6 introduces a hierarchical data structure for GPU-based data compaction, the HistoPyramid. Originally only a byproduct of fast histogram computation of image features, we can demonstrate how a traversal of a HistoPyramid can circumvent many architectural restrictions of graphics hardware in the compaction of data arrays to selected elements. The chapter continues with first applications: Real-time point cloud extraction (published in [ZTTS06]) and Streamline visualization for volume data (published in [ATR*08]), and concludes with future applications of the discovered data structure and algorithms.

Chapter 7 demonstrates how HistoPyramids prove useful even in data expansion, a concept for replication and modification of input while writing to an output stream. After explaining the necessary modifications, we present two key applications: Real-time iso-surface extraction from volume data via the marching cubes algorithm (published in [DZTS08]), and Light Wavefront Simulation through Refractice index volumes via adaptive ODEs (published in [IZT*07]).

Chapter 8 introduces QuadPyramids, a descendant of HistoPyramids, which is used to cluster common regions in input data. With the help of these data structures, compact quadtrees can be extracted directly into graphics memory. We show results from real-time video analysis (published in [ZDTS07]), sketch relevant applications, and discuss how different methods of feature clustering affect the outcome.

Chapter 9 presents the QuadPyramid´s logical extension to 3D, the OcPyramid. Beyond the promotion of the original algorithm to three dimensions, we explain how algorithmic extensions enable pointer octree generation on graphics hardware, and can thus contribute to a renaissance for the use of octrees in real-time volume graphics.

Chapter 10 summarizes the contributions of the presented research and concludes the thesis with an outlook on the future development of GPU algorithms in the wider context of hardware evolution.

1.2. Supervised student theses

Related Chapter	Reference	Student thesis
5	[Arn07]	M. Arnold. GPU-based 3D light marker tracking. Bachelor Thesis, Universität des Saarlandes, 2007.
5	[Sch07]	M. Schilz. High-Resolution multiple points tracking on the GPU. Bachelor Thesis, Universität des Saarlandes, 2007.
5	[Hei07]	L. Heidenreich. Real-Time Hierarchical Stereo Matching on Graphics Hardware. Diplomarbeit, Universität des Saarlandes, 2007.

1.3. Publications

Related Chapter	Reference	Publication venue
3	[ZLA*04]	G. Ziegler, H. Lensch, N. Ahmed, M. Magnor, H.-P. Seidel. Multi-Video Compression in Texture Space. In: 11th IEEE International Conference on Image Processing (ICIP 2004), Singapore, pp. 2467-2470, 2004.
3	[ZLM*04]	G. Ziegler, H. Lensch, M. Magnor, H.-P. Seidel. Multi-Video Compression in Texture Space using 4D SPIHT. 6th IEEE Workshop on Multimedia Signal Processing, Siena, Italy, pp. 39-42, 2004.
3	[TZMS04]	C. Theobalt, G. Ziegler, M. Magnor, H.-P. Seidel. Model-Based Free Viewpoint Video: Acquisition, Rendering, and Encoding. Proceedings of Picture Coding Symposium (PCS 2004), San Francisco, USA, December 2004.
3	[TAZS07]	C. Theobalt, N. Ahmed, G. Ziegler, H.-P. Seidel. High-Quality Reconstruction from Multiview Video Streams. IEEE Signal Processing Magazine, pp. 45-57, November 2007.
4	[ZHMS05]	G. Ziegler, L. Heidenreich, M. Magnor, and H.-P. Seidel. GeoCast: Unifying depth video with camera meta-data. In 2nd Workshop on Immersive Communication and Broadcast Systems, Berlin, Germany, October 2005.
6	[ZTTS06]	G. Ziegler, A. Tevs, C. Theobalt, H.-P. Seidel. GPU Point List Generation through Histogram Pyramids. Tech. Rep. MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.
6	[ATR*08]	T. Annen, H. Theisel, C. Rössl, G. Ziegler, H.-P. Seidel. Vector Field Contours. Proceedings of Graphics Interface, Volume 322, pp. 97 - 105, 2008.
7	[IZT*07]	I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt, M. Magnor, H.-P. Seidel. Eikonal Rendering: Efficient Light Transport in Refractive Objects, ACM Transactions on Graphics (Siggraph'07), 2007. http://www.mpi-inf.mpg.de/resources/EikonalRendering/index.html
7	[DZTS08]	C. Dyken, G. Ziegler, C. Theobalt, H.-P. Seidel. High-Speed Marching Cubes using Histogram Pyramids. Computer Graphics Forum, Issue 27, number 8, pp. 2028-2039, 2008.
8	[ZDTS07]	G. Ziegler, R. Dimitrov, C. Theobalt, H-P. Seidel. Real-time Quadtree Analysis using HistoPyramids. Proc. of SPIE Electronic Imaging conference in San Jose, USA, Jan 2007.

2. The GPU - An Introduction

2.1. The History of Graphics Hardware

Ever since the era of home computers began, specialized video hardware has been used to handle graphics-specific data operations [Var08a]. The first important task of the video chip was to produce screen output from a text buffer stored in memory. Later, the increased chip area and memory sizes allowed to display graphics directly. This was particularly useful for computer games: Figure 2.1 shows example output from the VIC-II, the graphics chip used in the home computer Commodore 64. Three quarters of its chip area were dedicated to handle *sprites*, small graphical objects that moved independently of the content in the graphics framebuffer [Var08b].

Figure 2.1: Hardware accelerated game graphics on a home computer (VIC-II, Commodore 64) .

On personal computers, a similar development took place, albeit with some years of delay. Because of their business purpose, graphics hardware of the PC originally lacked advanced capabilities. Later, the framebuffer display hardware was complemented with hardware-accelerated 2D operations to speed up the windowing operations of a 2D graphical interface, as shown in Figure 2.2. Typical operations were fast region copy, 2D rescaling, or video-overlay, sometimes even hardware video decoding [Var09a].

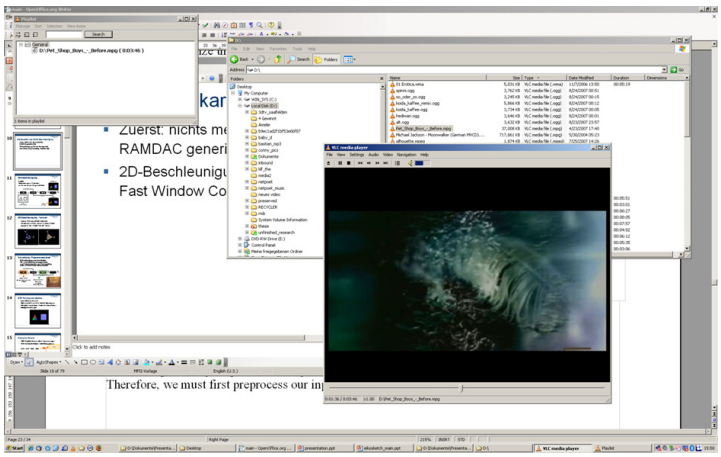


Figure 2.2: Graphical user interface with video acceleration, here: Windows XP.

Around 1999, 3D acceleration was integrated into graphics hardware. Its main purpose was to assist games in rasterizing *triangles*, which often form the basis of 3D models in computer graphics. A

schematic overview of an accelerated 3D graphics pipeline can be seen in Figure 2.3. The underlying graphics pipeline originated from chip architectures designed for workstation and supercomputer graphics, such as Silicon Graphics´ O2 and the Infinite Reality board [Kil97].

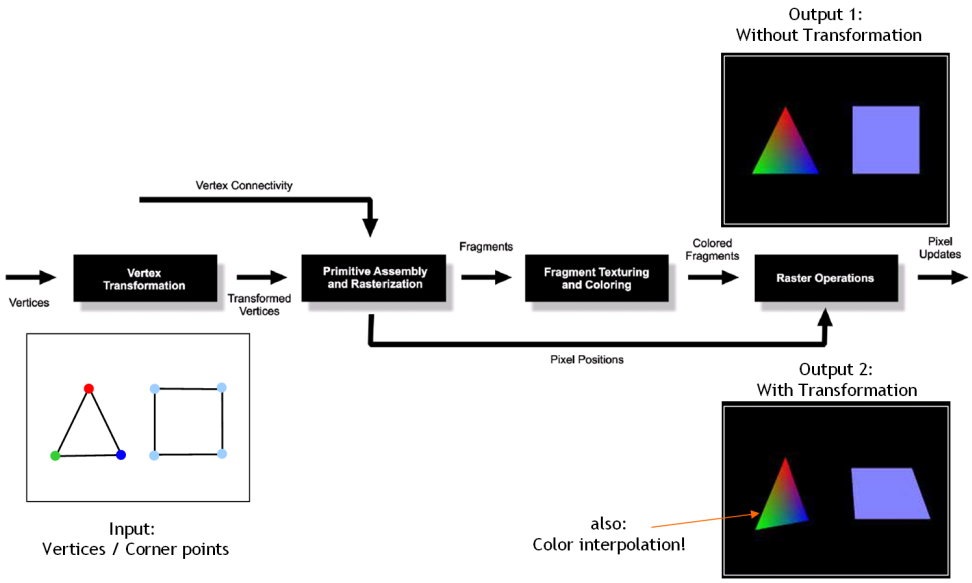


Figure 2.3: Basic 3D rasterization.
3D vertices of a triangle are transformed as the application pleases, and then converted into color pixels that are written into the 2D framebuffer.

To improve realism beyond the rasterization of simple RGB colored triangles, *texturing* was introduced. It allows the colouring of 3D meshes with 2D arrays of RGBA color data, in a way that resembles "wrapping" onto 3D surfaces, a principle pioneered by Catmull [Cat74]. This is achieved via per-vertex *texture coordinates*, which are interpolated inside the triangle. During triangle rasterization, an indirect lookup into the texture's 2D color array yields the actual surface color, as shown in Figure 2.4.

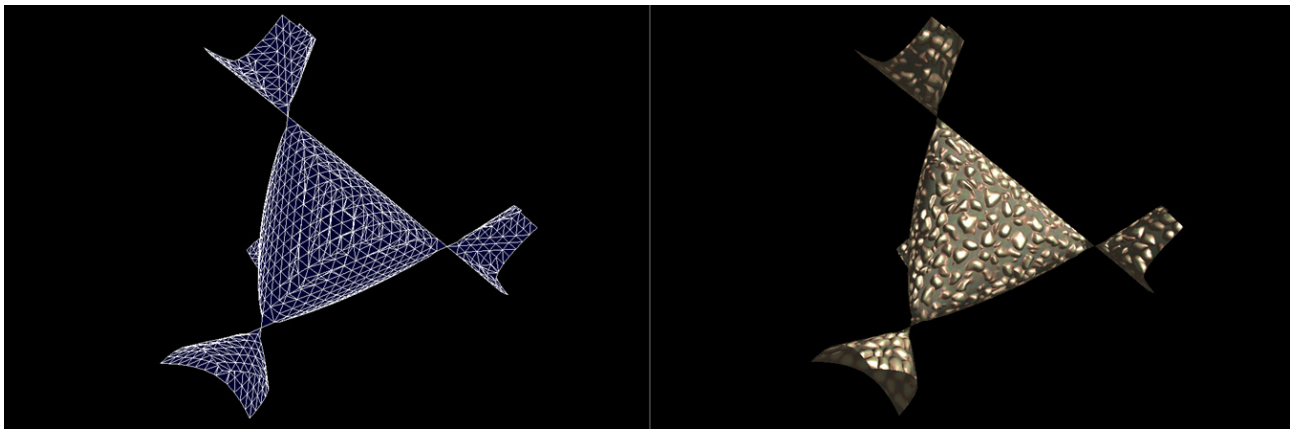


Figure 2.4: Texture application. A 3D mesh on the right is supplied with texture coordinates and a 2D texture map, which result in a textured 3D object after rasterization.

In early hardware, 3D transformation of vertices and lighting calculations had to be done by the CPU. As floating point units started to fit onto the graphics chip die, T&L-units (Transformation and Lighting) implemented these calculations in hardware, such as the ones used in NVIDIA's

GeForce 256 [NVI99]. In the following years, more and more 3D graphics functionality was taken over by the graphics processing unit (GPU). Figure 2.5 provides a rough timeline.

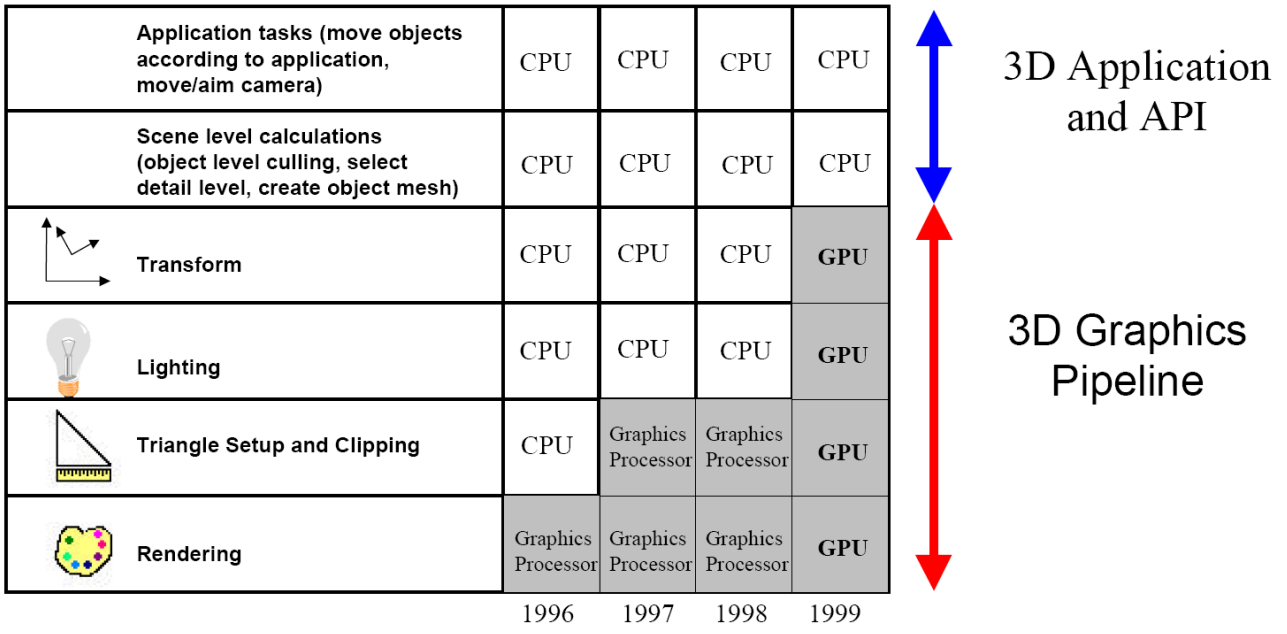


Figure 2.5: Early GPU evolution. The CPU was more and more relieved from 3D graphics computations.

But most of the graphics generation was hard-wired, with few options, in a so called *fixed function pipeline*. As a response to increasing demands for graphics generation flexibility from computer game developers, insights from the Renderman system [Ups90] were used to make the GPU *programmable*. As a first step, vertex transformation became programmable via a special assembly-like language [LKM01]. Vertex transformation took place in vertex shader programs. Later, also the fragment/pixel stage became programmable, introducing fragment shaders. See Figure 2.6 for a pipeline diagram from 2001.

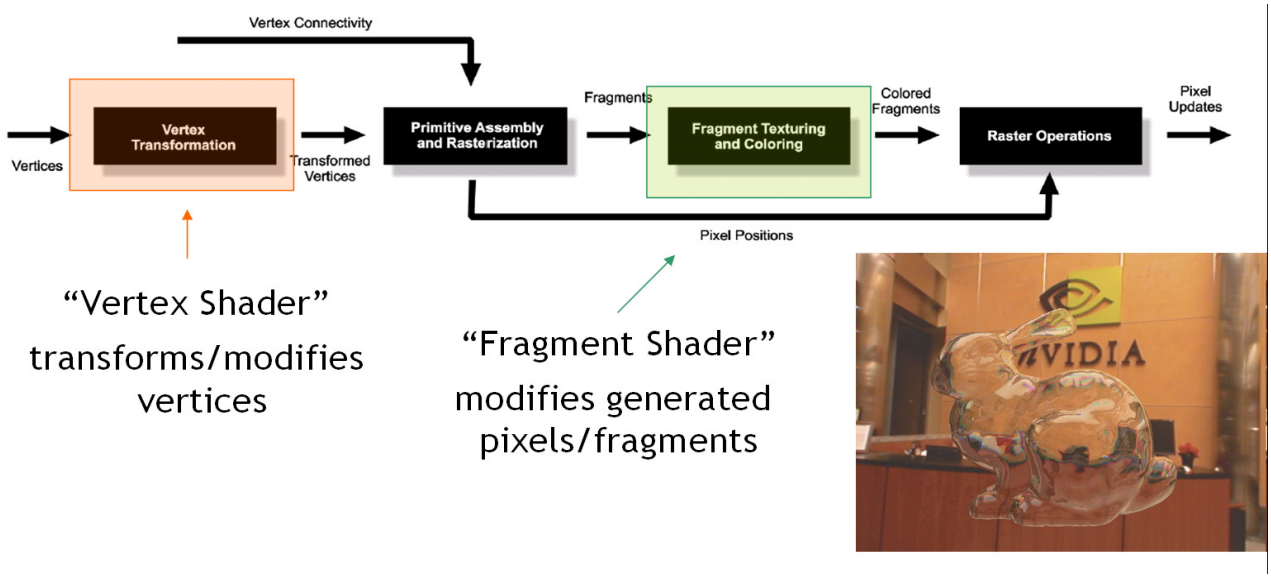


Figure 2.6: Programmable graphics hardware in 2001.

2.2. General Purpose Computing on the GPU

Around the time when the GPU had become programmable, its increasing computational performance had also captured interest in other areas of computer science.

2.2.1. Early Studies

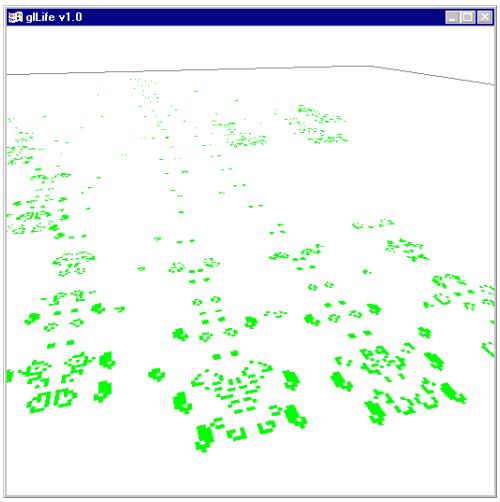


Figure 2.7: Game of Life as OpenGL program [Gre01].

First attempts to use the GPU for non-rendering and image-processing tasks were published. Two prominent examples are the z-buffer based Voronoi diagrams of Hoff et al. [HCK*99], and Green's implementation of the automata simulation “Game of Life” for OpenGL [Gre01], see Figure 2.7. In 2004, Harris coined the term GPGPU for ”General Purpose Computation on the GPU” [Har04].

The first studies showed that hardware-accelerated graphics APIs could accelerate some image processing tasks. Still, two key components were missing for good performance in general-purpose calculations: *Render-to-texture*, for efficient computational feedback in graphics memory, and *floating point output*, which made it easier to maximize numerical precision. In OpenGL, render-to-texture is implemented via the framebuffer objects (FBO) extension.

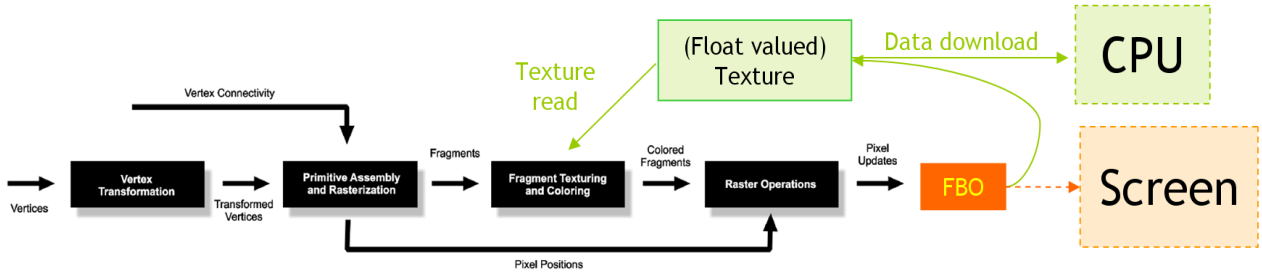


Figure 2.8: Render-to-texture in OpenGL.
Green Path: Graphics output can be redirected via framebuffer objects (FBO) into textures, and thus enable floating point computation throughout. Later, results can be downloaded to the CPU.

Render-to-texture capable graphics hardware can re-direct its graphics output into textures, instead of displaying them on the screen. In effect, such graphics hardware can write computational output into 2D data arrays. This output can be read back in the next computational pass, which allows keeping intermediate results in graphics memory, see Figure 2.8. As the framebuffer for the screen

could not be represented as floating point values, render-to-texture was even required for advanced calculations that required intermediate results stored as floating point.

With this feedback path, results from GPU computation did not have to be downloaded to the CPU after every computational pass anymore, which had often impaired speed gains from GPU-based computation before. Render-to-texture thus made the GPU a much more usable co-processor for data-parallel operations.

The second key feature towards GPGPU was the presence of floating point precision in all parts of the calculation pipeline, including graphics output. This was important because up to then, internal shader calculations were programmable, but of unclear precision – and shader output restricted to 8 bit RGBA color. In 2002, ATI introduced 24-bit based floating point calculation with the Radeon 9700. NVIDIA followed soon after with 32-bit floating point handling in the GeForce FX (NV40) architecture.

In the meanwhile, high-level shader languages emerged which strongly resembled the C language. While the initial Cg (C for Graphics) was to be used with both OpenGL and DirectX [MGAK03], both API:s received their own language later on: HLSL for DirectX, and GLSL for OpenGL [Kes06]. Buck et al. also created a tool to measure graphics hardware performance in shader execution [BFH04].

2.2.2. Data-Parallel Architecture

Graphics rendering tasks involve the fast processing of a large number of data elements, and high throughput had thus always been important. Luckily, many vertices and pixels can often be processed in a similar fashion. By applying the same instruction code to a fixed number of data elements (e.g. 4), SIMD (single instruction, multiple data) provided a first way to parallelize data processing cheaply, . The SSE extensions expose such a SIMD programming model on desktop CPUs.

But the sheer amount of graphics primitives required even more parallelism, as the same vertex shader programs would execute for thousands of triangles and pixels, and even more. Therefore, the GPU programming started to employ *data-parallelism*, a concept which assumes that the same program code (the *kernel*) is applied to many data-elements simultaneously, while not defining the number of present processors explicitly. This came with certain restrictions, but enabled the GPU to scale the number of available processors over time.

Up to the GeForce 7, the GPU architecture was mostly structured after the graphics pipeline. While the chip was already programmable, it contained separate processing units for vertex and fragment manipulation, each specialized to their task and a specific assembly-level shader language. While this enabled task-level parallelism, i.e. the parallel processing of vertices and fragments, and both vertex and fragment units could handle multiple elements each, it is easy to conceive graphics loads where the hard-wired ratio between vertex and fragment units caused bottlenecks. The restrictions for vertices and fragment processing differed, which made the programming model unusually complicated for general-purpose computations.

As a consequence, NVIDIA's G80 architecture reflects the increasing utilization of the GPU for non-rendering applications in two design choices: First, a unified shader model for vertex and fragment processing, partially also due to requirements in the DirectX 10 specification [Thi06]. This unified shader model is able to execute all types of shaders, including the new geometry shader that is responsible for GPU generated geometry. Second, sophisticated thread management now takes care of the load-balancing between geometry, vertex and fragment processing, and removes graphics pipeline bottlenecks for unusally vertex- or fragment-heavy applications. Massive on-chip

multi-threading for both memory transfers and ongoing computation ensures maximal bandwidth and ALU utilization for data-parallel programming. See Figure 2.9 for an overview.

Additionally, the vector-processing capable units of previous generation have been replaced by scalar units (128 in the GeForce 8800 GTX, 240 in the GeForce GTX 280). This is caused by the observation that in many non-graphics applications, vector processing is rarely expressed in program code, or would not follow the four element-vectorisation that was the mode of operation in previous graphics hardware generations.

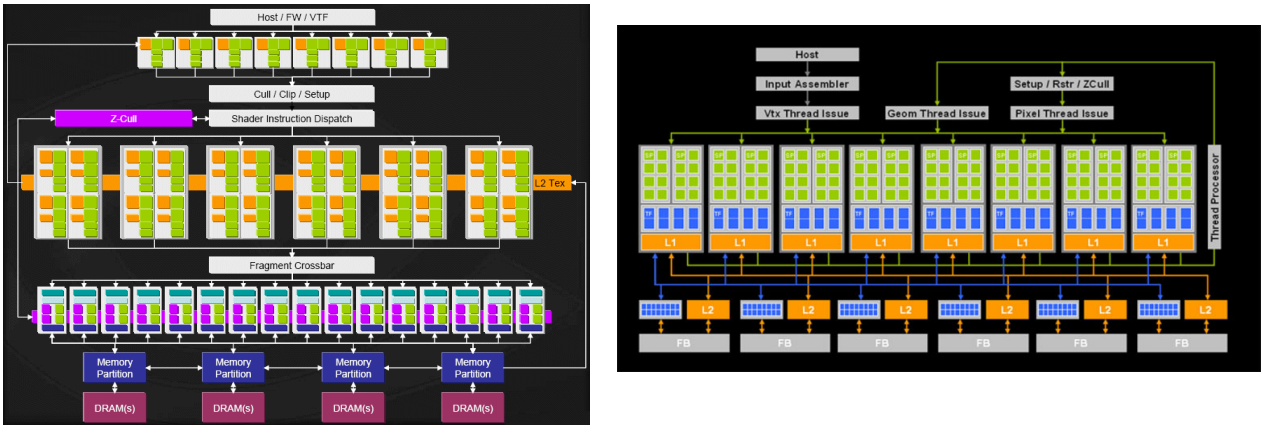


Figure 2.9: Comparison of task-specific and task-agnostic GPU architectures.
Left: NVIDIA GeForce 7 (aka NV40), with separate vertex and fragment processors.
Right: NVIDIA G80 architecture (aka NV50). Source: NVIDIA.

In data-parallel processing, also known as stream processing, the same operations are simultaneously applied to as many data elements as computational cores are available. As an example, GeForce 8 graphics hardware sports 128 scalar processing units (SPs), grouped into 16 multiprocessors of 8 SPs each, as shown in Figure 2.9. While the numbers of cores already ranges in the dozens, the data-parallel programming paradigm can utilize even more cores, as long as the data element workload exceeds the number of cores.

The current GT200 generation of NVIDIA hardware slightly improves on above architecture with double-precision computation (albeit at a ratio of one DP unit per 8 SP units), and a total number of 30 multiprocessors with 8 SPs each, resulting in 240 scalar processors.

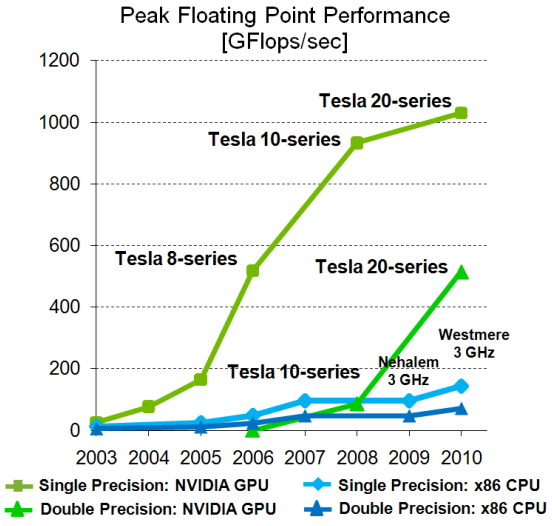


Figure 2.10: The rapid evolution of GPU math performance (peak GFLOPS, single-precision).

With these changes, current graphics hardware is now able to process millions of vertices and tens of millions of pixels per second. The raw computational performance at a low price makes them a prime candidate for massively parallel data-parallel computation, and far exceeds desktop CPU performance, as Figure 2.10 shows.

2.2.3. GPU vs. CPU

Of course, since the semiconductor factoring process is identical for CPU and GPU, drastic performance differences for general purpose computation were only possible via on-chip parallelization, and corresponding *trade-offs* in the programming model.

We have already mentioned the SIMD concept, where a single instruction is applied to multiple data elements, typically four or eight of them. But this is too little parallelization for graphics purposes. To increase the parallelism, one could imagine a many-component SIMD, a *wide-vector* unit, so that more vertices and pixels can be processed at the same time. But often, there are small *divergences* in the processing of individual elements, often expressed by if() clauses inside vertex and fragment programs. SIMD thus doesn't work efficiently for a large number of input elements.

To handle these divergences, the CUDA programming model introduces the term SIMT (Single Instruction, multiple *threads*), which describes the notion that data elements *usually* are processed by similar instructions in multiple threads of execution, but that these threads also must be capable of divergence. To reward coherent processing and keep processing efficient, divergence comes with a *penalty*: First, cores are batched into multiprocessors, and share a common instruction stream. For the most of the program these cores operate in unison. If their common program diverges for the processed data, all cores of a multiprocessor would execute *both* branches of an if() clause, and each core simply *ignores* the outcome of the "wrong" branch. This means that conditional execution, while possible, is penalized by slower processing. Note that each multiprocessor has its own instruction counters, and can thus diverge completely, i.e. divergence of larger groups of elements is handled well. There are 30 multiprocessors in a high-end GT200 chip, with each multiprocessor supervising 8 cores each.

The second issue is memory latency. On the CPU, this is solved through large caches for both reading and writing, in the hope that external DRAM accesses are avoided as much as possible. The GPU, on the other side, hides this memory latency through multi-threading: It keeps thousands of threads resident on-chip, and can quickly switch between them. Whenever a memory access stalls a batch of threads, other ones are activated to occupy the cores with computation. Note that these threads are very light-weight, and can thus be swapped within cycles.

In short, CPU and GPU are designed for two different purposes: For the CPU, source code can be serial and complex in its data dependencies. Since the computational dependencies in serial code require low latency for efficient execution, instruction/data caches are present, but also branch prediction and speculative execution which use redundant computation to reduce latency as much as possible. All these serial acceleration techniques require transistor capacity that is not used for the actual data processing. This is why a CPU is better at processing small amounts of data with complex dependencies.

On the GPU, huge amounts of data shall be processed with the same source code. Here, the overall data throughput is most important, and individual thread latencies more irrelevant. Therefore, emphasis is put on maximizing computational throughput: As the instruction flow is largely the same for all data elements, the cores can share a common instruction pointer. At the same time, memory latencies can be hidden by scheduling other threads for computation. It is thus the programmer's responsibility to ensure as much data-parallelism in the application as possible.

2.2.4. Related Architectures

PRAM (Parallel random access machine) is a hypothetical processor architecture for the research of (data-)parallel algorithms [KKT01]. It explicitly defines resolvement for write conflicts, i.e. the outcome of several processors writing to the same output location. The PRAM machine is used to describe the computational complexity of parallel algorithms that can be executed by an arbitrary number of processors. PRAMs are therefore highly suitable for exploring the theoretical complexity of data-parallel algorithms. One of the largest realized PRAM machines is the SB-PRAM [BBF*97], which currently sports 4 GB of global memory and has 64 physical (2048 virtual) processors.

One of the actual hardware ancestors to programmable graphics hardware is the Imagine processor [KDR*02], a highly parallel stream processor architecture. It strongly separates the read-only input and the output of data elements, and requires that its *kernels* (the program code that is executed for all data elements) have fixed output locations. This is more restrictive than the previously described PRAM, but completely evades the PRAM´s write-conflict problems, since a readback of other cores' output is first allowed after a kernel execution has finished. Stream processors have the advantage that their data communication with external DRAM is often highly predictable. They can therefore be pipelined by dedicated hardware, a property that is also exploited by the GPU's programmable vertex and pixel processing. The Imagine architecture later inspired the Merrimac processor, which explored the use of stream processors in scientific computing in more depth [DHE*03].

Another popular architecture for stream processing was the Cell Broadband Engine, consisting of a CPU and several coprocessor engines connected by an internal ring bus [Gsch06]. While its eight coprocessor engines (SPEs) had a higher amount of local memory available, they lacked a direct connection to external memory, and thus had to be carefully maintained by the operating system on the Cell's CPU to achieve the intended performance.

The scalar, data-parallel nature of the NVIDIA G80 architecture and beyond was influenced by the MasPar architecture [Var09b]. The MP PE-2 binds together 32 ALUs with a single instruction decoder, thus enabling effective throughput of 1 instruction per cycle. Each of its ALUs had direct access to external memory, allowing them to fetch data from different locations even though they all executed from the same instruction stream - a feature which distinguished the MP PE-2 from vector processors, and simplified data-parallel processing.

For the reason of scalability, GPU algorithms must be agnostic about the number of available processors, and more research is required to explain the application domain of such algorithms. Beyond the theoretical PRAM machine, chip architecture research has recently put increased focus on *many-core architectures*, which describes all architectures with a massive amount of processor cores [ABC*06]. In this context, an interesting manycore-architecture is the *Larrabee* processor from CPU manufacturer Intel [SCS*08]. It is intended as a collection of x86 cores with attached vector units, capable of both graphics and general purpose workloads.

The work in this thesis uses graphics hardware with NVIDIA GPU architecture. Among the competitors, one of them is usually very close in graphics performance: ATI hardware by AMD. AMD has introduced unified shaders as well, and thus computes vertex, fragment and geometry workloads on the same processors. But in contrast to NVIDIA, AMD remained true to the vector processing model, and now employs a 5-way vectorization in its processors, and improves constantly on its compilers to exploit these vector units in graphics workloads.

2.2.5. GPU Programming Peculiarities

Due to the radical changes in the programming model, new programming techniques are required, even for standard programming tasks. In many cases, the data-parallel design of graphics hardware requires considerable rethinking of data structures and algorithms. Interestingly, techniques from earlier processor architectures, e.g. vector machines, have now become the method of choice again. Below follows a short summary of the main considerations for GPU software development, and refer to related literature for more elaboration [OLG05].

2.2.5.1. Available Programming Models

As a consequence of the GPU's original use in 3D graphics rendering, GPUs have traditionally been programmed in graphics APIs for desktop gaming and visualization applications, the Windows-specific DirectX and the more platform-independent OpenGL. Both APIs were originally intended for graphics pipelines of fixed functionality, even though CPU-based implementations such as Mesa existed. Nowadays, both APIs feature their own programming language for vertex and pixel processing, **HLSL** (High Level Shading Language) for DirectX and **GLSL** (GL Shading Language), respectively, for OpenGL.

In this thesis, we focus on the OpenGL API [OGL21] and its shading language GLSL [Kes06] - but all presented algorithms could be implemented in DirectX and its language HLSL as well.

More recently, non-graphics GPU programming models such as C for CUDA have been introduced for general purpose computation on NVIDIA graphics hardware [BP04]. Development in C for CUDA has two modes of operation: CUDA runtime API allows mixing of CPU and GPU source code in one file, while the more classic driver API still separates CPU and GPU programs. Both provide an API for memory management, error handling and device query. C for CUDA makes use of certain features that only the G80 architecture and beyond are capable of, such as writes to arbitrary memory locations and explicit thread control, which makes the API better suited for aggressive optimization, but also excludes other OpenGL-compatible hardware as application platforms. C for CUDA also removes a certain portion of *graphics functionality*, making it harder to mix graphics primitives with computation. A recent addition to the family of general purpose computing languages is OpenCL, which is standardized on the initiative of Apple through the Khronos consortium. It is platform-indepent, separates CPU and GPU code, and provides an API similar to both the CUDA driver API, and the graphics API OpenGL.

On AMD hardware, the assembly-like GPU language CAL (Compute Abstraction Layer) was long the only method of choice for general purpose computing, occasionally augmented by the stream processing library Brook+. While AMD does not provide C for CUDA, it has recently introduced support for the platform-independent computing language OpenCL, just like NVIDIA has.

As this thesis originally had a graphics-related purpose for its general purpose computations, we mostly keep to implementing GPGPU via the OpenGL graphics API. However, we show for same cases how our algorithms can be easily ported to CUDA, provided the graphics interoperability is not of major importance in a given context.

The common denominator is that while both OpenGL and C for CUDA enforce data-parallel programming in order to achieve GPU performance, which forces an algorithmic redesign of many image processing algorithms, the actual API used in the implementation doesn't matter that much.

2.2.5.2. GPGPU Computation in OpenGL

As OpenGL is a graphics API, general purpose computations have to be “disguised” as graphics operations. In OpenGL 2.0, there are two programmable units, the vertex shader processors and the

fragment shader processors. The name “fragment shader processor” for pixel processors stems from the fact that their output is *fragments*, not pixels. Fragments first have to pass various fixed function tests before they are written to the framebuffer. The most prominent of these tests is the depth test, which determines if the current pixel in the framebuffer is closer to the user's viewpoint than the pixel candidate stored in the fragment.

In previous graphics hardware, e.g. NVIDIA's NV40 architecture, fragment shader processors were more plentiful than vertex shader processors. Additionally, only fragment shaders had access to textures. This is why GPGPU computation traditionally happens in the fragment shader. However, we will show in later chapters how new vertex shader capabilities, such as vertex texture access, can prove beneficial for GPGPU computation.

To trigger GPGPU computation, the application first chooses a texture output target via the OpenGL framebuffer object (FBO) extension. Then, it prepares the shader that shall be executed, and binds all input textures and the output texture to it. Now, the application issues a drawing command for a quad that covers the whole area of the texture. This way, the shader program is now executed in as many threads as there are fragments to rasterize. When the quad has completed rasterization, all threads of shader execution have finished, and the result can be found in the output texture.

Note that shader programs, or: shaders, are GPGPU terminology for the data-parallel program code that is executed by the hardware. In CUDA and later GPGPU algorithms, the same data-parallel program code is called a *kernel*. The terms shader (program) and kernel are used synonymously in this thesis.

2.2.5.3. Fixed Thread Write Location

The GPU´s inherently parallel nature can only be utilized if the output of several independent computation units is combined. As the theoretical PRAM machines demonstrate, this can lead to all kinds of data hazards during data-parallel program execution. OpenGL therefore introduces a certain restriction to avoid memory inconsistency: Each thread is *only allowed to write to a certain output position, and may not read back other threads´ output during this pass.* This prevents unpredictable data dependencies and racing conditions among threads, and thus allows arbitrary suspension of threads that wait for memory reads.

Put differently, in the OpenGL shader programming model, it is thus *not allowed to exchange data* between shader threads in a given execution pass. However, the output from previous computation passes can be read randomly from corresponding textures.

2.2.5.4. Mapping

GPUs have originally been designed for image generation, and their I/O is mostly handling 2D data arrays, such as textures and the framebuffer. While other data array types such as 1D and 3D textures exist, they are either restricted in features or inefficient in usage – as an example, on NV40 hardware, 1D textures support only 4096 entries, and 3D textures have a maximum resolution of 256^3. Complete render-to-texture is only possible for 2D textures, while 3D textures can only be accessed slice-by-slice. Even the texture cache structure is geared towards 2D arrays, and works suboptimally in 1D and 3D texture handling. For all these reasons, it is important to use mapping techniques for conversion of other dimensionality to 2D and back. These seemingly expensive mapping calculations for data accesses are comparably light-weight, as MAD (multiply-and-add) instructions take one to few cycles, and calculation performance far outpaces DRAM access latency.

2.2.5.5. Reduction

Non-local calculations, some of which are virtually trivial on single-thread systems, are often hard to process on the GPU. Many times, global evaluation of a data array is needed, such as finding a maximum, the minimum, or the average value of all array elements. Serial implementations are trivial, but would not utilize the massive parallelization of graphics hardware.

The concept of data pyramids led to viable approaches for e.g. computation of a global sum of values: With the reduction operator in [BP04], groups of four cells are repeatly added in a pyramidal data array until only one cell prevails. In general, similar operators can be found for other tasks. While the number of data accesses is higher for parallel reduction as for serial approaches, it has to be considered that several calculations happen at once. Thus, complexity is always reduced by the number of processing units, **K** (a consideration in later algorithms).

One prominent example of reduction are mip-maps, which provide precalculated color averages to avoid aliasing in textured triangle rasterization. Their reduction operator repeatedly computes the average of four incoming color cells.

2.2.5.6. Bus Transfers

In current PC architectures, graphics hardware usually has its own memory, and connects to the CPU via the data bus, commonly PCI-express. While GPU-to-CPU bus transfer bandwidth has on the whole improved, bus bandwidth still a bottleneck in contrast to GPU memory bandwidth. This restriction must be considered in algorithm design: Many times, it is wise to use a slower algorithm on graphics hardware and avoid CPU assistance, as this would involve massive bus transfers and thus cause an overall lower performance.

In OpenGL, vertex buffer objects (VBO), pixel buffer objects (PBO) and textures are all aspects of memory storage in graphics memory. VBOs hold vertex data, while PBOs are used to rapidly move data from the CPU to graphics DRAM memory and back. CUDA improves on this by introducing page-locked main memory, allowing asynchronous DMA transfers to graphic hardware memory [NV07].

2.2.5.7. Immediate Output Readback

As we already mentioned, GPU threads are *not allowed to read back output* from other threads in the middle of a GPU program execution. This is because in data-parallel programming, no execution order is guaranteed. Output can thus be written in any order, and consequently, one shader thread cannot expect another thread to have written any particular output. Partially, this can be circumvented by using pixel blend operations in OpenGL. In CUDA, atomic memory operations are possible for inter-thread communication, but come at a large performance penalty of hundreds of cycles.

2.2.5.8. Random Access Writes

Random-access writes to GPU memory, also known as scattering, from within GPU programs is available on graphics hardware, but not performant. In OpenGL, they are possible by generating point primitives, one for each random access write, with vertex shaders determining their position in the output. In CUDA, global memory write access are available from within kernel code [NV07]. But random access writes come at a price in both cases: bundled memory access is disturbed by smaller accesses than memory bandwidth would allow, severely reducing memory I/O performance (depending on locality of access).

And despite scattering capability, the *write order* is still not deterministic amongst threads writing to a single location. Scattering on graphics hardware thus causes the same problems as PRAM hardware has for concurrent memory writes [KKT01]. We therefore discourage from using this capability in algorithm development. In many cases, scattering can be replaced by gathering, which executes one thread per *output* location. These threads can then use indirect reads to *fetch* input data, process the data, and then write it to the thread's fixed output locatoin. This way, read accesses can even be cached, by using textures that have been bound to the concerned memory areas. The texture cache can feed multiple threads with data from close memory locations, without issuing separate memory transactions for each thread. A description of the texture cache model in G80 architecture is provided by Volkov et al. [VD08].

2.3. Conclusion

In this chapter, we have demonstrated how increasing demands on graphics applications have made display hardware ultimately evolve into a processor for data-intensive general purpose computation. We introduced the term *GPGPU* for the use of graphics hardware in general purpose computation.

Data-intensive tasks can cause calculational bottlenecks on the CPU, either due to high memory bandwidth requirements or massive computational demands. Luckily, data-intensive tasks often process many data elements in a very similar way. This poses a well-suited challenge for the data-parallelism of modern graphics hardware, which had originated from parallel vertex and pixel processing.

But in exchange for high performance, algorithmic design is more constrained in data-parallel architectures. For example, OpenGL restricts thread writing locations to avoid data hazards from parallel write conflicts. As a consequence of arbitrary execution order, threads cannot read each others' output for a given GPU execution pass. Such restrictions to the programming model have long been seen as *so grave* that sophisticated data processing was considered impossible on graphics hardware.

In this thesis, we will show that even general data processing on graphics hardware is possible with the right algorithm. But first, let us introduce more conventional applications of graphics hardware to computer vision tasks.

3. Free Viewpoint Video Compression

In the last decade, users of video processing applications have seen tremendous improvements in computational performance. They are no longer restricted to 2D image operations, but can even start manipulating the original, three-dimensional *content* shown in the video footage. To achieve this, computer vision and 3D computer graphics are used in tandem to both extract original object content from the original footage, and embed new content into the video stream. In the same measure, graphics hardware can not only accelerate 2D video operations, but also assist in the 3D video processing that is required for content extraction and embedding of modified and new content.

In the following, we elaborate on how real-time graphics concepts such as mip-maps, projective texturing, or dependent texture lookups can benefit advanced 3D video processing for multi-view video (MVV), and subsequently even *accelerate MVV video compression*, which comprised the topic of research in this chapter.

The two compression methods that we summarize in the following chapter were presented in two publications [ZLA*04, ZLM*04]. The methods result in a much higher compression ratio for MVV footage than compressing the MVV footage's individual camera views into separate 2D video streams (also known as the MPEG simulcast reference). Two publications on the complete MVV acquisition and compression proposal followed, with the later one elaborating on sophisticated BRDF material reconstruction from MVV textures [TZMS04, TAZS07].

3.1. Introduction

Free Viewpoint Video (FVV), is a novel way to view scene content that has been recorded from multiple camera views, in multi-view video (MVV) footage. In free viewpoint video players, the viewpoint can be freely chosen, as long as the limits of the overall viewing angles and resolution limits of the recording setup are not exceeeded, see Figure 3.1. An FVV-encoded theatre play can thus be viewed from any viewpoint that the user prefers [TZMS04].

A few of the FVV viewpoints can be covered by the original camera views, the *input* views. For *virtual* camera views, i.e. viewpoints that do not match an original camera from the setup, content must be generated via interpolation.

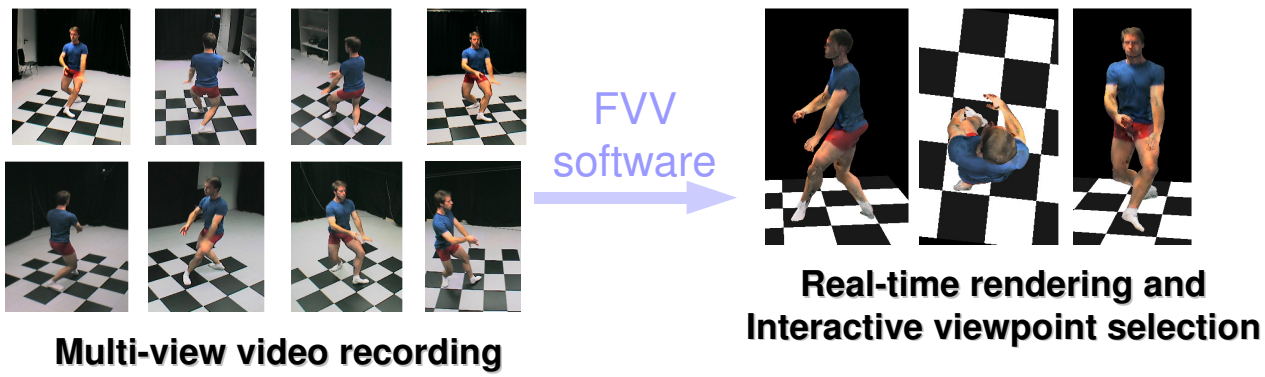


Figure 3.1: From Multi-View Video to Free Viewpoint Video.

The quality of such content interpolation depends not only on the interpolation algorithm, but also on the number of available input views that are close to the novel viewpoint. Naturally, a maximum

quality would be achieved if a complete lightfield of a scene was acquired, containing all possible viewpoints of the scene [LP96]. But due to both physical and acquisition/storage limitations, this is often not possible. Instead, sophisticated 3D image processing is used to both extract object geometry and shape from the available input views and to re-insert this content into the novel viewpoints, in an attempt to re-construct the missing novel viewpoint as realistically as possible.

View interpolation can generally be implemented via view morphing, which requires correspondences between the input views. Theoretically, these correspondences could be acquired via stereo correlation, or optical flow methods. However, optical flow methods are too computationally expensive to apply to video streams, and a stereo correlation method will often not succeed since the input views and their footage are not aligned along epipolar constraints. Instead, an animated 3D model is matched to the footage, using the methods by Carranza et al. [CTMS03]. This creates the necessary correlation for view interpolation, and was readily available for the works presented in this chapter.

While FVV playback is made possible by above view interpolation techniques, efficient storage and transmission has not yet been addressed. For the transmission from the footage producer to the consumer, wide data transmission channels would be required for this multi-view video footage. But these are often not present; and even larger storage media such as the DVD would not suffice for more than low-resolution snippets of raw MVV footage. Therefore, data compression is required before MVV footage can be transmitted to provide content for FVV playback software.

Like all video compression methods, MVV compression has to exploit data correspondences in the video footage. In MVV, multiple cameras produce simultaneous footage of a common scene, which produces inter-view correspondences beyond temporal correspondences between frames. But unlike stereo camera recordings, inter-view correspondences are not based on a clear camera parallax. Instead, camera correspondences can be arranged arbitrarily, which yields little compression for the motion compensation approaches of standard 2D video compression.

In our MVV compression technique, we assume that a common object of interest is shown in the MVV footage. With the same motion reconstruction techniques that Carranza et al. used for FVV view interpolation, we can attain a 3D model and its animation parameters, thus providing additional input to the compression algorithm. For inter-view compression, a pixel-wise correlation of input views is required. We observe that the 3D model's *surface* poses a common base of correlation for all input views, and can use a texture parameterization of the 3D model's surface to create a canvas onto which all camera views can be projected. If the 3D model matches the recorded scenery, then the camera views will match on the 3D surface, which effectively achieves a complex 3D motion compensation.

3.2. Related work

Using multiple images as texture is a relatively new research area. The benefits of having multiple images available to render view-dependent effects have first been shown by Debevec et al. [Deb96]. Compression of multi-view image data in the texture domain has been explored by a few researchers. Nishino et al. apply eigenvector decomposition to a number of textures created from images and a 3D model of the object [NSI99]. Wood et al. investigate vector quantization and principal function analysis to compress local reflection characteristics [WAA*00]. Magnor et al. make use of a 4D wavelet decomposition to exploit textural coherence also between textures [MRG03]. All previous work was concerned with encoding multiple textures of a static object. This paper, in contrast, presents an efficient approach to compress multi-video image data of a dynamic scene in the texture domain.

3.3. MVV Texture Generation

During playback in model-based FVV, it is common to re-project the input views of MVV footage onto the 3D-model [CTMS03]. However, such re-projection techniques are not commonly used in the *compression* of MVV. Our solution, used in both presented approaches [ZLM*04, ZLA*04], projects multi-view video footage from the 2D camera views into the texture domain of an animated 3D model to establish motion compensation.

This is accomplished in two steps: First, just as in the FVV playback techniques of Carranza et al. [CTMS03], the camera views' projection onto the 3D model is calculated via the projection matrices of the input views, and used to texture the triangle mesh of the animated 3D model. For FVV playback, the textured mesh would now be rasterized according to the model's current 3D position in the stage scene. Instead, we rasterize the triangles according to their *2D texture coordinates*, see also Figure 3.3. This effectively re-samples the cameras' video footage into the 2D texture domain, and creates partial textures of the 3D model for all camera views, *multi-view video textures*.

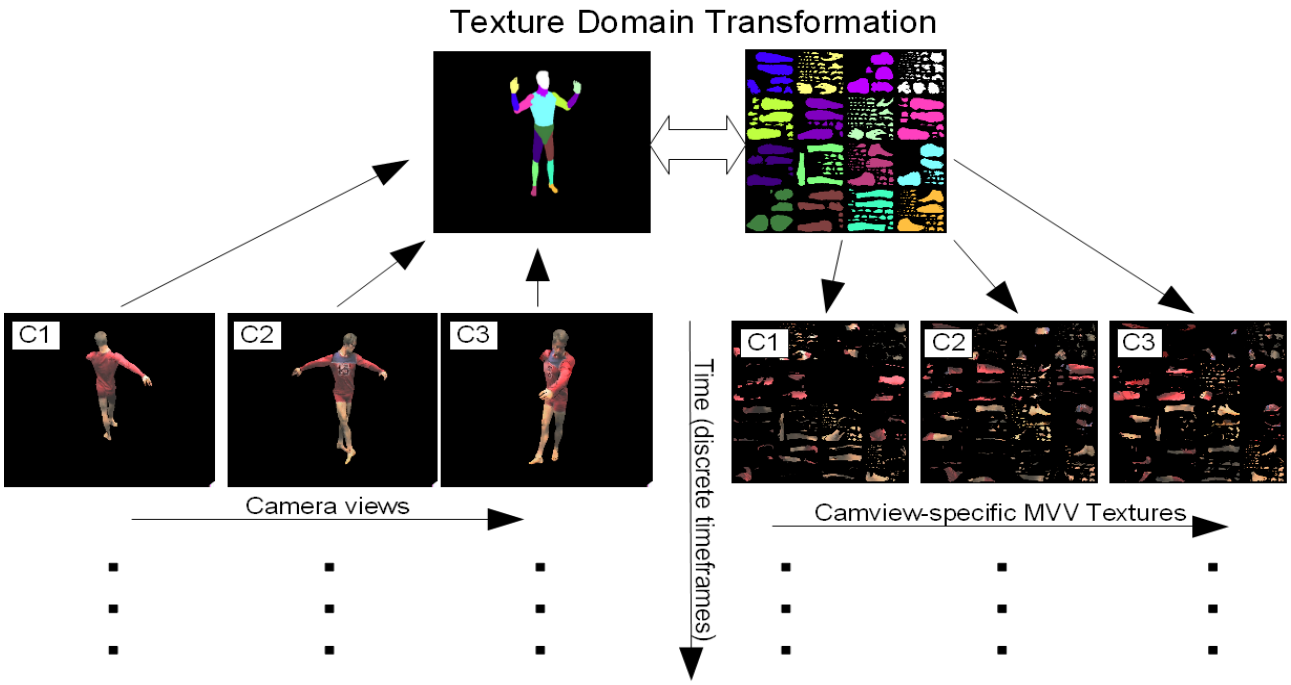


Figure 3.2: Camera views are transformed into an MVV texture, comprised of partial textures generated by GPU rasterization of the textured 3D model triangles into the 2D texture domain.

For each timestep, we can acquire a one-dimensional array of textures, where each entry represents the partial texture that has been reconstructed from a given input view, see Figure 3.2. With this 3D video preprocessing step, motion compensation for MVV has now become much easier, since the 3D correlations have been collapsed into overlapping texture coordinates in the 2D texture domain. Taking time into account as additional dimension (symbolized as purple dots in Figure 3.2), we have thus acquired a four-dimensional array of textures, with four axes: time, viewing angle, and the two dimensions of the texture parameterization. This is what we call the Multi-View Video Texture, short: MVV texture. It is generated and used in both our compression approaches [ZLA*04, ZLM*04].

An MVV texture is usually not completely filled with data. There are two sources for empty texel areas: Either, an area has not been allocated to any triangles of the 3D model, or a section of the 3D model has not been exposed in this camera view of MVV footage. Wth a *visibility mask,* the

actually projected-upon, or *relevant* texels, are marked as shown in Figure 3.3. Its use improves compression performance, since it allows irrelevant texels to take on arbitrary values in their compressed representation. Visibility masks can be generated from *shadow mapping*, a real-time 3D graphics technique that is normally used to simulate shadows from point light sources [NV05].

In our case, the camera position is used as light source position, and generates a depth map that holds the distance of all triangles to the camera. Then, all triangles must be re-rasterized in the texture domain in order to generate the MVV texture. During this rasterization, the triangle fragments' distance to be camera view is calculated, and compared to the camera depth's map using hardware shadow mapping. If they happen to be further away from the camera than the camera visibilitys' depth map has recorded, then they were not exposed in this camera view and are left empty. Only fragments which match the recorded depth in the camera view are written into the MVV texture, and are marked with a binary one in the visibility mask.

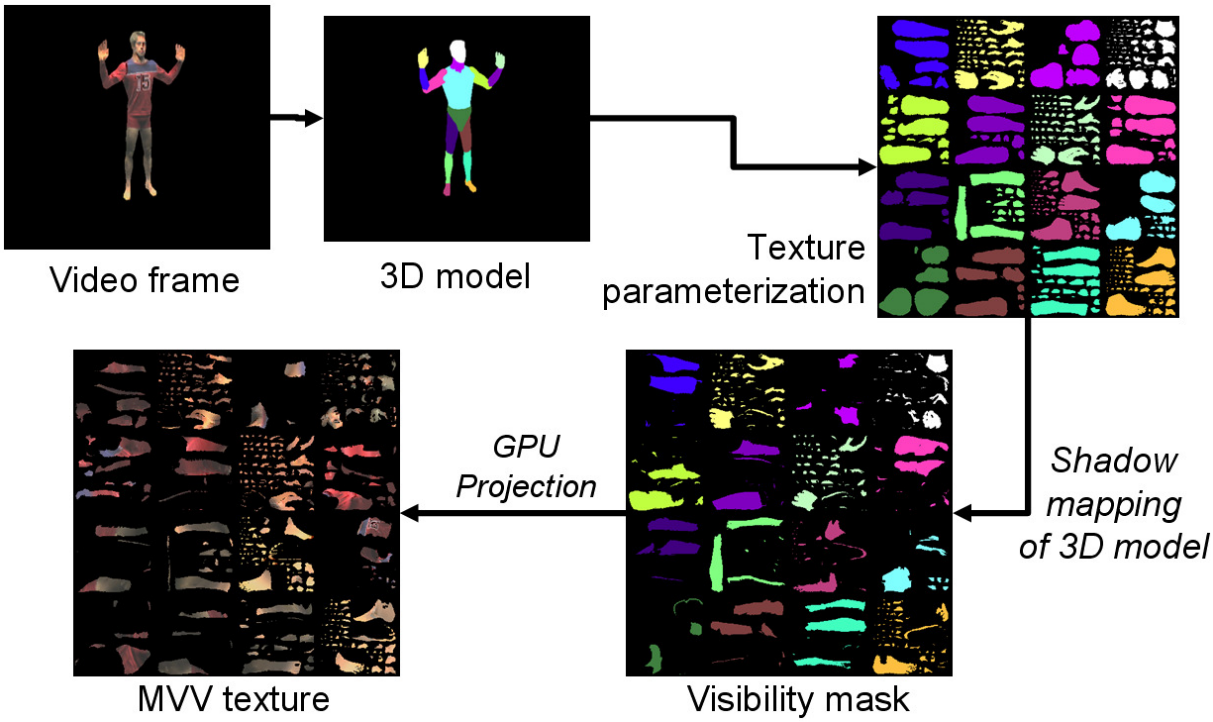


Figure 3.3: Resampling into the MVV texture atlas. (a) Color coded body parts (b) Original frame (320x240) (c) Corresponding regions in texture space (d) Resampled frame considering visibility (512x512).

3.4. MVV Texture Compression

In the following, we present two different schemes for the compression of MVV textures. Both result yield a much higher compression for the FVV footage than compressing the MVV's individual camera views into separate MPEG-2 video streams (also known as the simulcast reference, here as MPEG-2 simulcast).

3.4.1. 2D-Diff / Wavelet Compression

In our first approach [ZLA*04], we apply a simple, yet effective, scheme to compress the 4-dimensional texture array of an MVV texture, see Figure 3.4. By averaging the relevant texels from all camera views, we generate a *time average texture* for a certain time step, `T'(t)`. The time average textures themselves are averaged again, yielding a *master texture*, `T''`. From this master

texture, residuals of time average textures and camera view textures can be generated. Then, the master texture and the residuals are compressed using the BISK wavelet compression algorithm [Fow04]. BISK wavelet compression ratio increases slightly if a visibility mask is present.

```
for all timesteps t:
  average all camera images c → T'(t)
average over all timesteps → T''
encode, decode T'' → T̂''
for all timesteps t:
  T̂'' − T'(t) → R'(t)
  encode, decode R'(t) → R̂'(t)
  reconstruct T̂'' + R̂'(t) → T̂'(t)
  for all camera views c:
    T(t,c) − T̂'(t) → R(t,c)
    encode R(t,c)
```

```
decode T̂''()
for all timesteps t:
  decode R̂(t)
  reconstruct:  T̂''() + R̂(t) → T̂(t)
  for all camera views c:
    decode R̂(t,c)
    reconstruct:  T̂'(t) + R̂(t,c) → T̂(t,c)
    write/apply T̂(t,c)
```

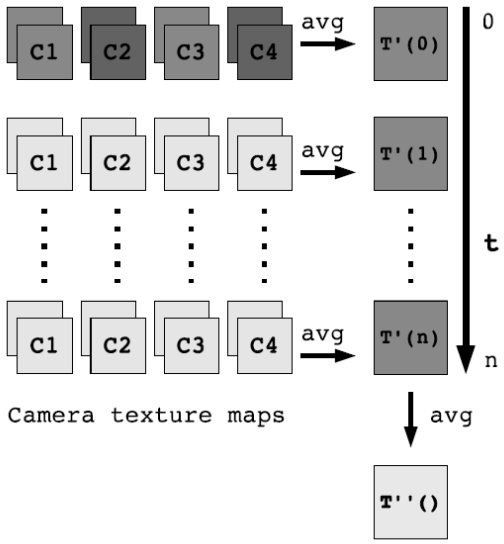


Figure 3.4: 2D-Diff generation, pseudocode and diagram approximation.
Top left and right: Encoder pseudocode and dataflow. Bottom left: Decoder pseudocode.

For decoding, time averages for a given time step are reconstructed, from which the final camera view residuals can be generated, see also the pseudocode in Figure 3.4.

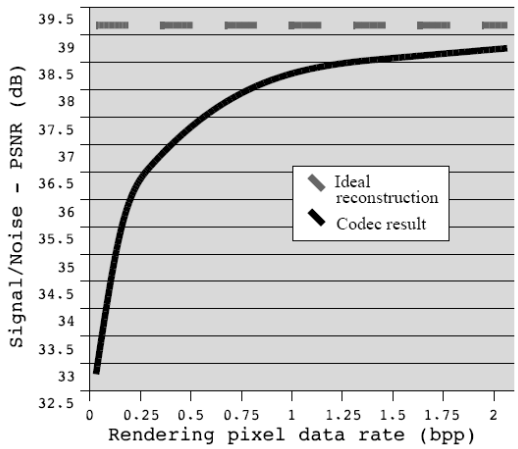


Figure 3.5: Quality comparison between renderings generated by uncompressed MVV footage (ideal reconstruction) and renderings from 2D-Diff compressed MVV footage (Codec result).

Quality comparison and verification of the encoder/decoder were achieved on the basis of a 350 frames long dancer scene, for which a matching 3D model had been reconstructed [CTMS03]. Their MVV texture, 3D model, camera views and animation parameters are available for download [TAG04]. To acquire an indication of the maximum quality achievable with the rendered result, we mapped the MVV texture back to the 3D model without using any compression pipeline inbetween. This resulted in a maximum limit for the achievable image quality, called ideal reconstruction.

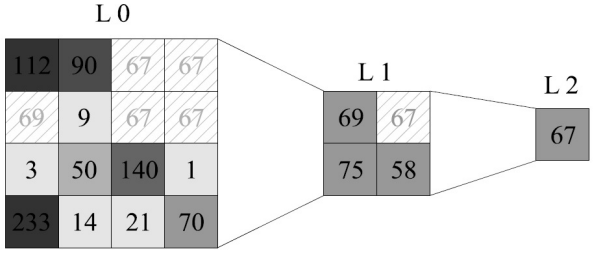
Then, we encoded the MVV texture at various bitrates, and mapped the decoded results back to the 3D model to gain an impression of how much the rendering quality was affected, see Figure 3.5. We can infer that a compression rate of 1 bpp can safely be used without sacrifices in rendering quality. The compression ratios of 50:1 to 8:1 compare well to the ratios attained from MPEG simulcast video compression, with the strong advantage that arbitrary views can be rendered from the input data using FVV techniques and the animated 3D model. The areas used in the texture maps and the rendered images compare at a ratio of 3.1:1.

3.4.2. 4D-SPIHT Wavelet compression

While 2D-Diff compression results were promising, they did not explicitly exploit the temporal correlation within camera views. Therefore, in order to encode the whole 4-dimensional texture array as one data entity and exploit interframe coherence in *both* temporal and spatial dimension, we implemented a more sophisticated 4D SPIHT wavelet compression algorithm [ZLM*04]. Our own SPIHT encoder implementation very much resembles its well-known 2D counterpart [SP96], with one exception: Since the 4D dimensions may be of different size, the codec has to be able to detect boundary conditions that generate a different number of descendants in the wavelet data traversal. In related work, a shape-adaptive SPIHT algorithm in three dimensions had previously been implemented by Danyali et al. [DM03].

Even in the SPIHT compression approach, it is useful to generate and transmit a visibility mask that marks irrelevant texels in the MVV texture. To improve wavelet compression, irrelevant texels can be filled by the average of their closest neighbours. This is achieved by the *sparse fill* procedure, part of the 4D Haar wavelet separation that poses the first step of SPIHT compression. First, the texture is averaged in a mipmap-like reduction process. Then, reduction reverses, and fills irrelevant texels at finer resolution levels with averaged texels from the levels above, see pseudocode and example in Figure 3.6. The sparse fill feature thus improves compression because the extracted SPIHT wavelet coefficients become smoother and span larger areas, which makes their position description require less storage in the bitstream.

```
for all timesteps of camera images T(t,c):
  for each color component Y, U and V:
    group s cameras with t timesteps into
      4D array (textures, u x v texels)
    downsample 4D array log(max(s,t,u,v)) times.
    upsample selectively, filling unused texels
    apply 1D Haar wavelet transformation
      repeatedly on (cam, time) dimensions s, t
    apply 1D Haar wavelet transformation
      repeatedly on (image) dimensions u, v
    apply 4D SPIHT onto 4D array
      result: 4ST code stream
```



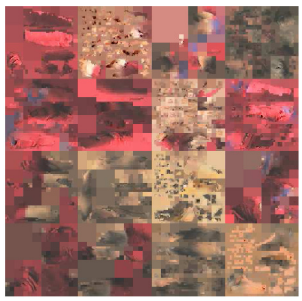


Figure 3.6: Left: Implementation pseudocode with figure.
Sparse fill intermediate output, before 4D-SPIHT compression commences.

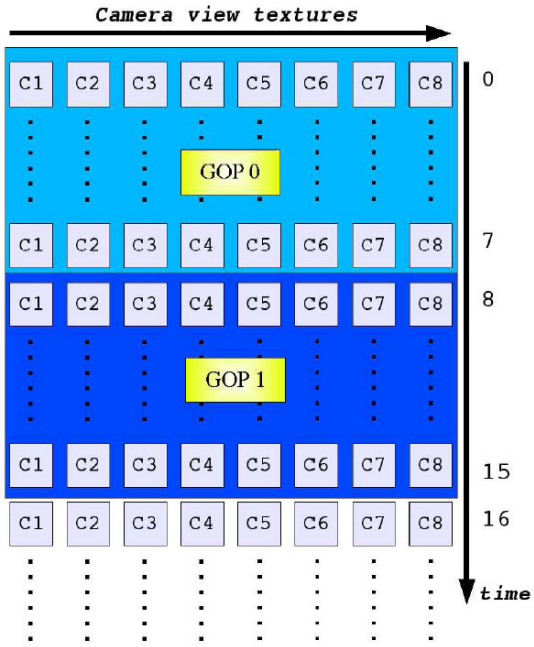


Figure 3.7: Right: the GOP (group of pictures) arrangement for a given MVV texture input.

Figure 3.7 shows how MVV texture components are grouped together into groups of pictures (GOPs), before compression commences. The decoder reverses this processes: If e.g. camera view C4 of time step 15 is required, then GOP 1 will first be decompressed as a whole. Compression gains thus come at the expense of random access. On the upside, one of our codec's advantage is synchronous access to the texture maps, which is harder to achieve with parallel MPEG2 decoders used in MPEG simulcast approaches (which are often not designed for frame-exact decode-and-resume, e.g. by lacking frame-exact seek ability).

```
for all required camera images and timesteps:
  for each color component Y, U and V:
    choose 4ST code stream containing T(t,c)
    restore 4D array with 4D SPIHT
    apply 1D Haar wavelet transformation
      repeatedly on (image) dimensions u, v
    apply 1D Haar wavelet transformation
      repeatedly on (cam, time) dimensions s, t
    extract T(t,c) from 4D array
  (apply shape mask to T(t,c) )
```

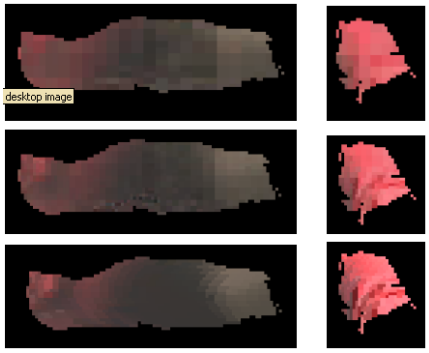


Figure 3.8: Decompression results. Left: Pseudocode. Middle: Texture decompression at varying bitrates.

For testing the codec performance, we used parts of an MVV texture from a 350 frames long dancer scene, for which a matching 3D model had been reconstructed by Carranza et al. [CTMS03], just as in Section 3.4.1. Our experimental results with eight texture streams, each of 40 frames, show that the compression ratio ranges up to 288:1, if a texel data rate of 0.05 bpp, the sparse fill procedure and lower UV bitrate are used. See Figure 3.8 for decoder pseudocode and a detail of the MVV texture decoded at varying bitrates. In the compression mode, the PSNR of the textures reached 23.5 dB. At a texel data rate of 0.30 bpp, a PSNR of 30 dB can be reached. But it is important to keep in mind that in order to reverse the sparse filling procedure at the decoder side, either visibility masks *or* an animated, textured 3D model and camera parameters must be transmitted.

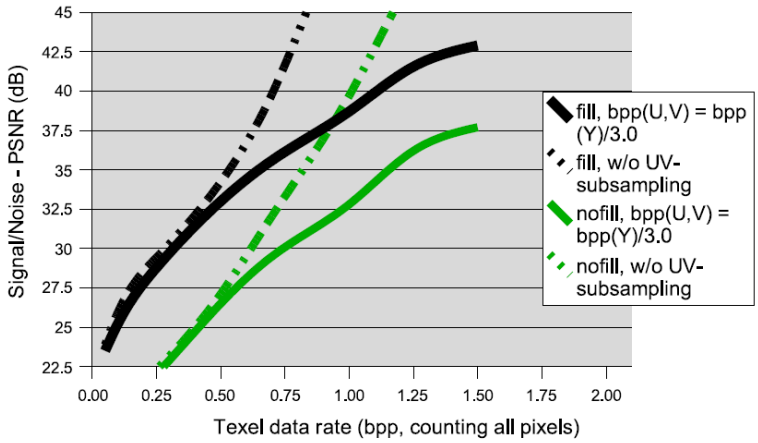


Figure 3.9: Decoder quality for a number of compression modes.

Without UV bitrate reduction and the sparse fill procedure, the usable compression ratio ranges up to 50:1, as Figure 3.9 shows. This mode can be used if no visibility mask shall be transmitted and authentic color resolution is of high importance. Here, an overall compression rate of 20:1 (Y: 0.75 bpp, U & V: 0.25 bpp) can be safely used without sacrifices in reproduction quality (PSNR: 30 dB).

3.5. Visibility Mask Reconstruction

In both compression approaches, only relevant texels need to be considered in compression. Whenever the compression algorithm may treat unfilled or shadowed texels as irrelevant in decoder output, compression can be increased considerably. In the 2D-Diff-approach, this is achieved by passing the visibility mask to the BISK wavelet encoder, while in the 4D-SPIHT approach, the fill feature will exploit the visibility mask to fill irrelevant texels with compression-optimal values. However, in both cases, this visibility mask must be known to the decoder. Therefore, it must usually be transmitted along with the code stream. Another option is to use the fact that this visibility stems from the knowledge of 3D model and camera view positions, usually transmitted in the same stream for FVV playback purposes. With this input data available, the GPU can reproduce all camera exposures at the decoder side, and generate visibility masks from scratch. Visibility mask can thus be omitted from transmission, provided that decoder and encoder agree exactly on how visibility masks are generated. Note that decoder-side visibility generation increases computational requirements and needs to be traded off against the obtained gain in compression.

3.6. Conclusion

We have presented two compression approaches for a new data modality, multi-view video footage, which is the input for Free Viewpoint Video playback software. Our method relies on having an approximate 3D model of the dynamic scene available, and uses it to map the video frames into an array of surface textures. By working in texture space, we compensate for disparity between camera views as well as for object motion over time. With hardware shadow mapping, visibility mask computation becomes feasible for the MVV texture generation process, which further increases data compression. By following a two-level hierarchical coding approach, we enable random access to any view by decoding just two texture difference maps.

Compression ratios up to 250:1 have been verified experimentally, with the commonly usable range below 50:1, all without taking YUV subsampling into consideration. This compares well to MPEG simulcast video compression, with the additional advantage of reproducing arbitrary viewpoints via FVV playback techniques.

3.7. Future work

The next step is to accelerate decoding to allow for real-time rendering from the compressed bit stream. For this purpose, graphics hardware could be involved in further parts of the bitstream decoding process, such as the expansion of SPIHT wavelet coefficients or the summation of residual texture data. It would also be worth investigating if U and V values should be subsampled prior to wavelet encoding, or if a reduced bitrate suffices. Another topic would be to find an area ratio between texture map and rendered image that does not affect signal-to-noise, but which does not oversample, either.

More sophisticated texture parameterization methods create continous texture atlases over large areas of the mesh surface (e.g. whole torso, arms, legs, head laid out in only one texture each, altogether 6 continous body textures). Such larger atlases provide better efficiency, as less discontinuities (and thus fewer high-frequency coefficients) have to be encoded in the SPIHT compression.

If the underlying 3D model does not suffice for some more complex deformable objects present in the scene, then multi-chart geometry images, as introduced by Sander et al. [SWG*03] might prove useful. Even these can be compressed with the presented compression methods if they are adapted to compress a six data channel texture (XYZ, RGB) instead of the 3 color channel texture.

To further improve coding efficiency, we could retrieve surface reflectance characteristics that are common to a particular area of the MVV texture. These characteristic often resemble object materials, are partially responsible for differing texel colors under varying camera exposure angles, even if the surface is perfectly inter-view correlated [TAZS07]. By compensating for these reflectance characteristics, compression can be further increased.

In future implementations of the presented MVV codecs, graphics hardware could also aid in data compression. If this approach is pursued, a replacement of the four sequential1D Haar wavelet transforms with two 2D Haar transforms should be considered to fit the GPU's native 2D image processing capabilities. With data compaction techniques presented later in this thesis, even the SPIHT coefficient separation and bitstream generation could be implemented on graphics hardware.

4. Color and Depth Reprojection

In the previous chapter, we have projected camera images back into world space, and used this technique to reconstruct a 3D model's texture from multi-view video. The important insight we gained was that camera imaging is *reversible*, and that acquired images can just as well be projected back from the camera´s recording position. This *Camera/Projector dualism* can be exploited in various ways, and has previously been used by others to reconstruct images from a projector view [SCG*05], or to compensate for a non-uniform projector canvas [GB08].

In this chapter, we demonstrate how the camera/projector dualism can be expressed with the OpenGL graphics API, leading to GPU acceleration for applications of this dualism. Section 4.1 explores a yet unpublished application of the camera/projector dualism: Calibration of camera images onto a common 3D scene with the help of graphics hardware. The interactive aspect and immediate user feedback allows for a fine-grained calibration that fully automatic algorithms can rarely achieve. In Section 4.2, we extend the camera/projector dualism to depth values, and show how OpenGL concepts can be used to simplify *depth reprojection*. In the course of this section, we describe an interchangeable description format for camera setups with multiple (depth) cameras, published in 2005 [ZHMS05]. Section 4.2.5. shows how framebuffer accumulation of depth camera sweeps can enable future view-planning for depth acquisition. Section 4.2.6 extends the discussion on novel viewpoint generation from multi-view depth video with a previously unpublished algorithm for FVV reconstruction using methods from constructive solid geometry (CSG).

4.1. Interactive Multi-Camera Calibration

While calibrating a camera's intrinsic and extrinsic parameters using acquired calibration footage, it can often be hard to make the algorithm automatically align to small features in the footage. Manual intervention is required, but calibration software (e.g. the Camera Calibration Toolbox for Matlab [Bou]) is often not designed for manual adjustments of the calibration process. This makes repeated calibration “experiments” necessary, with feedback often taking dozens of seconds as the CPU recomputes the corresponding image warp. Additionally, camera calibration software usually considers one camera only, and does not take other camera views of the same object into consideration.

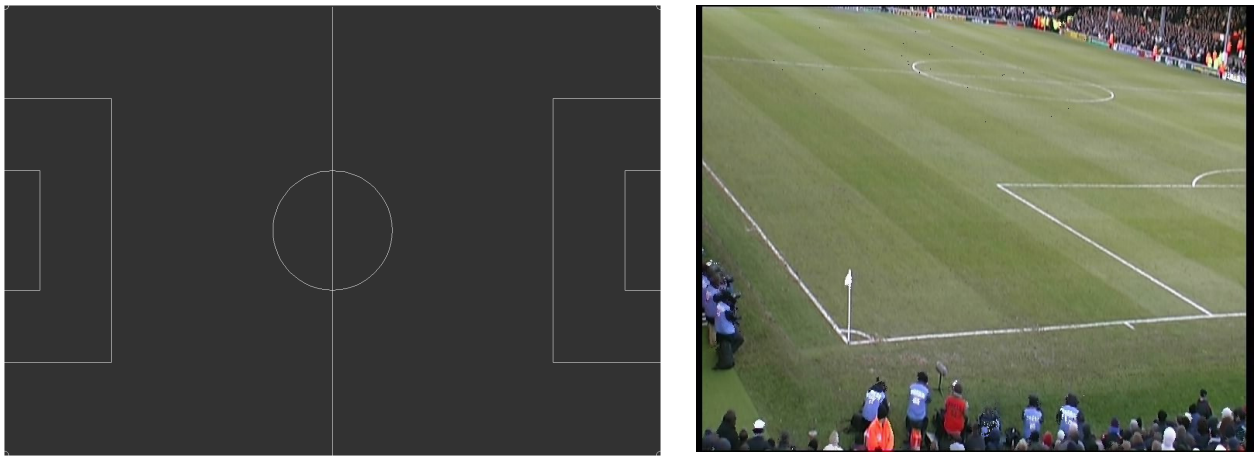


Figure 4.1: Input data to interactive camera calibration.
Left: Soccerfield proxy. Right: One of the camera views.

To improve on this time-consuming process of interactive calibration, we explored the possibilities of reprojecting camera footage onto a 3D model of the scene that it was acquired from, and let the user adjust the camera parameters freely, resulting in changed image reprojection onto the 3D model. Due to GPU accelerated projective texturing [Eve02], the user will immediately receive feedback on how his/her camera parameter changes have affected the alignment of the images with actual scene features. The camera model also provides the change of intrinsic camera parameters like lens distorsion, as the GPU is able to pre-warp the images to correct for a given lens distorsion before projecting them into the scene using a pinhole camera model.

For given real-world video footage, it is often hard to calibrate a camera accurately, or calibration has not even been attempted before. However, since many image-based rendering methods require camera parameters, calibration has to be acquired in post-processing. While it is common to fit a 3D model to recorded camera footage, such as demonstrated in Debevec et al. [Deb96], we focused on the *opposite* problem: Camera pose reconstruction, given a 3D model of the scene. Related augmented reality research tries to achieve automatic pose estimation for dynamic camera movement [KKS*05], which is also known as match moving [GBM*04, SGA*99], but these approaches often simplify the camera parameter space and tolerate tracking errors to counter the limited computation time available, or merely require an approximate calibration for motion prediction [GBM*04]. Our work focuses on maximum quality camera calibration for static camera setups, where even very faint features are utilized to improve camera calibration. Since such faint features are highly dependent on context, we involve the user in an interactive feedback loop to provide for satisfying results. By using a known 3D model of the recorded scene, the user is enabled to "fit" the original camera image to the 3D model within minutes.

In our current implementation, the user is allowed to manipulate camera roll, translation, field of view and radial lens distortion. For every modification, the camera modelview and projection matrix are re-calculated, and the footage is un-warped to compensate the user-provided lens distortion. Then, projective texturing is used to visualize the footage on the 3D model within milliseconds on graphics hardware. Since all camera parameters can be adjusted interactively, a human user is able to fine-adjust the camera alignment until no more improvement is possible, either due to 3D model inaccuracies or camera model restrictions, see Figure 4.2.

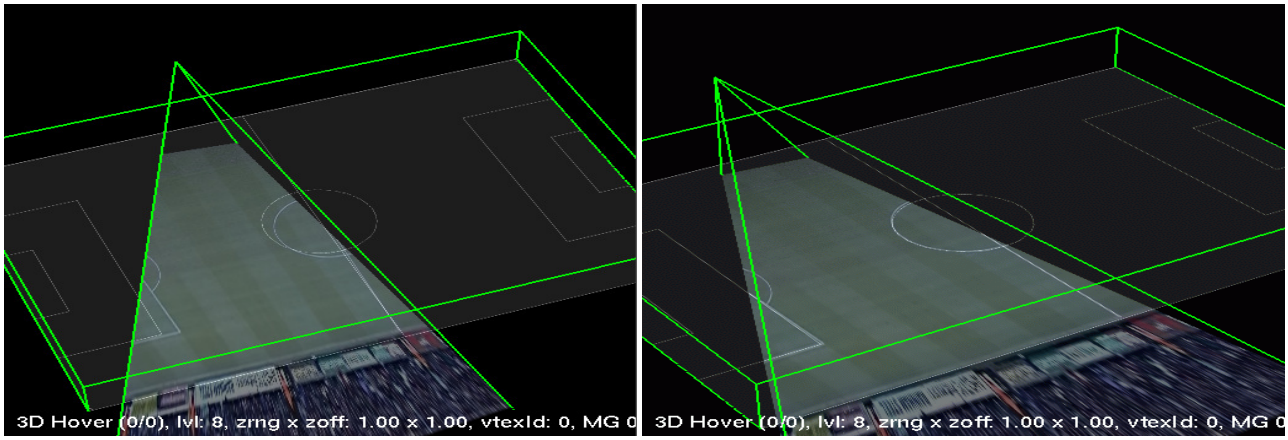


Figure 4.2: Iterative calibration. Left: Rough pose estimate. Right: User-refined improvement.

With the help of this immediate feedback, the user is then able to iteratively re-adjust the parameters. This is repeated until results satisfactory for a human user, which typically happens within a matter of minutes. While this seems a long time compared to automatic approaches, it allows for the calibration of very ill-conditioned footage, as is the case for the far-away opposing team´s goal in Figure 4.3.

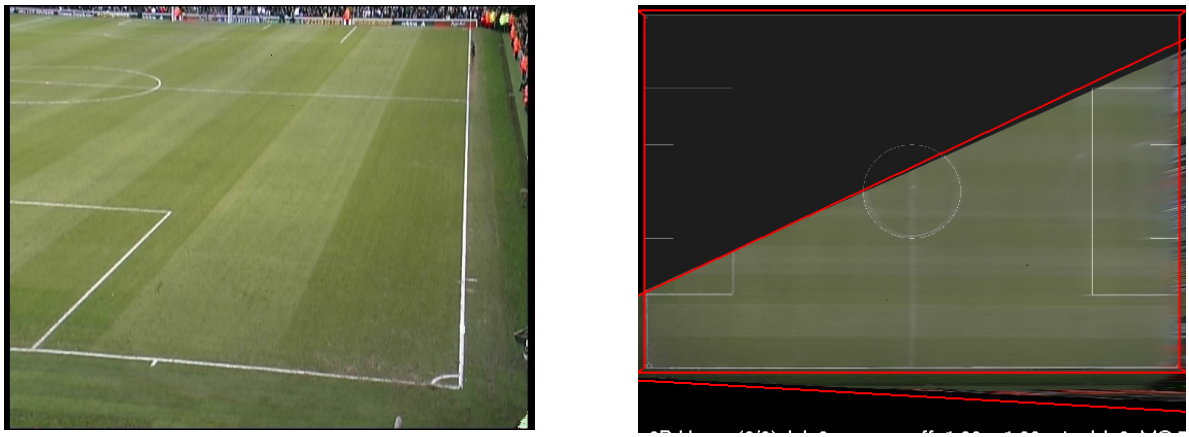


Figure 4.3: Calibration of ill-conditioned footage. Left: Footage. Right: Calibration result.

First tests with real data show that interactive calibration can assist the user, and achieve calibration results even for harder cases. While the search space is typically 6-9 dimensional, it doesn't take a user more than 2 days to get acquainted with the user interface, and to learn how to fine-adjust each camera to a given 3D scene within approximately 10 minutes. Figure 4.3 demonstrates how a virtual top view, and a projection of the camera image onto the soccer field proxy make it possible to fine-align the blurred and small features of the original camera image. Figure 4.4 shows how two camera views can be aligned by projecting onto the geometry proxy of the 3D model. This system thus allows for post-readjustment of coarsely aligned camera setups, even for inexperienced users. After and during calibration, the camera setup can be stored in a set of GeoCast meta-data files, see also Section 4.2.3. for more details.

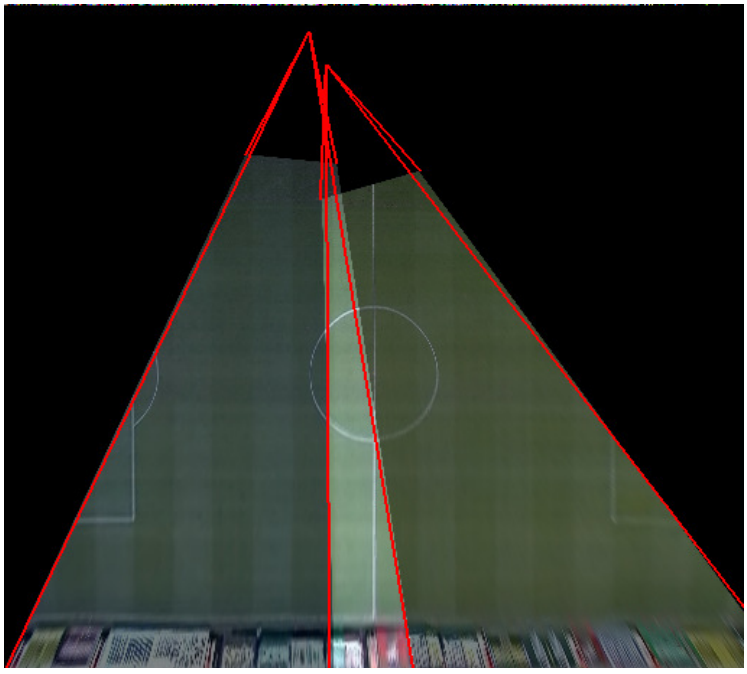


Figure 4.4: Calibration of two camera images via a geometry proxy, serving as ground truth.

In summary, the system enables computer-guided camera calibration for human users, which proves highly usable in situations where automatic camera calibration algorithms fail. We are also experimenting with shadow mapping to allow for a correct reprojection onto multiple 3D objects, enabling human users to verify that a camera image correctly depicts the scene's objects. Already

now, though not shown above, the system can use projective geometry from depthmaps as calibration proxies, and thus enable 3D scenes with non-trivial occlusion situations.

Further down the road, this system could be turned into an automatic camera calibration algorithm, which uses feedback between the ground truth of 3D scene model proxy and the current reprojection to hierarchically search the camera's parameter space for the correct camera setup, much in the same way a human would approach the calibration task.

4.2. Depth Reprojection

As cameras become capable of *depth acquisition*, new questions arise in the use of the camera/projector dualism. Depth cameras must have some kind of projectors as a dual, but what does *projected depth information* actually imply? While this might be hard to imagine in the real world, it is easy to visualize in the virtual worlds of computer graphics.

Before depth maps can be projected, information has to be gathered on the original depth acquisition. First, the camera position and parameters must be known, to relate the camera view to the actual world scene. Luckily, this is a common task in computer vision, which provides the neccessary mathematical framework to project an object onto an image plane, either in orthographic or perspective projection. But the original mathematical framework of computer vision has its limitations: It does often not consider the representation of *depth values*, i.e. the distance from the object surface to the corresponding image point. While this seems trivial to add, the finite resolution of depth values also requires the introduction of a *Z-near* and *Z-far plane* into the perspective projection. Another, less severe restriction is that image sensor resolution is often included in the imaging matrix, which implies that it has to be regenerated for every change of footage resolution. In OpenGL, on the other side, the image is referenced with the normalized image coordinates [0,1.0] in both dimensions, and allows for later change of image dimensions. For further reading, Li et al. has compared camera decriptions of computer graphics and computer vision [Li01].

To accomplish such reprojection of camera footage in a consistent way and connect more easily to content editing and playback software, we have decided to utilize the OpenGL standard to describe the object to image transformations, including object surface distance to depth values. With OpenGL notation present in the acquisition information, it also becomes easier to visualize the created content on graphics hardware.

In Section 4.2.2., we describe how different camera types can be represented in this notation, both synthetic (e.g. from 3D modelling software) and real (e.g. with lens distortion). Depth video receives special attention, as its depth channel requires an additional format description. In the actual data storage, the recorded footage can contain color video or color/depth streams (here: depth video), and is paired with the camera parameters, also called camera *meta-data.*. For this purpose, we have designed GeoCast, a description format for acqusition information and depth data storage. With the meta-data, all kinds of multi-view camera footage can be converted into a common representation, such as multi-view video footage, lightfield footage, and multi-view depth video.

We also demonstrate methods to display this footage on graphics hardware, and include screen shots of a small rendering system that visualizes GeoCast based RGBZ video in its original world space setting, see e.g. Figure 4.9. Our system utilizes projective geometry to make virtual projectors “cast” depth data from the original camera position. This way, novel views of the scene can be recreated as seen by the multiple cameras combined.

Finally, we exemplify several applications of multi-view depth projection from image-based rendering. We start with a basic implementation of depth projection on graphics hardware in Section 4.2.4., and explain our notion of depth sample connectivity and depth surface culling. In Section

4.2.5., we describe how the Z-buffer of graphics hardware can be utilized to accumulate depth camera sweeps over time, thereby generating novel views of objects from partial depth camera observations. Section 4.2.6. describes how Free Viewpoint Video playback from multi-view color/depth footage can be improved with ideas from Constructive Solid Geometry (CSG). In Section 4.2.7., we provide an outlook on future work, including a rough sketch of geometry image projection.

4.2.1. Related work

With dropping prices for imaging sensors, multi-camera acquisition has become more viable in content creation and imaging research. Lightfield camera arrays and multi-view video setups all aim at recording a common scene from several viewpoints simultaneously [WJV*05]. First prototypes of depth acquisition cameras are available [3DV] [3DIP], and they will soon be put into multi-view configurations as well.

A multi-view data set requires the presence of correlation information for the recorded camera views in order to be used in 3D scene reconstruction. Computer vision defines all the mathematical tools for describing the correlation between light entering the camera and actual image pixels [HS97]. Light ray/pixel mapping from computer vision can also be used to determine how an image with depth information (RGBZ image) maps into world space, but it often lacks information on how a depth value, i.e. the distance from camera sensor to the object surface shall be represented. Inter-camera calibration algorithms, such as the ones by Ihrke et al. [IAM04], provide a correlation matrix between the different camera views as a number of translations and rotations, and a textual description on how to convert these parameters to a matrix. Unfortunately, this is too loose a definition to allow for machine-based exchange of camera setups.

Most often, because of the lack of common standards, visualization and post-processing developers need to communicate intensely with content creators to determine how provided video footage can be projected back into world space in order to reproduce the original ray field [3DTV]. In case of depth video, an extended description of the mapping between camera z-axis and stored depth values becomes necessary. With this mapping unavailable, a 3D depth surface cannot be reconstructed from the camera footage. There is currently no way to describe multi-view camera setups in a standardized form, without possibly ambiguous textual descriptions, especially if dynamic camera movement is involved. However, textual descriptions are common, and examples can again be found in the sample data section of the 3DTV project [3DTV] and our free viewpoint video footage [TAG04].

Several display methods for regular depth data on graphics hardware have been published. Parajola et al. [PSM04] and Bogomjakov et al. [BG04] apply quadtree-based depth-map preprocessing to create triangle representations. While this is more data-efficient, it also requires preprocessing steps that range in the hundreds of millseconds, and thus are not viable for processing at video framerates. The treatment of discontinuities in depth maps, when viewed off-angle from the original viewpoints, differs strongly. While Oliveira et al. were not at all concerned with discontinuities, and used disjunct surface samples with increasing holes as the off-angle increases [OBM00], Parajola et al. propose triangle meshes to fill the discontinuities, and use a certain depth discontnuity thresholding to counter unwanted rubberband effects [PSM04].

4.2.2. Camera Representation

In reprojection, acquisition information beyond the camera footage is necessary to reconstruct how image pixels map to the light rays that were originally recorded from the acquired scene. The most important one is the spatial *correlation between camera views* and the depicted scene, as defined by

camera position and orientation. But intrinsic camera parameters are necessary to map image pixels to their correspondence in world space, such as field-of-view and imaging model (e.g. the pinhole camera model). In case of footage including depth values, the acquisition information must also describe how depth values map to object surface samples at the time of acquisition.

To summarize, camera footage thus needs to be bundled with *meta-data* to allow for reconstruction of 3D geometry and generate novel virtual views through depth projection. In static camera setups, this meta-data is constant for the whole footage; but in the case of camera movement or intrinsic camera changes (e.g. zooming), meta-data actually becomes frame-specific, and thus needs to accompany every video frame.

The camera's imaging model determines how objects from the scene are projected onto the camera sensor. To counter the issues of the camera model of computer vision, we introduce several concepts from OpenGL for the description of camera setups. What follows is a summary of the OpenGL concepts we employed; we recommend the OpenGL specification [OGL21] for details on the OpenGL coordinate system and its image formation pipeline,.

Given object coordinates from the scene as input, the *modelview matrix* is applied first, a 4x4 matrix which transforms world coordinates to eye coordinates. In our camera meta-data, the OpenGL modelview matix is stored as such. This explicit use of the modelview matrix avoids problems that other camera position descriptions using translations and rotations produce, such as gimble lock and the need to define the order of operations. Of course, the disadvantage is that the camera position is only provided implicitly in a modelview matix, and thus not easily human-readable. It can, however, be easily extracted by software, and written into any required format.

Next in the coordinate transformation comes the OpenGL projection matrix, a 4x4 matrix, to define how eye coordinates result in a image representation, including color and depth sensor positions, but still independent of camera sensor resolution due to normalized coordinates. Any wide baseline or lightfield camera arrangement imaginable can be represented with this 4x4 projection matrix and a preceding modelview transformation as long as all transformations are linear. If intrinsic camera parameters such as non-linear lens distortion are present, a step of image warping can ensure the linearity of above matrices. This image warp is typically implemented as a shader on graphics hardware, see also section 5.3.1.

The available camera models are as follows, with the OpenGL specification providing a more detailed definition [OGL21]:

Pinhole camera: For real-world footage from a single camera, this is the only possible imaging model. Among the available parameterizations, we chose to represent such cameras with the parameter set of OpenGL's `gluPerspective()` call: Field of view, the image aspect ratio and their near and far clipping plane for depth values.

Orthogonal camera: This imaging method fits well for space carving methods and other voxelization methods, which divide world space into uniform 3D grids, such as the one used by Curless et al. [CL96]. Synthetic depth video from 3D geometry can also be exported with this simple image-to-world mapping. The parameters specify a horizontal and vertical window size and a Z-near and Z-far clipping plane which is handled by the OpenGL `glOrtho()` call.

General 4x4 projection matrix: Some imaging methods cannot be represented in the models above, including the sheared multi-frustum that is used in lightfield representations, which resamples multiple imaging sensors onto a common plane [LP96]. However, each individual frustum can be represented by a general 4x4 projection matrix, and often even be generated by the OpenGL `glFrustum()` call. For this and other unusual camera imaging models, we allow for direct

specification of an OpenGL-style projection matrix. Figure 4.5 shows a lightfield, represented as a multitude of sheared camera frusti focusing on one plane.

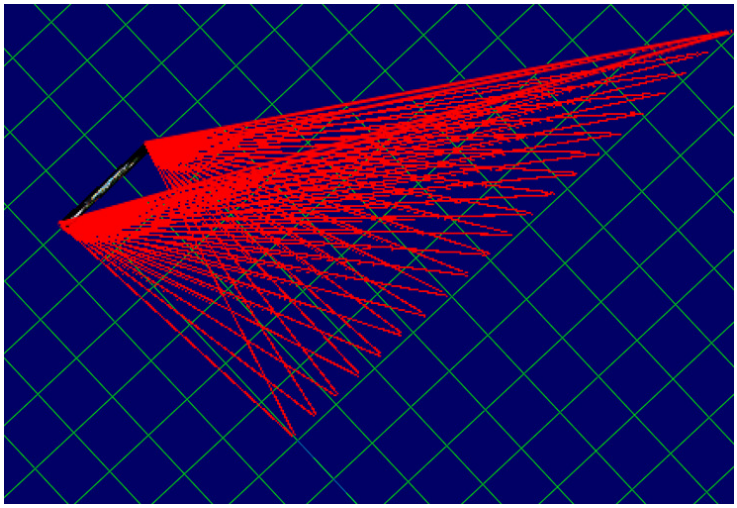


Figure 4.5: A lightfield, represented in GeoCast
as a multitude of sheared projections (red) onto a common canvas (black).

The OpenGL coordinate transformation maps to normalized image coordinates instead of pixel positions. The presented matrices map object coordinates into OpenGL's floating-point pixel addressing domain of x=[-1, 1], y=[-1, 1] instead of the camera image's pixel coordinates x=[0, xres], y=[0, yres]. As a consequence, we are able to omit image size from our camera meta-data, and retrieve image resolution from the specific storage format of the footage.

The 4x4 projection and modelview matrices also transform depth values, a vital component for depth projections. Since the generated image coordinates are normalized, depth samples are represented in a normalized fashion as well: z=-1 refers to the Z-near clipping plane, and z=+1 to the Z-far clipping plane. However, depth representation in the image format may be different, which is why the camera meta-data needs to define a mapping from stored values to OpenGL depth samples. For example, the OpenEXR Exporter Plugin of 3DStudio exports depth values as world-space distance from the camera plane, instead of depth values as an outcome of the pinhole camera imaging transformation.

4.2.3. Data Storage: GeoCast

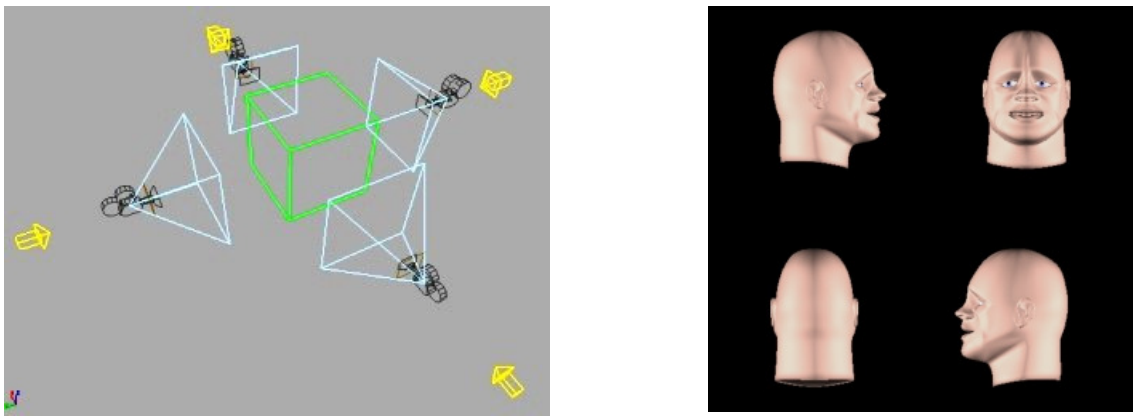


Figure 4.6: Data storage of multi-view camera setups as GeoCast meta-data and footage.
(a) Camera setup, as stored in the meta-data. (b) Footage from four camera views (depth data omitted).

For the later applications and as a proposal for 3DTV research, we defined GeoCast, a storage concept for footage and its associated meta-data [ZHMS05]. GeoCast footage encompasses video streams with all camera parameters necessary to re-project color and depth video in the way they were acquired, including camera position, bearing and imaging method. It can hold one or several camera views, and can thus put several camera views into the context of a common world space. The design goal is to simplify data exchange and ensure fast adoption by storing all the footage and its acquisition details as individual, machine-readable text and data files within a GeoCast scene. GeoCast data sets can thus be exchanged easily inbetween different parties, without any additional textual information that would require human interpretation. Figure 4.6 shows an example for the data contained in a GeoCast.representation: Camera position, orientation, imaging model, depth sample representation, and other camera parameters such as radial lens distortion. It also describes the naming convention for the footage's individual video frames and their storage format.

A GeoCast *stream* consists of one camera´s footage, and one or several GeoCast meta-data files which define the footage acquisition information in the previously described, OpenGL-inspired representation. These meta-data files hold the camera parameters for the original acquisition and consequently, through the camera/projector dualism, those for the footage's reprojection.

The GeoCast *scene* binds together all involved GeoCast streams, see Figure 4.7 for a graphical overview. It also contains all parameters that are valid for the entire footage, such as the number of frames in a multi-view video sequence. The scene does not specify camera positions; instead it refers to the GeoCast meta-data which accompanies each camera's footage.

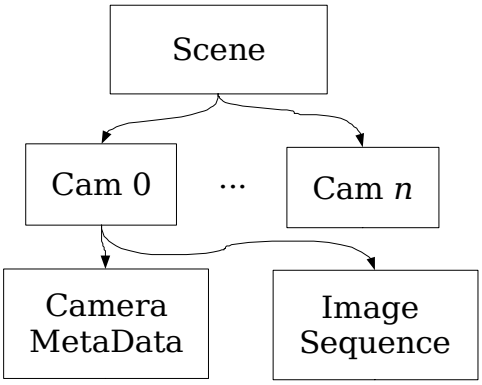


Figure 4.7: General structure of GeoCast file storage.

The footage of all cameras in a given GeoCast scene is held by GeoCast streams. Streams are stored as a sequence of files for each video frame, which makes random access possible. All GeoCast streams in a scene must have the same number of frames to simplify viewer implementation. The image file format for the frames is not specified, and the choice depends on the viewer and the acquisition application's I/O capabilities. We have used OpenEXR, a multi-layered floating point format introduced by Industrial Light & Magic [OpenEXR]. It stores float-valued RGB image data together with a depth channel. OpenEXR features intra-frame data compression, and its multi-layered nature permits additional, experimental data channels without endangering the basic color and depth data exchange.

Footage from the mentioned data sources usually requires conversion to fit the interfaces of image processing and computer vision software. We believe that GeoCast can simplify this data exchange. To exemplify this, we converted the Kung-Fu girl data set [BIG*03] and the Stanford CD data set [Sta] to GeoCast meta-data and can thus provide a *unified* camera setup description for multi-view and lightfield footage, as demonstrated in the GeoCast representation of a lightfield, Figure 4.5. A

Blender software plugin prototype has been developed to export synthetic GeoCast streams from 3D models, and thus to assist multi-view video research [GS04].

4.2.4. Basic Implementation

Depth projection aims to recreate the original surface in world space by reversing all the camera imaging operations. Before projection starts, intrinsic parameters such as lens distortion are accounted for via shaders on graphics hardware: the image's texture coordinates are warped such that the original lens distortion is compensated. Since we employ the OpenGL standard in our data storage, its projection and modelview matrices already fit the OpenGL API and can be simply inverted for the use in projection.

However, the representation of depth values in various input formats requires special treatment. For example, the OpenEXR Exporter Plugin of 3DStudio MAX exports depth values as world-space distance from the camera plane. This implies that the depth values delivered by this image format must *not* be transformed by the projection matrix; instead, they are injected into the eye coordinate vector that results from feeding image coordinates to the inversion of the projection matrix.

In low-level image-based rendering, depth information is used to warp video data, and thus create new viewing angles. In early approaches, pixels were shifted individually, creating ugly holes if the viewing angle deviated too strongly from the original depth axis [OBM00]. Post-processing can cover this, but not completely. Later, raytracing-like approaches were used to remove holes produced by displacement [POC05]. But intersection tests and preprocessing both slowed down the visualization process and make implementation cumbersome. Neither do we create a triangulation from the depth data like Parajola et al. [PSM04], which would require CPU preprocessing.

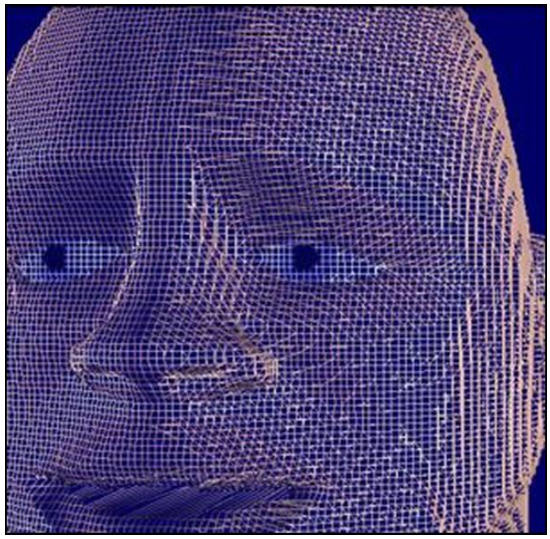


Figure 4.8: A quad mesh is used to connect all valid depth samples of a camera view.

With the GPU´s high-speed geometry processing and projective texturing [Eve02], a much more simple approach can be employed. We use a quad mesh of the same resolution as the incoming video's depth data and displace the mesh vertices by inverting the camera projection matrix. Doing this, we have effectively projected out depth data from the original depth camera position, see Figure 4.8 and 4.9.

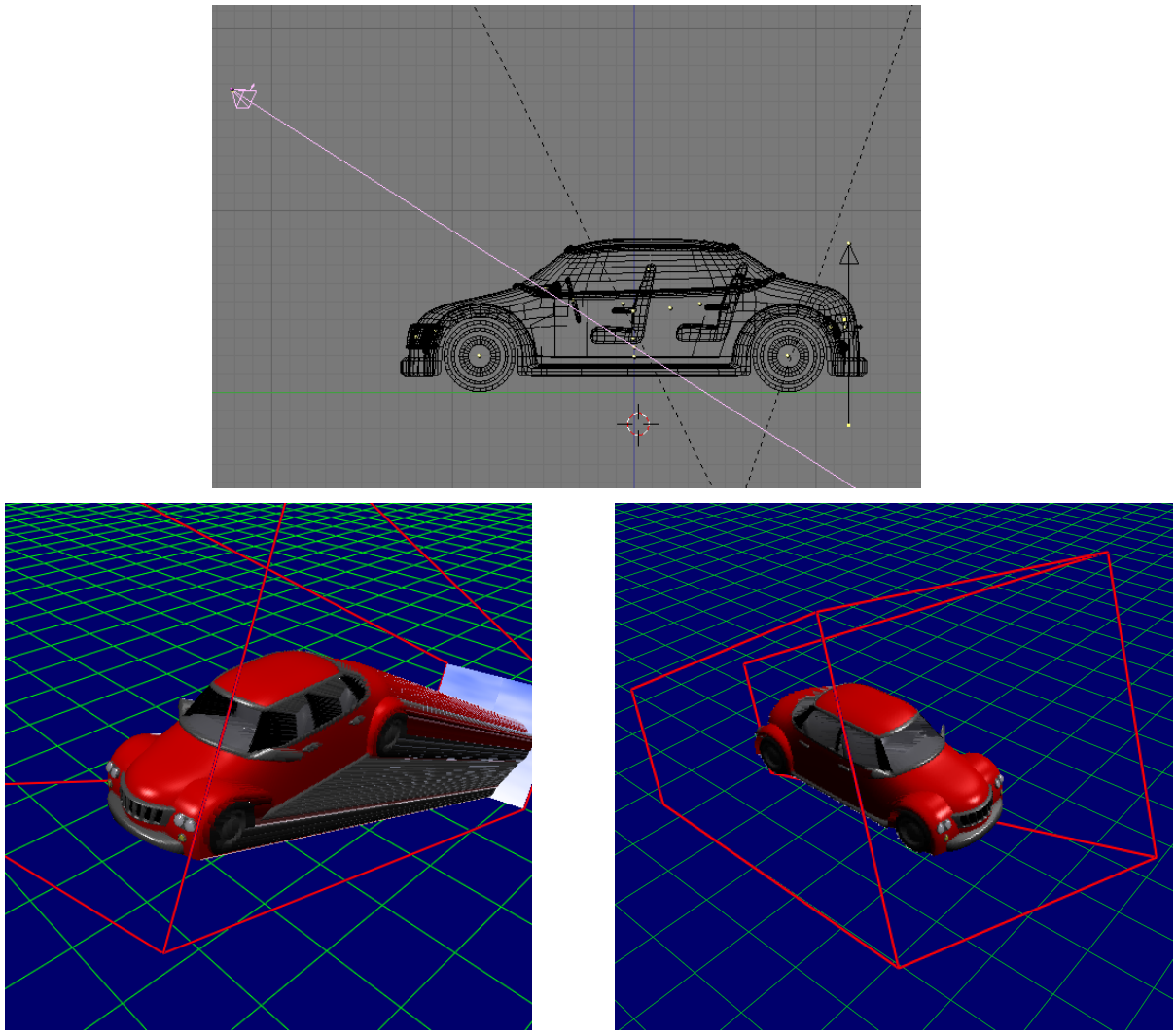


Figure 4.9: Visualization of a GeoCast stream, containing a camera sweep of a 3D model.
Top: 3D model and camera. Middle: A view without background clipping. Right: A view using Z-based clipping.

In our approach, we assume that adjacent depth samples are connected with each other, and thus *always* connect them with a quad, projecting out a connected quad mesh, such as in Figure 4.8. The only exception are depth samples that lie on the Z-far clipping plane, whose connecting quads are culled immediately. Figure 4.9 demonstrates the effect of Z-based clipping in two renderings of a car's 3D model, with the camera meta-data generated by our GeoCast output plugin for Blender [Geo04]. We used our GeoCast viewer to visualize the resulting footage on graphics hardware [GS04]. Rendering of this 256x256 GeoCast stream proceeds at interactive frames on NVIDIA GeForce 6800 hardware, providing free viewpoint choice while the image data is streamed from the harddisc and decompressed on the CPU.

4.2.5. Depth Sweep Accumulation

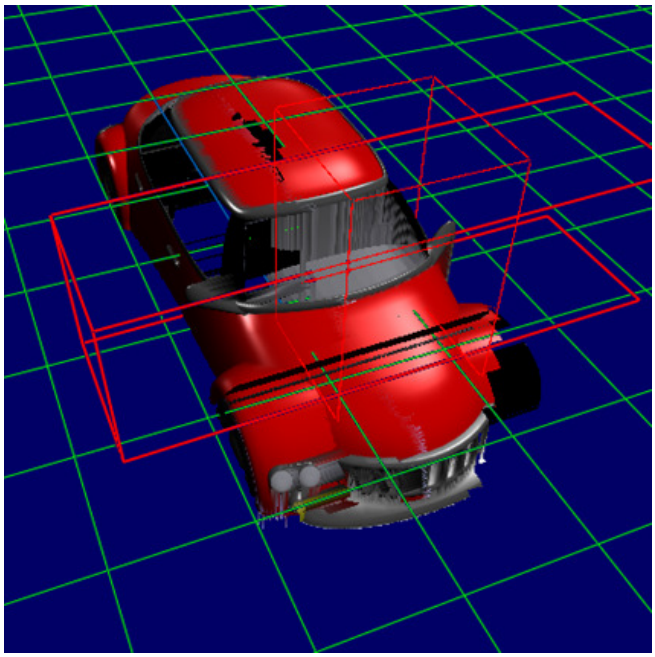


Figure 4.10: Sweeps of depth cameras can be accumulated to reconstruct complete novel views. Red lines: Current frustum positions of the two ortographic depth cameras.

In Figure 4.9, we have already shown how a depth projector can reproduce a depth camera sweep over the 3D model of a car, and how the novel viewpoint can be changed at any point in time. In Figure 4.10, we demonstrate a novel viewpoint reconstruction using *framebuffer accumulation.* Our input is a sweep of two synthetic depth cameras along the same 3D model of the car as before, but with both cameras only providing a very limited field of view of the car, at any given point in time. Thus, a single timeframe of the camera views would not suffice to imagine what the object looked like. However, we can provide an impression of the complete car by choosing a novel viewpoint still, and let the rendered depth projections from the two depth cameras *accumulate* over time in the framebuffer. Note that such an accumulation is only possible because the quad mesh of the depth projection is transformed and rasterized as a stream of finely tesselated graphics primitives, compare Figure 4.8, which contain a Z value that can be used to detect inter-camera occlusions (Z-buffer test). Please note how the accumulation has not been able to reconstruct the complete 3D object, since the camera sweeps have not been able to catch every angle of the input. Framebuffer accumulation is thus able to point out areas that have not yet been acquired, making it a useful tool for *view-planning* in depth acquisition .

4.2.6. CSG-assisted Novel Viewpoint Generation

Free Viewpoint Video playback requires the generation of novel viewpoints from multi-view video. In the following, we improve on FVV viewing of projected depth footage by using concepts from constructive solid geometry to generate novel views from several partial views of the same object.

While projection from several sources is well-defined, it is not completely determined how new virtual views shall be generated from the multi-view footage. Oliveira simply puts the partial view that is closest to the novel viewpoint on top of the other partial views [OBM00], a technique that we used to generate the novel viewpoint in Figure 4.11 to the right. However, this produces sudden changes as the novel viewpoint rotates around the object and the topmost partial view changes.

Other approaches, like Parajola et al. blend the partial views, with blending weights correponding to how close the depth projector's frustum is to the novel viewpoint's viewing direction [PSM04].

Figure 4.11: Real-time fusion of multi-view depth video.
(a) Camera setup, as stored in GeoCast meta-data. (b) The four camera views (depth data omitted).
(c) Images arranged as in the camera setup. (d) Final view, a real-time merger.

However, neither of the previous approaches made use of the depth samples that the transformed depth meshes of the depth projectors produce. Therefore, after experimenting with various methods for the fusion of partial views and mostly unsatisfactory results, we became interested in how differing depth contributions from partial views should be treated in the generation of novel viewpoints.

First, we need to define how the depth surface produced by a depth projector is rasterized. We already mentioned in our basic implementation in Section 4.2.4. that we assume all depth samples to be connected, except for the ones at the Z-far clipping plane. Note that this contrasts with Parajola et al. [PSM04] that use the heuristics of rubber sheet assumptions, implying that large depth discontinuities are very probably not connected at all.

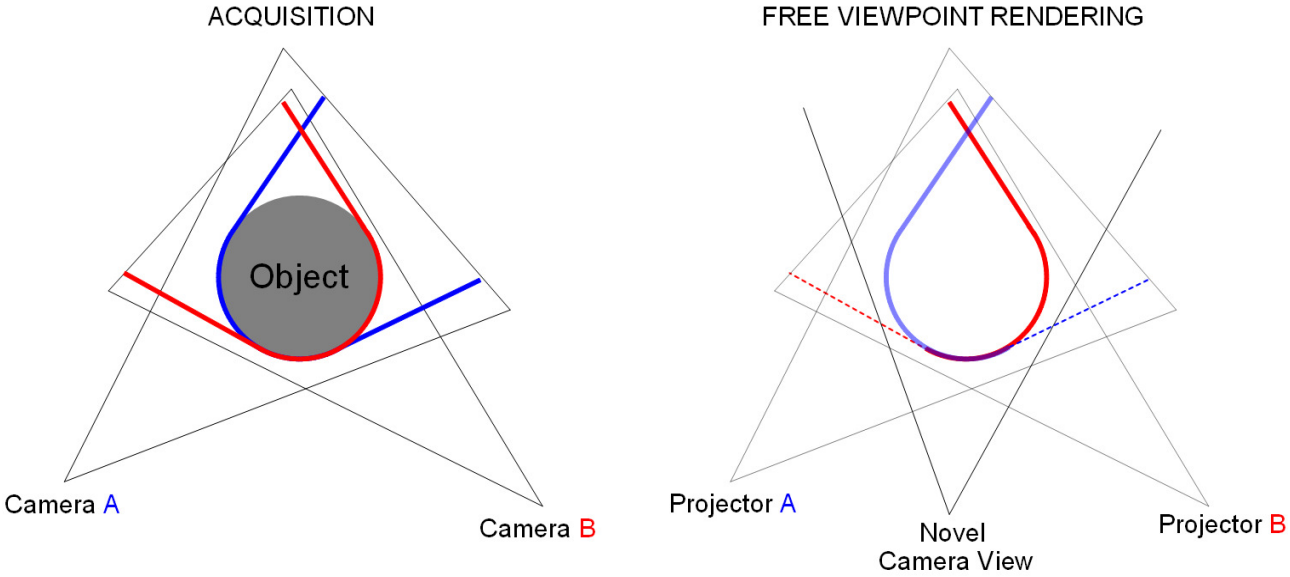


Figure 4.12: CSG-based novel view generation from two acquired depth views of a sphere.
Left: Acquisition from depth camera positions A and B. Right: CSG-based merge culls the dashed meshes from A (red) and B (now acting as depth view projectors) due to differing depth, while blending middle part (purple).

Next, we observe that the space between depth projection origin and the correspondent depth surface is *guaranteed* to be empty (if not, the surface color information would not have reached the camera sensor). As this observation holds for all camera views, there are cases when a connected surface is *not* correct for the novel view: Every time a second camera view provides insight into areas that were occluded from the first camera view, we have to *break up* the connectivity

assumption. Figure 4.12 provides an example, where two depth cameras have acquired their views of a sphere. Both of these depth views are of course incomplete, and the outermost corners of their acquired depth surfaces stretch back to the far-Z clipping plane. Now, if a novel view is generated from inbetween the two depth camera positions, then the two linear prolongations of their depth surfaces to the far-Z clipping plane must be *culled* to provide a more accurate reproduction of the original spherical object. Note that this connectivity culling might as well occur inbetween depth samples that are *not* connected to the Z-far clipping plane, e.g. whenever the acquired object has a cavity that was occluded in one of the camera views.

This insight allows us to improve rendering of the novel viewpoint. However, we do not attempt to generate a visual hull-based mesh, as the required preprocessing would not allow for interactive frame rates from raw depth-video footage.

Instead, we continue using the *rasterized, transformed* depth meshes of all participating camera views to generate a novel view of the object with an improved impression of the original object surface. To achieve this, we first determine the participating camera footage for a given view upon the scene. In Figure 4.12, this was trivial, but for the animated head in Figure 4.11 (c), two camera views participate in the novel view to the right: the frontal, and the left view, which shows in Figure 4.11 (b). Then, we transform and rasterize all participating camera views independently, much like Oliveira et al. [OBM00], but preserve color *and depth values* from all partial views (in the implementation, this is achieved by rendering to textures). In previous approaches, the partial views would now be blended together, with blending weights often globally chosen for the whole view, depending on viewing angle. In our approach, however, we execute a fragment shader for every pixel of the novel view, and provide it with the rasterized output from all contributing partial views, in the form of textures. For every output pixel, the shader now samples all partial views for their color and depth contribution. If only one view has valid content to contribute, it is passed through directly. If all contributing views have the same depth value to provide, then they are blended together. In all other cases, the participating views compete for providing the *farthest view*. In our example in Figure 4.12, blending occurs in the purple, middle part, as both input views contribute with the same depth. In the left and right zone, however, only one input's contribution remains, while the other input's contribution is ignored by the fragment shader due to different depth values (dashed lines hint at the ignored contributions from either view).

Figure 4.13: CSG methods in novel view rendering.
Left: Simple Blending/Overlay of two camera views (akin to relief texturing approaches).
Middle: CSG-based rendering. Right: Per-pixel choices of input views (Blue: only left view, Red: only right view)

Figure 4.13 demonstrates a result, with a comparison to previous approaches: To the left, a globally blended composition of two partial views. Several regions are problematic: For example, the

contribution from the side view (compare Figure 4.6) did not acquire any of the beret's brim.But despite its warped depth surface not contributing to the beret's brim, it is *still* blended into its rendering in the novel output view. In the middle, our new approach: Especially the rendering of the eye and the beret has improved substantially. To the right of Figure 4.13, color codes indicate the input views that have been chosen for the individual pixels of the final novel view. It clearly shows that the frontal view (red) dominates the novel view's rendering of the beret's brim and the eye.

Note that our algorithm cannot be implemented on triangle level, i.e. by filtering irrelevant triangles *before* rasterization, as it is not known if some of their rasterized fragments might still be needed. A triangle filtering in preprocessing would also require a comparison with $O(n^2)$ complexity, or a spatial data structure that reduces this comparison complexity. Since such a spatial data structure does not exist for raw depth video footage, we believe that the spatial arrangement and subsequent per-pixel comparison of our approach is faster than any complex preprocessing.

4.2.7. Future Work

In the future, we would like to investigate novel viewpoint generation from a large number of depth projections, e.g. to represent complex objects or scenes. One approach to efficient surface representation would be PolyCube-maps, which define individual, ortographic 2D planes surrounding the object [THC*04], that allow for more complex 3D object surfaces that cannot be represented in a single depth camera view. In this context, camera meta-data could also specify the footage's visibility in the scene to improve rendering speeds. We have, for example, already started investigating how a *viewing cone* for depth video could allow early culling in the rendering process, and thus improve the rendering of complex scenes with a multitude of depth projectors.

Beyond depth video projection, we would be interested in how geometry images could be projected [GGH02]. The footage input would thus provide explicit XYZ coordinates for every pixel instead of only a depth value. For projection, the modelview matrix would still set out the projected surface's 3D coordinate system, and even the projection matrix can be applied, if necessary. As a default, a quad mesh would connect all pixels of the footage input. Holes and discontinuities could be implemented by “invalid” footage pixels that would avoid the rendering of a quad connecting the pixels, and thus separate independent regions. The approach could e.g. be used to store multiple depth layers in a single GeoCast stream.

4.3. Conclusion

In this chapter, we have investigated how GPU acceleration of color and depth projection can enable new applications of the camera/projector dualism. Section 4.1 investigated the use of color reprojection from camera images onto scene geometry as a tool for fine-calibration of camera parameters, and presents multi-camera calibration results which would very probably not have been possible to achieve with an automatic algorithm, while still only taking minutes for an experienced user with the immediate feedback from GPU scene renderings. In Section 4.2, we explored the concept of depth projection. After defining the concept and introducing a data format for depth video storage (GeoCast), we described how raw depth video footage can be rendered immediately using quad meshes, and provided several renderings of novel viewpoints from synthetic depth footage generated by our software plugins for 3D modelling software. Later, we observed that depth camera sweeps can be accumulated in the framebuffer via Z-buffer testing, and developed a new rendering algorithm for a depth-dependent composition of novel viewpoints from partial views. Using synthetic data, we have thus provided an outlook on issues that novel viewpoint generation from multi-view depth video will need to deal with as depth cameras become commonplace in the future.

5. Hierarchical GPU Image processing

Even in image processing, we wanted established graphics hardware concepts to play a central role, and to ensure that performance converged towards interactive levels in the near future. To achieve this, we took note of crucial differences between CPU and GPU image processing: for example, projective texturing is computationally inexpensive, and so is bilinear interpolation. In contrast, serial dependencies in the algorithm severely limit graphics hardware performance, making scanline based approaches from CPU implementations unattractive. Redundant computation is not as computationally expensive for graphics hardware as any serial dependencies that reduce parallelization. The data-parallel architecture of graphics hardware also favors hierarchical approaches which allow adaptive refinement of initially coarse results, but nonetheless retain a regular data arrangement for each computation pass.

Plane-sweep algorithms are therefore a good approach to stereo reconstruction on graphics hardware, and form the first part of this chapter. We start by introducing our first approach to stereo reconstruction with mipmap-based plane sweeps in section 5.1.2, and demonstrate previously unpublished results on how the camera/projector dualism significantly simplifies depth reconstruction from multi-view image sets such as acquired by the Stanford camera array, in section 5.1.3. Section 5.1.4 summarizes the refined version of our stereo reconstruction algorithm based on plane sweeps over image stacks, as documented in the diploma thesis of Heidenreich [Hei07].

The latter part of the chapter demonstrates further novel uses for hierarchical image processing. Section 5.2 presents a data-parallel algorithm for hierarchical feature clustering, enabling stable tracking of a large feature region at interactive framerates on graphics hardware. Two applications of the feature clustering algorithm have been published in the bachelor theses of Arnold [Arn07] and Schilz [Sch07]. Section 5.3. introduces previously unpublished work on reduction-based histogram generation from 2D data arrays. While the use of histograms in image color analysis is widely known, we can demonstrate a novel utilization in semi-automatic lens compensation.

5.1. Depth Reconstruction via Plane Sweep

While the previous chapter showed that virtual viewpoints could be reconstructed from multi-view color/depth video footage in real-time, all the input material was of synthetic origin - at the time of investigation, sufficiently calibrated multi-view color/depth footage was not easily available. First prototypes of depth cameras, such as the ZCam [3DV], did already exist, but their depth output had only low resolution and was too noisy to be usable in scene reconstruction.

For these reasons, it was natural to continue research towards depth reconstruction from images. Even in depth reconstruction, we wanted established graphics hardware concepts to play a central role. One thing we observed is that the GPU's texturing capabilities favor any reconstruction approach that uses *projective techniques*, since image resampling is hardware-accelerated, and projective matrix calculation is a common element of real-time graphics. This led us to the investigation of plane-sweep methods.

As a first step towards real-time depth reconstruction, we focused on depth reconstruction from stereo images and on exploiting the images' correlation to the maximum. Scan-line based algorithms would be a common approach to calculate image disparity on the CPU, but they introduce serial dependencies and require the stereo images to be rectified along epipolar lines. Also, depth values have to be reconstructed from image disparity in an extra pass after image correlation has finished. However, our original purpose for the depth reconstruction is the

reprojection of depth data from the point of acquisition into the original 3D scene. Plane-sweep based algorithms use such reprojection already *during* depth reconstruction, and thus ensure that the results can be reprojected into a 3D scene at a later time. The required software framework for plane-sweep based depth reconstruction also makes it easier to assess the depth results in 3D visualization, see Figure 5.1.

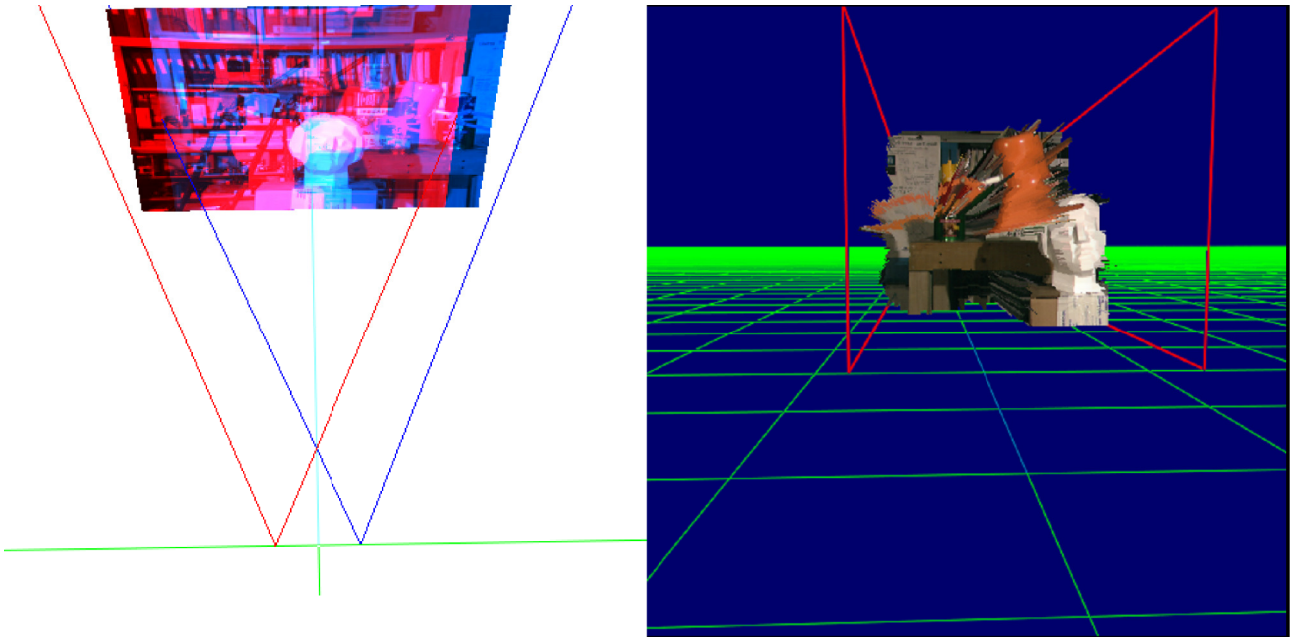


Figure 5.1: Usage of GeoCast in stereo vision.
Left: GeoCast meta-data defines camera setups for image reprojection in depth reconstruction (red lines and image: left camera/projector image, blue lines and image: right camera/projector image).
Right: Visualizing a part of the reconstructed disparity map as depth surface in world space.

Depth reconstruction from images uses two or several images of the same scene, taken from different camera angles, to reconstruct depth maps. The plane-sweep approach reconstructs depth values by projecting camera images from their original viewpoints onto a *depth hypothesis*, a hypothetical geometry which represents the yet unknown depth map. This depth hypothesis is iteratively refined until it matches both camera images on its surface, and thus very probably corresponds to the original object surface that the cameras had seen, see Figure 5.4. One can also say that comparing two image projections on a depth surface, while moving the surface along the Z-axis, corresponds to the comparison along epipolar lines in scan-line based algorithms (where projection has been accounted for in the image rectification beforehand).

While examining the problem, we observed that the camera/projector dualism from Chapter 4 can be used in unusual ways: While it is obvious that the camera images are projected back into the scene during reconstruction, even the depth hypothesis itself can be interpreted as a *depth map that is projected out from a camera view* - and thus allows use of the techniques in Section 4.2.

The question is how many parts of a plane sweep algorithm can be implemented on graphics hardware. Obviously, the GPU can assist with image projection computations; but with nowadays' programmability, even depth hypothesis management and the intermediate match evaluation can be implemented on graphics hardware - as long as the task remains *regular*. While a high depth map resolution maintains this regularity, it would be too computationally expensive to evaluate all possible depth values in a depth map of the final resolution. Instead, we pursue a coarse-to-fine approach to depth reconstruction that remains regular at all times, but which forwards coarser depth

map results to the finer resolution levels. This limits the search range at finer resolution levels, which counters reconstruction noise and reduces the computational burden.

5.1.1. Related work

We were inspired by the works of Henkel et al. on depth reconstruction with neural networks that simulate human vision [Hen97, Hen00]. In these works, Henkel describes how simulated neurons can be connected in disparity stacks to attempt image matching at several disparity levels *and* resolution levels simultaneously. His neural network then outputs the resulting depth values as a series of pulses. If these pulses remain stable, a conclusive solution has been found. For optical illusions, the neural net behaves the same way as a human's visual system for input that is ambigous - it switches between several unstable states. The presented examples in his work demonstrate how half-transparent surfaces cannot yield a single, disambigous depth value per image point, and how occlusions generate regions with highly unclear depth values. It also introduces the *cyclopean view,* a virtual image placed exactly inbetween the stereo image pair. This cyclopean view holds the most image samples common to both views, and thus is a good choice for depth reconstruction. The depth map associated with this cyclopean view therefore corresponds to the depth hypothesis in our algorithm. We also made use of the concept of coarse-level disparity stacks that are only fed with *pre-filtered image input*, instead of aggregating a sum of differences from individual pixel values over larger areas. This led to our idea of pre-computing filtered image output to quickly compute coarse depth estimates, before further depth refinement occurs. In our implementation on graphics hardware, we achieve iterative refinement of depth data via pyramidal data structures that are commonly used in image processing and computer vision [Tan75, Ros84].

Section 5.1.2 and 5.1.3 use mipmap-based prefiltering even for the input images in order to generate fast approximations of larger support windows, which is also an aspect of Ruigang and Pollefeys' approach [YP05]. To improve on the reconstruction of the coarser depth resolutions, Section 5.1.4 replaces mipmaps with recurvise box filtering for input image prefiltering, generating full resolution image stacks of varying filter convolution sizes.

A further inspiration was Jones and Malik, which argue for the use of multi-scale image input in stereo reconstruction [JM92]. In other related work, the approach of Woetzel et al. is particularly interesting for its lens compensation and quite GPU-centric multi-camera implementation that utilizes only as many cameras as necessary [WK04]. It still lacks a hierarchical approach however.

5.1.2. Mipmap-Based Plane Sweep

Our first implementation of plane-sweep on graphics hardware follows the iterative mesh tesselation and depth refinement by Zach et al.[ZKHK03]. Zach et al. use a mipmap to solve for a disparity map in a coarse-to-fine manner, see also Figure 5.3, and we make use of the same. However, our algorithms differ in several other details: While Zach et al. assume that their input images are rectified, we let our algorithm compensate for lens distortion in a pre-warp process on graphics hardware, as demonstrated in Figure 5.27. We also solve directly for a depth map instead of a disparity map, using 3D reprojection instead of a pure 2D matching process. The last major difference is that we don´t use the CPU to evaluate the intermediate results and update vertex positions in the depth mesh. Instead, we let a fragment shader write intermediate results from depth reconstruction to a texture, the intermediate depth map, which is *directly* used in the next depth refinement pass. For every pass, a vertex shader generates the mesh positions of the next, more refined depth hypothesis from the provided intermediate depth map, a technique resembling the depth surface projection in Section 4.2.

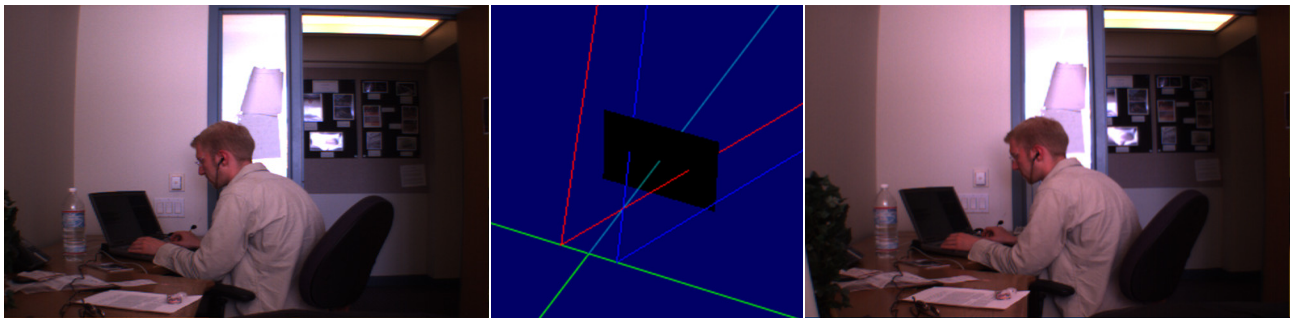


Figure 5.2: Left and right: Input images (a slightly unfavorable stereo view of the author). Middle: the camera view frustums in red and blue, with the initial depth hypothesis in black.

The algorithm proceeds as follows: First, a pair of images and their camera parameters are retrieved from the input source, in our case the Bumblebee IEEE stereo camera by PointGrey, which delivers pairs of 640x480 images and has a fixed stereo calibration. These images are first warped in a fragment shader to account for lens distortion, as demonstrated in Figure 5.27.

Then, a mesh of quads is set up for the depth hypothesis at various resolution levels, see Figure 5.3. This mesh acts as a proxy for the depth surface that we plan to reconstruct, and is projected from a *virtual depth camera*, which can share its frustum with the left or right input views, or be totally independent from them. One good choice is the cyclopean view, which sits inbetween the two stereo camera positions, and is a good alternative frustum for the depth values, as it evenly compromises between occlusions of the left and the right view.

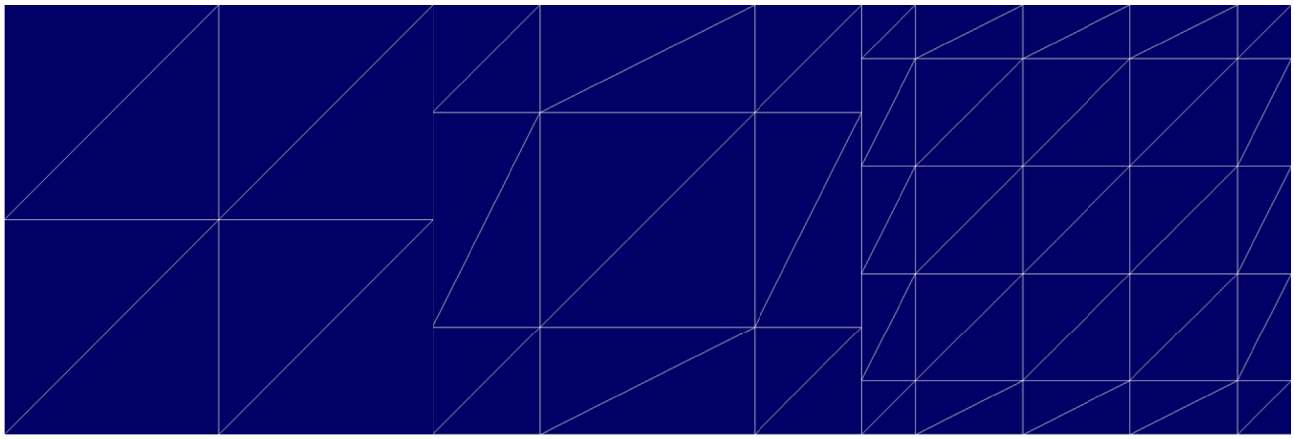


Figure 5.3: Iterative refinement of the depth hypothesis, for tutorial purposes from 1x1 to 2x2 and 3x3. Note the special border tesselation, required for coarse-to-fine depth hypothesis propagation.

Now, to reconstruct depth, the depth hypothesis slowly iterates through all possible depth values, while both images are projected on it. This is implemented by displacing the vertices of a quad mesh in a vertex shader, see Figure 5.4 for a visualization. The mesh of depth hypothesis, initially a plane, thus *sweeps* through the search space of possible depth values.

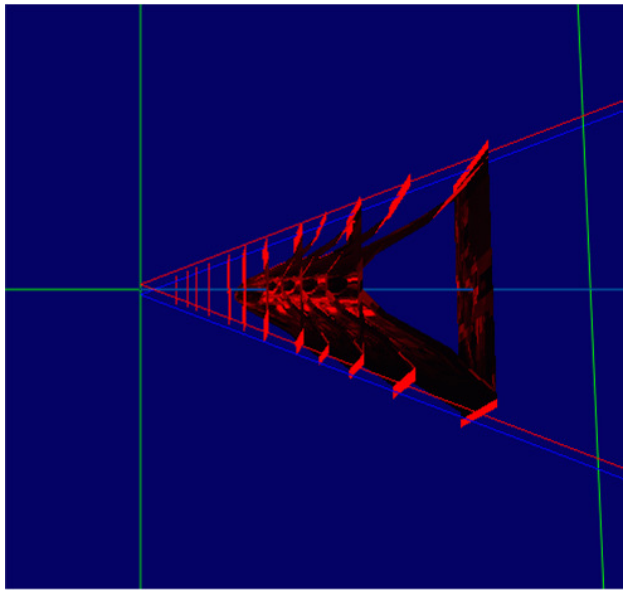


Figure 5.4: Evolution of a depth hypothesis, shown as red surface.
Blue and red lines: left and right camera view extents.

Figure 5.4 demonstrates a bird eye's view of the scene, where the depth map is placed into a 3D scene. But this is not the view in which actual depth evaluation takes place. Instead, the actual evaluation is performed in the depth view, with the depth camera as primary view (in Figure 5.2, the depth camera view corresponds to the black area set inbetween the two stereo views). In this view, a match texture acts as render target for the pixel rasterization of the quad mesh. To implement the projection of the camera image onto the depth hypothesis, a fragment shader is called for each rasterized pixel, which fetches samples from both input images via projective texturing [Eve02].

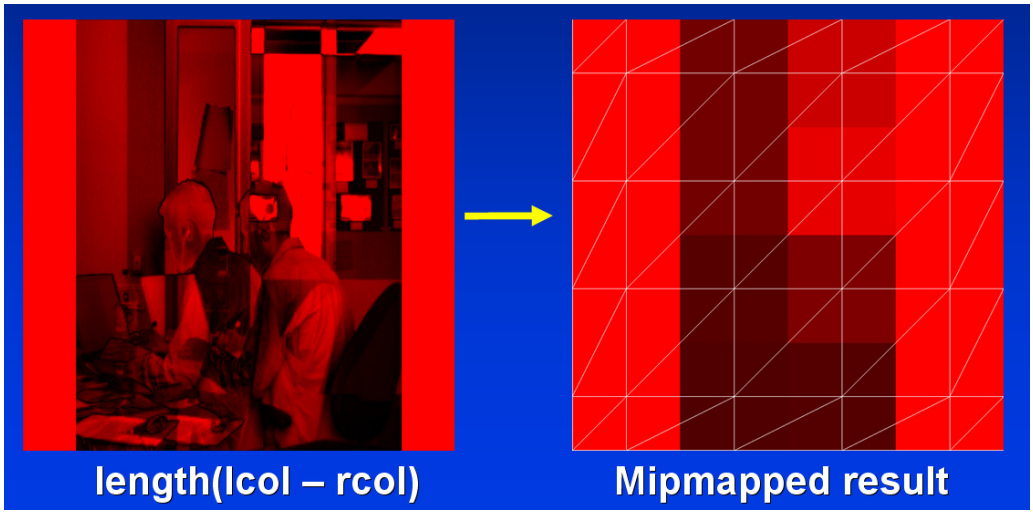


Figure 5.5: Depth error feedback via an SAD-based match texture.
Left: SAD evaluation of stereo input into match texture,
via difference between left and right image color vector (lcol, rcol).
Right: Reduction of match texture to quad mesh resolution for depth evaluation.

At every texel of the match texture, the sum of absolute differences (SAD) of the two image projections is calculated, using the builtin shader function `length()` on the difference of the input image color vectors `lcol` and `rcol`. Note that the match texture oversamples the quad mesh resolution to minimize outliers in the forthcoming depth evaluation, see also Figure 5.5. For

example, if the quad mesh has a resolution of 4x4 depth displacements, the match texture could have a resolution of 16x16 texels, always depending on the oversampling factor.

But this oversampling is not using the stereo image's resolution: In order to accelerate computation, it is the stereo images' *mipmaps* that are used to compare the two pixel areas, i.e. the texture lookup for `lcol` and `rcol` uses a mipmap level of the left and right input image that corresponds to the current resolution of the match texture. After the SAD evaluation for each texel has completed, the match texture is downsampled to quad mesh resolution, using a reduction operator which simply adds up the input values. This local sum of SAD becomes the *match score of the current depth hypothesis*, stemming from the SADs of stereo image samples that surrounded the actual quad mesh displacement. The following fragment shader compares this current match score to the previously stored match scores for this location. If the incoming score is smaller, i.e. the current depth hypothesis had produced a better local match, then the previously best match and its associated depth value are replaced. This is repeated until all depth value offsets have been evaluated.

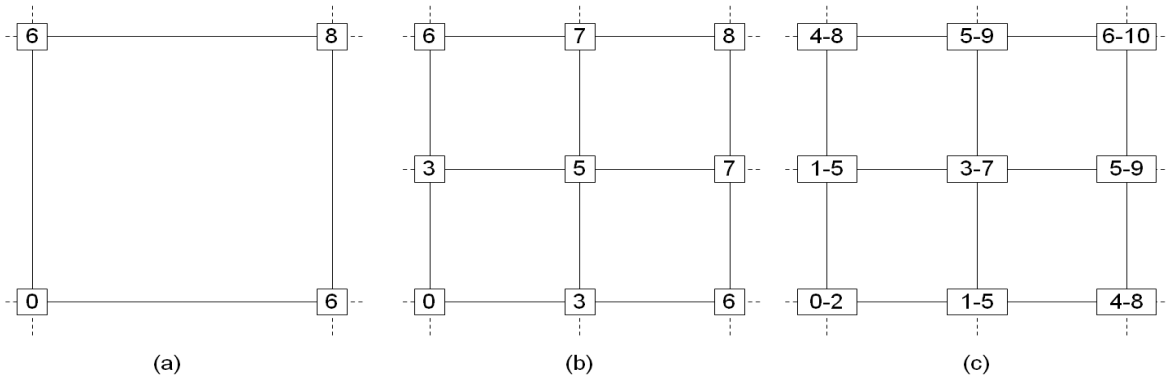


Figure 5.6: Preparation of next depth hypothesis.
Left: Previous, coarser depth result. Middle: New depth values in mesh via bilinear interpolation.
Right: A new, smaller depth search offset range (±2) determines the potential values of the new depth hypothesis.

The next step is depth hypothesis refinement, which implies that the previous depth values will be used as base for a more fine-grained depth hypothesis, see Figure 5.6 (a).First, the old depth values must be upsampled to serve as the new depth base values. In practice, we use bilinear texture interpolation to generate appropiate depth base values, see Figure 5.6 (b). Then, quad mesh resolution is increased for the new displacement positions. Following these preparations, depth hypothesis evaluation can proceed again, but this time with a *more restricted* depth search range, and centered around the depth results from the previous iteration. Thus, depth values from the coarser depth evaluation are now regarded as the base solution, and all new depth evaluations are *offset* from this base, , see Figure 5.6 (c). During evaluation of every iteration, we thus have a depth base value (stemming from the previous iteration) and a depth offset which is varied, and these are combined to the current depth hypothesis which is tested via image matching. The reason for this progressive reduction of depth search range is that due the increasing resolution of the match texture, SAD evaluation will cover increasingly smaller areas. While a smaller coverage area might make depth evaluation more sensitive to details in the stereo images, local image noise may cause sudden SAD fluctuations at the same time. Therefore, the smaller evaluation area of the SADs is compensated by reducing the search range for the depth offset, causing fewer spurious wide-disparity matches between unrelated image noise at wide-apart image locations.

5.1.2.1. Results

Our first results were computed on a Dell Precision M70 laptop with Nvidia Quadro FX Go 1400 and 256 MB video memory, connected over PCI Express. It contained an Intel Pentium M (2.13 Ghz) and 2 GB of main memory. The stereo input itself was provided by a PointGrey Bumblebee

stereo camera, connected over an IEEE 1394 Firewire bus. The camera's software provided calibration data to compensate for lens distortion in both cameras. We used the calibration information to compensate the radial lens distortion of the input images via a dependent texture lookup in a fragment shader. For the actual reconstruction, the depth view was placed in the center of the two input views, as already shown in Figure 5.2.

Figure 5.7: Depth map reconstructed from stereo views.

Figure 5.7 shows the depth reconstruction from a stereo view shown in Figure 5.2. Problems remain in the transparent area of the room entrance, but already Henkel states that even a human visual system cannot reliably determine one single depth value in such areas of transparency. Instead, it would constantly switch between the two stable states [Hen00]. Thus, with the assumption of only one depth value for every location, including an increasing restriction of the depth value range during refinement, the system cannot possibly reconstruct a correct value in these cases.

5.1.3. Multi-View Depth Reconstruction

Since our just presented stereo reconstruction also relied on re-projection, we combined the two concepts into a solution for depth reconstruction from multi-view image sets. In this reconstruction approach, we use *several* camera views of the same scene to reconstruct depth maps. Such data input is not very common, at least not with an inter-camera calibration, but e.g. provided by the Stanford camera array [WJV*05].

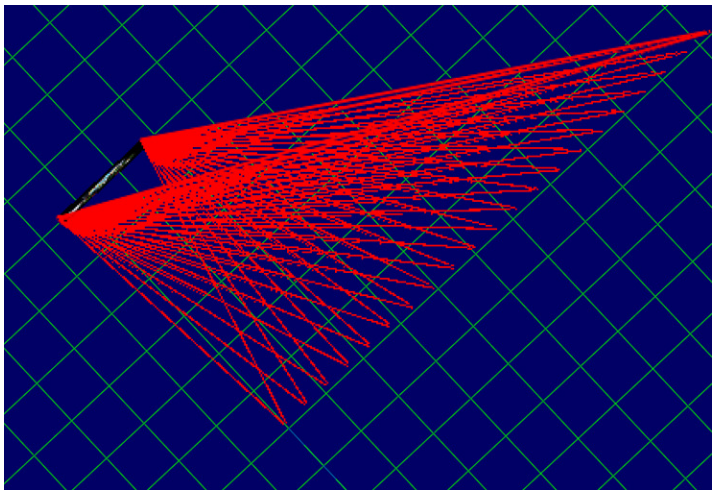


Figure 5.8: Reprojection of a lightfield can be implemented as a multitude of sheared projections from multiple camera views (in red).

Depth reconstruction resembles the previously presented reconstruction from stereo images in many ways, but now has to accommodate multiple camera views. In section 4.2.2. of the previous chapter on GPU-based camera/projector dualism, we already established that the reprojection of lightfield data, in the form delivered by the Stanford camera array, can be expressed as multiple 2D reprojections with non-symmetrical frusta. With this insight, the Stanford camera array's image sets can be reprojected on a depth hypothesis. In Figure 5.8, the depth hypothesis appears as black surface to the upper left, while the camera frusta origins are situated at the origin of the red lines, to the lower right.

Multiple image input to the depth hypothesis has several consequences. The major difference lies in the depth match evaluation: As a number of n views are involved, there are also $n^2/2$ possible sum of area differences (SAD) that could be computed between any two particular image projections from the lightfield. Since a *single* matching score is aftersought for depth reconstruction, we settled for minimizing the sum of SADs from the common *input color average*. This input color average is computed by averaging the projections of all contributing camera views on the depth hypothesis. A *contributing* camera view is defined as having a *valid* projection onto the depth hypothesis at a given surface point. Invalid projections are surface points outside the camera's frustum, or projection samples which were *ignored*, either due to low exposure, or because they were considered irrelevant to the depth map (e.g. background samples).

The meaning of reconstruction confidence also had to accommodate multiple views. While the final match score is a good indicator for reconstruction quality, it is also important to know how *many* views contributed to the matching score. We thus created two indicators for reconstruction confidence: Match score (sum of SADs from input color average), and number of contributing camera views.

In initial experiments, we used an orthogonal projection model for the depth view, resulting in a linear depth representation which subdivided the search space for depth evenly and allowed for easier debugging and tuning of depth hypothesis evolution and its parameters. Unfortunately, this linear depth representation doesn't properly reflect the non-linear relation between world-space depth differences and the disparity in the camera images, as introduced by the perspective imaging model of the cameras. The discrepancy resulted in insufficient depth resolution of the depth view in near depth values, while it over-resolved far away depth values, and made a proper parameter choice for increasing restriction of the depth search space impossible. To resolve this, we settled for depth views with a perspective imaging model, and a similar field of view as the original cameras.

A last issue was the detection of contributing camera views for the depth hypothesis in the presence of image noise. As depth reconstruction refines, SAD support regions become smaller, and if the remaining local image information is only noise, then random matches in the depth evaluation occur, leading to random noise in the depth value refinement. A camera view's color contribution should thus be regarded as invalid during fine-grained depth reconstruction if the noise level of its color contribution is too high. To detect this case, it is necessary to segment the image data from camera views into usable and noisy parts.

Figure 5.9: Image segmentation by color threshold, ignored image areas in bright blue.

One candidate for a noisy region could be input view regions with low light exposure, such as the background curtains in the input material, see Figure 5.9. In general, such a noise-based segmentation can be achieved in various ways; in our particular application case of high-speed reconsutrction, we used a color threshold, designating very dark color values in the input material as regions where depth refinement should not take place. In Figure 5.9, ignored regions of the input material are marked by a bright blue color.

5.1.3.1. Result

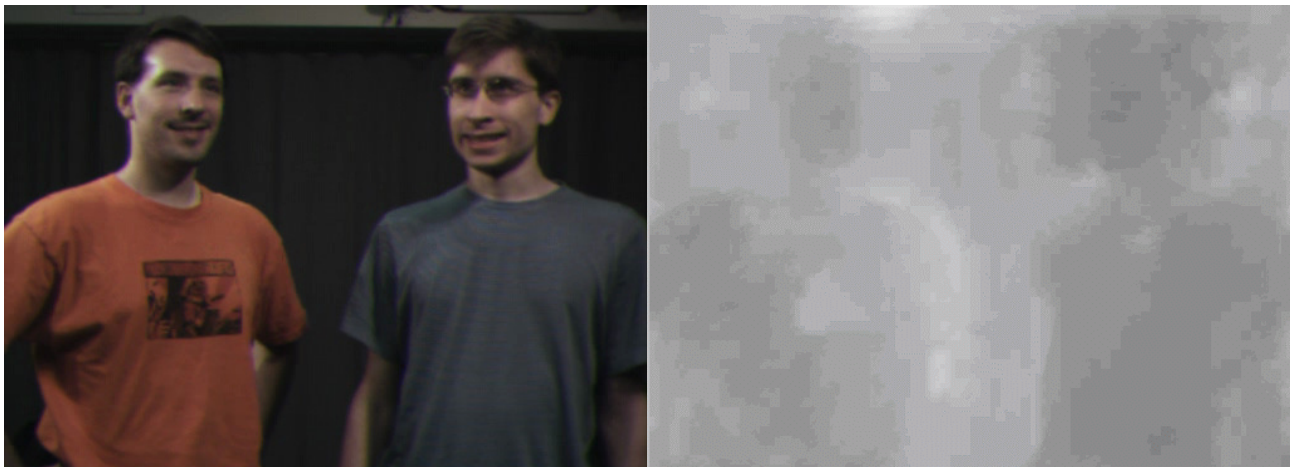


Figure 5.10: Depth reconstruction from a multi-view image set produced by the Stanford camera array. Left: One of the input image as projected upon implicit depth surface. Right: Reconstructed depth map.

Results from this approach were promising, as Figure 5.10 and 5.11 shows, with the depth image using darker color for values closer to the camera. Note that the left image does not depict one of the images directly, but shows one of the inputs as projected upon the final depth surfaces. By projecting the input images onto the final depth surface, the user can verify that the depth reconstruction produced the desired result of a match of image projections on the surface.

Our noise segmentation mostly led to the desired result: The initial depth estimates, which still were based on large SAD support regions that *also* included *valid* image samples, remained stable throughout the refinement, and provided depth continuity for regions where no further details were present. The technique thus provided a *plausible depth guess* from adjacent image input, instead of introducing random noise into this region's depth values. One exception to the good impression is

the depth map of the T-shirt's logo worn by the left individual. We suspect that erroneous matching with the incompletely noise-segmented background took place. Otherwise, the depth image corresponds roughly to how a human would perceive the scene.

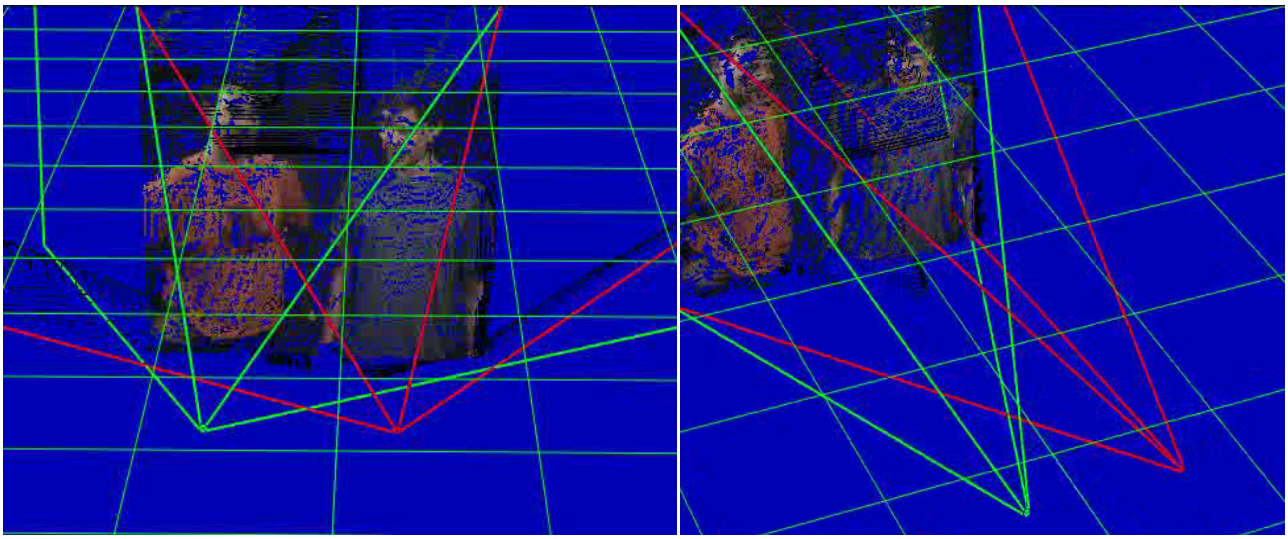


Figure 5.11: Point cloud visualization of final depth surface, using one of the input views for color. The red and green lines mark depth view frustum (green), and one of the input view frustums (red).

5.1.3.2. Remaining issues

While experimenting with different choices of input camera views and depth views, we encountered a strange problem: If the depth view was too far from the center of all included camera views, depth estimation would not converge to reasonable depth maps as the one in Figure 5.10. At first, we blamed this on occlusion issues, but found later that the likely cause was rooted in the mipmap-reduction of the input images.

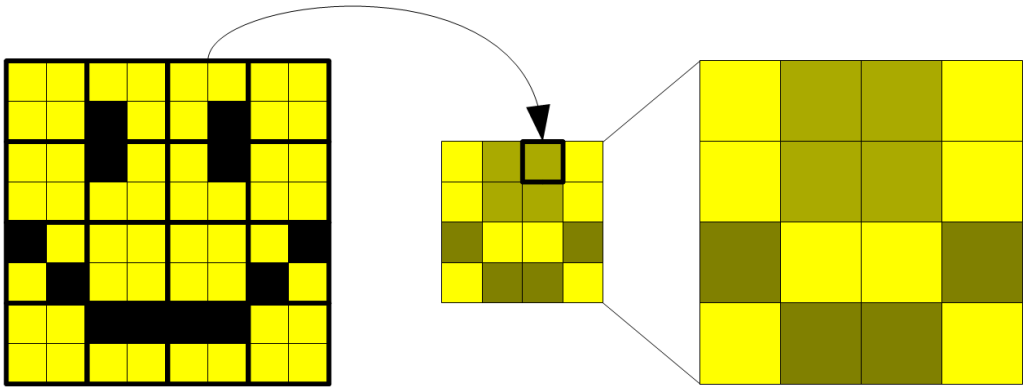


Figure 5.12: Mipmaps can be used like averaging windows at a fixed grid spacing.

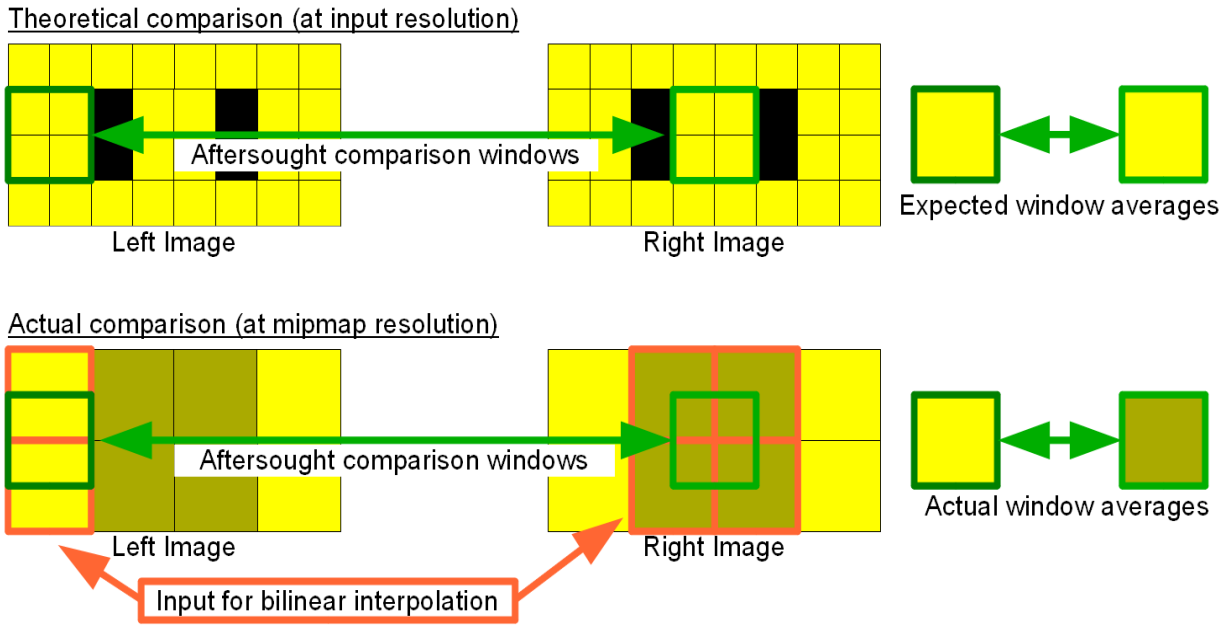


Figure 5.13: Comparisons of averaging windows outside the mipmap grid can fail despite bilinear interpolation.

We had used mipmaps to accelerate SAD computations for the larger support regions of the coarser depth estimates, see Figure 5.12. During depth estimation, these image mipmaps are accessed at the original resolution of the input image, to yield an average of the input image pixels in this area. In graphics, a mipmap is actually used to approximate a box filter convolution. But the problem lies in this approximation: While some positions in the image, the ones that remained the centers of mipmap reduction, yield the *exact* results that a box filter would generate, the texture unit must *fill in* missing results for all other positions, using bilinear interpolation during texture lookup. Unfortunately, for strongly varying input areas, depth estimation actually requires the *exact* results of a box filter convolution at *every* image position and not an approximation based on bilinear interpolation, see Figure 5.13. Interestingly, as long as the depth view lies in the center of the participating camera views, this mipmap sampling issue will evenly affect all input views. But if the depth view lies off-center, then some of input views' mipmaps are more often estimated that others, leading to biased SAD calculations. This insight led us to replace mipmaps with full resolution, mean-filtered image stacks, as the next section will show.

5.1.4. Coarse-To-Fine Plane Sweep Reconstruction

In order to improve depth matching accuracy even for larger SAD support regions, we decided to replace mipmaps with a recursive box filtering approach. We now maintain the input image resolution even as the support regions for the box filter increase, in contrast to the mipmap-like reduction of our previous approach that only created *approximations* to a box filter operation, as already mentioned by Yang et al. [YP05]. We can thus attain a *complete* filter result for the input image instead of the approximation that a bilinearly interpolated mipmap delivers. While such a box-filter convolution usually becomes quickly expensive for increasing filter support regions, computational cost can be contained by the *recursive application* of filtering operation which re-uses results from smaller support regions. For example, the first iteration of the filter would generate a 3x3 box filter convolution for every pixel of the input images. By re-using these 3x3 filter results, a 9x9 box filter convolution can be generated from 9 lookups into the 3x3 filter result, instead of the 81 lookups that would be necessary if the 9x9 filter convolution was generated from the original image. This way, mean filtering at full image resolution can even be used in real-time image processing, see the upper row of Figure 5.14 for an illustration.

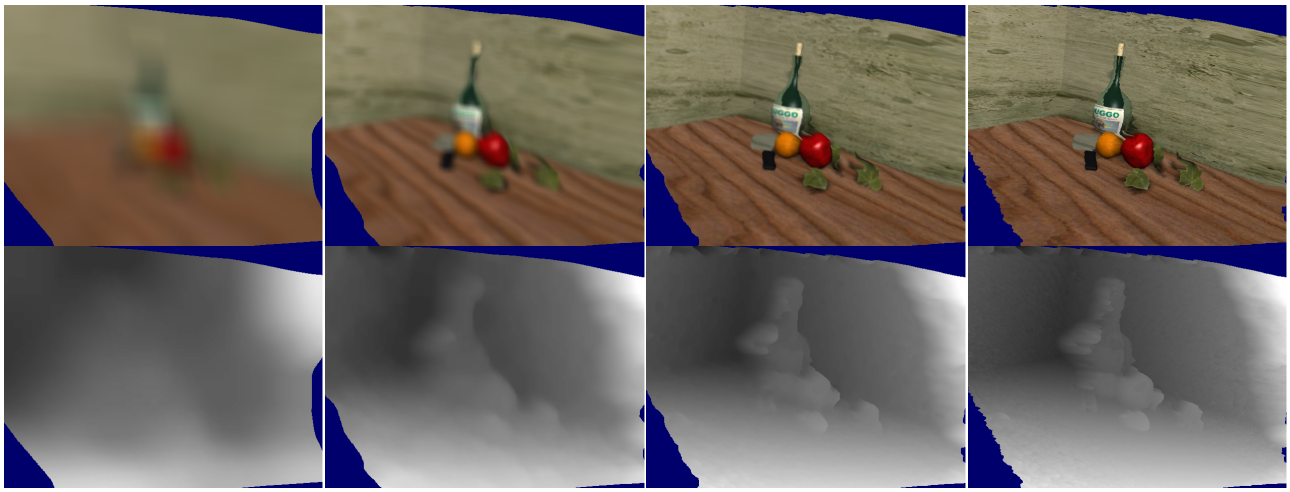


Figure 5.14: Mean-filtered input images provide SAD support regions of varying size for depth reconstruction. [Hei07]

The box-filtered results of all filter sizes, and the input images, are now bundled together into an *image stack*, which represents the input for actual depth reconstruction. Depth reconstruction itself is performed similarly to section 5.1.2: The algorithm still starts at the coarsest resolution, projecting the most coarsely filtered images onto a depth hypothesis, which is initially a simple plane. While the depth hypothesis assumes all possible depth values in the search space, the best matches between the (filtered) image pairs are recorded. The depth offsets of the best match pose the first depth estimates. This process repeats at increasing detail, and uses the depth values from previous passes as a base for the refined depth estimations.

The reason for this box-filtering of images is that in coarse-to-fine depth estimation, sums of color values at given image positions and their surroundings shall be compared with each other. This is generally achieved by generating a sum of absolute differences (SAD). However, for the coarser resolution, it suffices to compare the *aggregated* pixel values over a larger area with each other, instead of comparing individual pixel values. Previously, this aggregation of pixel values has been achieved with mipmaps. With mipmaps, these sums are however only exact at reduction centroids, i.e. those image positions that remain the center of aggregation throughout the reduction process that generates the mipmaps. The aggregated sums at all other image positions are *estimated* by bilinear interpolation in the later texture lookup, compare Figure 5.13 from the previous section. If *only* these reduction centers were used in SAD computation, then mipmaps would suffice. However, this cannot be guarantueed: For a given depth value, corresponding image positions in the camera views must be sampled, and these camera positions rarely correspond to reduction centroid positions, which implies that interpolated, or estimated, aggregation sums are used in the depth estimation. Especially coarse-level depth estimations, which only have very small mipmaps available, might thus become flawed. Therefore, the image needs to keep its full resolution at all box filter kernel sizes. Note that even with this change, we still follow Henkel's introduction of multi-resolution matching, as he did not describe how input aggregations from larger areas were to be generated [Hen97].

Beyond this change from mipmaps to full resolution box-filtering, we considered that triangle rasterization is expensive at high depth map resolution. To eliminate triangle rasterization and speed up the computation of depth estimations, we decided to never generate any kind of mesh for the depth hypothesis. Instead, we exploit the fact that in the depth view, the depth mesh has a planar, unoccluded projection image. For every pixel of the depth view, the depth value is *directly* used to generate a projective texturing lookup into the two input images. In effect, we have replaced a rasterized depth surface with an *implicit* depth surface.

To reduce computations in depth reconstruction further, projective matrix multiplications are partially precomputed. Since the image projection is a linear operation, any projection on a plane with a constant depth value can be linearly interpolated from four corner values. Hence, we can use bilinear texture interpolation to quickly calculate a 2D image offset from a set of four projective texture coordinates, instead of having to use a projective matrix multiplication for each depth sample. These four corner values are precomputed for all possible depth values.

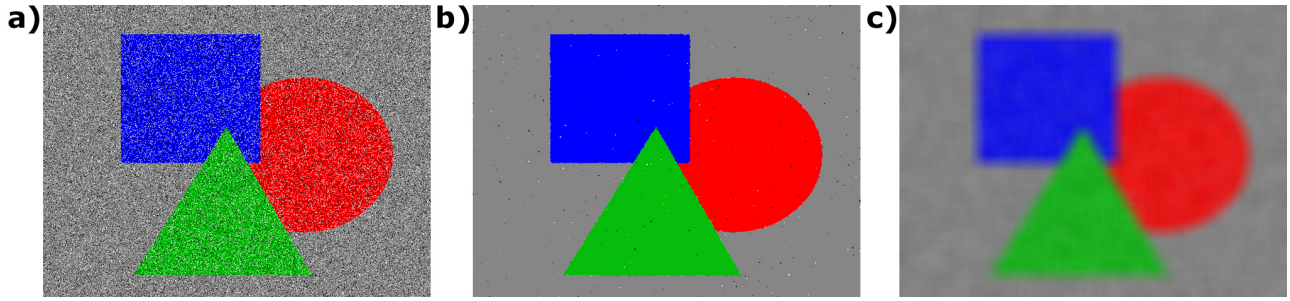


Figure 5.15: A color analogy for the median and mean filtering of depth values.
Left: input image. Middle: Median-filtered output. Right: Mean filtered output. [Hei07]

Just like in previous approaches, depth estimation iteratively refines the depth map by increasing the depth view's resolution, and simultaneously limiting the variation in depth offsets and shrinking the SAD regions. In order to have a depth base at this increased resolution, the previously estimated depth values must be upsampled. In previous approaches, we plainly used nearest neighbour or bilinear interpolation to fill in the missing depth values at the higher resolution. Here, the depth values are also *median filtered* before the upsampling step. Figure 5.15 sketches why this is a good choice for depth noise reduction: In contrast to mean filtering, discontinuities are often not affected by median filtering. This proves very useful in the suppression of noise around local depth discontinuities, as Figure 5.16 demonstrates. Note that in principle, mean filtering can still destroy depth discontinuities. But median filtering has considerably smaller complexity than an upsampling filter that is more aware of discontinuities, such as e.g. bilateral filters. With real-time performance in median, and based on the fact that median filtering is only used to generate the base for new depth estimates here, we chose to remain with the faster filter method. After median filtering and upsampling of depth values has completed, more fine-grained depth estimation can take place, see Figure 5.14. Depth refinement is repeated until even the original, unfiltered stereo images have contributed to the depth reconstruction.

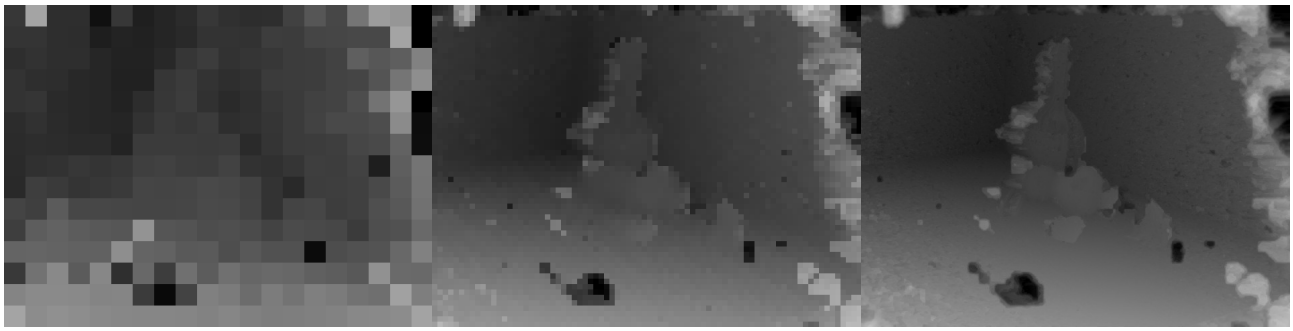


Figure 5.16: The importance of median filtering in depth hypothesis refinement.
Top: Mean filtering propagates the depth hypothesis to higher levels. Bottom: Median filtering result. [Hei07]

For the comparison with other stereo approaches, disparity values are often necessary. A disparity map can easily be produced by generating the 2D texture coordinates that would otherwise be used by projective texturing to conduct the color value lookups from both input images - of course, this requires an epipolar situation.

5.1.4.1. Results

Results of our coarse-to-fine approach with recursive box filtering are definitely in the upper league with comparable algorithms [Hei07]. While the approach cannot compete with the best algorithms in stereo matching, it performs soundly amongst the real-time algorithms. In this speed class, only few algorithms are able to handle lens distortion and non-epipolar matching in real-time, solved here via shader programs and projective texturing onto an implicit depth surface. Figure 5.14 demonstrates that our box filtering in a recursive approach worked as expected, and was fast enough to be used in real-time processing. The implicit, positive effects of median filtering on depth hypothesis refinement are clearly demonstrated in Figure 5.16. This, and the fact that the complete algorithm ran at up to 10 frames/second on GeForce 6800 hardware underlines that we have been able to implement a high-speed, yet fairly sophisticated stereo reconstruction on graphics hardware using plane-sweep methods and a coarse-to-fine approach.

The resulting depth maps can be directly visualized with displacement mapping, as Figure 5.17 demonstrates. Noise stability results for a synthetic ground truth are also available [Hei07].

One might argue that the presented stereo approach is too simple, and that more sophisticated algorithms such as segmentation-based stereo matching are needed to properly reconstruct depth maps of real world objects [HC04]. But we side with the results of Goesele et al., and argue that there is only a limited amount of depth information contained in stereo images [GCS06]. With this reasoning, more accurate object information can and should rather be extracted by fusing *multiple* incomplete depth reconstructions from multi-view stereo images. We therefore endorse a simple depth reconstruction for each stereo view, and recommend to concentrate efforts on the 3D reconstruction from *multi-view* stereo images of the same object. With this line of reasoning, the presented algorithm provides one of the building stones for a forthcoming real-time 3D reconstruction of complete objects.

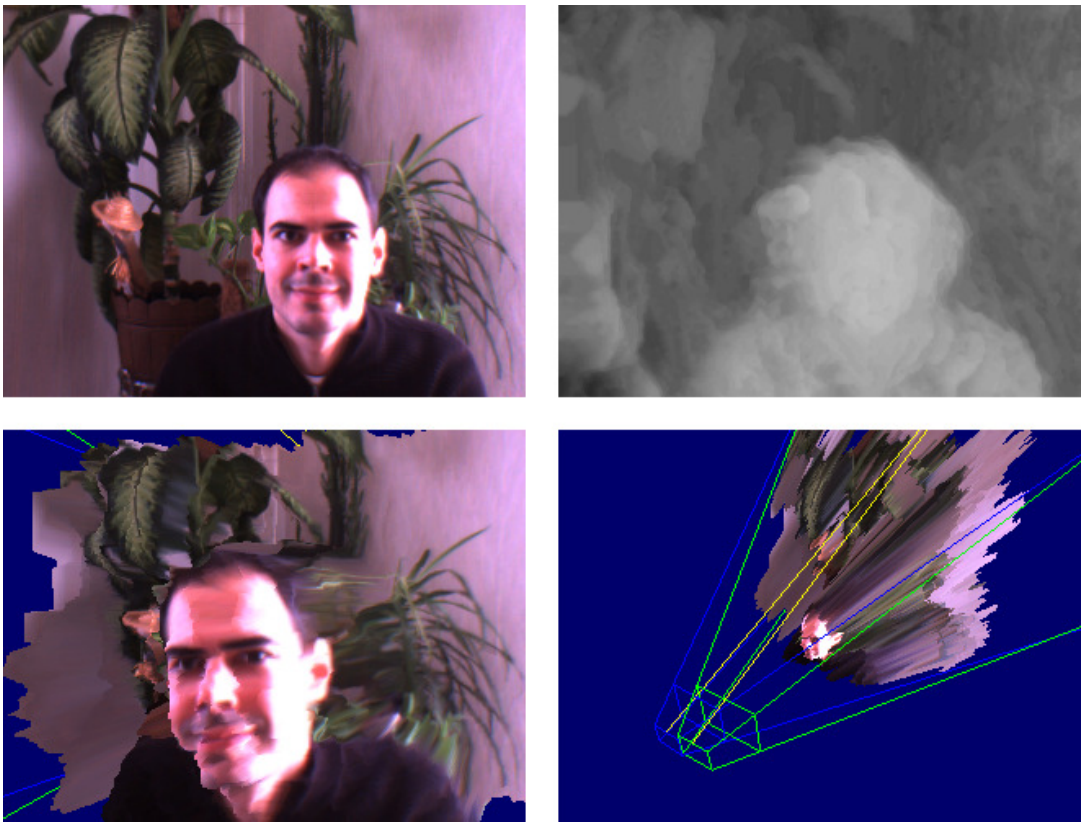


Figure 5.17: A snapshot from a real-time stereo reconstruction. [Hei07]

5.1.5. Future work

In the future, we aim to improve depth reconstruction while retaining the computational performance of our current implementations in as far as possible.

The early detection of untextured input regions would reduce depth noise caused by spurious image matches due to image noise. Sophisticated solutions do exist, with one example being the noise-driven stereo matching by Gimel'farb et al. [GMDL05], but these sophisticated solutions can usually not reach real-time performance. Instead, one possible implementation more suitable for our real-time algorithm could be based on the observation that a human would not attempt any depth refinement for image regions with little perceptible texture content, and assume a continuous depth value for this region instead. The procedure would be as follows: if the added SAD from an image region never rises above a certain threshold during a whole iteration, then the remaining texture content is considered to be image noise, as the resulting matches would only produce spurious depth discontinuities. Depth hypothesis refinement is thus terminated early, and the previous depth base value retained as the final result. This way, we could probably avoid that image noise destroys a good depth result in later passes of refinement.

Improved matching results can also be attained by replacing the box kernel filtering with a recursive implementation of Gaussian filters [VYV98, Der92]. Due to its close relation to human vision processing, Gaussian filtering was also employed by Henkel in image stack generation [Hen97].

We would also like to investigate depth reconstruction from more arbitrary camera constellations than the one used in the Stanford camera array shown in Figure 5.8, and use the multiple input views to increase the accuracy of 3D scene reconstructions. For example, multiple depth surfaces can be quickly reconstructed from pairs of camera views with the help of graphics hardware. Depth reconstruction could then store the final SAD of the depth reconstruction along with the actual

depth values. Later, in the interactive 3D reconstruction of the object, depth values with a high SAD could be ignored in depth map projection, and the CSG techniques from Section 4.2.6. would be used to cover the ignored depth samples with samples from other depth views with lower SAD, and thus yield higher overall reconstruction confidence. Such an approach would follow the reasoning of Goesele et al, where they compared the quality of a simple, but fast multi-view depth reconstruction with a more sophisticated, but stereo-based depth reconstruction [GCS06].

Many of the techniques that we explored in Section 5.1.4 are also applicable to multi-view reconstruction. The box-filtered image stack very probably makes multi-view depth reconstruction more accurate, and the implicit depth hypothesis would certainly speed up the necessary computation. However, since a common depth view easily causes occlusions for one or several of the participating camera views, a mesh-based depth hypothesis is still preferrable in wide-angle camera arrangements, Following this reasoning, we plan to include occlusion detection in the selection of contributing views. This would be achieved by creating shadow maps of the depth hypothesis in all camera views before depth map refinement commences, and by using these shadows maps to invalidate occluded views in depth evaluation. Multi-view depth reconstruction would thus move away from classic stereo matching to space carving.

5.2. Hierarchical Feature Clustering

In many applications of computer vision, objects with certain features shall be detected and tracked over a number of image frames, e.g. a yellow tennisball moving over the court. The first step in such image processing is first to detect all features independent of their global context in the image, i.e. local feature detection. This is usually achieved via image filters that use 2D convolution kernels, an operation that is easily implemented on graphics hardware due to its locality [JO04].

But some of the detected features may only be noise, and not at all belong to the object of interest. Therefore, it is necessary to to separate features from background noise by setting the detected features *into a global context*, i.e. relating them to each other. One important technique is to exploit the *spatial clustering* of features that belong to an object, to separate them from those feature indicators that originate from image noise.

We set as our task here to find the largest connected cluster of features in a 2D image. In terms of performance, we want this task to be accomplished for TV or HDTV resolution, and at more than 10 frames per second. To relate this to our example of a tennisball moving through a court, we would make the assumption that the largest present object of yellow color would be the tennisball. If that is the case, then an algorithm solving our task would detect the tennisball at real-time speed, wherever present in the scene. Our example in Figure 5.18 provides a second example: there we aim to detect the largest object of red color. In this case, we want an algorithm to return the upper left region as the final result.

The task setting is not new, and many feature clustering techniques solving this task have been conceived before. However, some of them have a high complexity that is too high for the expected performance. For example, while a connected components algorithm could easily solve above task (provided that the size of the connected regions is tracked) given execution times of seconds, it would very probably have a run-time that bars it from being used in real-time application. Also, would

Other algorithms are simpler, and correlate features while a single thread processor parses through the image. One of them is the *scanline-based tracker*, which holds a list of all connected regions of feature found in the current scanline, and a largest region so far. While parsing through the scanline, it updates the list of connected regions that it just found. All regions that are “discontinued” in a given scanline, are compared to the largest region found so far, before being discarded from the

tracker's list. The tracker can thus locate the largest region of features in the image in one top-to-bottom, left-to-right parse of the image, which promises a speedy implementation.

Unfortunately, this algorithm has one flaw, making it very hard to reach performance in a graphics hardware implementation that must utilize data-programming: it was designed with a single-thread processor in mind, and uses serial dependencies in its feature correlation.

In our approach to this task, we want to avoid serial dependencies, and use a reduction to *merge feature regions* on ever larger scales. Reduction, already mentioned in Section 2.2.5.5, is a versatile tool for the data-parallel computation of a global result for a given data array. Typically, the reduction operator is simple and only used for single-valued results such as averaging, but with programmable graphics hardware, more complex reductions are just as feasible. In our algorithm, the associated reduction operator becomes a *feature clustering operator.* Consequently, we name our algorithm *Hierarchical Feature Clustering.*

We examined the implementation, performance and applications of hierarchical feature clustering in the bachelor theses of Schilz and Arnold [Sch07, Arn07], The feature filter was kept simple to focus on the algorithmic evaluation, and retrieves the largest region of a given color.

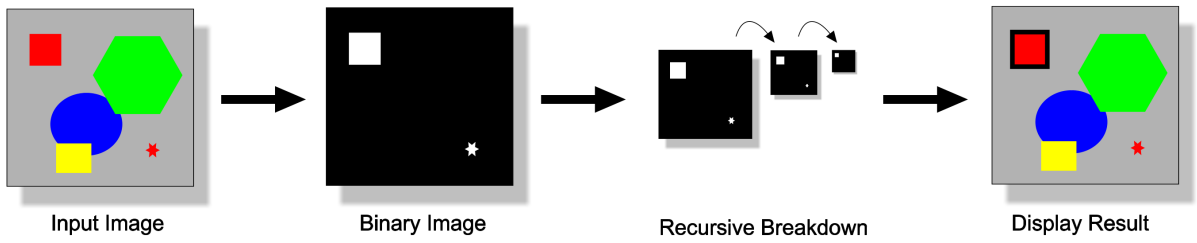


Figure 5.18: Algorithm overview.
The input image is first filtered to detected all features matching the criterion (here: red color). Then, feature clustering through reduction is performed until the largest region has been found. [Sch07]

Figure 5.18 provides a rough overview over the algorithm's individual stages. As a first step, the input image is filtered for features; in the figure, the filter applies a red color threshold to find all pixels of red color. The filter marks all areas that pass the filter criterion, which are red pixels in our example. The pixels that have passed the criterion are now very small *feature regions,* and represented by bounding boxes that enclose the detected features. In the next stage, reduction commences, which merges the bounding boxes of these feature regions if they happen to be adjacent. If the feature regions are not adjacent, then reduction will select the largest feature region of the presented ones. With repeated reduction, the number of output cells diminishes, and so does the number of prevailing feature regions. In the top output cell, only one feature region has remained. This single feature region describes the *largest connected feature region* of the input image, bearing a loose resemblance to a maximum value reduction. The result can be conveniently displayed in the input image, or downloaded to the CPU for further use. In the following, we will explain the individual steps involved, and refer to Schild and Arnold for a more detailed description, including shader sourcecode [Sch07, Arn07].

5.2.1. Feature Detection

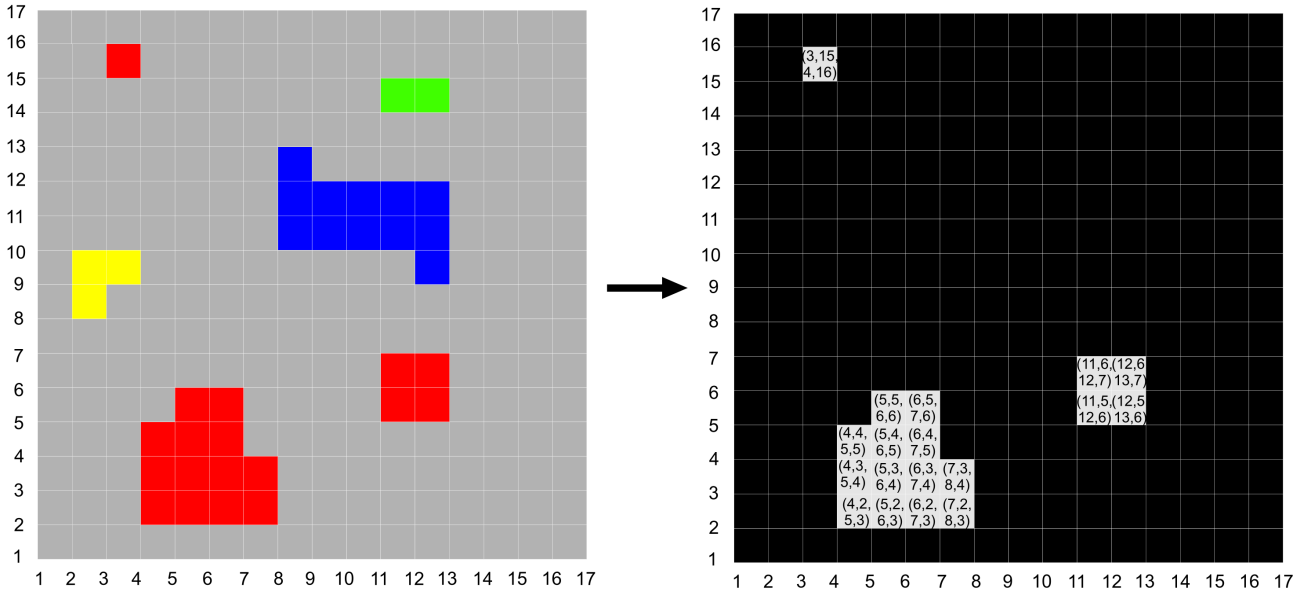


Figure 5.19: Filter output, as described in [Sch07]. The color values of the input image that match the filter criterion (red) are directly converted into small feature regions of size 1x1.

Before feature clustering can commence, features have to be detected in the image input first. The type of filter is irrelevant, as long as it produces binary output; some examples are a threshold-based color or motion detection. But note that the binary filter's output is never written as an explicit binary value. Instead, the filter will output *feature regions:* A 2D bounding box that encloses the feature's area with a bounding rectangle. Figure 5.19 shows an example, the transformation of a color image through a filter for red color. Note that the filter will not output the binary values 0 and 1, but directly generate the coordinates of small rectangles that bound the found feature. One detected, valid feature is the single red pixel in the upper left. The filter writes the coordinates (xmin, ymin, xmax, ymax) = (3, 15, 4, 16) to describe a bounding box that encloses the feature's region. All featureless regions are filled with the invalid feature region (0,0,0,0).

5.2.2. Reduction

In the next stage, feature regions are repeatedly merged or selected until only one feature region remains. The reduction operator is slightly more complex than in usual cases, and has two stages: *region merging* and *region size competition*. In region merging, incoming bounding boxes are first tested for adjacency, and merged as far as possible. Figure 5.20 shows the possible merger outcomes, where red color marks connected regions.

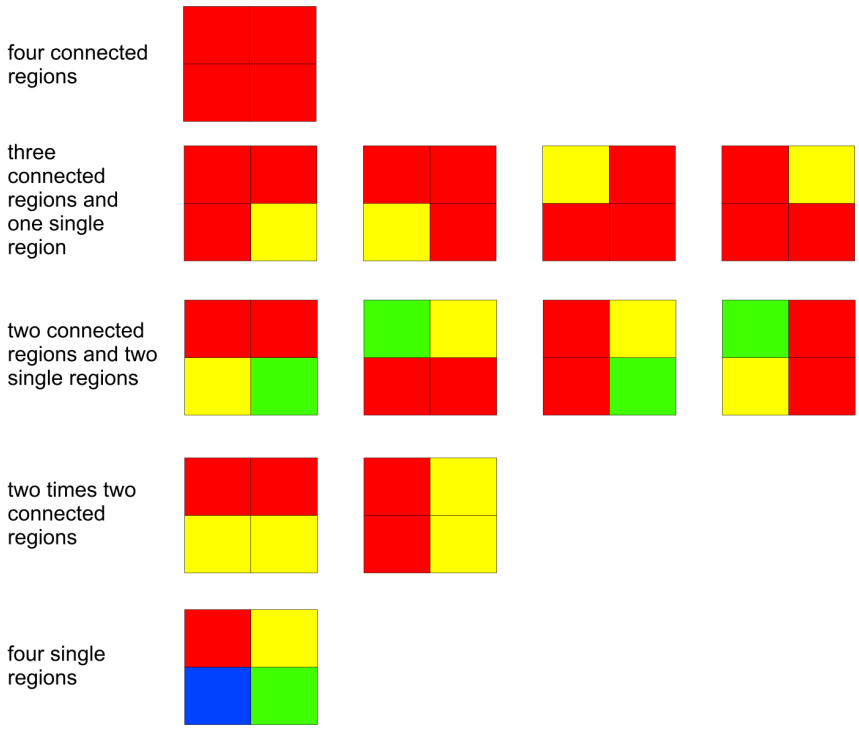


Figure 5.20: All possible combinations of feature region merges.
Adjacent feature regions, marked in red and yellow, are merged in this stage. [Sch07]

Then, the merged feature regions are tested for their size in a region size competition. The largest region amongst the candidates prevails in the output. Note that merged feature regions do not necessarily have to be the largest one if their contained bounding boxes are small: In Figure 5.20, it could e.g. be that a cell marked in green contains a bounding box that is larger than the merged regions of the two, merged bounding boxes stored in red cells.

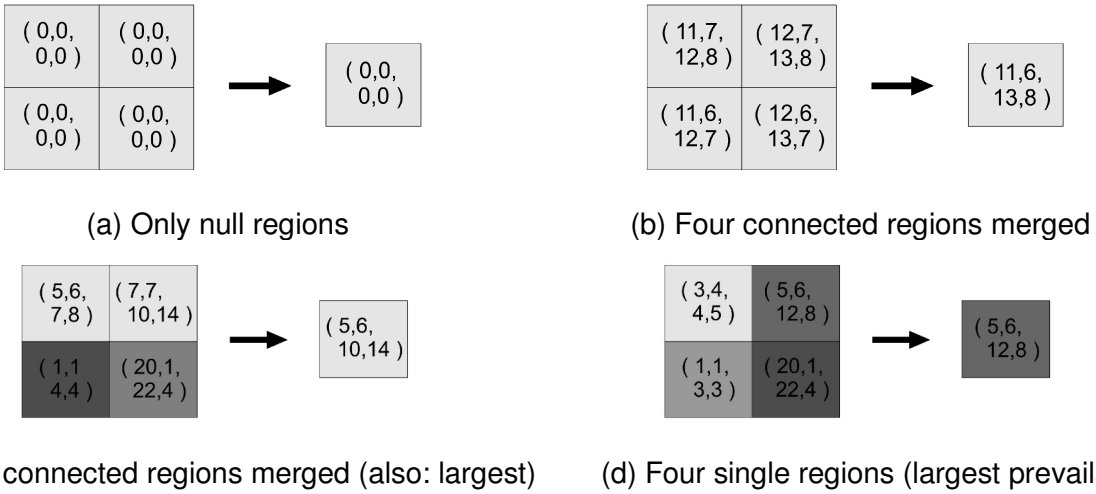


Figure 5.21: Example outcomes from the reduction for various combinations of feature region bounds.
Bounding box values are denoted as (xmin, ymin, xmax, ymax) [Sch07]

Figure 5.21 demonstrates some examples, with the participating bounding boxes described as (xmin, ymin, xmax, ymax). Case (a) describes a merger of four featureless regions: The result is a new featureless region. Case (b) describes the merger of four regions that are adjacent to each other, because the left cell's xmax=12 is identical the right cells' xmin=12. The result contains xmin of the

left cells, and xmax from the right cells; the same applies to ymin and ymax for top/down orientation. In case (c), only the upper left and right cells' stored feature regions are adjacent to each other. Since their merged bounding box (5,6,10,14)|size=5x8 also happens to be larger than (1,1,4,4)|size=3x3 and (20,1,22,4)|size=2x3, it is written to the output cell of the reduction. In case (d), the sizes of the cells are (3,4,4,5)|size=1x1, (5,6,12,8)|size=7x2, (1,1,3,3)|size=2x2 and (20,1,22,4)|size=2x3, respectively. Thus, the upper right cell prevails the competition, and is written to output. Note that these examples do not cover all of the cases in Figure 5.20, but is sufficient to understand the principles.

Reduction ends when the data array has been reduced to a single cell. The bounding box in this cell describes the largest connected feature region in the area. The result can now be used for subsequent stages, such as a higher-level algorithm for object tracking. The result's data resides in graphics memory, but is very small and can easily be downloaded to the CPU.

5.2.3. Results

Since our object detection based on feature clustering is very fast on graphics hardware, it was natural to track something that can move quickly as a first test, such as a lightbeam's trace on a wall.

Figure 5.22: Tracking an LED light beam, with tracking algorithm's output shown as a red square.

The tracking results shown in Figure 5.22 demonstrate that the algorithm accurately tracks the *main* spot of the lightbeam. This cannot simply be explained with good thresholding, as the image clearly contains several other areas of similar color. It is the feature clustering, the second stage after thresholding, that ensures that only the largest connected feature region prevails. The side whiskers of the light beam are properly ignored in all of the three images, and the beam's main focus bounded accurately in the last two images, which likens a human's assessment of this input. In related experiments with the fast-moving beams of laser pointers, camera exposure time was too long to locate the beam's position on the wall. Despite that difficulty, our algorithm correctly bounded the blurred trace of the laser pointer, with its centroid at the laser pointer's approximate position in the middle of camera exposure time. This corrresponds to the best guess a human could do for this degenerate case. Note that in both experiments, with lightbeam and laserpointer, the beam would always be located by the algorithm, *independent of speed of movement* from frame to frame. This is because each frame of the camera input is analyzed completely and no assumptions are made on previous object positions.

Beyond these first results, in their theses, we examined two applications of the algorithm: head tracking [Arn07] and gesture recognition [Sch07]. Since both applications share a similar physical and algorithmic setup in regard to feature clustering, we concentrate on head tracking as sufficient example for both.

While 3D head tracking through triangulation is well-known, our approach was novel through its noise suppression and speed provided by hierarchical feature clustering. All data was processed on graphics hardware, which freed CPU ressources to maintain interactive speeds for the main application. Input was provided from two Firewire-based cameras by PointGrey. The operating system was Linux, and libdc1394 was used to capture the image input to main memory. From there, the OpenGL pixel buffer object mechanism helped in speedily transmitting the images to the GPU, an NVIDIA GeForce 6800. The intrinsic and extrinsic camera parameters had previously been acquired from a stereo camera calibration. The color thresholds were acquired similarly. All settings remained fixed throughout the experiment. The reports can of course provide further details.

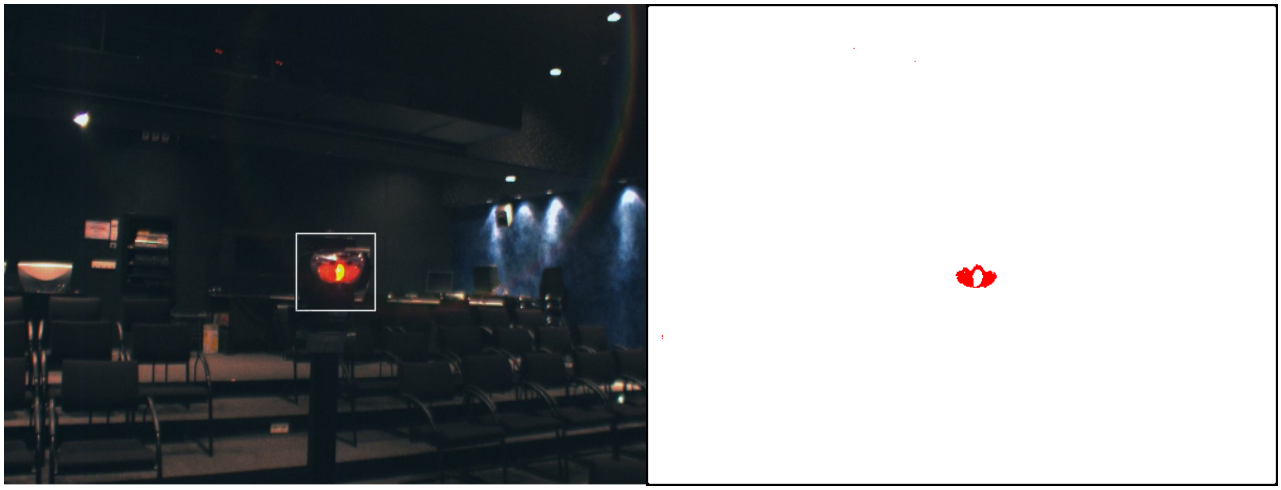


Figure 5.23: Left:The headlight, marked in white, produces red light.
Right: The algorithm's binary color filter thresholds for this red light color. [Arn07]

Whenever a new image pair from the cameras finished uploading to the GPU, a shader program is was run to account for lens compensation and color space conversion via dependent texture lookups. Afterwards, pixel-sized feature regions were extracted via color threshold filtering, see also Figure 5.23. The result was then fed to the stage that conducted feature clustering via reduction, as described above. When feature clustering was complete, the center of the single remaining feature region from both cameras could be used for a stereo triangulation of the user's viewpoint, see Figure 5.24.

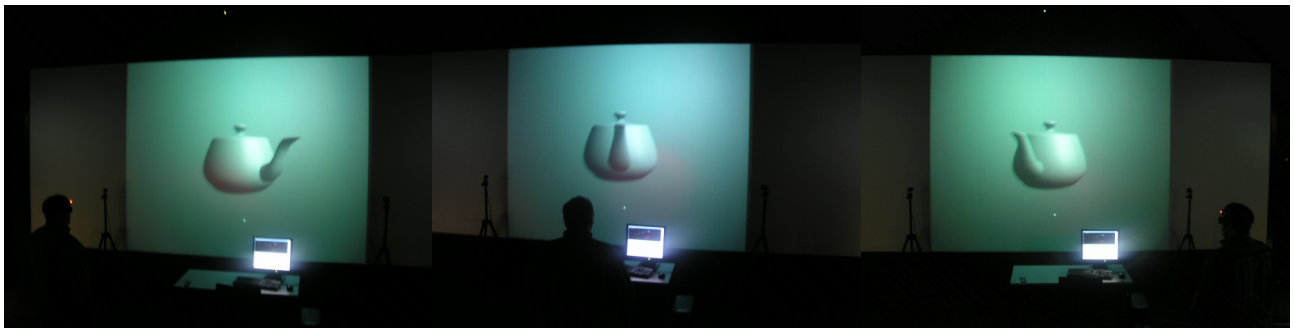


Figure 5.24: View-dependent visualization via robust head tracking, based on this algorithm. [Arn07]

Input resolution *(Stereo)*	CPU handling, max. frames per second	GPU feature clustering, frames per second	Feature Clustering Texture (Power of 2)
320x240 *x 2*	122	25	512x512
640x480 *x 2*	40	25	1024x1024
800x600 *x 2*	11	11	2048x2048
1024x768 *x 2*	11	11	2048x2048

Table 5.1: Timings for CPU input handling (without GPU), and GPU-based feature clustering for different input sizes, including required texture sizes for feature clustering. [Sch07]

The processed frames per second are listed in Table 5.1. Note that the input images are twice the data amount due to the stereo camera setup. The observed framerates strongly correlate to the texture size used in feature clustering: Timings are equal for 1024x768 and 800x600, as both use 2048x2048 texture during feature clustering. For 320x240 input , a 512x512 texture suffices. For the larger input resolutions, a latency of approximately one second emerged between user movement and change of viewing angle on the canvas, probably caused by the lack of onboard memory on the deployed graphics hardware, or Firewire readback timeouts. The lower resolutions lacked such latencies.

	CPU scanline algorithm (fps)	GPU algorithm (fps)
256x256	195	156
512x512	100	128
1024x1024	34	72
2048x2048	9	25

Table 5.2: A coarse comparison between CPU-based and GPU-based object detection. [Sch07]

In an additional experiment, the GPU software prototype was compared with the CPU implementation of a scanline-based object tracker to underline previous statements on the algorithm's performance. The scanline-based object tracker was implemented as described in the introduction. Table 5.2 shows the results. Despite allowing the CPU approach more assumptions on the shape of feature regions, it quickly loses in framerate for increasing image sizes. Our GPU approach can thus outperform the CPU algorithm.at input sizes akin to webcamera resolution and beyond.

5.2.4. Conclusion and Future Work

The presented algorithm shows that reduction operators can even prove useful in more complex image processing tasks that require global results, such as object the detection via the largest feature region of an image. While a GPU-based object detection approach, based on reduction, has been proposed before [BDL07], it was not yet capable of separating disjunct regions, and therefore weighed in all found features throughout the image into the final centroid and its estimated area.

In contrast to common scanline-algorithms for the CPU [JB94], our algorithm retains the two-dimensional nature of the input throughout the processing steps, which yields advantages in data-parallelism and 2D cache locality during execution on graphics hardware. In contrast to the more general connected component labelling [ST88], the required running time is known in advance. By combining the algorithm's data-parallelism with the massive memory bandwidth of GPU memory, complete frames can be continously processed at interactive framerates.

The algorithm handles various input data modalities, and other feature classes than the presented can be clustered simply by changing the input filter. For example, a gradient operator and a threshold could be combined with above feature clustering to detect the largest moving object in a scene. In another application, thresholding on a low color difference between adjacent pixels could mark connected pixels of *any* color, and thus allow for the detection of the largest *arbitrarily colored* object in the scene.

In the future, we plan to improve on the algorithm's limitation to clustering a single feature at a time. With the OpenGL extension for multiple rendering targets, graphics hardware can render to multiple textures simultaneously. This way, up to 8 distinct features can be clustered in parallel.

5.3. Histogram Buildup via Reduction

The computation of an image histogram can be achieved in various ways. Occlusion queries are an OpenGL mechanism to count rasterized pixels [NV04b]. In combination with a fragment shader that discards unwanted pixels before they are rasterized, occlusion queries can quickly generate a total histogram count over an image. However, for multiple histogram bins, they require one rendering pass for each of the bins, as occlusion queries only have one result channel. Fluck improves on this by using a fragment shader to generates all output channels in the same image plane [Flu06].

But global operations, such as the computation of an image histogram, are often best achieved with reduction operations, as already described in section 2.2.5.5. On graphics hardware, reduction is typically implemented in 2D, as this is the most efficient datatype in OpenGL and the input often originates from 2D images. During reduction, a mipmap-like data structure is built. While mipmaps contain the average of their input values, the data structure in histogram computation contains sums of its input values. For each of the histogram bins, one such data structure is used. Henceforth, we call them Histogram Pyramids (short: HistoPyramids).

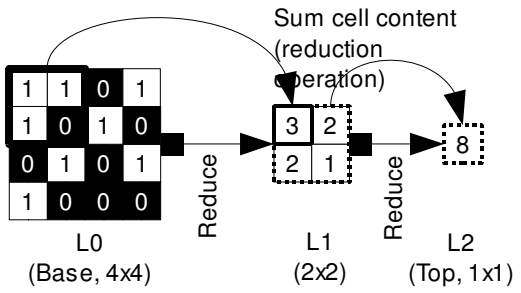


Figure 5.25: Basic HistoPyramid buildup process.
Input is entered as L0, while L1 and L2 are the reduction levels.

Figure 5.25 provides an overview over the reduction process. The bottom level of the pyramid, the *base level,* contains the reduction's input. This input can generally originate from various data sources, but in our application, such as the thresholded filter output from an image convolution.We let the filter output "1" if the filter's threshold has been passed. Otherwise, a "0" is written.

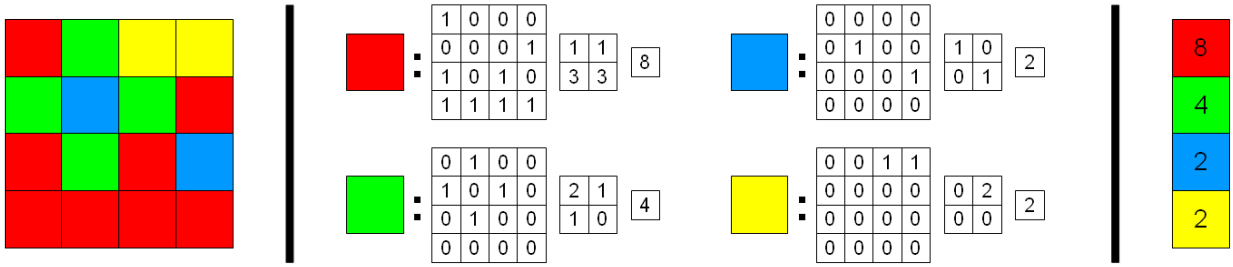


Figure 5.26: Color histogram via Histogram Pyramids. Left: Input image. Middle: Color-specific histogram pyramids. Right: Final histogram, composed on the CPU from each histogram pyramid's top cell.

Figure 5.26 presents an example with four possible input colors. For histogram generation, a separate pyramid is created for each color. After all color filters have been applied and thresholded, reduction of the pyramids commences. While generating the next pyramid level, the GPU sums four adjacent cells into one, thereby halving resolution until only one top cell remains in each pyramid, see again Figure 5.25. When reduction has finished, the single remaining top cells of each pyramid will contain the histogram entries. These cells are then read back to the CPU, where the complete histogram is created. Note also that while the CPU is involved in this final step, all heavy image processing has been performed on graphics hardware.

5.3.1. Lens Compensation

We will now present an application for GPU-computed histograms outside the widely used color histogram computation. A classic problem in camera calibration is the compensation of lens imaging distortion, short *lens compensation*. The most common lens distorsion is the radial lens distorsion of quadratic nature, where image coordinates are distorted as

$$x_d = x_u + (x_u - x_c)\boldsymbol{k}\, r^2 \text{ and } y_d = y_u + (y_u - y_c)\boldsymbol{k}\, r^2 \text{ and } r = \sqrt{(x_d - x_c)^2 + (y_d - y_c)^2}$$

with x_d, y_d as the distorted image coordinates, x_u, y_u as the undistorted image coordinates, k as the distortion parameter, and x_c, y_cr as the center of distortion, which usually lies in the middle of the acquired image, depending on the camera sensor properties. Figure 5.27 provides two examples of the effects that radial lens distorsion can cause.

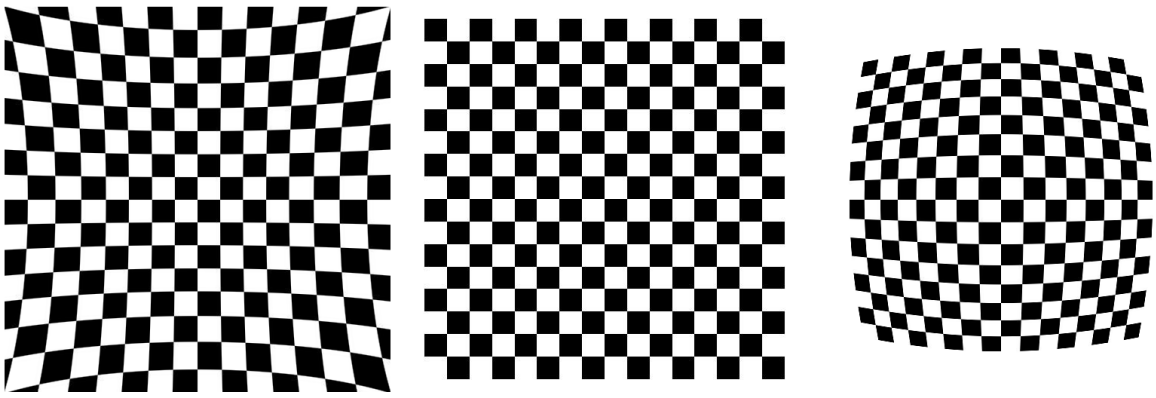


Figure 5.27: Radial distortion warp of a checkerboard pattern, with distortion parameter k. Left: k < 0. Middle: Original, k = 0. Right: Positive values of k > 0.

With the help of a checker board that is placed in front of the camera plane, different image transformations can be applied until the checker board appears even - at this point, lens distortion has been compensated. Traditionally, these lens compensation parameters are chosen by a human

through careful experimentation, possibly assisted by a half-automatic parameter proposal based on user-defined feature points. But this is a time-consuming process, and far from happening at real-time speeds. The necessary image warp for radial lens distortion is often run on the CPU, which typically requires up to a second to complete it. The choice of various undistortion parameters thus becomes a tedious process. On the other side, graphics hardware is able to warp the image in real-time, and can thus accelerate intermediate visual feedback during the user-guided search for lens compensation parameters.

But beyond that initial usage of graphics hardware, we had the following insight for image undistorsion: Given a straight-up view of a checkerboard from a camera, image undistorsion should be successful if all checker contours are straight. For a checkerboard, having straightened all checker contours implies that all edges in the checkerboard image follow *two major directions*. If an algorithm would thus be able to tell that all contours in an image start following a few major directions, it could imply that they have been straightened, and thus provide guidance for the user during the calibration process, or even find image undistorsion parameters *automatically*.[1]

A common approach to detecting straight lines in an image are Hough transforms, which already have been implemented on graphics hardware [SIM03]. But we found it hard to make graphics hardware interpret the output of the transform, and resorted to a different approach instead that mimics the human approach: Tracking the overall behaviour of contours in the image.

To track contour directions in the image, we need to detect discontinuities. Our discontinuity detection starts by convolving the image with simple filter kernels to highlight potential contours in several directions. Our contour filter design is inspired by the linear spatial filters that mimic human visual system responses, as proposed by Jones and Malik [JM92]. The filter outputs are then thresholded to pinpoint local contour directions. These local countour directions are then classified into *bins of direction*, from which a *histogram of contour directions* can be created, see also Figure 5.26. An input image has thus been analysed for local image contours and their directions.

A human would now seek to straighten the contours in the image. With the histogram of contour directions as feedback, a higher-level algorithm can do similar: it would alter the lens compensation parameters until *few prominent bins* form in the contour direction histogram. This is because a clustering of contour directions indicates that the image contains mostly straight lines in few major directions, as all other line paths would show a more even distribution of contour directions. Thus the optimal choice of lens compensation for a checkerboard lies in a *maximization of variance in the contour directions*, between nearly empty and very highly filled histogram bins. Note that it does *not* matter if the checkerboard is held vertically and horizontally aligned into the camera, or not: If the checkers appear at an angle, then the histogram bins correspondings to the checkers' angles will attain maximum values as soon as the checkers are straightened in their angled positions in the image.

Naturally, this exhaustive search for undistortion parameters must run at acceptable speeds to be feasible. Preferrably, the whole operation, from image warping to histogram computation, should execute on graphics hardware. The detection of contour directions via image convolution and filter output thresholding is a very local operation, and thus easy to implement.

For the histogram of contour directions, which is computed from the whole image, we use the presented histogram pyramid. For each contour direction, a separate pyramid is created; see Figure 5.28 for an example with four possible contour directions. After all contour filters have been applied and thresholded, reduction of the pyramids commences. While generating the next pyramid level,

1 Unfortunately, this lens compensation algorithm was discovered *after* our works on stereo reconstruction had been concluded, and could thus not assist in the necessary camera calibrations earlier in this chapter.

the GPU sums four adjacent cells into one, thereby halving resolution until only one top cell remains in each pyramid, as previously described in Figure 5.25.

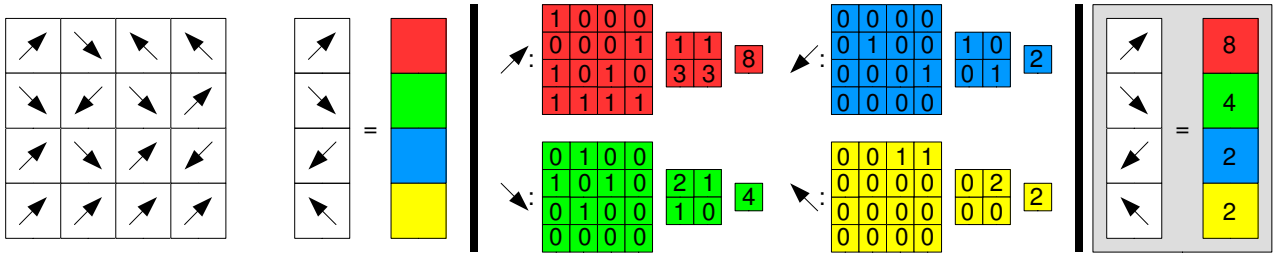


Figure 5.28: Edge direction binning via Histogram Pyramids. The arrows depict valid edge gradients.

In the OpenGL implementation, four contour directions can be put into color channels (here: RGBA), and thus be reduced in parallel, see Figure 5.28. But for the overall application, four contour directions do not suffice. Luckily, the OpenGL extension for multiple rendering targets (MRT), provides supplemental output channels. With MRT, up to 4 x 4 =16 (NV40) or 4 x 8 = 32 (G80) color channels can be processed in parallel, which enables the computation of equally many histogram entries in one iteration. When reduction has finished, the color channels of the single remaining top cell will contain the histogram entries. This cell is then read back to the CPU, where the complete histogram of contour directions is created. When the full histogram is available, decisions for lens compensation values in the next iteration can be taken in the higher-level algorithm.

In the current implementation, the higher-level algorithm iterates through a range of radial lens compensation values k, and searches for the *maximum distance* between maximal and minimal filter response. Note also that while the CPU is involved in this final step, all heavy image processing has been performed on graphics hardware.

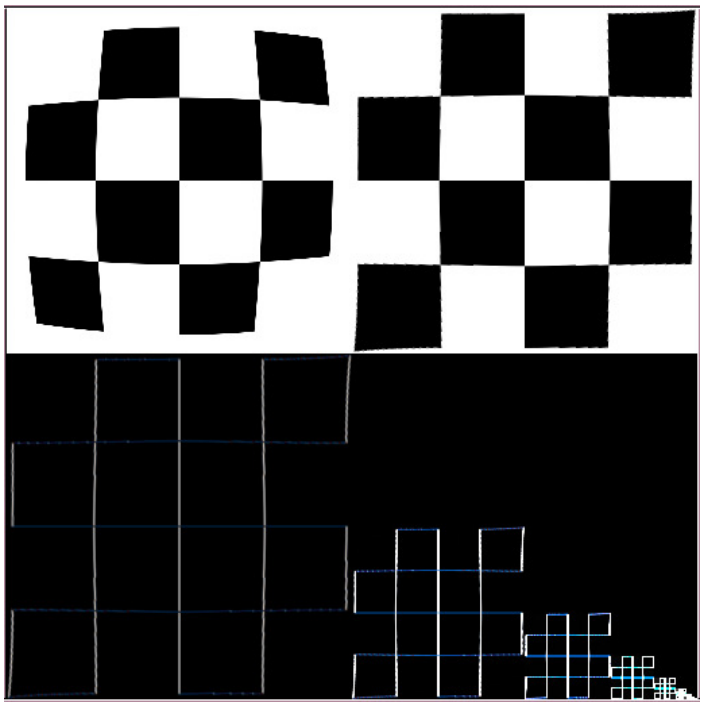


Figure 5.29: Application screenshot, k=-0.085, the final result for this 512x512 checkerboard image.
From upper left to lower right: Input image, Warped input,
Filter response (RGB = three contour directions), Reduction pyramid for histogram feedback.

A screenshot of a typical result from the high-level lens compensation process is provided in Figure 5.29. As input for lens compensation, we used a checkerboard image that we had artificially distorted in the image processing software GIMP.

Gradient angle	k=-0.165	k=0.045	k=-0.085
0	3272	3606	4207
11.25	3583	3965	4593
22.5	4073	4536	5238
36.75	4509	5021	5797
45	**4903**	**5461**	6304
56.25	4509	5021	5797
67.5	4073	4536	5238
78.85	3583	3965	4594
90	3272	**3606**	**4207**
101.25	3583	3616	4631
112.5	4073	4464	5297
123.85	4507	4934	5867
135	**4901**	5365	**6381**
146.25	4507	4934	5867
157.5	4073	4464	5297
168.85	3583	3916	4631
MIN	3272	3606	4207
MAX	4903	5461	6381
DIST	1631	1855	**2174**

Table 5.3: Histogram response for varying lens compensation values.

The columns of table 5.3 list the feedback from the image analysis for a given radial lens compensation value k. It contains a histogram of local gradient angles, collected from the whole image. The final choice, k=-0.085, is based on the image analysis for the warped image in Figure 5.29, top right, which shows to have the maximum distance 2174 between minimum filter response 4207 (gradient angle 90 degrees) and maximum filter response (gradient angle 135 degrees).

We would like to make several possible improvements in future implementations. One is to mask filter responses in regions where image content is missing after lens compensation, such that the transition betwen image content and empty background is not.wrongly counted as a checker contour. We would also like to extend the available choices of lens compensation techniques, which should provide better overall lens compensation for the wide range of available lenses and imaging sensors.

5.4. Conclusion

In this chapter, we have presented several successful algorithms employing hierarchical image processing on graphics hardware.

In depth reconstruction, plane sweep methods showed to be utilizing the bilinear interpolation and projective texturing capabilities of graphics hardware well for both stereo and multi-view image input. By borrowing both the concepts of mipmapping and, later, recursive image filtering and medical filter-based upsampling of the depth hypothesis, we demonstrated that depth reconstruction also benefits from a hierarchical depth refinement, leading to improved depth accuracy and depth noise suppression while maintaining high reconstruction performance.

In our experiments with object tracking, we discussed the problem of relating local image features to each other in order to suppress detection noise. We introduced a complex reduction operator that is able to detect the largest connected region of an aftersought feature in an image by repeatedly merging adjacent feature regions, and selecting the largest if feature regions are not adjacent. Using

this data-parallel approach, we could present a graphics hardware implementation that is able to perform this image analysis, and thus object tracking, at interactive speeds for high resolution camera input.

In the last section, we investigated the use of data-parallel reduction for the computation of image histograms. We explained how the histogram bin entry computation can be decoupled from the actual histogram computation, and how multiple reductions can be conducted at once by using texture color channels and multiple rendering targets. Later, we presented a novel application of histogram computation in the automatic compensation of lens distortion, through the repeated transformation and analysis of a checkerboard image.

In all three of the presented image processing tasks, the hierarchical dataflow has allowed us to avoid serial dependencies in the GPU implementation, while still retaining the regular data access patterns that are beneficial for thread workload balancing on graphics hardware.

In the next chapter, we will explain how even *irregular* data structures can be created after passing through a hierarchical reduction process.

6. GPU Data Compaction

Previous chapters demonstrated that the image manipulation abilities of graphics hardware can assist in computationally intensive image processing, especially if all processing is related to 3D vision tasks. The results underlined that image scale-space operations, 3D projection and 2D image manipulation are straight-forward to accelerate. But the ultimate goal of visual computing on graphics hardware must be to port the *complete* image processing pipeline, which promises both CPU offloading and higher performance due to reduced bus loads.

Unfortunately, the GPU was not originally designed for general purpose computation. One of the main restrictions is the fixed-output location of GPU shader programs, which makes it hard to directly port the more advanced data-processing parts of any visual computing algorithm, such as list operations.

In this chapter, we show how previously introduced concepts, such as reduction and their intermediate data structures, can be combined with scatter-by-gather ideas to implement general purpose computing on graphics hardware.

The work presented in this chapter resulted in a publication on GPU-based point cloud extraction from 3D meshes [ZTTS06], and was later followed by a publication on the original design purpose of this algorithm, interactive vector contour extraction from 3D vector fields [ATR*08].

6.1. Basic Problem Setting

In signal/image/volume processing, a common task is to locate features in an array of samples. Often, a *filter kernel* and a *filter output criterion* is provided, which indicate the samples that hold desirable features. The task is to create a *feature list* with the locations of all desired samples.

In general, such a feature list can be seen as an output of *data compaction*: From a list of input samples, we only keep the ones that are relevant. In the example in Figure 6.1, we only want to keep all blue elements. Thus, our filter criterion, or *Classifier*, is blue color.

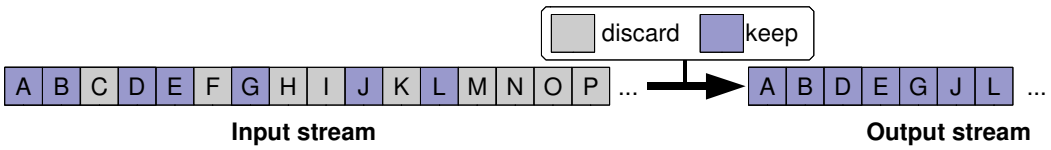


Figure 6.1: Data-compation, from input to output stream.

An initial approach is to consider how a single CPU thread would implement this search. By traversing the elements one by one and keeping track of the current location, it could easily create a feature list, advancing the output pointer as features are found. If the memory for the written output does not suffice, it would interrupt the search, allocate more memory for the output, and resume the search.

The next approach is to use several CPU threads for this task. One way would be to divide input evenly amongst the available cores, and let each one write a separate feature list. After all input has been processed, these lists would have to be merged. Already here, it shows that parallelization requires double as many write operations as the single core approach, and the final concatenation can still only be done by a single processor.

Another way would be to divide the input, but let the threads share a *common* feature list to write to, with a semaphore to *serialize write access* to the list.

But imagine how clumsy and cumbersome both approaches become when the number of active cores goes into the hundreds. For the first parallel approach proposed above, memory re-allocation and the final merge would become highly tedious, while for the second approach, a common lock on a single pointer, shared by hundreds of threads, would surely ruin any performance gained from this massive parallelization.

Therefore, it is wise to *separate* output *allocation* from output *writing*. This way, the parallel implementation for above problem becomes conceptually simple: Let each thread count its contributed features first, allocate its output in a common output list, *then* start writing the actual features. On data-parallel systems, we would use *reduction* for counting the input elements that shall be kept, and thus size the output correctly, see Figure 6.2. Each thread is responsible for one input element, and then later participates in the higher levels of reduction, as long as there is enough reduction work to do. Note that we are using a 4-to-1 reduction here to remain closer to the 2D implementation which follows later. Reduction ends when only a single output cell remains. Based on this resulting value, the CPU allocates the necessary space for the output.

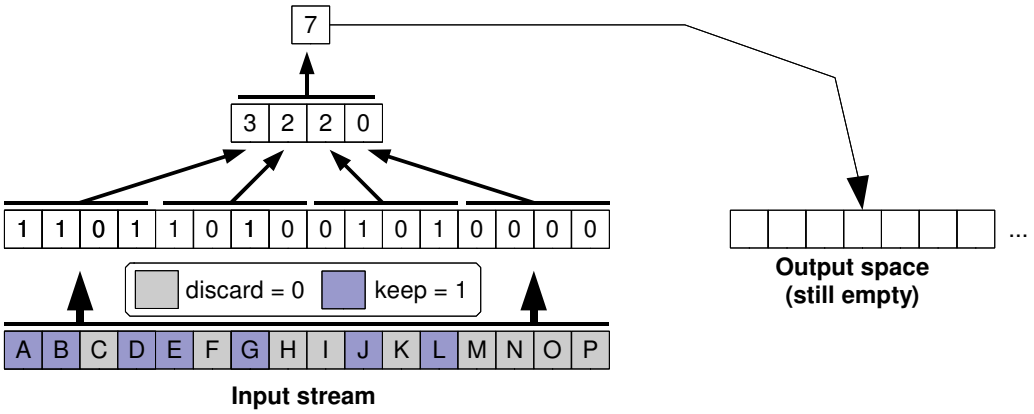


Figure 6.2: Data compaction. First step is the classification of input elements, then output elements are counted through a 4-to-1 reduction.

Output has then been properly allocated. But how does each thread know where to write its output in this commonly allocated output list? Basically, each threads needs to add up the offsets from previous threads´ contributed elements. For hundreds of threads, this becomes a non-trivial task. And we face another challenge: In graphics APIs such as OpenGL, each thread is only allowed to write to a fixed location.

To solve the restriction of a fixed output location, we will look at the problem from a *gather perspective*: Assume that a thread has been assigned an output location - how will it find the corresponding location in the input to read from?

It shows that the reduction's intermediate output can serve as a lookup structure! For this purpose, we visit the reduction operation once more: The four inputs to reduction are the children to the reduction result, which is their parent. And we now re-interpret the parent's value as an indicator for the number of indices that its children have requested. We will now see that this suffices to track down a corresponding, unique input element for each output element.

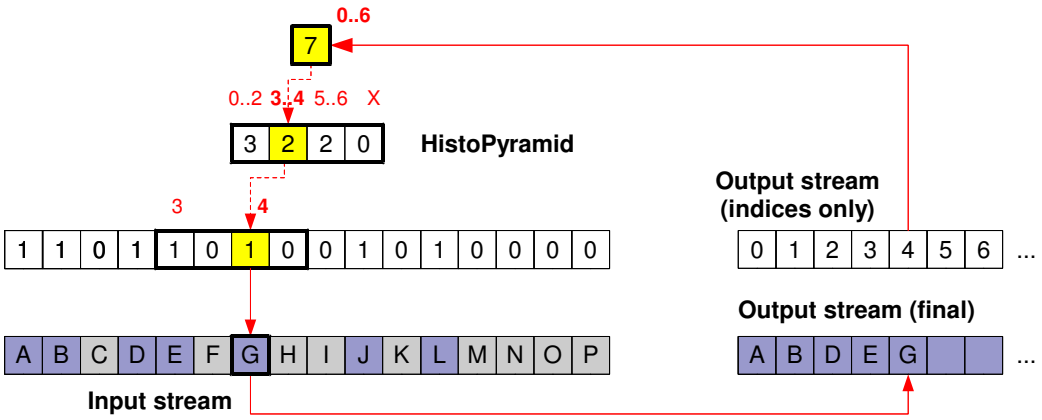


Figure 6.3: Data compaction, from input to output stream.

Figure 6.3 gives an example for writing the output element with index 4, or: the fifth output location. The top cell of reduction indicates that output indices 0..6 are found among its children - this is obvious, as it covers the complete output allocation. From there, we take a look at the top cell's children and establish a certain order of their requested indices, here: left to right, to create partial index ranges. Amongst these ranges, index 4 is to be found as a child of the second element. Therefore, we examine its children, and find that input element 6, with content “G”, corresponds to our requested output element with index 4.

Thus, by re-interpreting the reduction sums as index ranges, establishing a top-down hierarchy, and descending into those children that cover the index we are looking for, we can locate the input element that corresponds to a given output location. In short, we have built a lookup structure from the reduction that originally counted the relevant input elements – after allocating the output, we can thus fill it with a top-down traversal of this very reduction output.

We will show later that this algorithm solves feature list generation considerably faster on the GPU than comparable hybrid CPU/GPU- or CPU-based solutions. Of course, this only holds for sample data already residing in graphics memory, which might seem as a restriction for practical applications. But graphics hardware is often used for fast signal processing anyways, as the GPU heavily outperforms the CPU on these data-parallel tasks. Therefore, we see the graphics memory input storage as no obstacle for real-world applications.

Note that above figures do not correspond to the actual OpenGL implementation on graphics hardware, which must use two dimensional arrays, textures, to gain optimal performance. This 2D algorithm will be described and used in the following sections.

6.2. Overview

Figure 6.4 illustrates the workflow between the different computation steps. All data is being processed on the GPU – the CPU only handles data if the input data originates at the CPU, or if the algorithm's output shall be downloaded for further processing in a non-GPU application.

Note that in contrast to the introduction, two-dimensional arrays are used for input, output, and reduction pyramid. While the algorithm is independent of dimensionality, it is implemented as two-dimensional to fit graphics hardware paths best. On the GPU, they are implemented as 2D textures.

Interestingly, the data structure is identical to the reduction pyramid that counts the histogram of features in Section 5.3.. We continue using its name **Histogram Pyramid**, or **HistoPyramid** even here.

The **Classifier** determines if a cell's content is regarded as relevant for the list or not. It generates an integer array of the same size as the input, which acts as the base level for the HistoPyramid. The array cells may be of arbitrary type (single/RGBA, byte/float), as long as the Classifier is able to handle cells of such type.

HistoPyramid buildup creates the pyramid levels through 2D reduction. Starting with the base layer, at the resolution of the original input image, it repeatedly adds up groups of four input cells throughout the whole input image, until only a single cell remains.

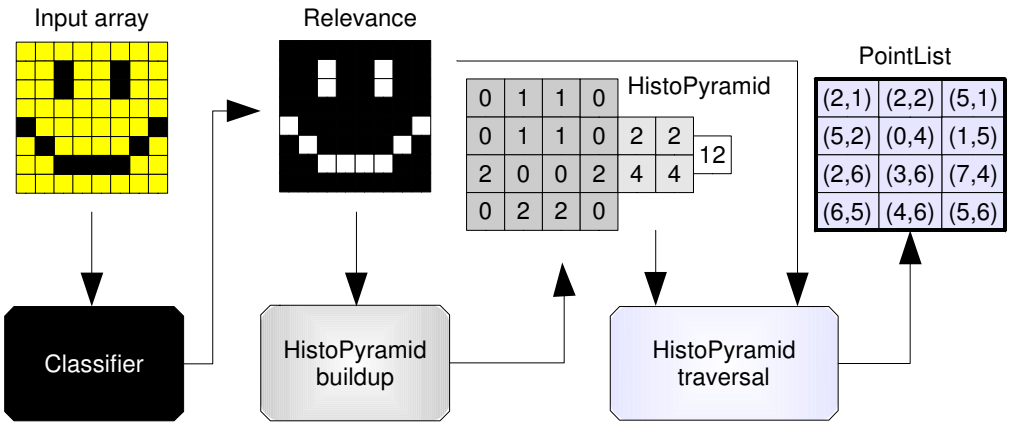


Figure 6.4: Workflow for a simple example of HistoPyramid-assisted list .

HistoPyramid traversal or List Extraction creates the output, here called PointList. First, it creates an empty list in the form of a 2D array, based on the number of input elements that have been counted as relevant. Afterwards, it fills the list via a top-to-bottom traversal of the HistoPyramid, starting one thread for each output element. We describe the details of traversal in diagrams, and present a log of traversal decisions in two examples.

Later, we summarize the current performance results that we obtained by running the algorithm's variants on graphics hardware, and analyze the runtime behaviour. To test the algorithm's real performance, we introduce *Dicer*, a tutorial application which converts arbitrary 3D models into particle clouds in real-time. We also present another research prototype from volume processing, where GPU data compaction was used in volume streamline extraction.

Data compaction, which might sound simple from the CPU perspective, opens a whole new range of other applications if available for graphics hardware. Therefore, we detail several real-time applications that become feasible with GPU-based data compaction.

Major algorithmic extensions will follow in later chapters.

6.3. Related Work

The hierarchical building process for the mentioned HistoPyramid is adopting the well-known “parallel reduction operation”. It is usually applied in mipmapping [BP04], and processes n^2 elements in $\log_2(n)$ passes. Our reduction operator uses additions to build a pyramid of partial histograms.

Image pyramids have been used in Binary Tree Predictive Coding before [Han85], [Rob97], where they served as an early terminator for transmission. For example, after reduction that built a quad tree, a quad tree leaf can signal that all of its descendants are identical, and therefore skip the transmission of its descendants. Our algorithm uses similar ideas to skip empty regions during output list generation.

Bitonic merge sort can be used for point isolation in sparse images by giving seed points a different sorting key than invalid points [GHL*04]. However, since this sorting algorithm is optimized for a plentiful of key values, it runs suboptimally ($O(n(\log n)^2)$ steps) for input streams requiring compaction, where only a binary partitioning between 0 and 1 is required.

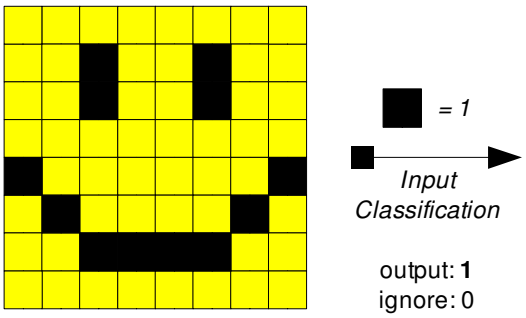
Horn's early approach to GPU-based stream compaction uses a prefix sum method to generate output offsets for each input element [Hor05]. For each output element, the corresponding input element is gathered using binary search. While the gathering approach, shows similarities to our traversal, output offset generation has a complexity of $O(n \log(n))$ and does not perform well on large datasets. Horn's approach requires $\log_2(n).n$ elements to represent its intermediate data. With our mipmap-like data structure and by producing at maximum 2n elements, a considerable reduction in temporary data size is achieved. However, while Horn's approach requires only the last level of his intermediate results, we have to use *all* intermediate data levels to generate the output - but this is fortunately a graphics hardware feature, which separates it from classic streaming processors.

Lefohn proposes a 1D version of data compaction with a similar reduction as ours [Lef06]. It utilizes the concept of prefix-sum scan. Prefix Sum (Scan) uses a pyramid-like up-sweep and down-sweep phase, where it creates, in parallel, a table that associates each input element with output offsets. Then, using scattering, the GPU can iterate over input elements and directly store the output using this offset table. Harris designed an efficient implementation in CUDA [Har07]. However, it requires data scattering, which is not feasible in a pure OpenGL environment, and reserved for later data-parallel computing languages such as CUDA.

[BH95] defines an algorithm for *interval allocation*, which runs on PRAM machines (parallel random access machines) and conducts a task similar to Harris' approach. By allowing holes in the output allocation, it can further speed up output index generation. Output *writing* is not mentioned explicitly, which is why the author probably assumes that scattering is used. While scattering is just a simple operation in the PRAM model, it hits the same performance obstacle in OpenGL as Harris does.

6.4. Classification

The subsequent stages operate on *integer* data, where each input element that is deemed relevant is marked with a value of "1". Therefore, we must first preprocess our input. In our example in Figure 6.5, a color template is applied to color input samples to find all black image pixels. These results in a "1", signalling output relevance, while all other values result in a "0". Beyond this simple classification, any operator which maps 2D image cells into integers can be utilized here. We refer to [ZTTS06] for a more extensive survey of classifiers.



0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0

Figure 6.5: A classifier converts input samples into a HistoPyramid base level.

6.5. HistoPyramid Buildup

As already described, the HistoPyramid is a pyramid of partial sums, with the Classifier's output as its base. On this *base level*, each cell is treated as list-relevant ("1") or irrelevant ("0"). To create the other levels, reduction is applied. The reduction operator repeatedly sums up four underlying cells and writes the result into an output array of half the input's size, until the top level only consists of a single cell. At this final pyramid level, reduction terminates. In our example in Figure 6.6, three levels are created until only a single top cell remains. Since reduction was used to sum up all of the classifier output, the value in this top cell contains the *output element count*. Thus, the output list can now be allocated from this output element count. Since we use a two-dimensional layout for the output, a certain over-allocation of output elements can occur.

For the GPU, a 2D texture with 32bit float values is the most optimal format.

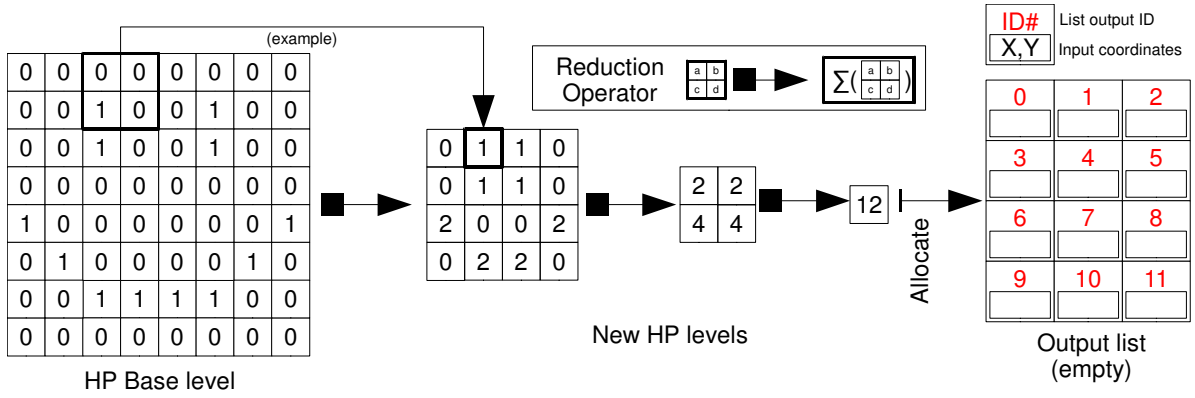


Figure 6.6: Basic HistoPyramid building process.

6.6. HistoPyramid Traversal

After allocation, output generation can commence. For every output element to be written, one thread is started. Each thread is assigned one single output element to write to, the *output index* **k**. In our example setup in Figures 6.7 and 6.8, twelve threads are started in total.

Each thread now accesses the HistoPyramid to find the corresponding input element for the output. At first, the top elemen is read. This element holds the total output element count. If k exceeds the output count, the thread immediately terminates. The reason for this safety check is that in a 2D layout for the output, unused entries can exist due to allocation spill.

The thread will now traverse the HistoPyramid from the top down to the corresponding input element at the base level. Traversal requires several more variables: We let **m** denote the number of HistoPyramid levels. The traversal maintains a 2D coordinate **p** and a current level **level** to address the HistoPyramid. It also keeps an index offset **off**, to describe an offset to the investigated cell values during traversal.

Traversal starts at the HistoPyramid's top level, **level = m,** and **p** points at the single cell in the top level. Since traversal is newly started, the accumulated offset **off**=0.

Since **k** was deemed valid in the safety check, traversal now descends down the pyramid. It decrements **level**, and after trivial upscaling, uses **p** now addresses a 2×2 block of cells, the children

of the previous cell. We label these four cells as $\begin{matrix}\mathbf{a} & \mathbf{b}\\ \mathbf{c} & \mathbf{d}\end{matrix}$ and use the values of these cells to form index ranges **A**, **B**, **C**, and **D**, defined as $\begin{matrix}[A] & [B]\\ [C] & [D]\end{matrix} = \begin{matrix}\mathbf{[0,a)} & \mathbf{[a, a+b)}\\ \mathbf{[a+b,a+b+c)} & \mathbf{[a+b+c,a+b+c+d)}\end{matrix}$.

Then, it examines which of the four ranges **k-off** falls into. If, for example, **k-off** falls into range **B**, we make **p** point to cell **b** and add **B**´s range start to **off**, here: **off=off+a**. This adapts the local index range for the next iteration. Now traversal descends in the HistoPyramid, by subtracting one from **level**, adapting **p**, and repeating the range evaluation. This repeats until the base level is reached, with **level** = 0. When traversal terminates there, one final range evaluation is done. **p** now points to a cell in the base level, which holds the corresponding input element for **k**. Due to the accumulation in traversal and the final range evluation, **off** equals the aftersought output index **k**. The input coordinate **p** can now either be written directly to the output, or it is used to retrieve additional information from the original input.

We now provide two examples for HistoPyramid traversal.

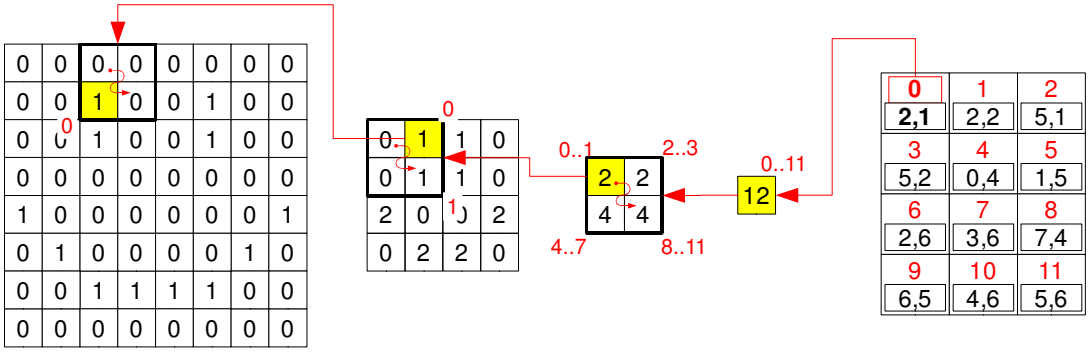


Figure 6.7: HistoPyramid-assisted list generation, example traversal for entry 0. Yellow: chosen subcell. Red: entry indices, calculated during traversal.

The first example, Figure 6.7, is traversal for output index k = 0. We start at level 3, and enter the pyramid since k < 12. Next, we descend to level 2, with the offset still at **off=0**. The four cells at level 2 form the ranges **A** = [0 , 2), **B** = [2 , 4), **C** = [4 , 8), **D** = [8 , 12). We see that **k** is in the range of **A**. Thus, we adjust **p** to point to the cell a, the upper left cell. **off=**0. Then, we descend to level 1. The four cells there form ranges **A** = [0 , 0), **B** = [0 , 1), **C** = [1 , 1), **D** = [2 , 3). The ranges A and C are *empty*, since they both include *and* exclude the single index they cover. As **k** falls into range B, we make **p** point to cell b, with the offset still remaining **off**=0. The group of four texels in the base level form the ranges **A** = [0 , 0), **B** = [0 , 0), **C** = [0 , 1), **D** = [1, 1). The ranges A and B are empty. Consequently, **k** = 0 falls into C. Since we are at the base level, a final range evaluation is done without further descend, resulting in **p** = [2,1].

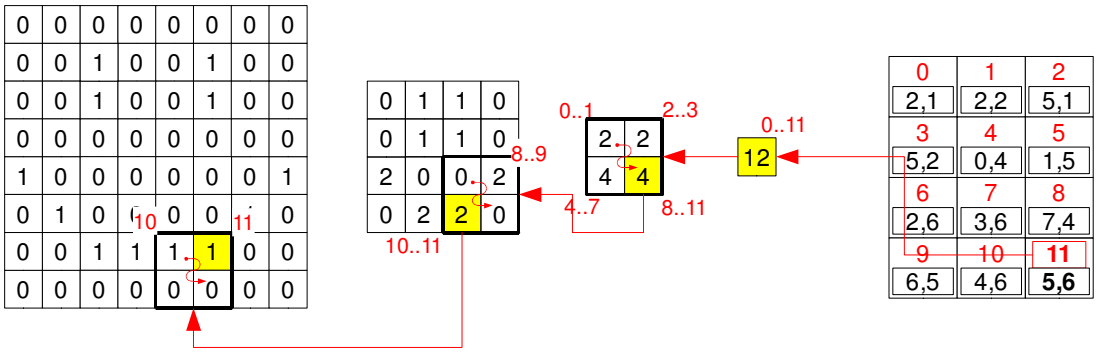


Figure 6.8: Example traversals for HistoPyramid-assisted list generation, entry 11.
Note how the local traversal order, shown as red round arrows, affect the global indexing order.

In the second example, Figure 6.8, we follow traversal for key index **k** = 11, where the local index offset **off** will come into play. We begin at the top of the HistoPyramid and descend to level 2. Again, the four cells form the ranges **A** = [0 , 2), **B** = [2 , 4), **C** = [5 , 8), **D** = [8 , 12), and **k-off=11** falls into range **D**. We adjust **p** to point to cell d, and add **D**'s range start 8 to **off**, resulting in **off**=8. Then, we descend to the level 1. The four texels there form **A** = [0 , 0), **B** = [0 , 2), **C** = [2 , 4), **D** = [4 , 4). k-off=11-8=3, which causes us to choose range **C**, with **off=off**+2=10. The four cells in the base level, form the ranges **A** = [0 , 1), **B** = [1 , 2), **C** = [2, 2), **D** = [2, 2). Here, **k-off** = 11-10 = 1 which falls into range **B.** We adjust **p** the last time. Traversal terminates, with **p** = [5,6].

We are thus able to generate all output elements independently, as there are no data dependencies between traversals. All threads only read from the HistoPyramid, which has thus become a read-only data structure.

The final result is an array, containing data and/or coordinates of all input array cells that classification had originally selected. In OpenGL, this traversal is implemented through a shader for the output array, which is stored in a 2D texture. After invocation, each shader thread uses its 2D coordinate of execution to deduce the output index **k**.

6.7. Discussion

6.7.1. Traversal Order

Note that in the two-dimensional case of our presented traversal, output order of elements is not the same as the input order. But for data compaction, element order is irrelevant, as long as all input elements are included in the output. This ensured because all HistoPyramid traversals use the same traversal order, shown as rounded arrows in Figure 6.8. In the description, we traverse from left-to-right and top-to-bottom, a zig-zag traversal which creates a self-similar indexing pattern, also known as Z-order, or Morton-order [Mor66]. If input order shall be retained, the algorithm can of course also employ row-wise traversal. In that case, the reduction operator processes four horizontal cells instead of a square of 2x2. However, hampers texture cache performance during HistoPyramid buildup and traversal, and requires special row-crossover handling for the 2D layout.

6.7.2. Caching Considerations

The 2D texture layout of the HistoPyramid fits the native capabilities of graphics hardware. At each descent during traversal, we inspect the values of four texels, which amounts to four texture fetches. In the domain of normalized texture coordinate calculations, it can be easily seen that the texture fetches for a parent cell overlap with fetches for its children from the level below. Therefore, many of the parallel shader threads access the same texture locations, especially in the beginning of

traversal. This allows the 2D texture cache to assist by serving many threads simultaneously, and thus increases the algorithm's traversal performance.

6.7.3. Complexity

The 1D version of our algorithm requires $4\log_4(input_{count})$ lookups from the HistoPyramid, while the 2D algorithm implemented on the GPU requires $4\log_2(max(input_{width}, input_{height}))$ HistoPyramid accesses to generate one output element. The binary search used in Horn's approach [Hor05] always uses $\log_2(input_{count})$ accesses. In contrast to Horn's approach, we can also modify traversal complexity by either increasing dimensionality (here: from 1D to 2D) or reduction factors (here: 4-to-1). For example, a three-dimensional HistoPyramid would yield $8\log_2(max(\text{size}_x, \text{size}_y, \text{size}_z))$ in traversal complexity, since a parent has 8 children cells, if a 2x2x2-to-1 reduction is applied to build the 3D HistoPyramid. A modified two-dimensional HistoPyramid with reduction factor 3x3-to-1 would have complexity $9\log_3(max(\text{size}_x, \text{size}_y, \text{size}_z))$, since for every iteration, each sidelength is reduced by a factor of 3, and at every level, 3x3 children cells have to be examined.

6.7.4. Variants

Several optimizations of the algorithm are possible. Since we always fetch 2×2 blocks, we can also use a four-channel texture and encode these four values as RGBA values instead [ZTTS06]. This halves the size of all levels along both dimensions, and allows four-times larger HistoPyramids within the same texture size limits. In addition, since we quarter the number of texture fetches, we increase cache coherence, and reduce complexity to $\log_2(max(input_{width}, input_{height}))$. Graphics hardware is very efficient at fetching four-channel RGBA values which usually implies a traversal speedup..

6.7.5. Memory Requirements

The memory requirements of a HistoPyrampid with a 4-to-1 reduction ratio are similar to including a 2D MipMap-pyramid with a 2D texture, i.e. 4/3 of the input array's size. For example, if the input array would have 512x512 elements, then the base level of the HistoPyramid would be 512x512 elements, too, and the other levels 256x256, 128x128, etc. Other reductionratios (e.g. 2-to-1, the 1D case) have a different size limit, but in general, the overall memory required never exceeds twice the size of the base level. One interesting insight is that since the number of required bits is small at the lower levels, a composite pyramid could use 8 bit for the lowest levels and16 bit for some levels inbetween, before using 32 bit values for the top levels.

6.7.6. CPU-GPU bus bottlenecks

In volume processing, the location of features in an three-dimensional array of samples is a common task. Often, a *filter kernel* is applied to the sample volume, with a certain *filter output criterion* that determines where desirable features are located. The task is then to create a *feature list,* with the locations of all desired samples.

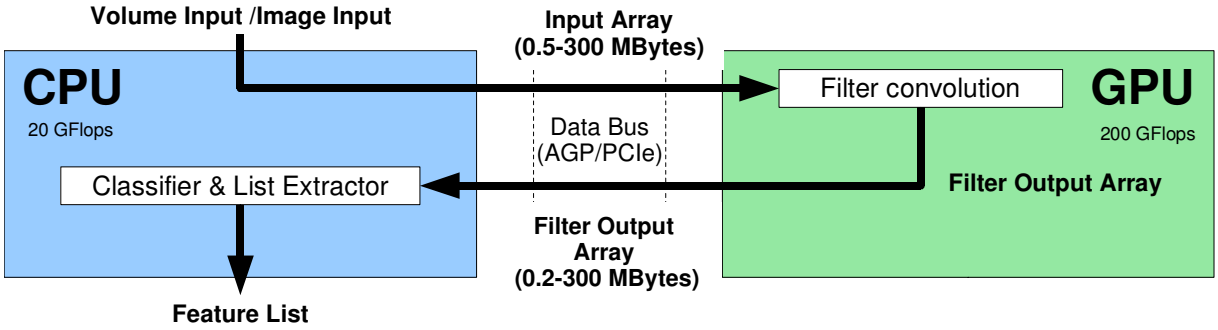


Figure 6.9: Previous approaches to CPU/GPU computer vision only used the GPU for simple data filtering, requiring heavy bus transfers to solve the remaining tasks on the CPU.

Already previously it had been possible to use the GPU for the input sample convolution, a very local operation, which can be applied to each sample in parallel. However, feature list generation requires a dynamically growing list, created from serial processing of samples, and could thus not be easily parallelized and run on graphics hardware. Hence, GPU-assisted volume processing applications have traditionally delegated this task to the CPU. However, with the GPU only responsible for the sample pre-processing, much of the performance gains were eaten up by massive data transfers, see Figure 6.9.

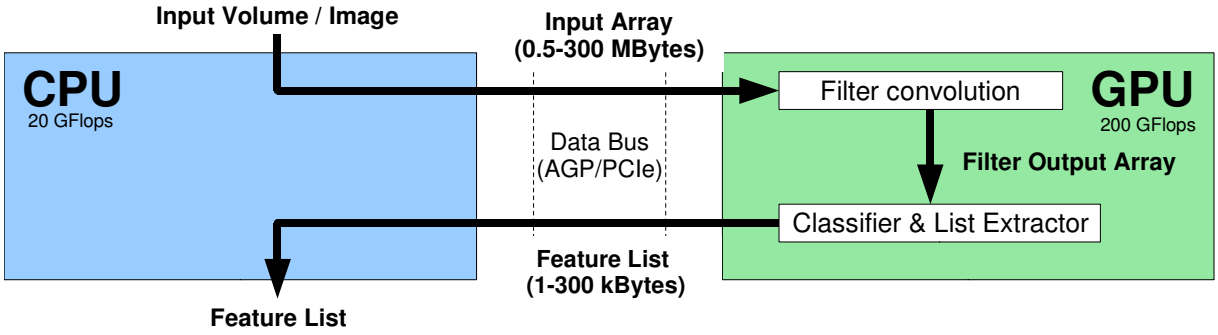


Figure 6.10: Sophisticated image processing, enabled by GPU data compaction, substantially reduces bus loads.

To eliminate this bottleneck, feature list generation has to be ported to graphics hardware. As already discussed in this chapter, HistoPyramids allow the creation of feature lists on the GPU. As Figure 6.10 shows, the expensive download of half-processed image data can thus be avoided, and only the discovered feature list needs to be transferred.

6.8. Implemented Applications

In the following, we demonstrate how data compaction proves useful in the generation of irregular data structures from regular input data. We chose to implement two applications from the domain of computer graphics and volume processing due to their intuitive task setting. Without data compaction techniques, the last computing step of both applications would have to be run on the CPU, and had obstructed interactivity for any but the smallest mesh and volume sizes. Note that the usability of GPU-based data compaction reaches beyond these domains, and is a necessary component in any application that requires the selection of elements from a larger input array of data.

6.8.1. Point Cloud extraction

One of the most obvious applications for HistoPyramid-based data compaction is feature detection. Since we use a computer graphics API, the most intuitive way to visualize such feature detection is to use graphics; therefore, we created Dicer, an application to create point clouds from 3D meshes, in *real-time*. This might sound trivial, and certainly, one could simply take the mesh's vertex positions to create a point cloud. But through the voxelization into a 3D volume, we are able to ensure a *minimum sampling rate* of the object mesh and thus create a *dense* point cloud, independent of the mesh triangles' sizes.

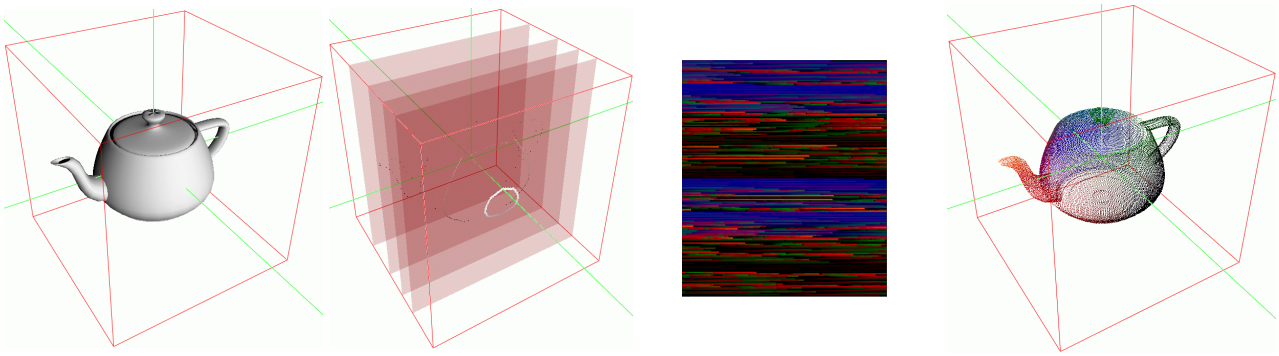


Figure 6.11: Dicer decomposes a 3D model into a point cloud on the GPU.
Slice rendering produces the 3D mesh voxelization into a 2563 volume.
The teapot surface occupies ca. 34000 points, which are extracted via GPU data compaction.
Left to right: 3D model of teapot; volume slices at z=70/130/190/201; feature list as 2D array;
particle cloud based on list. The false colors stand for the voxels' 3D coordinates.

In order to test the real-time behaviour of our algorithm, we implemented *Dicer*, a small Linux application that converts 3D models to point clouds. The 3D model, a teapot generated with `glutTeapot(0.6)`, is stored in a display list to maximize geometry throughput during voxelization - of course, any 3D mesh could be used as input to the algorithm.

The software voxelizes the mesh by rendering it into 256 2D slices of 256x256 each, spanning a volume of [-1,-1,-1] to [1,1,1] in world space (see also Figure 6.13). The output is put into 16 x 16 tiles of an 8-bit RGBA texture at 4096x4096 resolution. All pixels belonging to the 3D model are marked with `alpha=1.0`.

After voxelization, the algorithm analyzes the resulting 2D texture, and starts by building a HistoPyramid on top of it. Alpha thresholding is a local and inexpensive operation, and was therefore integrated into the first stage of the HistoPyramid buildup. This saves storage space and calculation time, since classified results never need to be written to video memory.

HP traversal bears two modifications from the original algorithm: After the 2D coordinates of occupied voxels have been located in the texture atlas, we convert them back into 3D voxel space coordinates and output these coordinates instead. Since HP buildup had never written the classification output explicitly, HP traversal has to *redo* these operations on the base level to determine the correct target cell. Therefore, it is only advisable to omit explicit base level writeout if classification is negligible in comparison to writing and re-reading the base level.

HP traversal's output contains all the input mesh's surface points in one compact 2D texture. For visualization purposes, they can be rendered as a GPU particle system, simply by feeding a number of OpenGL points to the vertex shader, and having it read the actual vertex positions from the point cloud texture that had just been generated by HP traversal.

The tests were conducted on a Dell Precision M70 laptop with Nvidia Quadro FX Go 1400 and 256 MB video memory, connected over PCI Express. It contained an Intel Pentium M (2.13 Ghz) and 2 GB of main memory. AGP download timings came from an Athlon XP2400 system with an Nvidia GForce 6600, AGP 8x.

Additionally, we experimented with smaller texture sizes to measure the performance scaling, effectively producing volumes of 256x128x128 (2048x2048) and 256x64x64 (1024x1024).

We compare three variants of the algorithm:

dicer_single is the most classic implementation, and follows the basic algorithm. It uses the OpenGL texture format GL_TEXTURE_2D, which provides mipmaps and render-to-texture, but current restrictions force it to build the HistoPyramid in a 32 bit-float RGBA texture, even though only one data channel is used.

dicer_vec4 is similar, but makes better use of the four 32-bit components by storing partial sums in the RGBA vec4, effectively delaying the cell sum-up by one level (see also Section 6.7.4). This accelerates PointList Builder, as the tree traversal has do to less texture lookups to make its branching decisions.

dicer_rect utilizes GL_TEXTURE_RECTANGLE, a texture format without mipmaps - but render-to-texture allows 32-bit single float textures here, which saves considerable amounts of memory. Since PointList Builder needs to access all levels in one pass, we were forced to create a pseudo-mipmap layout in a single texture (see Figure 6.12).

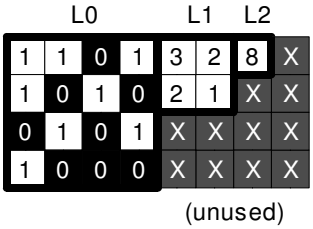


Figure 6.12: dicer_rect's pseudo-mipmap layout for rectangular textures without mipmap capability.

We could not test the algorithm on ATI graphics hardware without intense redesign due to the ATI driver's OpenGL API restrictions.

Table 6.1 lists the timings common to all implementations. Slicing (the voxelization of the 3D model into a 2D texture atlas) dominates all timings, presumably due to the repeated geometry processing. GPU timings were acquired via the GL_EXT_timer_query extension [Gre05b].

For Table 6.2, we implemented a classic CPU loop to compare our GPU algorithm with a standard single-thread implementation. Here, the CPU downloads the 2D volume texture as RGBA8 (over AGP or PCI express, depending on the system), generates an output list after line-wise traversal, and uploads it to the GPU again. Aggressive compiler optimization accelerates the CPU based analysis. No SIMD techniques were used. CPU timings were taken as virtual process time by the `getitimer()` function of Linux systems. It is clear that for large textures, the texture download greatly outweighs the actual analysis.

Table 6.2 shows the GPU time spent for creating the HistoPyramid. Both dicer_single and dicer_vec4 suffer heavily from the restriction to RGBA, 32-bit float textures: The texture data is obviously being swapped to main memory, causing large performance penalties for 4096x4096. Only dicer_rect can use single-component 32-bit float textures, and therefore scales as expected.

Even without memory restrictions, render-to-texture shows to be considerably faster for single components than for RGBA.

	4096x4096	2048x2048	1024x1024
Voxelization	470 ms	470 ms	470 ms
Voxel count	33989	8595	2130

Table 6.1: Timings for voxelization, required preprocessing for CPU and GPU variant.

	4096x4096	2048x2048	1024x1024
AGP fetch	560 ms	142 ms	36 ms
PCIe fetch	172 ms	40 ms	12 ms
CPU traversal	25 ms	25 ms	24 ms

Table 6.2: CPU: Timings for whole extraction traversal.

	4096x4096	2048x2048	1024x1024
dicer_single	~2000 ms	20 ms	6 ms
dicer_vec4	~2000 ms	20 ms	6 ms
dicer_rect	30 ms	10 ms	2 ms

Table 6.3: GPU: HistoPyramid creation (first part of traversal).

	4096x4096	2048x2048	1024x1024
dicer_single	16 ms	12 ms	~6 ms
dicer_vec4	14 ms	7 ms	~6 ms
dicer_rect	9 ms	6 ms	~2 ms

1. **Table 6.4:** GPU: Point list extraction (second part of traversal).

Finally, table 6.4 documents the timings of point list extraction. Here, results are more comparable, and dicer_vec4 can outperform dicer_single due to its improved traversal algorithm. However, dicer_rect *outperforms both*, and as soon as dicer_single is able to render to single-component textures, it will probably be in the same speed ranking. Therefore, additional tests are required to verify the gain of dicer_vec4's increased storage and bandwidth consumption for volume analysis. The situation can be different for future binning operations, where the whole volume of data needs to be rearranged and no data will be thrown away.

An intermediate data compaction on the CPU actually slows down processing, which is verified by *summing up* the timings of table 6.3 and 6.4, and comparing them with table 6.2: For example, a volume data fetch over the PCI express bus and CPU-based data compaction would take 172+25 = **197 ms**, while the same can be achieved in 30+9 = **39 ms** on the GPU.

GPU-based image/volume analysis has thus become competitive with the help of HistoPyramids. The speed advantage is only small for medium-sized textures, but for large textures, the impact for CPU texture download is so profound that it pays to process the data on the GPU. Further, it saves both memory and CPU time to have the GPU process data that already resides in graphics memory, such as the algorithm's output from mesh voxelization.

Finally, in our real-time demo *HeartBreaker*, which we used in our publication [TZ06], we demonstrate how our method can deliver new and unusual visual effects, such as particle explosions of arbitrary geometry models. It works similar to *Dicer*, except that it downloads the particle cloud to the CPU to animate it there. See Figure 6.10 for screenshots of the animation sequence.

Figure 6.13: Screenshots from our FX demo HeartBreaker.
Left to right: Solid (5000 triangles); Point cloud (1872 points, in a 256x64x64 grid, first iteration: 90 ms, subsequently: 25 ms); Particle explosion effect.

6.8.2. Vector Field Contours

In the forthcoming evaluation of real-time 3D volume visualization, graphics hardware is utilized for the generation of *vector field contours*, a visualization concept that extends the notion of surface contours to 3D vector fields. By selecting stream lines that run perpendicular to the current observation angle, view-dependent surface contours for a vector field are extracted, see Figure 6.14 for a first example. In the forthcoming summary of this work, we concentrate on the involvement of graphics hardware in the overall algorithm; in particular, we explain the role of HistoPyramids in the selection of seed points for the surface contour lines. A more extensive explanation of the proposed NPR technique and its mathematical background can be found in the related publication [ATR*08].

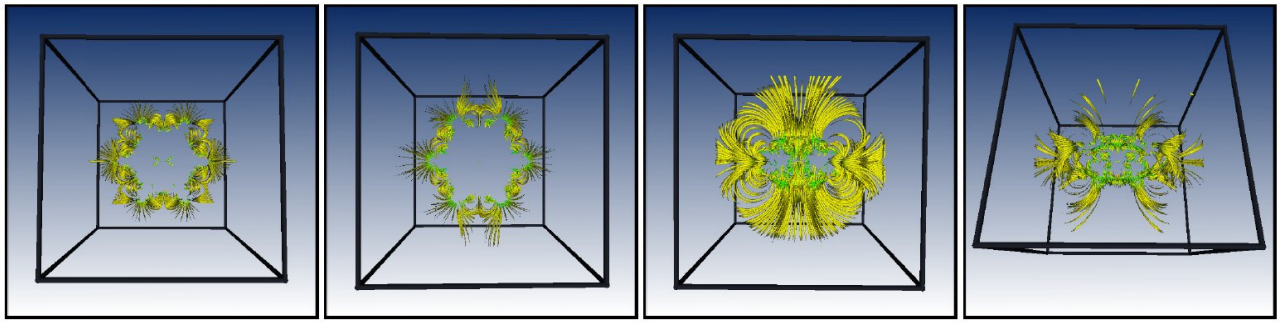


Figure 6.14: Vector field contours of the electrostatic field around a benzene molecule from different view directions. Our new technique allows selection of those stream lines which have a tangent direction and osculating plane perpendicular to the current view direction, seen from the seeding point.

6.8.2.1. Related work

Any direct approach to streamline visualization must carefully select the most significant stream lines for the rendering, even for moderately complex input. This leads to stream line seeding methods, which are well-established for 2D vector fields. However, good direct visualization of 3D flow fields is much more involved, and we are so far only aware of one nontrivial seeding strategy [YKP05]. At the same time, NPR techniques and contours in particular have emerged as a powerful tool for the visualization of surfaces and 3D scalar fields. Here, the prioritization of features is guided by human visual perception, which leads to only relatively few line primitives being required to convey the essential information. We are aware of one other approach that takes advantage of such NPR-like techniques to emphasize features in 3D flows [SJEG05].

6.8.2.2. Overview

Extracting seeding structures associated with a certain view direction involves several GPGPU rendering passes before the visualization rendering can be generated. An explanatory overview of our GPU-based rendering of vector field contours is given in Figure 6.15.

The input to the algorithm is a 3D vector field, designated by **v** and stored in a 3-dimensional grid. From this input, we derive two supplemental vector fields over the same grid: The vector movement tendency **w = (∇v) v**, and curvature samples **κ**.

w uses central differences to estimate **∇v** in the grid points. During read-out, samples of **w** are trilinearly interpolated inside the grid cells.

The 3D position-dependent, but scalar curvature values **κ** are computed as $\kappa = \frac{\|v \times w\|}{\|v\|^3}$.

Note that **w** and the scalar field **κ** are not dependent on the viewing direction, and can thus be precomputed once for a given data set.

Beyond **v** and **w**, the viewing direction vector **r** is necessary for streamline selection.

Now, the streamline seedpoints **l** can be extracted as follows: First, we want to select all streamlines from the vector field that act as *vector field contours for a given viewing direction* ***r***. We cannot tentatively generate all possible streamlines; instead, we need to select those seedpoints that will have the highest probability for generating vector field contours after extrusion into a streamline, see the right of Figure 6.15.

To act as a good seedpoint for a vector field contour, a given vector field location must hold a vector perpendicular to the viewing direction. It must also *retain* such a perpendicular direction even after extrusion into a streamline has progressed one step further. **w** indicates the vector forthcoming direction change. Thus, we are interested in all intersections of the iso-surfaces **r ·v=0** and **r ·w=0**.

This is achieved in several steps, as the left of Figure 6.15 outlines:

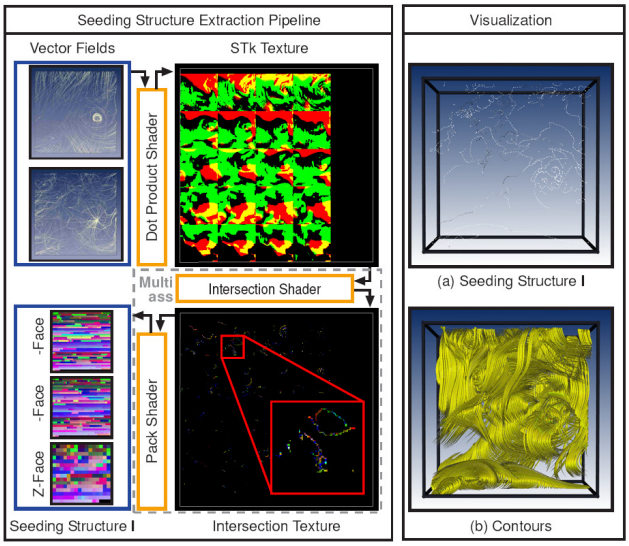


Figure 6.15: The pipeline for vector field contour computation (left) and visualization (right). A multi-pass process locates intersections of tand converts sparse intersection textures into a packed seedpoint list **l**. To the right, (a) shows the seedpoints found during the extraction process, while (b) illustrates the final contours that were extruded from these seeding points, rendered as streamtubes.

The *DotProduct Shader* computes the dot product between view direction **r** and vector field **v**, and **r** and **w**. Results are stored in a new array with three channels : S = **r** ·**v** is stored in the red channel, and T = **r** ·**w** in the green channel. The blue channel, labeled k, contains the view-independent constant κ which is only filled in once. On the GPU, this array is stored in an off-screen texture, and hence labelled STk-texture. The three dimensions of the array are packed as a Flat3D layout into a 2D texture [HSBL03].

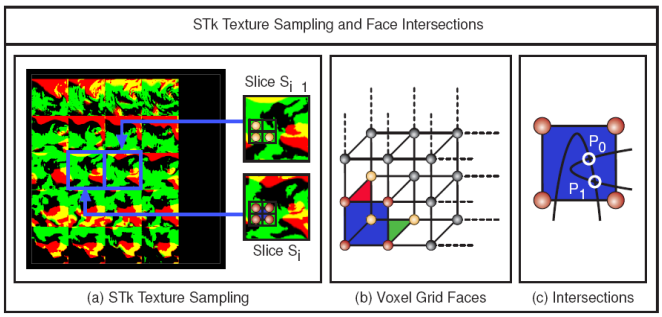
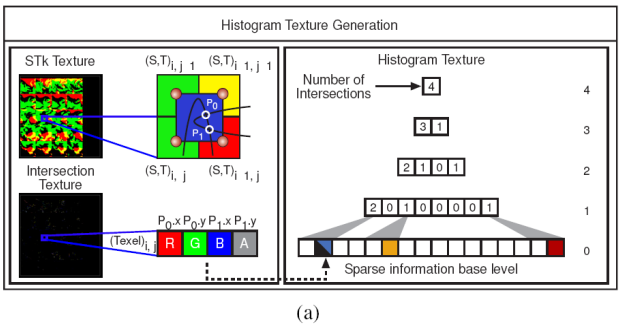


Figure 6.16: Grid cell faces and adjacency sampled in 2D.
Four vertices from the z-face correspond to four texels in the STk Texture slice Si.
The other three vertices, required to compute all intersections, are located in the next slice Si+1.

For later computation, each entry in this **STk-texture** represents a vertex of a grid cell which is shared by three cell faces, see Figure 6.16.

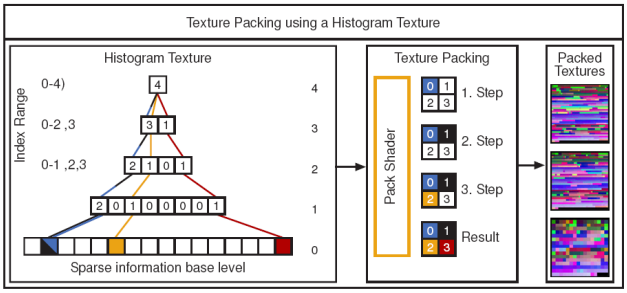
The *Intersection Shader* now determines in three rendering passes where the isosurfaces S=0 and T=0 intersect in one of the three cell faces. Due to the hyperbolic nature of the isosurface, only two intersections P0 and P1 are possible in each grid cell. The output is the Intersection Texture, a texture with only very few texels that actually encode intersections.

While the Intersection Texture already contains the seed points required for final visualization, it cannot be downloaded to the CPU for streamline generation due to bus bandwidth limitations. Instead, we let the Pack Shader take advantage of HistoPyramid-based data compaction to generate the final, compact seedpoint list **l**. Figure 6.17 explains this in detail: Using the 2D coordinates of the Flat3D mapping, each entry of the intersection texture encodes up to two face intersections in its four color channels. The found intersections are now counted through data-parallel reduction, building up a HistoPyramid inside the Histogram Texture.



(a)

Figure 6.17: Building a HistoPyramid from the Intersection Texture. Left: Encoding of iso-surface intersections into texture channels. Right: Counting the intersections in a HistoPyramid, as stored in a Histogram Texture.



(b)

Figure 6.18: The Pack Shader utilizes a Histopyramid traversal to extract the compact seedpoint list from the sparse Intersection Texture.

Figure 6.18 explains how the Pack Shader extracts the compact seedpoint list from the Histogram Texture and Intersection Texture: As usual in HistoPyramid traversal, it re-interpretes the entries of the Histogram Texture as output index intervals, and can thus locate the corresponding input entry in the Intersection Texture. While writing the final result, the shader converts the intersection point coordinates from 2D texture space into 3D object space coordinates, thus reverting the Flat3D mapping that was used during intermediate processing.

The seedpoint list $\mathbf{l}$ is downloaded to the CPU as soon as data compaction has completed. There, the necessary streamtubes are generated by extruding forwards and backwards from the provided seedpoints. To achieve a better consistency during view direction changes, kurvature $\boldsymbol{\kappa}$ can be used to shorten streamtubes before they completely disappear (as this is not GPU-based, we refer to the publication for more details [ATR*08]). The generated streamtube meshes are then submitted to the GPU for final rendering.

6.8.3. Results

To prove the feasibility of the implementation, we applied our approach to a number of data sets.

Figure 6.14 visualizes the electrostatic field around a benzene molecule from varying view directions. This data set was computed on a regular grid of 100^3 elements using the fractional charges method described by Stalling et al. [SS96]. This data set is highly symmetric and rather complex. It has been used for demonstration of other visualization techniques recently [TWHS03, WTS*05].

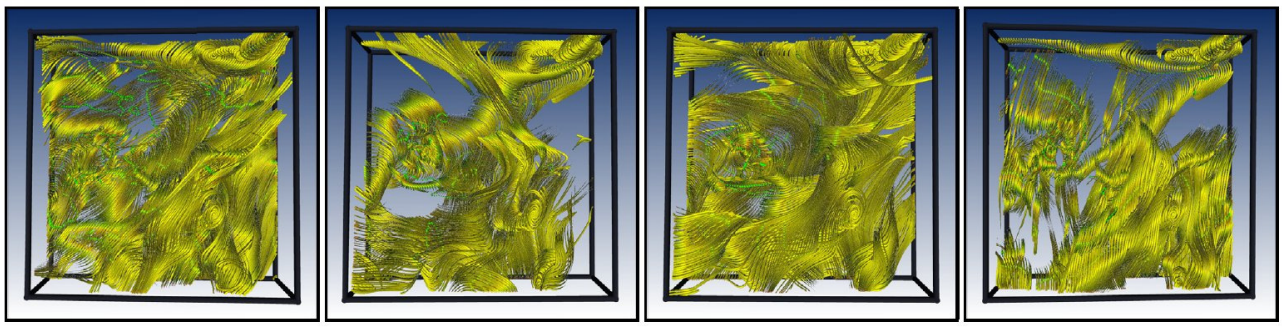
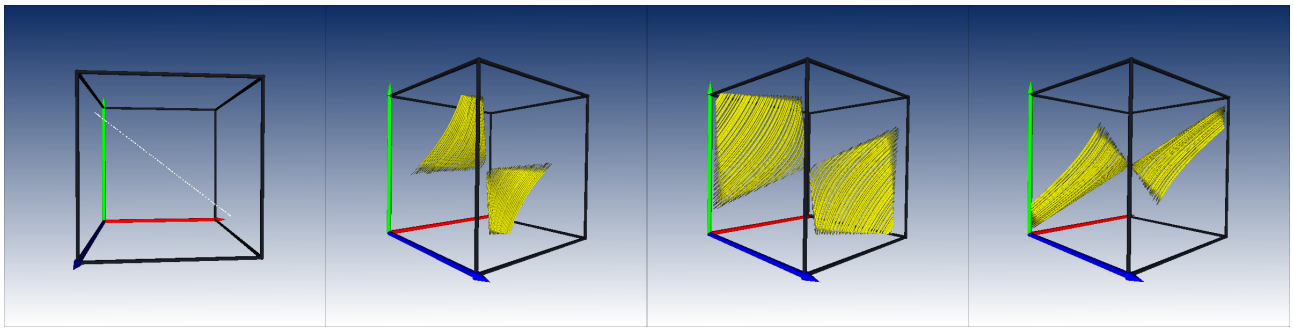


Figure 6.19: Hurricane Isabel data set: vector field contours rendered as stream tubes.
Please note that in certain views, the tunnels of the storm are well separated from other stream lines.

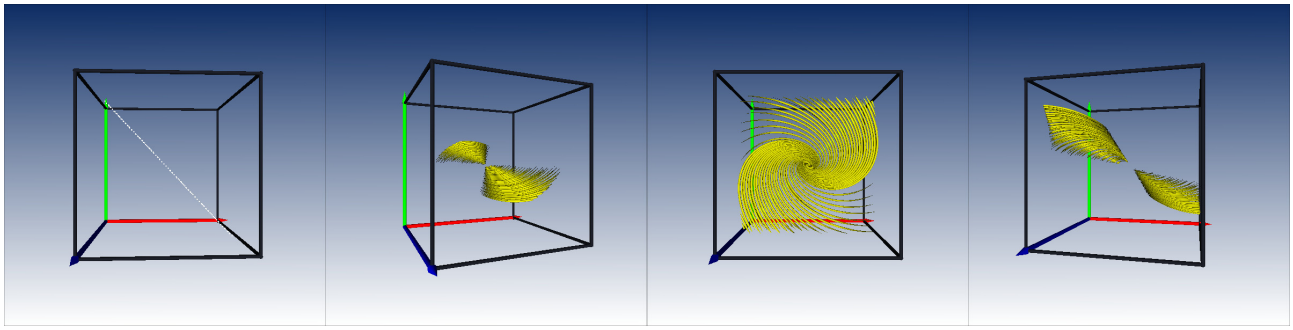
Figure 6.19 depicts visualizations of the Hurricane Isabel data set, which recently has been used in a number of papers [DMH04, GM04]. Using vector field contours, the wind tunnels are well visible

under the according view directions. We used the vector field at the starting time of the simulation on a regular grid.



(a) (b) (c) (d)

Figure 6.20: A saddle. (a) seed points. (b), (d) two sweeps through the eigenplane. (c) the eigenvector $(1,0,-1)^T$ of the linear vector field.



(a) (b) (c) (d)

Figure 6.21: A repelling focus. (a) seed points. (b), (d) Two sweeps through the eigenplane. (c) the eigenvector $(0,0,-1)^T$ of the linear vector field.

Figure 6.20 and 6.21 show vector field contours of a saddle and a repelling focus, respectively. Our visualization technique can reveal inherent structures of linear vector fields: by interactively moving view direction **r**, the eigenplanes can be found by searching for directions where the contours have a maximal number and length. This technique is explained and analyzed in more detail in our publication [ATR*08], which also contains one further visualization for the ABC (Arnold-Beltrami-Childress) flow field, which has recently attracted attention in the fluid dynamics community.

Name of Data Set	**Size**	**Extraction**	**Visualization**	**Extrusion Length**	**Seedpoints**
Isabel	100^2 x 50	30	2-8	200	< 5519
Benzene	100^3	21	5-8	250	< 9852
Saddle	64^3	64	8-16	400	< 186
Focus	64^3	63	11-13	400	< 180

Table 6.5: Summary of our example data sets. Columns show the name, size, extraction and visualization timings (both in frames per second), (maximum) stream line integration steps and number of seedpoints, respectively.

Table 6.5 lists the timings of our GPU-based visualization for the various datasets, as measured on a 2.6GHz AMD CPU and a GeForce FX7800 graphics card with 512 MByte memory, and confirm the efficiency of our approach.

Interframe coherence is improved by scaling the length of stream lines depending on a view-dependent relevance rating for each seed point [ATR*08]. Please see the supplementary videos for an illustration of the missing temporal coherence when such a mechanism is absent [VecCont].

Our results also show the considerable potential of vector field contours for flow visualization, using a sophisticated GPU implementation that enables real-time visualization. By utilizing latest graphics hardware features and a novel algorithm, we are able to map our entire seed point extraction to the GPU. In fact, seeding *and* rendering is achieved in real-time, which had previously not even been achieved for 2D vector fields. Further possibilities for performance tuning lie in moving even stream line generation to the graphics hardware by the use of data expansion techniques as described in Chapter 7.

Other research directions would be an extension of above approach to time-varying data, an extension with suggestive contours [DFRS03], and the introduction of other GPU-based interactive techniques, e.g. particle tracing to vector field visualization.

6.9. Other Applications

Due to the algorithm's generality, data sources of all kinds can be analyzed by graphics hardware. This enables interactive runtimes for several applications that were previously restricted to partial preprocessing and offline rendering.

6.9.1. Sparse Matrix Creation

Krüger et al. have demonstrated how to process large sparse matrices by packing them into a special representation [KW03]. Until now, it had not been possible to *create* such sparse matrix representation on graphics hardware. Our algorithm can be used to convert matrices into such sparse matrix representations, and thus save memory and computation time. With ideas from QuadPyramid processing (see Chapter 8) and push-pull techniques, even implicit multiplication of sparse matrices should be possible.

6.9.2. Level-Set Techniques

Even GPU-assisted level-set identification is unproblematic. To do this, the level set must be provided as an explicit function of space, providing a clear indication if a given point in space belongs to a level set **l**. Then, a classifier can be applied to extract all points belonging to this level set **l**, similar to the one used for mesh surface identification in Section 6.4. For implicit descriptions, we refer to Chapter 7, where a light wavefront simulation uses data expansion to make a mesh follow a level-set evolution, and continously adapts its tesselation so that a minimum sampling error is guaranteed.

6.10. Conclusions and Future Work

We have presented a novel and fast GPU algorithm for data compaction/feature list extraction from n-dimensional data arrays, including 2D images and 3D volume data.

We have shown that the generation of a *list* of selected elements in a 2D array is in effect a *data compaction* operation, with input elements being marked as *relevant* for the output stream (1) or not (0). By using an intermediate data structure that is built via data-parallel reduction, we can separate the output allocation from input retrieval, i.e. the search for relevant input elements. The HistoPyramid, a data structure that we had already presented in Section 5.3 proves to be this intermediate data structure. Output relevance classification is achieved by the Classifier, which assigns a "1" to all input elements that are relevant for the output list, and “0” to all others. After the base level of HistoPyramid has been initialized with these relevance values, reduction commences, until a single element remains at the pyramid's top. This single element holds the number of elements relevant for the output list. A HistoPyramid can thus be used to calculate the output list's

size, with its top element holding the output allocation size, it serves output allocation. But the second, more crucial observation is that the HistoPyramid can serves as an *implicit indexing data structure* for input retrieval: Its partial sums, created during reduction of the HistoPyramid levels, can be used to *locate* relevant input elements. The algorithm requires $4\log_2(max(\text{size}_x, \text{size}_y))$ HistoPyramid accesses to generate one element in the output list.

The 2D version of the HistoPyramid buildup and traversal fits perfectly with the 2D texturing capabilities of graphics hardware, and uses traversal during input retrieval yield high coherency in the texture cache. Since the algorithm is data-parallel and agnostic to the number of processors, it scales even for future GPU generations.

Through experiments we have also shown that for input generated by a GPU processing stage, such as volume data, data compaction in a purely GPU-based implementation is significantly faster than a CPU implementation, because bus transfers severely limit the maximum attainable performance in a hybrid CPU/GPU implementation. The GPU algorithm's intermediate processing stages do not matter, as GPU memory bandwidth and texture caching help to keep execution times low.

Additionally, we have applied GPU-based feature list creation in a new approach to 3D flow visualization, which uses vector field contours [ATR*08]. Our method computes certain seeding structures based on ideas from NPR, in particular, a novel notion of contours for flow data is used to select contour-like stream lines for rendering. Most importantly, and contrary to previous work, our visualization method is *not static* but designed as a tool for dynamic, view-dependent exploration of flow data.

On the whole, new visual effects for computer games are now feasible - the rapid generation of point clouds from 3D models provides a first example. But even non-graphics applications are easy to conceive, ranging from 2D image analysis and feature detection over 3D volume processing to improving efficiency in general GPGPU calculations through data compaction. The algorithm can thus complement software frameworks for graphics hardware in the areas of image analysis, data compression and general purpose computation, for instance as an extension to the Glift library [Lef06]. A HistoPyramid implementation has recently been included in OpenVIDIA, a framework for GPU-based computer vision by Fung et al. [FM05].

7. GPU Data Expansion

In applications operating on lists, list elements are repeatedly replicated, modified and deleted. One such application is geometry generation, such as the one used for dynamic particle cloud generators or dynamic mesh tesselation based on level of detail. In data-parallel research, lists are also known as *streams*. If they are implemented as one-dimensional arrays then they map to previously introduced GPU data structures, textures.

In this chapter we show how arbitrary list modification, including 1-to-m replication of list elements, can be achieved on graphics hardware by extending the definition of HistoPyramids and slightly modifying its traversal.

We start by explaining the basic concepts of the algorithm in Section 7.1, using a hypothetical 1D implementation. This is followed by two applications of GPU data expansion: Section 7.2 presents a light wavefront simulation based on ordinary differential equations (ODEs), which we utilized in our SIGGRAPH 2007 paper on Eikonal Rendering [IZT*07]. The simulation makes use of a 1-to-4 data expansion to adapt the light wavefront to local ODE density. In section 7.3, we generalize the implementation to 1-to-m data expansion, leading to an extremely fast implementation of Marching Cubes, a common algorithm for mesh iso-surface extraction from 3D volumes, which we have published in Computer Graphics Forum [DZTS08].

7.1. Basic Concepts

As introduction to the algorithm, we assume a one-dimensional input, as in Figure 7.1. The algorithm's *input stream* is shown to the left. The *predicate function,* in the middle and determines how many output elements shall be generated for each input element. The predicate function thus determines the *output multiplicity* for each input element. In this example, the predicate function acts upon the color of the input elements to determine their output multiplicity.

If an input element allocates zero elements in the output stream, then the element is discarded. This is equivalent to the *data compaction* of Chapter 6.

But if the predicate function allocates more than one output element for a given input element, then the stream is *expanded*. This process is called *data expansion,* or input replication – in the example, element “M” in the Figure represents an element that is expanded, i.e. replicated in the output stream.

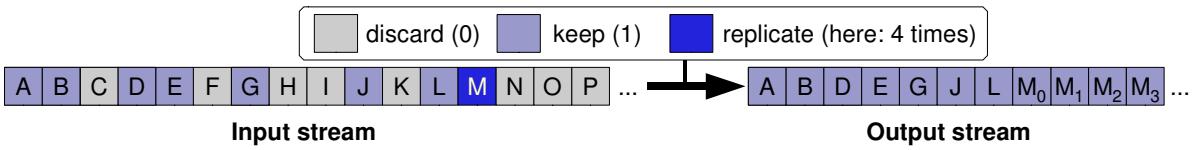


Figure 7.1: Stream data compaction and expansion. The rounded box contains the applied predicate function.

The generated output stream is shown to the right. The algorithm has retained all blue input elements, and created four copies of the input element “M”, with the indices for “M” showing which copy it is.

Note that the predicate function is effectively a generalization of the Classifier from Chapter 6: The Classifier allocated exactly *one* output element for a given input element, or *none*. The Classifier thus already acted as a simple predicate function.

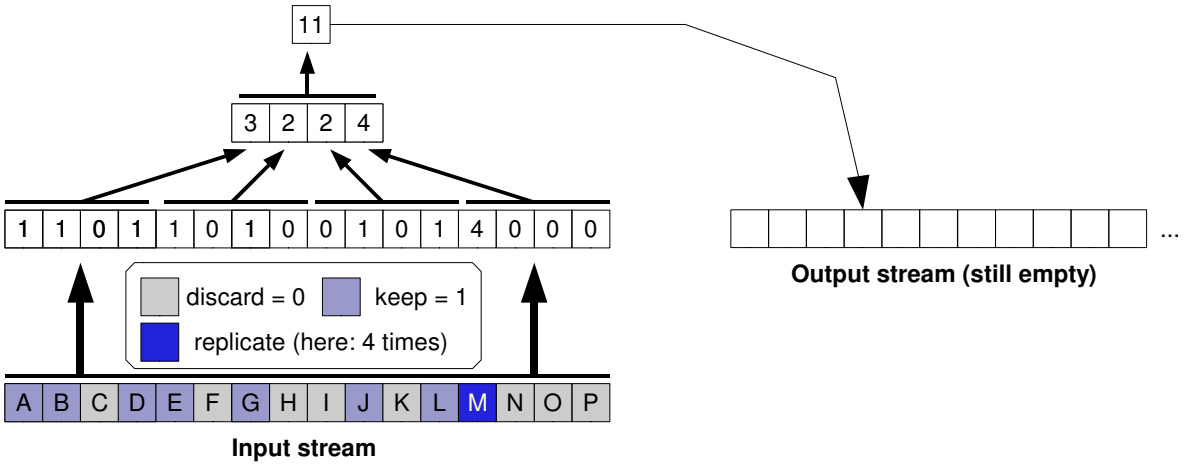


Figure 7.2: Buildup of the HistoPyramid is unchanged, but note the behaviour for element M.

Just as in Chapter 6, the extended HistoPyramid algorithm consists of two distinct phases, compare Figure 7.2 for an example. The first phase, Histopyramid buildup, is nearly similar to the process used in data compaction. The only difference is that the Classifier and its binary output has been replaced by a more universal *predicate function* with integer output. The predicate function determines how many copies of a certain input element shall be present in the output. After buildup through reduction with simple addition, the number of output elements is provided by the top element of the HistoPyramid: its value determines output allocation size.

In the second phase, output elements are extracted by traversing the HistoPyramid top-down. Traversal remains in principle unchanged, but only if simple replication of input data is sufficient. Often, traversal will track *local key indices* to allow for *numbering* copies of input elements. This proves useful where input shall be modified instead of replicated, e.g. for the tessellation of geometry.

This locates the input elements which correspond to a certain output, and works as described in Section 6.6, with the addition that we determine which *copy number* of the input element to generate. The new output stream is filled by starting one thread for each output element, which traverse the HistoPyramid in parallel. The output elements are enumerated with a key index **k**.

Traversal requires several variables, many of which are already known to the reader from Section 6.1: **m** denotes the number of HistoPyramid levels. Traversal maintains a coordinate **p** and a variable **level** to address cells in the HistoPyramid. We further maintain a local key index $\mathbf{k_L}$, which follows the local index range. Traversal starts at the top level, with **level** = **m**, **p** pointing at the single cell at the top level, and $\mathbf{k_L = k}$.

We decrement **level**, descending one level down in the HistoPyramid. After upscaling, **p** now points out four cells**.** We label these four cells as $[\mathbf{a}]$ $[\mathbf{b}]$ $[\mathbf{c}]$ $[\mathbf{d}]$ and use their four index ranges, defined as $[A]$ $[B]$ $[C]$ $[D]$=$[\mathbf{[0,a)}]$ $[\mathbf{[a, a+b)}]$ $[\mathbf{[a+b,a+b+c)}]$ $[\mathbf{[a+b+c,a+b+c+d)}]$.

Then, we decide which of the four ranges $\mathbf{k_L}$ falls into. Then, we make **p** point to the chosen cell, and subtract the cell's range start from $\mathbf{k_L}$, which adapts $\mathbf{k_L}$ to the local index range for the next iteration. For example, if $\mathbf{k_L}$ falls into range **B**, we let **p** point to cell b, and subtract the index start of range B, here: a, from $\mathbf{k_L}$**.**

The descent is now repeated, by subtracting one from **level** and repeating the evaluation, until **level** = 0 and the base level is reached. There, the index ranges are evaluated one last time before traversal terminates. From this last evaluation, **p** points at a cell in the base level, and $\mathbf{k_L}$ has been updated from the chosen cell's range start.

After the input element has been found, the output element can be written. Usually, input element data is copied, and/or input location **p** is written. In the case of pure data compaction, $\mathbf{k_L}$ is always zero after the last traversal evaluation. However, in the case of *data expansion*, the value in $\mathbf{k_L}$ holds the *copy number*., i.e. which numbered copy of the input element is about to be written.

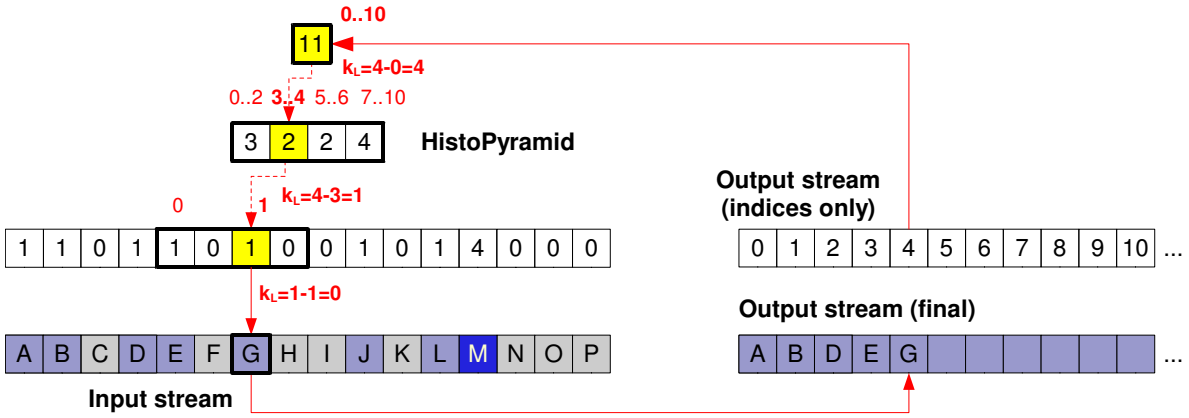


Figure 7.3: Traversal for index 4, copying element G to the output.

The first traversal, shown in Figure 7.3, is for the fifth element in the output stream, with key index $\mathbf{k}=\mathbf{k_L}=4$. We start at level 2, find that $\mathbf{k_L}$ matches the overall range and descend to level 1. The four cells at level 1 form the ranges **A**=[0,2), **B**=[3,4), **C**=[5,6), **D**=[7,10). We see that $\mathbf{k_L}$ is in the range of **B**. Thus, we adjust **p** to cell b, adjust $\mathbf{k_L}$ to the new index range by subtracting B's range start 3, which leaves $\mathbf{k_L}$=4-3=1. Then, we descend to the base level to access its children: The four cells at the base level form the ranges **A** = [0,1), **B** = [1,1), **C** = [1, 2), **D** = [2, 2). Ranges B and D are invalid, and thus ignored. Here, $\mathbf{k_L}$ = 1 falls into **C**, and we adjust **p** and $\mathbf{k_L}$ accordingly. Since we are at the base level, traversal terminates, with **p** = [6] and $\mathbf{k_L}$=1-1=0. Using **p**, we can now fetch the actual data of element G from the input stream, and copy it to the output location.

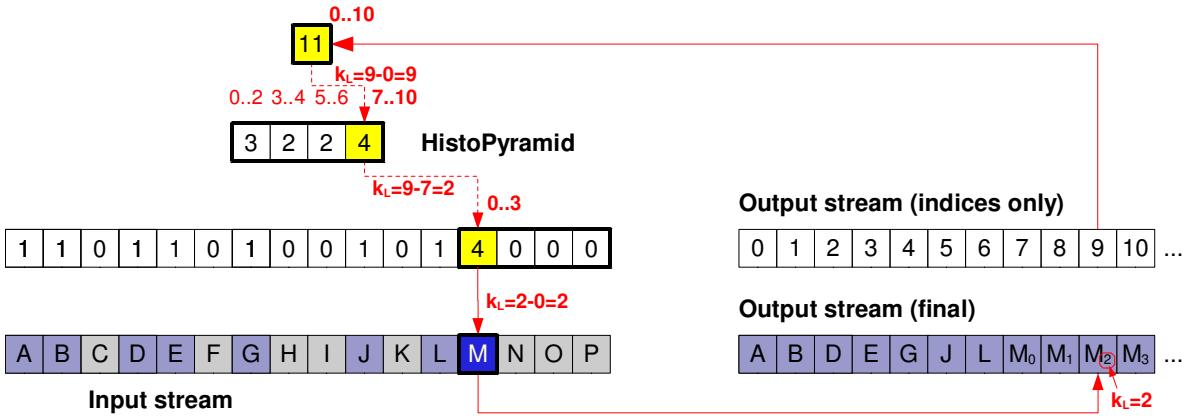


Figure 7.4: Traversal for index 9, a data expansion case, here: generating the second copy of M.

The second traversal, shown in Figure 7.4, is for the tenth element in the output stream, with key index $\mathbf{k}=\mathbf{k_L}=9$. We start at level 2, find that $\mathbf{k_L}$ is within the top cell's range [0, 11) and descend to level 1. The four texels at level 1 form the ranges **A**=[0,2), **B**=[3,4), **C**=[5,6), **D**=[7,10). We see that $\mathbf{k_L}$ is in the range **D**. Thus, we adjust **p** to point to cell d: **p**=3, and adjust $\mathbf{k_L}$ to the new index range by subtracting D's range start 7, which leaves $\mathbf{k_L}$=9-7=2. Then, we descend to the base level: The four cells at the base level form the ranges **A** = [0,3), **B** = [3,3), **C** = [3,3), **D** = [3,3). Ranges B, C and D are invalid. But range **A** actually covers *four* output entries, including the one we look after, , although we are at the base level. $\mathbf{k_L}$=2 thus falls into **A**, and we adjust **p** and $\mathbf{k_L}$ accordingly. Since

we are already at the base level, traversal terminates, with **p** = [12] and $\mathbf{k}_L$=2-0=2. We can now fetch the actual data of element M from the input stream using **p**=[12], and either copy it unchanged to the output stream location of this traversal, *or* use the numbered copy information $\mathbf{k}_L$=2 to *modify the input data* before writing it to the output.

The individual traversal threads will thus fill one output element each. Note that traversal only *reads* from the HistoPyramid, just like in the previous chapter on data compaction. There are no data dependencies between individual traversals; in particular, even the numbered copies of a single input element can be extracted independently and in parallel.

Geometry shaders can be emulated with above HistoPyramid traversal. Interestingly, the parallelism for generating individual copies of an input element is not present in geometry shader programs. There, a single input element is handled by a single thread, and a stream append method must be used to generate multiple output elements. Geometry shader programs are thus less parallelizable for high levels of output multiplicity. The results of HistoPyramid Marching Cubes in Section 7.3 show that above approach outperforms current geometry shader hardware functionality .

A comparison with the Scan algorithm [Har07] is also at place: Scan and HistoPyramids have some similarities, with the Scan up-sweep phase and the HistoPyramid construction being identical. But the difference lies in the output generation. Scan produces first the correct output indices for all input elements with a down-sweep phase, which can be implemented very efficiently in CUDA; afterwards, it picks all input elements which have corresponding output positions, and scatters them to their right position. For HistoPyramids, down-sweep phase and scattering do not exist. Instead, a log(n)-traversal of the HistoPyramid is started for each output element. Despite that algorithmic complexity, the HistoPyramid algorithm can utilize the texture cache very efficiently, reducing the performance hit of traversal. A second difference is that HistoPyramid traversal iterates over the *output* elements, instead of the input elements, which makes a difference if only a fraction of the input elements are relevant for output. This is the case in the Marching Cubes application, see Section 7.3.

We will now demonstrate how data expansion proves useful in real-time visual computing applications. Note that the actual *OpenGL implementation* of data expansion on graphics hardware is two-dimensional, for previously described performance reasons. While the two example applications below sport their own explanation of a two-dimensional implementation, the algorithmic concepts match the ones used in this introduction - only diagrams and terminology might differ slightly.

7.2. Light Wavefront Simulation

In our paper on Eikonal rendering [IZT*07], we introduced a novel algorithm to render a variety of sophisticated lighting effects in and around refractive objects, in real-time, on a single PC. This method enables us to realistically display objects with spatially-varying refractive indices, inhomogeneous attenuation characteristics, as well as spatially-varying reflectance and anisotropic scattering properties. With Eikonal rendering, we can reproduce arbitrarily curved light paths, volume and surface caustics, anisotropic scattering, as well as total reflection, by means of the same efficient theoretical framework.

7.2.1. System Overview

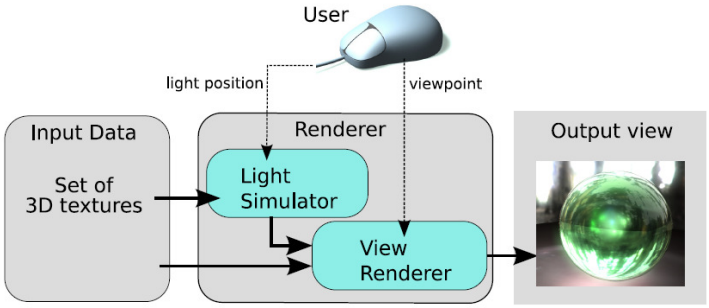


Figure 7.5: Workflow for our rendering system.

Our system for refractive object rendering is comprised of two components: The *light simulator* and the *view renderer*, see Figure 7.5 for an overview. The *light simulator* utilizes a framework based on ordinary differential equations (ODEs) to adaptively trace light wavefronts through the scene, and requires approximately 10 seconds for typical object-light constellations, see the first three images of Figure 7.6. The view renderer, shown to the right of Figure 7.6, is a fast GPU volume raycaster, with the additional ability of altering viewing ray trajectories. To do this, it uses ordinary differential equations, derived from the eikonal equation [BW99]. With this novel combination of techniques, it can create imagery in real-time (5-60 fps, depending on scene complexity), provided that the relation of object and light does not change.

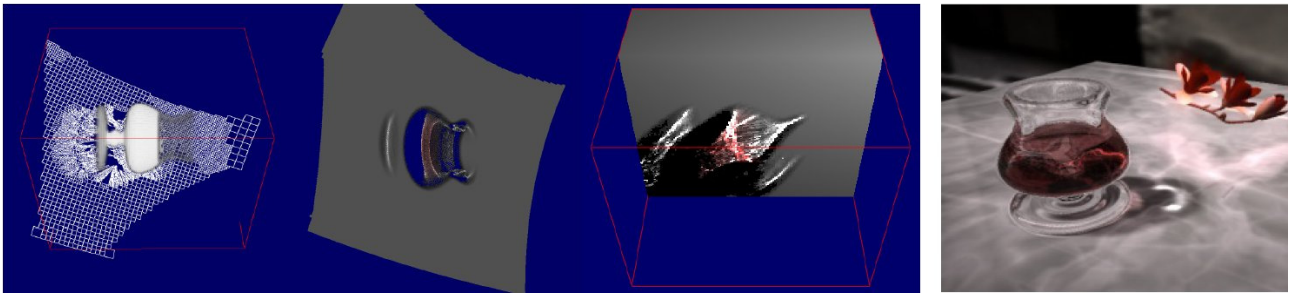


Figure 7.6: Left to right: Adaptive light wavefront simulation calculates the modified light distribution and protocols it into an irradiance volume. The real-time viewer finally embeds the refractive object and its lighting into a 3D-scene.

While the conference paper [IZT*07] focused on development of the numerical framework and the validation of our results, we will focus here on the novel concepts and data structures that enabled ODE light simulation via adaptive wavefront tracing on graphics hardware. The view renderer is only described in as far as data structures are concerned; for more details on its implementation we refer to Tevs [Tevs07].

7.2.2. Background

We model a light source with a three-dimensional vector field of local light directions l(x) and a scalar field of differential irradiance values $\Delta E_{\omega}(x)$ (compare Section 3.2 in [IZT*07]). These fields can be computed in several ways: A popular choice among computer graphics researchers is photon mapping [Jen01], with several GPU implementations available [GWS04] [PDC*03]. In the literature of computational physics and numerical analysis, a huge range of methods have been proposed to solve this problem. Choices range from purely Eulerian formulations using the eikonal

and transport equations [BK00], phase space methods [OCK*02] and hybrid Lagrangian-Eulerian approaches [Ben96] to adaptive wavefront tracing [ER03]. All methods, except for the purely Eulerian approach, deal with the inherent multivaluedness of the solution to the underlying equations.

We use adaptive wavefront tracing [ER03, Col97] for the computation of the local light directions and differential irradiance values because it offers the best trade-off between computation time and accuracy of the solution. A wavefront is an iso-surface of constant travel time of light originating from a light source, see Figure 7.7. In accordance with Fermat's Principle, light rays travel always normal to these wavefronts. Due to inhomogeneous material distribution of our simulated object materials, light rays and viewing rays are bent on their way through the scene volume.

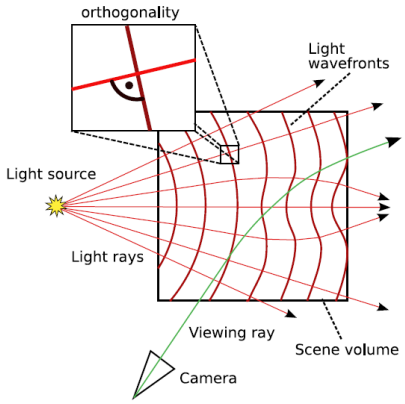


Figure 7.7: 2D illustration of our complex image formation scenario.
Light rays always travel orthogonally to the light wavefronts, i.e. the isosurfaces of constant travel time.

7.2.3. Scene Representation

To efficiently map our concepts onto graphics hardware, we use a volumetric scene representation to represent the relevant scene data, such as refraction indices and their local gradients. We also introduce attenuation values, which describe local light irradiance loss due to material properties.

In graphics memory, the input scenes are stored as a set of 3D volume textures. In a first set of volumes, the spatially varying refractive indices, as well as its gradient field are stored. In our test scenes, objects were created as solids of revolution, implicit surfaces, or triangle meshes that we rasterized into the 3D volume. Refractive index distributions can be defined interactively, or be derived directly from the implicit functions of generated objects. Prior to gradient computation, we smooth the volumes, using a uniform Gaussian filter kernel with a standard deviation of typically 0.5-1.0 voxels.For some of our test objects, we supplied spatially varying attenuation in the interior by applying a noise function or by manually defining a n attenuation for the 2D surface of revolution.

Other 3D textures contain the volumetric material boundary indicator, as well as BRDF parameters or emission descriptions. As they are not used by the light simulator, we refer to Ihrke et al. [IZT*07] for a detailed description, and Tevs [Tevs07] for their use in the implementation of the view renderer.

7.2.4. Light Simulator

7.2.4.1. Overview

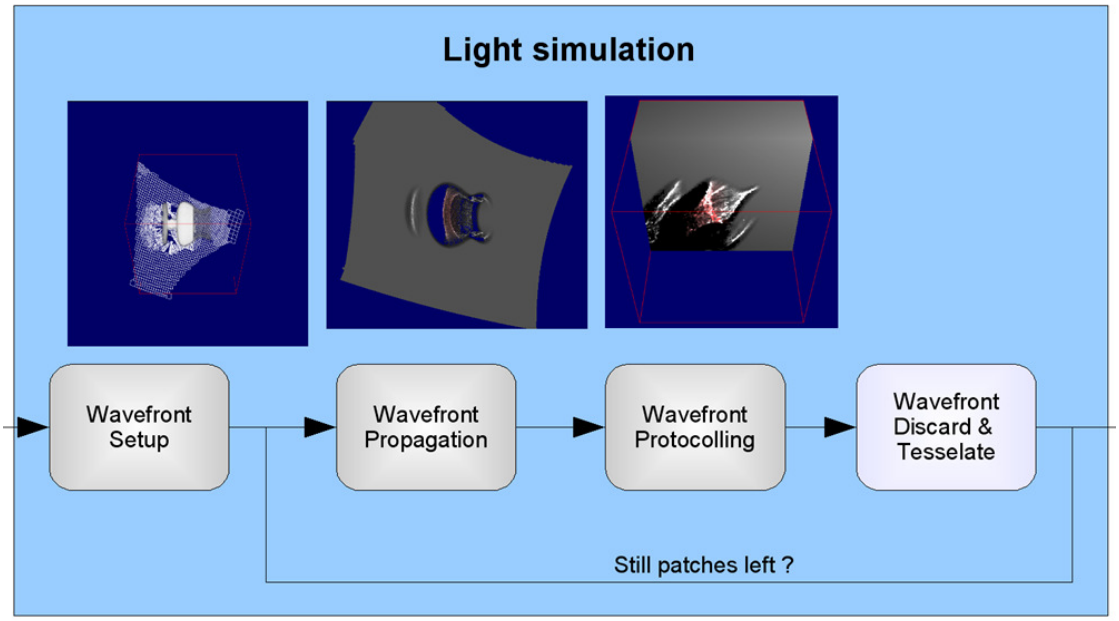


Figure 7.8: ODE Light simulation stages.

Our off-line light wavefront simulation implements fast ODE-based particle tracing through a volumetric object through the ray equation of geometric optics. Our particle tracing through the object shows similarities to photon mapping, however, the main difference is that we bind four particles into one entity, a *wavefront patch*, see Figure 7.11 (right), and track the light irradiance that these patches span. We also spawn new particles according to a minimum sampling criterion. As the wavefront is tracked over time, we can compute the differential irradiance at every point in space from the area of the wavefront patches that connect the particles. See Figure 7.8 for a stage overview.

At first, the wavefront is set up as a list of wavefront patches emitted from a light source, just about to enter the object volume. Section 7.2.4.2 describes this, and explains the texture representation of the wavefront. Then, the wavefront is propagated through the object. Section 7.2.4.3 explains both the involved rules of patch propagation and the criteria for splitting a patch into smaller patches. The wavefront can thus adaptively tesselate and still keep a low memory footprint during its traversal of the refraction volume. The final step in light simulation is wavefront protocolling, which updates the output volume of local irradiance and incoming ray direction. This is described in Section 7.2.4.5. Simulation ends when all patches have been culled from the simulation, either by losing too much irradiance or by leaving the volume.

This process is repeated until all wavefront patches have either left the volume of interest or been culled from the simulation due to above reasons.

7.2.4.2. Wavefront Setup

During initialization, we initialize the patch list to either represent a planar wavefront (directional light source) or a spherical wavefront (point light source). The initialization also readily ensures that the wavefront is large enough to cover the simulation volume. Other light source types (as multi-

directional light) can be implemented, as the wavefront patches are independent and thus can be spatially stacked on top of each other, while being randomly located in the patch list.

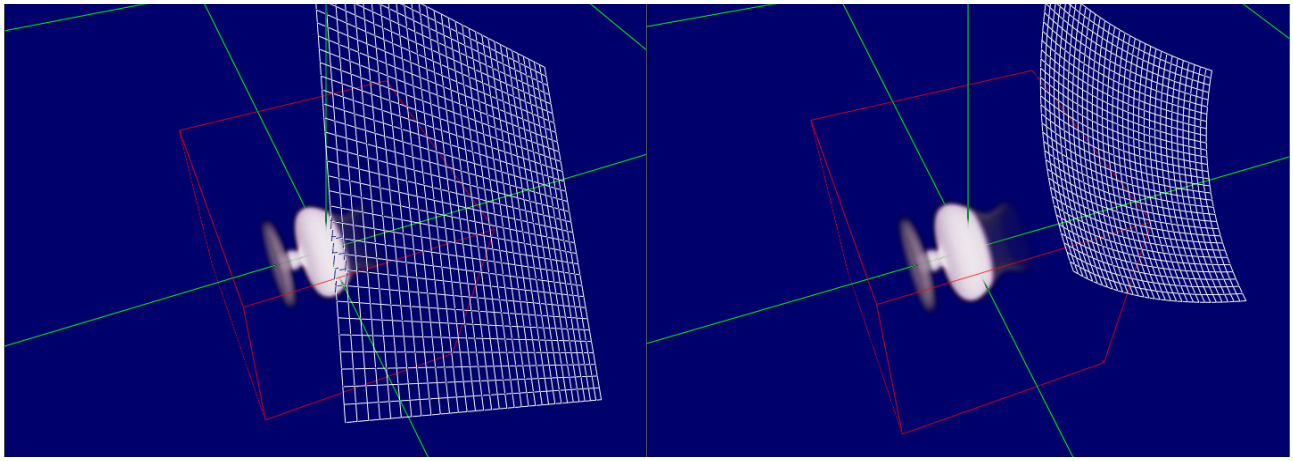


Figure 7.9: Wavefront setup (coarse 32x32 mesh only for demonstration).
Left: Directional light source. Right: Point light source.

$\Delta E_\omega(t)$, the discretized differential irradiance emitted by the lightsource, is discretized as well, and distributed amongst the wavefront patches. In the current implementation, we provide a spatially even distribution that is solely based on wavefront patch areas. Uneven distributions, representing partially masked or matted light sources, are of course possible, and require only minor changes in the initialization procedure.

The wavefront patch is stored in a number of textures, that we use as 2D data storage arrays. Since each particle requires a 3D position and motion vector, we use the RGB components of eight RGBA floating point textures. The remaining A components are used for the retained light irradiance of the patch, represented as R, G and B irradiance.

7.2.4.3. Wavefront Propagation

After initialization close to the light source, the wavefront must be propaged through the scene, as already sketched in Figure 7.7. We explain the numerical computation of this wavefront in two steps: First, we describe the path of an individual light ray through the volume with the ray equation of geometric optics. For computation purposes, we associate the ray with a light 'particle' that tracks the ray's progression. This describes the progression of the wavefront, it still misses how the wavefront's energy is distributed. Hence, we extend the wavefront description into a set of infinitesimal stream-tubes that follow the lightrays' paths. By discretizing the span area of the stream tubes and applying the intensity law of geometric optics, we can show that these stream-tubes can be discretized as a collection of wavefront patches, binding together four light particles at a time to track their span area. This allows us to simulate the light wavefront very similar to a particle system.

The motion of a light 'particle' in a field of inhomogeneous refractive indices, denoted by *n*, can be described by the ray equation of geometric optics, which has been previously used in computer graphics e.g. by Stam and Langu´enou [Sta95] and Gutierrez et al. [GMAS05]:

$$(8)\quad \frac{d}{ds}\left(n\frac{d\,\boldsymbol{x}}{ds}\right)=\nabla n$$

It is derived from the eikonal equation and the motion of a massless particle along the gradient of the eikonal solution. In equation (8), **x** describes the light particle's current position, while ds

denotes an infinitesimal step in the direction tangential to the curved ray. The scene's refraction volume provides refraction index n, and its local 3D gradient ∇n .

We reparameterize it to yield equitemporal discretization steps:

(13) $$n\frac{d}{dt}(n^2\frac{d\,\boldsymbol{x}}{dt})=\nabla n$$

A proof of this property is given in the appendix of [IZT*07]. The reparameterization is necessary to ensure that all particles stay on a common wavefront over time, a property which is necessary to apply the simple intensity law of geometric optics for energy computation, instead of more complex wavefront curvature tracking schemes used in other publications [MH92; Col94].

For numerical computation, we express Eq. (13) as a system of first order ordinary differential equations

(14) $$\frac{d\,\boldsymbol{x}}{dt}=\frac{\boldsymbol{v}}{n^2}$$

(15) $$\frac{d\,\boldsymbol{v}}{dt}=\frac{\nabla n}{n}$$

The wavefront's propagation is thus described here as a set of inter-connected particles which are propagated independently. For every light particle, we maintain a speed/direction vector v and a position x in three dimensions. The scene's refraction volume provides refraction index n, and its local 3D gradient ∇n ..

But while we now can describe the wavefront's propagation, the wavefront's internal *energy distribution* over time is still unknown. We therefore must extend the concept of light rays, and describe the wavefront as a set of stream-tubes, see Figure 7.10 (left).

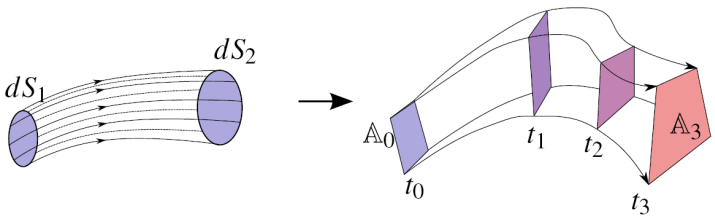


Figure 7.10: The intensity law of geometric optics (left) and its discretized version (right) in the form of a stream tube. The product of area and differential irradiance is constant along a tube of rays.

The intensity law of geometric optics [BW99] now states that in an infinitesimal tube of rays, the energy E_ω stays constant:

(16) $$dE_{\omega 1}\,dS_1=dE_{\omega 2}\,dS_2$$

For the simulation, we now replace the infinitesimal tube with a discretized version, and replace its infinitesimally small area dS with the time-dependent span area A(t) . To track this span area A(t), we bundle light particles into packets of four. This constitutes the *wavefront patch*, our basic unit of simulation.

For the *irradiance computation,* i.e. to update the energy contribution of wavefront patches during propagation, we use a discretized version of the intensity law. In this version, *dS* becomes the *span area* A(t) which denotes the area of a wavefront patch at time t and $\Delta E_\omega(t)$ the discretized differential irradiance associated with it.

Equation (16) then reads

$$\Delta E_\omega(t) = \frac{\Delta E_\omega(0) A(0)}{A(t)} \quad . \tag{17}$$

As a last step, we need to model material absorption of irradiance. Therefore, the final discretized differential irradiance for a wavefront patch is given by

$$\Delta E_\omega(t) = \frac{\Delta E_\omega(0) A(0)}{A(t)} \mathrm{e}^{-\frac{\int_0^t \sigma_t(c(\hat{t}))}{n} d\hat{t}} \quad . \tag{18}$$

On the GPU, we update the patches' corner positions **x** and directions **v** according to equations (14) and (15) for every time step. We further update the patches' held RGB irradiances E according to equation (18).

To summarize, we simulate overall wavefront propagation by first subdividing the wavefront into patches, whose corners are determined by light particles. Then we simulate how these light particles propagate through the scene, just like in any GPU-based particle simulation. The difference to a ”classic” particle simulation is that the area spanned by sets of four light particles, i.e. the area of the wavefront patches, is used for differential irradiance computation.

7.2.4.4. Wavefront Refinement

After each simulation timestep, the wavefront's patch list has to be reorganized for various reasons. For this purpose, each patch is evaluated by its state variables, and assigned a state that determines its further treatment. The actual reorganization is accomplished through GPU data compaction and expansion, based on HistoPyramids.

Tesselation due to divergence (Tesselate state): After each timestep, the wavefront patches are written into a 3D output volume to obtain a continuous volumetric representation of the light distribution,see section 7.2.4.5. But the light particle corners of a patch may have diverged in wavefront patch propagation, and patch's area thus become larger. If a patch area slides through a voxel without touching it with one of its corners, its corners have effectively ignored the influence of this voxel's refraction value. The wavefront would thus *undersample the volume of refraction indices.* To alleviate this, we adaptively split the wavefront patches once their corners grow further apart than one voxel, see Figure 7.11. In other words, we tessellate the wavefront to stay below a minimum sampling bound. In practice, our tessellation currently divides a patch into four smaller ones if its corners span more than one voxel in any direction, Figure 7.11.

This tesselation has another advantage: Since graphics hardware is currently unable to rasterize arbitrarily sized quads into 3D volumes, we need to use OpenGL point primitives to rasterize into the output volume. With the knowledge that a patch's corners must never be further apart than a voxel from each other, we can represent wavefront patches with their midpoints during voxelization. Sent as OpenGL points by the vertex shader to rasterization, the GPU then stores the differential irradiance and directional information of a wavefront patch as a single voxel sample. GPU-based voxelization is thus solved in conjunction with implementing adaptive wavefront tesselation.

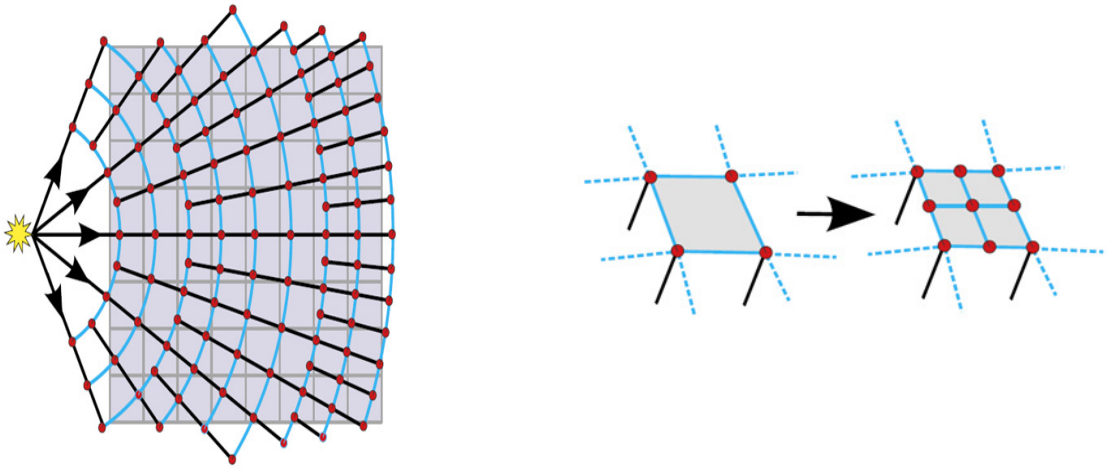


Figure 7.11: Adaptive wavefront refinement principles. Left: A 2D wavefront is represented by particles (red dots) that are connected to form a wavefront (blue lines). While advancing through the voxel volume (shown in gray) the wavefront is tessellated such that its patches span less than a voxel. Right: A 3D illustration of the tessellation for one wavefront patch.

Patch termination (Discard state): Whenever a patch holds too little irradiance we eliminate the patch from the wavefront if it doesn't pass an irradiance threshold, implicitly assuming it will not contract again and thus nevermore yield a noteworthy irradiance contribution.

Termination typically happens after too much tessellation or loss of irradiance due to repeated attenuation. We also eliminate patches which leave the simulation volume, since we assume that they will never re-enter simulation. Another cause for termination are wavefront singularities [BW99], where the physical model of ray optics breaks down, resulting in infinite irradiance at catastrophic points. We detect these areas by tracking patch orientation with respect to its propagation direction. In case orientation changes, a singular point must have been crossed by the wavefront patch, and we discard it from further simulation.

Retained as is (Keep state): This is the default case if none of the above applies.

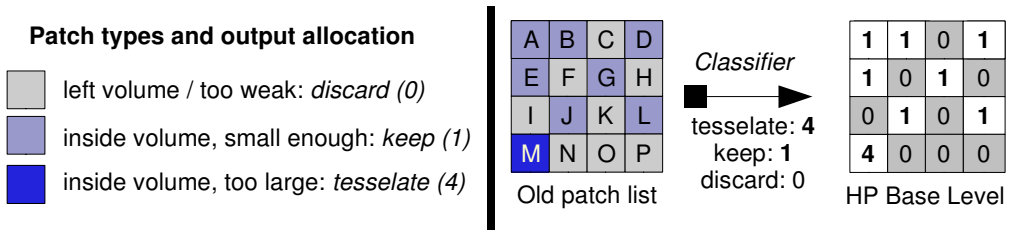


Figure 7.12: Wavefront patch classification.

In summary, this means that each patch can have three states before the next simulation step, with the numbers in brackets telling how many new patches are yielded from each input patch: Discard (0), Keep (1) or Tessellate (4). Patch classification takes place after the patches' corner directions have been updated, see also Figure 7.12. After classification, the patch list must be reorganized, which faces us with the non-trivial problem of data compaction and expansion on graphics hardware. We therefore use the algorithms presented in the beginning of this chapter, albeit switch to a 2D representation: The algorithm uses a mipmap-like data structure, the HistoPyramid, to construct a list of retained data entries (here: patches) without involving the CPU.

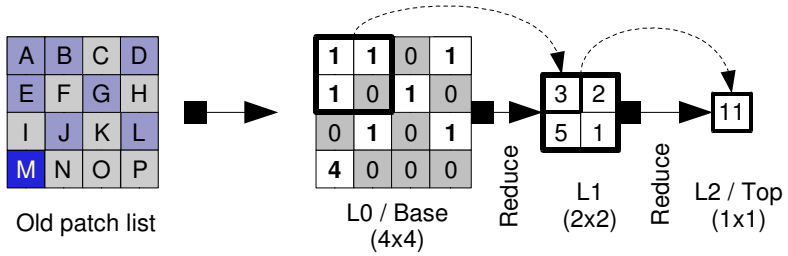


Figure 7.13: HistoPyramid buildup.

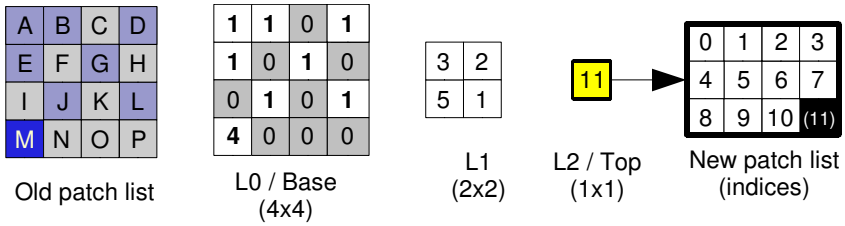


Figure 7.14: Allocation of new patch list from HP top cell.

Of course, replicated copies of input stream elements will not suffice for patch tesselation. Instead, input elements must be modified while they are written to the output stream, i.e. one large patch from the input must result in four smaller ones in the output, arranged as in Figure 7.11. Luckily, this is no problem given the local key index $\mathbf{k_L}$, as Figure 7.15 shows. This local key index allows us to write the correct output after the corresponding input element has been found. Therefore, if a patch has to be tesselated, each copy is assigned one quarter of the original irradiance and the area of the input patches, with corners calculated from bilinear interpolation of the original patch's corners.

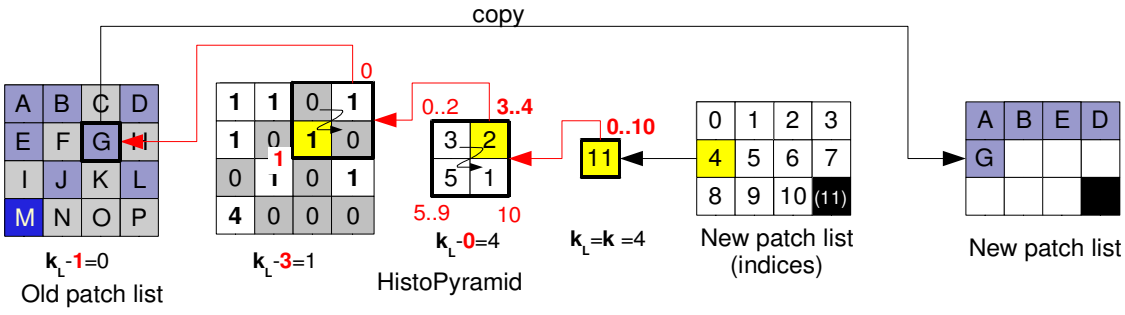


Figure 7.15: HP traversal for unchanged patches.

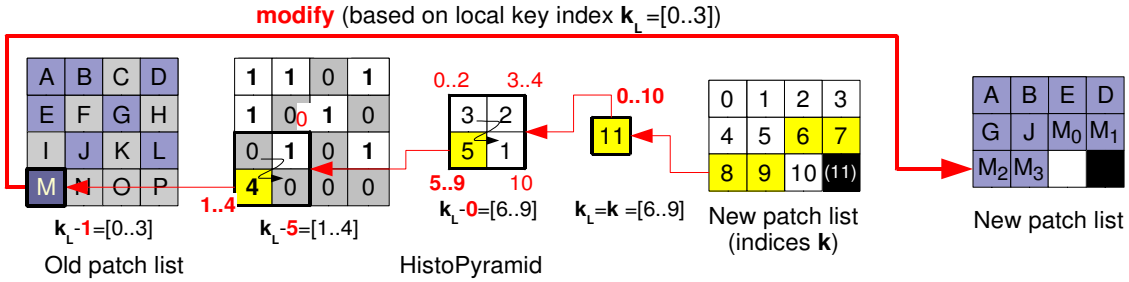


Figure 7.16: HP traversal for tesselated patches.

After the new patch list has been generated, it is updated to advance the light wavefront simulation further, using the previously described propagation framework. This repeats until no patches remain in the simulation volume. In Figure 7.17, we show a wavefront in various states of tesselation while it propagates through a wine glass. Note how the center of the wavefront is more tesselated than the outer parts, and how some wavefront patches have already been culled to the left. In the right image the computed irradiance values are shown as colors, and thus preview the yielded caustic patterns in and around the object.

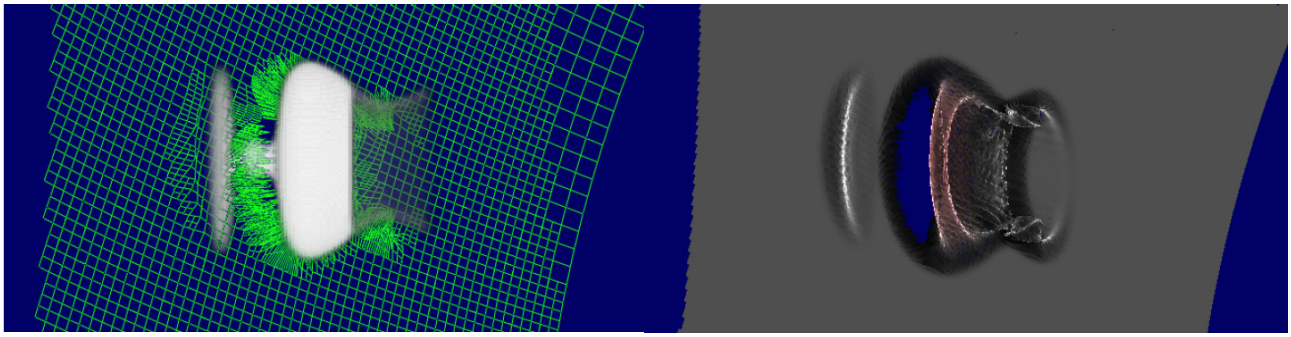


Figure 7.17: Left: The refractive index volume of the glass is approached by a spherical wavefront from the right. The adaptive tessellation of the wavefront is also visible. Right: When it passes through the object, caustic patterns appear in its irradiance distribution.

7.2.4.5. Wavefront Voxelization

After each update step, we need to protocol the wavefront patches into the 3D output volumes for irradiance and incoming ray direction. On graphics hardware, this is accomplished through point primitives that are rasterized into volumes. Since we can warranty each patch to be smaller than a voxel, we can treat them as OpenGL points and scatter them into the output 3D volumes. Since hardware cannot write directly into a 3D volume, we use the concept of Flat3D textures, introduced by Harris et al. [HSBL03]. During the write, all traversal passes can be accumulated, or selective writing can be used, e.g. to only store the highest-irradiance pass or a certain number of passes.

Further, we limit ourselves to storing only one incoming light direction, corresponding to the highest irradiance ray passing a particular voxel. This is justified by a statistical analysis: For the wine glass model, see Figure 7.6 and Figure 7.19 (right), only 5.6% of the voxels receive light from more than one direction. For these 5.6% of voxels, the highest irradiance ray contributes a mean of 81.6% of the total irradiance arriving at these voxels. Similar numbers hold for the other models. Before we commit a patch to the 3D volume, we check if it is allowed to overwrite the one already stored there (if any), based on the highest irradiance criterion.

7.2.5. View Renderer

Light simulation is typically completed after a few seconds, depending on volume resolution and lighting complexity. Afterwards, the results can be viewed as part of a 3D scene. The employed rendering techniques resemble volume raycasting from the user viewpoint, but keep using the mentioned set of ODE's to deflect the incoming viewing rays. On its path through the volume, each viewing ray accumulates light contributions from the precalculated irradiance volume, according to the complex image formation model described in Ihrke et al. [IZT*07], until it encounters an opaque surface or finally leaves the volume, finishing with an environment map lookup. Viewing ray propagation is described in detail in section 4.1 of Ihrke et al. [IZT*07]. Please note that no explicit ray-surface intersections are required. All these traversals are implemented as OpenGL fragment shaders, which are executed by rendering cube faces at the location of the refractive

volume object in the 3D scene. However since the input volumes have been smoothed, object boundaries between air and object can extend over 2-3 voxels, and thus appear blurry. If blurry boundaries pose a problem in the view renderer, they can be alleviated by either increasing the volume resolution, or by rendering a suitable proxy geometry (instead of cube faces) to trigger the raycasting. While proxy geometry improves the sharpness of the boundaries and results in higher frame rates, participating media surrounding the object can no longer be rendered.

In theory, the view renderer could handle arbitrary BRDF models, including parametric or tabulated representations. For example, for approximating anisotropic scattering effects, we employ the scattering-phase function model by Henyey and Greenstein [HG41]. Its parameters are also stored in volumetric textures. However, since our glass objects come close to perfect reflectors, and a good approximation of the first reflection is already visually pleasing, we use simple dynamic environment mapping. The Fresnel effect and the anisotropic scattering phase function are computed on-the-fly in the fragment shader. Through spatially varying as well as colorchannel-dependent attenuation, beautifully colored objects can be reproduced. Optionally, emission can be added, and dispersion effects can be simulated if the input data contain a separate refractive index field for each RGB channel.

After the viewing ray has finished volume traversal, we use its exit direction to conduct a lookup into a dynamic environment map to approximate the background radiance. All lighting computations are performed in high dynamic range and an adaptive tone-mapping based on [KMS05] is applied prior to display. See Figure 7.20 for example imagery, all screenshots from the video footage of our SIGGRAPH submission video [Eikonal]. A more detailed explanation of the view renderer can be found in Tevs [Tevs07].

7.2.6. Results

The results gained from this project were two-fold: First, the light simulation had to work properly, with its numerical stability and accuracy of the light simulation had to be ensured. It was also necessary to stay within computational limits of the graphics hardware; for example, the light simulation was not allowed to exceed 1 million wavefront patches due to texture size limitations.The second part regarded the final images generated from the view renderer, which had to be visually plausible and appealing.

7.2.6.1. Light wavefront simulation

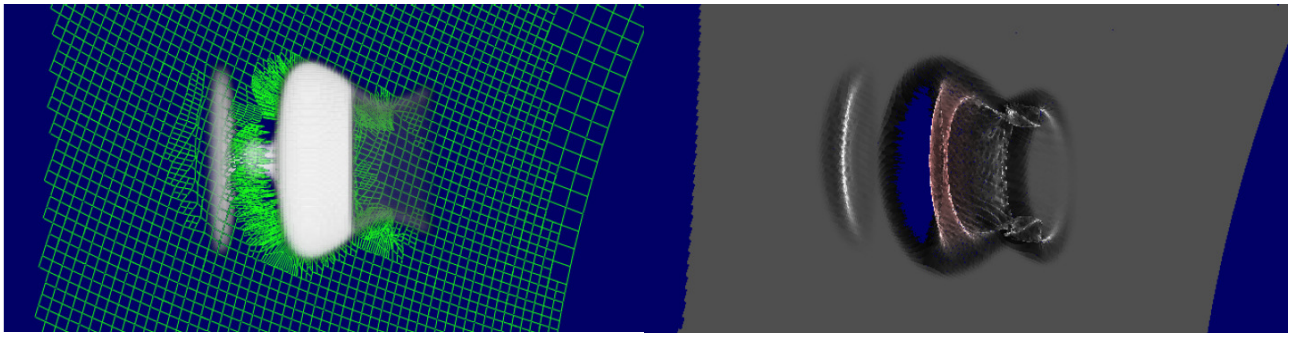


Figure 7.18: Left: The refractive index volume of the glass is approached by a spherical wavefront from the right. The adaptive tessellation of the wavefront is also visible. Right: When it passes through the object, caustic patterns appear in its irradiance distribution.

Figure 7.18 demonstrates a light simulation in progress, with a light wavefront propagating through a refraction index volume describing a wine glass. At the right of the figure, the computed

irradiance values show as colors, and thus preview the yielded caustic patterns in and around the object. Simulation took around 10 seconds on GeForce GTX 8800, and the number of wavefront patches peaked at 400000. For the final video, with a a moving lightsource required 400 similar simulations with varying lighting positions around the wineglass object, which all completed without numerical instability. A screenshot from this wine glass scene can be seen in Figure 7.20.

7.2.6.2. View renderer

After the lighting simulation has completed, the final views of the objects in the scene can be generated. Figure 7.19 and Figure 7.20 show several frames of the final video submitted to and accepted at SIGGRAPH 2007.

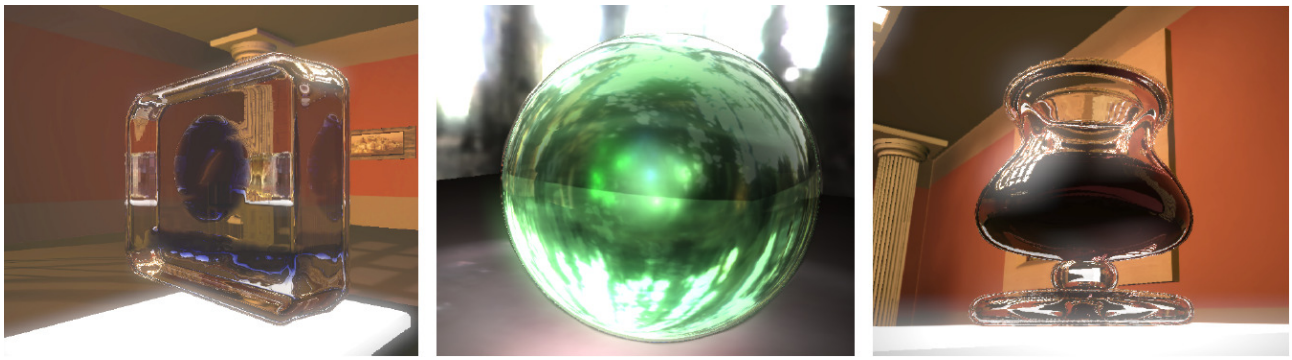


Figure 7.19: Left: Glass block with embedded SIGGRAPH logo of different refraction and attenuation, 15.5 fps, (5 objects in scene). Middle: Colored sphere rendered into an HDR environment map showing slight emission in addition to all other effects, 26.2 fps. Right: Complex refraction patterns in the glass, 13.7 fps, (5 objects in scene). Also note the surface reflections and the total reflections within, as well as the rounded cube being visible through the glass block.

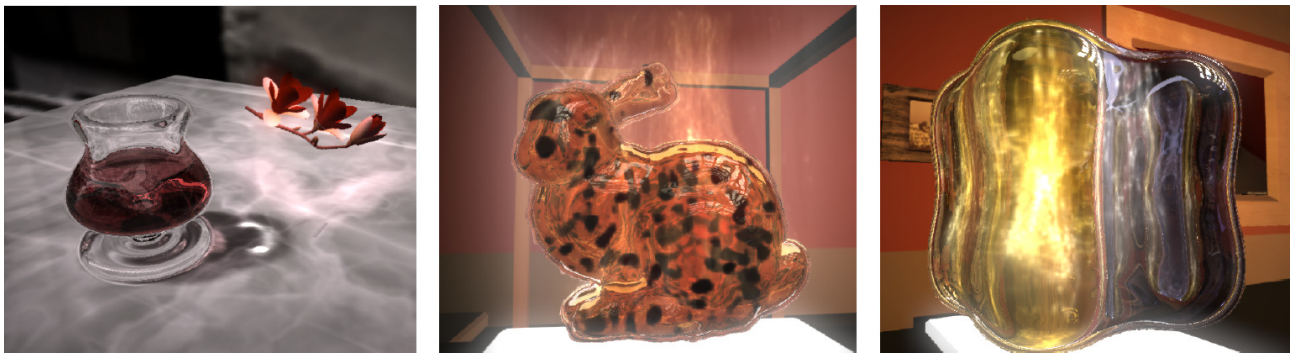


Figure 7.20: Real-time renderings of complex refractive objects. Left: Glass with red wine casting a colorful caustic, 24.8 fps. Middle: Amberlike bunny with black embeddings showing anisotropic scattering and volume caustics in the surrounding smoke and its interior, 13.0 fps. Right: Rounded cube composed of three differently colored and differently refracting kinds of glass showing scattering effects and caustics in its interior, 6.4 fps.

7.2.7. Future Work

We hope that our research prototype for Eikonal Rendering inspires novel computer graphics concepts for graphics hardware, and foresee interesting extensions into global illumination and raytracing. We are very interested in how octree data structures, like the one presented in Chapter 9, may be combined with Eikonal Rendering to yield more fine-grained volumes, possibly even spanning complete scene representations.

7.3. HistoPyramid Marching Cubes

After HistoPyramids had proven their usefulness in 1-to-4 tesselation of adaptive wavefront patches, we became aware that it could even be used for the more general 1-to-m replication of input data, and thus *replace* OpenGL geometry shader functionality [BL06].

We verified our idea by implementing a classic geometry-generation algorithm, Marching Cubes, entirely on graphics hardware. The results were published in Computer Graphics Forum [DZTS08]. The basis is a reformulation of Marching Cubes as a data compaction and expansion process: A stream, or: list, of a 3D volume's voxels is first *compacted* to only hold the iso-surface voxels relevant for the iso-surface, then *expanded* to a list of triangle vertices that describe this iso-surface.

The HistoPyramid algorithm for data expansion thus became the core of a highly efficient and interactive Marching Cube implementation for OpenGL 2.0 and comparable APIs. Surprisingly, it outperformed all other GPU-based iso-surface extraction algorithms in direct rendering for sparse or large volumes known at that time, even those that used the then-recently introduced geometry shader capabilities of modern graphics hardware. See Figure 7.21 for various screenshots from our GPU implementation..

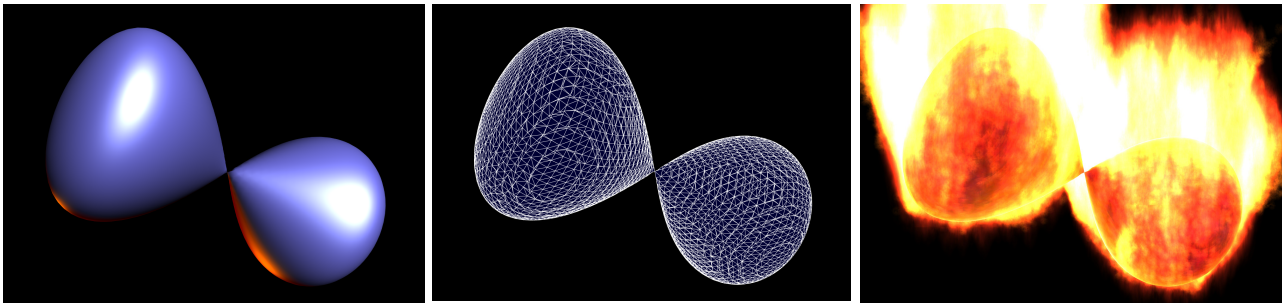


Figure 7.21: An iso-surface represented explicitly as a compact list of triangles (left) can be visualized from any viewpoint (middle) and even be directly post-processed. One example for such post-processing is the spawning of particles evenly over the surface (right). In all three images, the GPU has autonomously extracted the mesh from the scalar field, where it is kept in graphics memory.

7.3.1. Background

Iso-surfaces of scalar fields defined over cubical grids are essential in a wide range of applications, e.g. medical imaging, geophysical surveying, physics, and computational geometry. A major challenge is that the number of elements grows to the power of three with respect to sample density, and the massive amounts of data puts tough requirements on processing power and memory bandwidth. This is particularly true for applications that require interactive visualization of scalar fields. In medical visualization, for example, iso-surface extraction, as depicted in Figure 7.22, is used on a daily basis. There, the user benefits greatly from immediate feedback in the delicate process of determining transfer functions and iso-levels. In other areas such as geophysical surveys, iso-surfaces are an invaluable tool for interpreting the enormous amounts of measurement data.

Therefore, and not unexpectedly, there has been a lot of research on volume data processing on Graphics Processing Units (GPUs), since GPUs are particularly designed for huge computational tasks with challenging memory bandwidth requirements, building on simple and massive parallelism instead of the CPU’s more sophisticated serial processing.

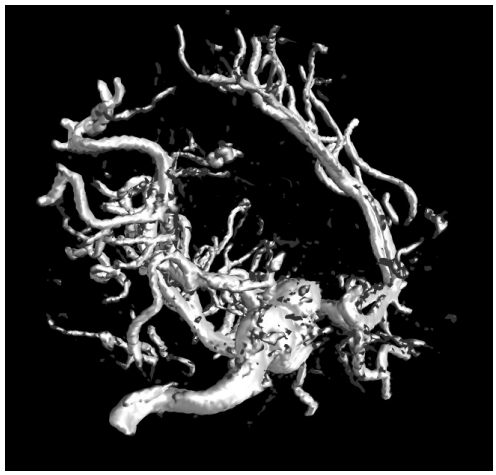


Figure 7.22: Determining transfer functions and iso-levels for medical data is a delicate process, where the user benefits greatly from immediate feedback.

Volume ray-casting is one visualization technique for scalar fields that has been successfully implemented on GPUs. While the intense computation for every change in viewport can nowadays be handled, ray-casting can never produce an explicit representation of the iso-surface. Such an explicit iso-surface is essential for successive processing of the geometry, like volume or surface area calculations, freeform modeling, surface fairing, or surface-related effects for movies and games, such as the one shown in Figure 7.21. In particular, two efficient algorithms for extracting explicit iso-surfaces, Marching Cubes (MC) [LC87] and Marching Tetrahedra (MT) [GH95], have been introduced. By now, substantial research effort has been spent on accelerating these algorithms on GPUs.

Here, we present a novel, though well-founded, formulation of the Marching Cubes algorithm, suitable for any graphics hardware with at least Shader Model 3 (SM3) capabilites. This allows the implementation to run on a wide range of graphics hardware. Our approach extracts isosurfaces directly from raw data without any pre-processing, and thus dynamic data sets, changes in the transfer function, or variations in the iso-level is handled directly. It is able to produce a compact sequence of iso-surface triangles in GPU memory without any transfer of geometry to or from the CPU. The method requires only a moderate implementation effort and can thus be easily integrated into existing applications, and currently outperforms all other known GPU-based iso-surface extraction approaches. For completeness,we also propose how this algorithm can be implemented in the general GPU programming language CUDA [NV07].

The main element of our approach is the Histogram Pyramid, which has shown to be an efficient data structure for GPU data compaction in Chapter 6.

As already mentioned in this Chapter´s introduction, we have extended the HistoPyramid to handle general GPU stream expansion. This simple, yet fundamental modification, together with a reformulation of the MC algorithm as a stream compaction and expansion process, enables us to map the MC algorithm onto the GPU.

We begin with an overview of previous and related work in Section 7.3.2, followed by a description of HistoPyramids in Section 7.3.3.. From Section 7.3.4. onwards, we describe the MC algorithm, its mapping to HistoPyramid stream processing, and implementation details for both the OpenGL and the CUDA implementations. After that, we provide a performance analysis in Section 7.3.8., before we conclude in the final section.

7.3.2. Previous and Related Work

In recent years, iso-surface extraction on stream processors (like GPUs) has been a topic of intense research. MC is particularly suited for parallelization, as each MC cell can be processed individually. Nevertheless, the number of MC cells is substantial, and some approaches employ preprocessing strategies to avoid processing of empty regions at render time. Unfortunately, this greatly reduces the applicability of the approach to dynamic data. Also, merging the outputs of the MC cells' triangles into one compact sequence is not trivial to parallelize.

Prior to the introduction of geometry shaders, GPUs completely lacked functionality to create primitives directly. Consequently, geometry had to be instantiated by the CPU or be prepared as a vertex buffer object (VBO). Therefore, a fixed number of triangles had to be assumed for each MC cell, and by making the GPU cull degenerate primitives, the superfluous triangles could be discarded. Some approaches used MT to reduce the amount of redundant triangle geometry, since MT never requires more than two triangles per MT tetrahedron. In addition, the configuration of a MT tetrahedron can be determined by inspecting only four corners, reducing the amount of inputs. However, for a cubical grid, each cube must be subdivided into at least five tetrahedra, which makes the total number of triangles usually larger in total than for an MC-generated mesh. Beyond that, tetrahedral subdivision of cubical grids introduces artifacts [CMS06].

Pascucci et al. [Pas04] represents each MT tetrahedron with a quad and let the vertex shader determine intersections. The input geometry is comprised of triangle strips arranged in a 3D space-filling curve, which minimizes the workload of the vertex shader. The MT approach of Klein et al. [KSE04] renders the geometry into vertex arrays, moving the computations to the fragment shader. Kipfer et al. [KW05] improved upon this by letting edge intersections be shared.

Buatois et al. applied multiple stages and vertex texture lookups to reduce redundant calculations [BCL06]. Some approaches reduce the impact of fixed expansion by using spatial data-structures. Kipfer et al. [KW05], for example, identify empty regions in their MT approach using an interval tree. Goetz et al. [GJD05] let the CPU classify MC cells, and only feed surface-relevant MC cells to the GPU, an approach also taken by Johannson et al. [JC06], where a kd-tree is used to cull empty regions. But they also note that this preprocessing on the CPU limits the speed of the algorithm. The geometry shader (GS) stage of SM4 hardware can produce and discard geometry on the fly. Uralsky et al. [Ura06] therefore propose a GS-based MT approach for cubical grids, splitting each cube into six tetrahedra. An implementation is provided in the Nvidia OpenGL SDK-10, and has also been included in the performance analysis.

Most methods could provide a copy of the iso-surface in GPU memory, using either vertex buffers or the new transform feedback mechanism of SM4-hardware. However, except for GS-based approaches, the copy would be littered with degenerate geometry, so additional post-processing, such as stream compaction, would be needed to produce a compact sequence of triangles.

Horn's early approach [Hor05] to GPU-based stream compaction uses a prefix sum method to generate output offsets for each input element. Then, for each output element, the corresponding input element is gathered using binary search. The approach has a complexity of O(nlogn) and does not perform well on large datasets. Prefix Sum (Scan) uses a pyramid-like up-sweep and down-sweep phase, where it creates, in parallel, a table that associates each input element with output offsets. Then, using scattering, the GPU can iterate over input elements and directly store the output using this offset table. Harris [Har07] designed an efficient implementation of Scan in CUDA.

The Nvidia CUDA SDK 1.1 provides an MC implementation using Scan, and we have included a highly optimized version in the performance analysis.

But, with the introduction of HistoPyramids, data compaction and expansion can be run on the GPU of SM3 hardware. Despite a deep gather process for the output elements, the algorithm shows to be extremely fast when extracting a small subset of the input data.

7.3.3. HistoPyramids

The core component of our MC implementation is the HistoPyramid data structure, introduced in [ZTTS06] for GPU-based data compaction, introduced in chapter 6. We extend its definition here, and introduce the concept of local key indices (below) to provide for GPU-based 1:m expansion of data stream elements for all non-negative m. The input is a stream of data input elements, short: the input stream. Now, each input element may allocate a given number of elements in the output stream. If an input element allocates zero elements in the output stream, the input element is discarded and the output stream becomes smaller (data compaction). On the other hand, if the input element allocates more than one output element, the stream is expanded (data expansion). The input elements' individual allocation is determined by a user-supplied predicate function which determines the output multiplicity for each input element. As a sidenote, in chapter 6, each element allocated exactly one output or none.

The HistoPyramid algorithm consists of two distinct phases. In the first phase, we create a HistoPyramid, a pyramid-like data structure very similar to a MipMap. In the second phase, we extract the output elements by traversing the HistoPyramid top-down to find the corresponding input elements. In the case of stream expansion, we also determine which numbered copy of the input element we are currently generating.

7.3.3.1. Construction

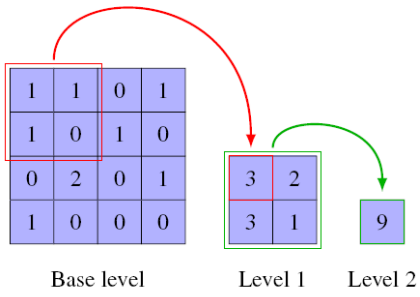


Figure 7.23: Bottom-up build process of the HistoPyramid, adding the values of four texels repeatedly. The top texel contains the total number of output elements in the pyramid.

The first step is to build the HistoPyramid, a stack of 2D textures. At each level, the texture size is a quarter of the size of the level below, i.e. the same layout as the MipMap pyramid of a 2D texture. We call the largest texture, in the bottom of the stack, the base texture, and the single texel of the 1×1 texture in the top of the stack the top element. Figure 7.23 shows the levels of a HistoPyramid, laid out from left to right. The texel count in the base texture is the maximum number of input elements the HistoPyramid can handle. For simplicity, we assume that the base texture is square and the side length a power of two (arbitrary sizes can be accommodated with suitable padding).

In the base level, each input element corresponds to one texel. This texel holds the number of allocated output elements. In Figure we have an input stream of 16 elements, laid out from left to right and top to bottom. Thus, elements number 0,1,3,4,6,11, and 12 have allocated one output element each (stream pass-through). Element number 9 has allocated two output elements (stream expansion), while the rest of the elements have not allocated anything (stream compaction). These

elements will be discarded. The number of elements to be allocated is determined by the predicate function at the base level. This predicate function may also map the dimension of the input stream to the 2D layout of the base level. In our MC application, the input stream is a 3D volume.

The next step is to build the rest of the levels from bottom up, level by level. According to the MipMap principle, each texel in a level corresponds to four texels in the level below. In contrast to the averaging used in the construction of MipMaps, we sum the four elements. Thus, each texel receives the sum of the four corresponding elements in the level below. The example in Figure illustrates this process. The sum of the texels in the 2×2 block in the upper left of the base level is three, and stored in the upper left texel of Level 1. The sum of the texels in the single 2×2 block of Level 1 is nine, and stored in the single texel of Level 2, the top element of the HistoPyramid.

At each level, the computation of a texel depends only on four texels from the previous one. This allows us to compute all texels in one level in parallel, without any data interdependencies.

7.3.3.2. Traversal

In the second phase, we generate the output stream. The number of output elements is provided by the top element of the HistoPyramid. Now, to fill the output stream, we traverse the HistoPyramid once per output element. To do this, we linearly enumerate the output elements with a *key index* **k**, and *re-interpret HP values as key index intervals*.

The traversal requires several more variables: We let **m** denote the number of HP levels. The traversal maintains a texture coordinate **p** and a current level **level**, enough to address specific texels in the HistoPyramid. We further maintain a local key index $\mathbf{k_L}$, which follows the local index range. It is initialized as $\mathbf{k_L} = \mathbf{k}$. The traversal starts from the top level, **level = m,** and descends down, terminating at the base level, **level** = 0. During traversal, $\mathbf{k_L}$ and **p** are updated continuously, and when traversal terminates, **p** points at a texel in the base level. In the case of pure data compaction, $\mathbf{k_L}$ is always zero when traversal terminates, as only one copy is produced. However, in the case of *data expansion*, the value in $\mathbf{k_L}$ holds the copy number of the input element for this particular output element.

Traversal starts as follows: **level = m**, and **p** points at the single texel in the top level. We subtract one from **level**, descending one step in the HistoPyramid, and after trivial upscaling, **p** addresses now a 2×2 block of texels in **level-1.** m−1 corresponding to the single texel p pointed to at level m. We label these four texels as $\begin{matrix}\mathbf{a} & \mathbf{b}\\ \mathbf{c} & \mathbf{d}\end{matrix}$ and use the values of these texels to form four index ranges **A**, **B**, **C**, and **D**, defined as $\begin{matrix}[A] & [B]\\ [C] & [D]\end{matrix} = \begin{matrix}\mathbf{[0,a)} & \mathbf{[a, a+b)}\\ \mathbf{[a+b,a+b+c)} & \mathbf{[a+b+c,a+b+c+d)}\end{matrix}$.

Then, we examine which of the four ranges $\mathbf{k_L}$ falls into. If, for example, $\mathbf{k_L}$ falls into range **B**, we let **p** point to texel **b** and subtract the index start of range **B**, here: **a**, from $\mathbf{k_L}$. This adapts $\mathbf{k_L}$ to the local index range of **B** for the next iteration. We iterate with a descend, subtracting one from **level** and repeating the process. This happens until **level** is zero, where traversal terminates. At termination, **p** points to the aftersought texel for the input stream element, and the value in $\mathbf{k_L}$ enumerates the current copy. Now, the output can be written. Usually, input element data is copied, and/or position **p** is written. In case of data expansion, $\mathbf{k_L}$ is pivotal if the output shall be a modified version of the input. In this case, the value of $\mathbf{k_L}$ determines the modification of the input, which is the crucial input for e.g. tesselation of input triangles. This way, geometry shaders can be emulated.

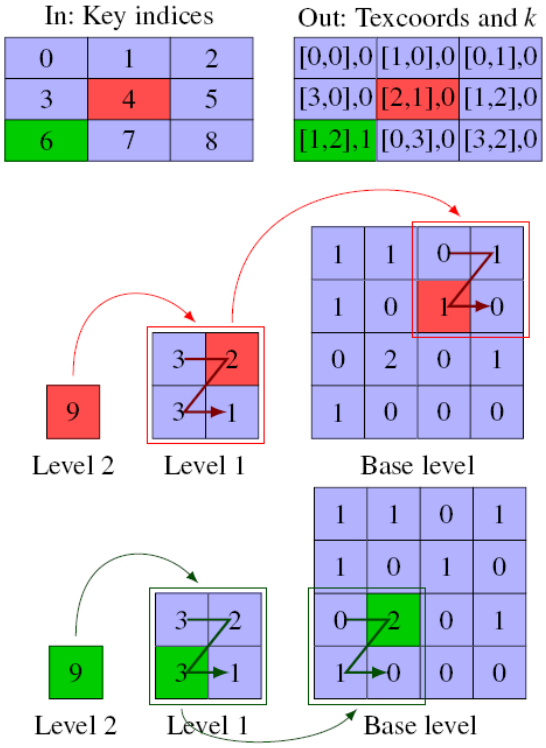


Figure 7.24: Element extraction, interpreting partial sums as interval in top-down traversal. Red traces the extraction of key index 4 and green traces key index 6.

Figure 7.24 shows two examples of HistoPyramid traversal. The first example, labeled in red, is for the key index k = 4, a stream pass-through. We start at level 2 and descend to level 1. The four texels at level 1 form the ranges **A** = [0 , 3), **B** = [3 , 5), **C** = [5 , 8), **D** = [8 , 9). We see that $\mathbf{k_L}$ is in the range of **B**. Thus, we adjust **p** to point to the upper left texel and adjust $\mathbf{k_L}$ to the new index range by subtracting 3, which leaves $\mathbf{k_L} = 1$. Then, we descend to the base level. The four texels in the base level, adjacent to the upper left texel of level 1, form the ranges **A** = [0 , 0), **B** = [0 , 1), **C** = [1 , 2), **D** = [2 , 2). The ranges A and D are empty. Here, $\mathbf{k_L} = 1$ falls into C, and we adjust **p** and $\mathbf{k_L}$ accordingly. Since we are at the base level, traversal terminates, with **p** = [2,1] and $\mathbf{k_L} = 0$.

The second example of Figure 7.24, labeled in green, is a case of stream expansion, with key index **k** = 6. We begin at the top of the HistoPyramid and descend to level 2. Again, the four texels form the ranges **A** = [0 , 3), **B** = [3 , 5), **C** = [5 , 8), **D** = [8 , 9), and $\mathbf{k_L}$ falls into the range **C**. We adjust **p** to point to **c** and subtract the start of range **C** from kl , resulting in the new local key index $\mathbf{k_L} = 1$. Descending, we inspect the four texels in the lower left corner of the base level, which form the four ranges **A** = [0 , 0), **B** = [0 , 2), **C** = [2 , 3), **D** = [3 , 3), where $\mathbf{k_L}$ now falls into range **B**, and we adjust **p** and $\mathbf{k_L}$ accordingly. Since we're at the base level, we terminate traversal, and have **p** = [1,2]. $\mathbf{k_L} = 1$ implies that output element 6 is the second copy of the input element from position [1,2] in the base texture, information that could be used for copy-dependent modification.

The traversal only reads from the HistoPyramid. There are no data dependencies between traversals. Therefore, the output elements can be extracted independently — even in parallel.

7.3.4. Marching Cubes

The Marching Cubes (MC) algorithm [LC87] of Lorensen and Cline is probably the most commonly used algorithm for extracting iso-surfaces from scalar fields, which is why we chose it as basis for our GPU iso-surface extraction. From a 3D grid of N ×M×L scalar values, we form a grid

of (N−1)×(M−1)×(L−1) cube-shaped “MC cells” inbetween the scalar values such that each corner of the cube corresponds to a scalar value. The basic idea is to “march” through all the cells one-by-one, and for each cell, produce a set of triangles that approximates the iso-surface locally in that particular cell.

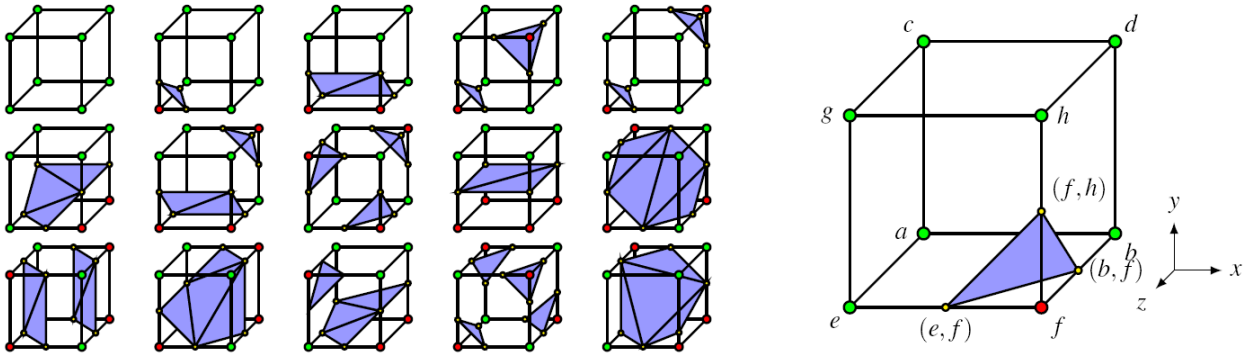


Figure 7.25: The 15 basic predefined triangulations [LC87] for edge intersections (left). By symmetry, they provide triangulations for all 256 MC cases. Ambiguous cases are handled by adding some extra triangulations [MSS94]. A MC cell (right) where only f is inside the iso-surface, and thus the edges (e, f), (b, f), and (f ,h) intersect the iso-surface.

It is assumed that the topology of the iso-surface inside a MC cell can be completely determined from classifying the eight corners of the MC cell as inside or outside the isosurface. Thus, the topology of the local iso-surface can be encoded into an eight-bit integer, which we call the MC case of the MC cell. If any of the twelve edges of the MC cell have one endpoint inside and one outside, the edge is said to be piercing the iso-surface. The set of piercing edges is completely determined by the MC case of the cell. E.g., the MC cell right in Figure 7.25 has corner f inside and the rest of the corners outside. Encoding “inside” with 1 and “outside” with 0, we attain the MC case %00000100 in binary notation, or 32 in decimal. The three piercing edges of the MC cell are (b, f), (e, f), and (f ,h).

For each piercing edge we determine the intersection point where the edge intersects the iso-surface. By triangulating these intersection points we attain an approximation of the iso-surface inside the MC cell, and with some care, the triangles of two adjacent MC cells fit together. Since the intersection points only move along the piercing edges, there are essentially 256 possible triangulations, one for each MC case. From 15 basic predefined triangulations, depicted left in Figure 7.25, we can create triangulations for all 256 MC cases due to inherent symmetries [LC87].

However, some of the MC cases are ambiguous, which may result in a discontinuous surface. Luckily, this is easily remedied by modifying the triangulations for some of the MC cases [MSS94]. On the downside, this also increases the maximum number of triangles emitted per MC cell from 4 to 5. Where a piercing edge intersects the iso-surface is determined by the scalar field along the edge. However, the scalar field is only known at the end-points of the edge, so some assumptions must be made. A simple approach is to position the intersection at the midpoint of the edge, however, this choice leads to an excessively “blocky” appearance, see the left side of Figure 7.26. A better choice is to approximate the scalar field along the edge with an interpolating linear polynomial, and find the intersection using this approximation, as shown in the right half of Figure 7.26.

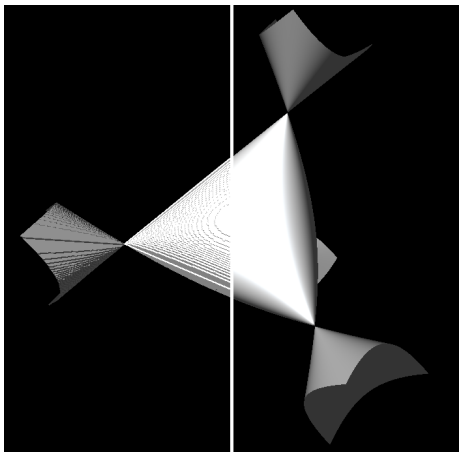


Figure 7.26: The quality difference in assuming that edges pierce the iso-surface at the middle of an edge (left) and using an approximating linear polynomial to determine the intersection (right).

7.3.5. Marching Cubes in Stream and HistoPyramid-Processing

Our approach is to implement MC as a sequence of data stream operations, with the input data stream being the cells of the 3D scalar field, and the output stream being a set of vertices, forming the triangles of the iso-surface. The data stream operations are executed via the HistoPyramid or, in one variant, the geometry shader, which compact and expand the data stream as necessary.

Figure 7.27 shows a flowchart of our algorithm.We use a texture to represent the 3D scalar field, and the first step of our algorithm is to update this field. The 3D scalar field can stem from a variety of sources: it may e.g. originate from disk storage, CPU memory, or simply be the result of GPGPU computations. For static scalar fields, this update is of course only needed once.

The next step is to build the HistoPyramid. We start at the base level. Our predicate function corresponds the base level texels with one MC cell each, and calculates the corresponding 3D scalar field coordinates. Then, it samples the scalar field to classify the MC cell corners. By comparing against the iso-level, it can determine which MC cell corners are inside or outside the iso-surface. This determines the MC case of the cell, and thus the number of vertices needed to triangulate this case.We store this value in the base level, and can now proceed with HistoPyramid build-up for the rest of the levels, as described in Section 7.3.3.2.

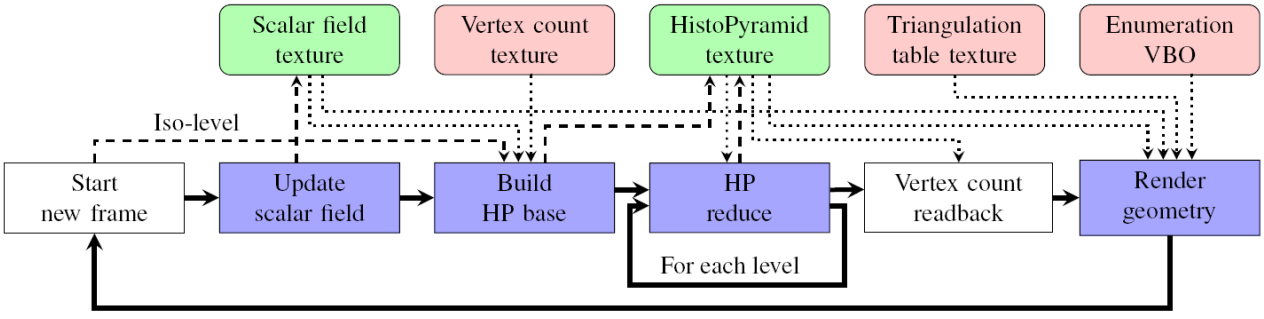


Figure 7.27: A schematic view of the implementation.
Thick arrows designate control flow, with blue boxes executing on the GPU and white boxes on the CPU. The dotted and dashed arrows represent data flow, with dotted arrows for fetches and dashed arrows for writes. Green boxes stand for dynamic data. Red boxes for static data.

After HistoPyramid buildup has been completed, we read back the single texel at its top level. This makes the CPU aware of the actual number of vertices required for a complete iso-surface mesh.

Dividing this number by three yields the number of triangles. As already mentioned, output elements can be extracted by traversing the HistoPyramid. Therefore, the render pass is fed with dummy vertices, enumerated with increasing key indices. For each input vertex, we use its key index to conduct a HistoPyramid traversal, as already described in the beginning of this chapter.

After the traversal, we have a texel position in the base level and a local key index k_L. From the texel position in the base texture, we can determine the corresponding 3D coordinate, inverting the predicate function's 3D to 2D mapping. Using the MC case of the cell and the local key index k_L, we can perform a lookup in the triangulation table texture, a 16×256 table where entry (i, j) tells which edge vertex i in a cell of MC case j corresponds to. Then, we sample the scalar field at the two end-points of the edge, determine a linear interpolant of the scalar field along this edge, find the exact intersection, and emit the corresponding vertex. In effect, the algorithm has transformed the stream of 3D scalar field values into a stream of vertices, generated on the fly while rendering iso-surface geometry. Still, the geometry can be stored in a buffer on the GPU if so needed, either by by rendering to a vertex buffer, or by using transform feedback buffers on SM4 hardware [BLW06].

7.3.6. Implementation Details

In detail, the actual implementation of our MC approach contains some noteworthy caveats described in this Chapter. We store the 3D scalar field using a large tiled 2D texture, know as a Flat3D layout [HSBL03], which allows the scalar field to be updated using a single GPGPU-pass. Since the HistoPyramid algorithm performs better for large amounts of data, we use the same layout for the base level of the HistoPyramid, allowing the entire volume to be processed using one HistoPyramid.

We employ a four-channel HistoPyramid, where the RGBA-values of each base level texel correspond to the analysis of a tiny 2×2×1-chunk of MC cells. Analysis begins by fetching the scalar values at the common 3×3×2 corners of the four MC cells. We compare these values to the iso-value to determine the inside/outside state of the corners, and from this determine the actual MC cases of the MC cells. The MC case corresponds to the MC template geometry set forth by the Marching Cubes algorithm. As it is needed in the extraction process, we use some of the bits in the base level texels to cache it. To do this, we let the vertex count be the integer part and the MC case the fractional part of a single float32 value. This is sound, as the maximum number of vertices needed by an MC case is 15, and therefore the vertex count only needs 4 of the 32 bits in a float32 value. This data co-sharing is only of relevance in the base-level, and the fractional part is stripped away when building the first level of the HistoPyramid. HistoPyramid texture building is implemented as consecutive GPGPU-passes of reduction operations, as exemplified in "render-to-texture loop with custom MipMap generation" [JS05], but instead of using one single framebuffer object (FBO) for all MipMap levels, we use a separate FBO for each MipMap level, yielding a speedup on some hardware. We retrieve the RGBA-value of the top element of the HistoPyramid to the CPU as the sum of these four values is the number of vertices in the iso-surface.

GLHP-VS, our SM3 implementation variant, uses the OpenGL vertex shader to generate the iso-surface on the fly (). Here, rendering is triggered by feeding the given number of (dummy) vertices to the vertex shader. The only vertex attribute provided by the CPU is a sequence of key indices, streamed off a static vertex buffer object (VBO). Even though SM4-hardware provides the gl_VertexID-attribute, OpenGL cannot initiate vertex processing without any attribute data, and hence a VBO is needed anyway. For each vertex, the vertex shader uses the provided key index to traverse the HistoPyramid, determining which MC cell and which of its edges this vertex is part of. It then samples the scalar field at both end-points of its edge, and uses its linear approximation to intersect with the edge. The shader can also find an approximate normal vector at this vertex, which it does by interpolating the forward differences of the 3D scalar field at the edge end-points.

GLHP-GS, our SM4 implementation variant of iso-surface extraction, lets the OpenGL geometry shader generate the vertices required for each MC cell. Here, the HistoPyramid is only used for data stream compaction, discarding MC cells that do not intersect with the iso-surface. To this purpose, we modified the predicate function to fill the HP base level with keep (1) or discard (0) values, since no output cloning is necessary for vertex generation.

After retrieving the number of geometry-producing MC cells from the top level of the HistoPyramid, the CPU triggers the geometry shader by rendering one point primitive per geometry-producing MC cell. For each primitive, the geometry shader first traverses the HistoPyramid and determines which MC cell this invocation corresponds to. Then, based on the stored MC case, it emits the required vertices and, optionally, their normals by iterating through the triangulation table texture. This way, the SM4 variant reduces the number of HistoPyramid traversals from once for every vertex of each iso-surface triangle, to once for every geometry-relevant MC cell.

If the iso-surface is required for post-processing, the geometry can be recorded directly as a compact list of triangles in GPU memory using either the new transform feedback extension [BLW06] or a more traditional render-to-texture setup, as shown in Figure 2.8 of the introduction. Algorithmically, there is no reason to handle the complete volume in one go, except for the moderate performance increase at large volume sizes which is typical for HistoPyramids. Hence, the volume could also be tiled into suitable chunks, making the memory impact of the HP small.

7.3.7. CUDA Implementation

Even though our method can be implemented using standard OpenGL 2.0 on Shader Model 3.0 class graphics hardware, we have noticed increased interest in performance behavior under the GPGPU programming language CUDA. In the following section, we describe some additional insights from porting our algorithm to CUDA. A thorough introduction to CUDA itself can be found in [NV07].

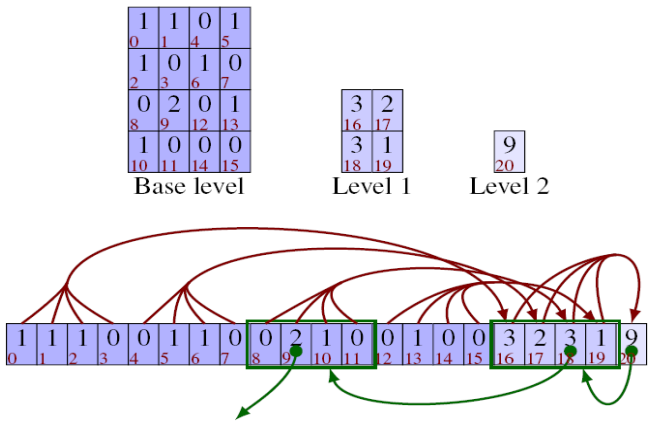


Figure 7.28: HistoPyramids in CUDA: A chunk.
Serialization of the HistoPyramid cells is shown in the cells' lower left. It corresponds to a one-allocation of each input cell. Using this layout, four numbers that form intervals lie consecutively in memory. Red arrows (above) show construction of the linearized HistoPyramid, while the green arrows (below) show extraction of key index 6, as exemplified in Figure 7.24 and explained in Section 7.3.3.2.

At the core of our algorithm lies again the HistoPyramid, a data structure that is best implemented on graphics hardware via 2D textures. Unfortunately, in the current release of CUDA, kernels cannot output data to a 2D texture without an intermediate device-device copy. Instead, the kernels

write output to linear memory, which can in its turn be bound directly to a 1D sampler. Therefore, we linearize the 2D structure using Morton-code, which preserves locality very well. The Morton-code of a point is determined by interleaving the bits of the coordinate values. Figure 7.28 shows a HistoPyramid with the Morton-code in the lower left corners of the elements. To improve the locality between MipMaplevels, we use *chunks*, HistoPyramids that are so small that all their levels remain close in memory. These chunks are then organized into *layers*, where the top level of the chunks in one layer forms the base of the chunks in the next layer. One example: using chunks with 64 base layer elements, we have one layer handle 3 levels of a comparable 2D HistoPyramid.

Using this layout, we can link our data structures to the CUDA computation concepts of *grids*, *blocks* and *threads*. Each layer of chunks is built by one grid. Inside the layer, each chunk is built with one block, using one thread per chunk base level element. The block's threads store the chunk base layer in global memory, but keep a copy in shared memory. Then, the first quarter of the threads continue, building the next level of the chunk from shared memory, again storing it in global mem, with a copy in shared mem and so on. Four consecutive elements are summed to form an element in the next level, as shown by the red arrows in Figure 7.28. HP Chunk/Layer Traversal is largely analogous to 2D texture-based traversal, as shown by the green arrows in Figure 7.28. In addition, for each chunk traversed, we must jump to the corresponding chunk in the layer below.

In our implementation variant **CUHP-CU**, CUDA populates a VBO based on data extraction from this traversal. In **CUHP-VS**, our alternate implementation, CUDA stores the layers of chunks in an OpenGL buffer object. These HP Chunk/Layer structures are then traversed in an OpenGL vertex shader.

In effect, we have transformed the HistoPyramid 2D data structure into a novel 1D HistoPyramid layer/chunk-structure. In principle, the memory requirement of the layer/chunk-structure is one third of the base layer size, just like for 2D MipMaps. But since empty chunks are never traversed, they can even be completely omitted. This way, the size of the input data needs only be padded up to the number of base elements in a chunk, which further reduces memory requirements. Furthermore, all layers do not need chunks with full 32-bit values. MC produces maximally 15 vertices per cell, which allows us to use 8-bit chunks with 16 base level elements in the first layer, and a layer with 16-bit chunks with 256 base level elements, before we have to start using 32-bit chunks. Thus, the flexibility of the layer/chunk-structure makes it easier to handle large datasets, very probably even out-of-core data.

We had good results using 64 elements in the chunk base layer.With this chunk size, a set of 2563 elements can be reduced using four layers. Since the chunks' cells are closely located in linear memory, we have improved 1D cache assistance — both in theory and practice.

7.3.8. Results

We have performed a series of performance benchmarks on six iso-surface extraction methods. Four are of our own design: the OpenGL-based HistoPyramid with extraction in the vertex shader (GLHP-VS) or extraction in the geometry shader (GLHP-GS), and the CUDA-based HistoPyramid with extraction into a VBO using CUDA (CUHP-CU), and extraction directly in the OpenGL vertex shader (CUHP-VS). In addition, we have benchmarked the MT-implementation [Ura06] from the Nvidia OpenGL SDK-10 (NV-SDK10), where the geometry shader is used for compaction and expansion. For the purpose of this performance analysis, we obtained a highly optimized version of the Scan [Har07]-based MC-implementation provided in the Nvidia CUDA 1.1 SDK. This optimized version (CUDA1.1+) is up to three times faster than the original version from the SDK, which reinforces the notion that intimate knowledge of the hardware is of advantage for application development in CUDA.

To measure the performance of the algorithms under various loads, we extracted iso-surfaces out of six different datasets, at four different resolutions. The iso-surfaces are depicted in Figure 7.29. The first three volumes, “Bunny”, “CThead”, and “MRbrain”, were obtained from the Stanford volume data archive [Sta], the “Bonsai” and “Aneurism” volumes were obtained from volvis.org [Vol]. The analytical “Cayley” surface is the zero set of the function $f(x,y,z)=16xyz+4(x+y+z)-1$ sampled over $[-1,1]^3$. While the algorithm is perfectly capable of handling dynamic volumes without modification, we have kept the scalar field and iso-level static to get consistent statistics. However, the full pipeline is run every frame.

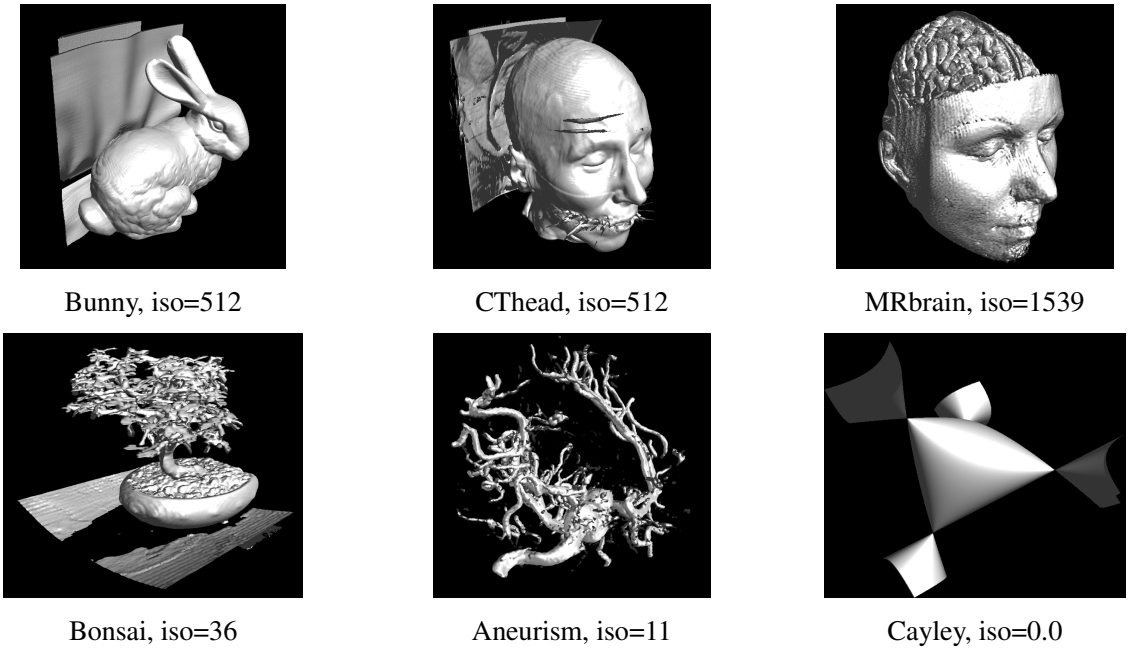


Figure 7.29: The iso-surfaces used in the performance analysis, along with the actually used iso-values used in the extraction process.

Table 7.1 shows the performance of the algorithms, given in *million MC-cells processed per second*, reflecting the throughput of each algorithm. In addition, the *frames per second*, given in parentheses, captures the interactiveness on current graphics hardware. Since the computations per MC cell vary, we recorded the percentage of MC cells that produce geometry. This is because processing of a MC cell that intersects the iso-surface is heavier than for MC cells that do not intersect. On average, each intersecting MC cell produces roughly two triangles.

All tests were carried out on a single workstation with an Intel Core2 2.13 GHz CPU and 1 GB RAM, with four different Nvidia GeForce graphics cards: A 128MB 6600GT, a 256MB 7800GT, a 512MB 8800GT-640, and a 768MB 8800GTX. Table 5 shows the results for the 7800GT and the 8800GTX, representing the SM3.0 and SM4.0 generations of graphics hardware. All tests were carried out under Linux, running the 169.04 Nvidia OpenGL display driver, except the test with NVSDK10, which was carried out on MS Windows under the 158.22 OpenGL display driver.

Evaluation shows that the HistoPyramid algorithms benefit considerably from increasing amounts of volume data. This meets our expectations, since we deem the HistoPyramid to be particularly suited for sparse input data, and in large volume datasets, large amounts of data can be culled early in the MC algorithm. However, some increase in throughput is also likely to be caused by the fact that larger chunks of data give increased possibility of data-parallelism, and require fewer GPU state-changes (shader setup, etc.) in relation to the data processed. This probably explains the (moderate) increase in performance for the NV-SDK10.

	Model	MC cells	Density	7800GT GLHP-VS	8800GTX GLHP-VS	8800GTX GLHP-GS	8800GTX CUHP-CU	8800GTX CUHP-VS	8800GTX NVSDK10	8800GTX CUDA1.1+
Bunny	255x255x255	16581375	3.18%	—	540 (33)	82 (5.0)	414 (25)	420 (25)	—	400 (24)
	127x127x127	2048383	5.64%	12 (5.7)	295 (144)	45 (22)	250 (122)	248 (121)	—	246 (120)
	63x63x63	250047	9.07%	8.5 (34)	133 (530)	27 (108)	94 (378)	119 (474)	28 (113)	109 (436)
	31x31x31	29791	13.57%	5.0 (167)	22 (722)	12 (399)	17 (569)	24 (805)	22 (734)	22 (739)
CTHead	255x255x127	8258175	3.73%	16 (2.0)	434 (53)	68 (8.2)	372 (45)	366 (44)	—	358 (43)
	127x127x63	1016127	6.25%	12 (11)	265 (260)	40 (40)	200 (197)	200 (196)	—	217 (213)
	63x63x31	123039	9.62%	7.6 (62)	82 (669)	23 (189)	58 (473)	76 (615)	25 (206)	70 (571)
	31x31x15	14415	14.46%	4.5 (310)	11 (768)	8 (566)	8.6 (599)	12 (857)	17 (1187)	12 (802)
MRBrain	255x255x127	8258175	5.87%	10 (1.3)	305 (37)	38 (4.6)	269 (33)	274 (33)	—	279 (34)
	127x127x63	1016127	7.35%	9.8 (9.7)	239 (235)	32 (31)	184 (181)	183 (180)	—	112 (194)
	63x63x31	123039	9.96%	7.4 (60)	82 (663)	21 (169)	57 (466)	75 (611)	26 (215)	70 (566)
	31x31x15	14415	14.91%	4.3 (302)	11 (771)	8 (546)	8.5 (589)	12 (837)	18 (1257)	12 (795)
Bonsai	255x255x255	16581375	3.04%	—	562 (34)	82 (4.9)	427 (26)	433 (26)	—	407 (25)
	127x127x127	2048383	5.07%	13 (6.3)	314 (153)	45 (22)	264 (129)	262 (128)	—	269 (131)
	63x63x63	250047	6.69%	11 (45)	148 (590)	32 (127)	103 (413)	132 (526)	29 (116)	119 (476)
	31x31x31	29791	8.17%	8.0 (268)	21 (717)	16 (529)	17 (578)	25 (827)	24 (805)	23 (783)
Aneurism	255x255x255	16581375	1.60%	—	905 (55)	134 (8.1)	605 (37)	598 (36)	—	510 (31)
	127x127x127	2048383	2.11%	29 (14)	520 (254)	98 (48)	396 (193)	427 (209)	—	372 (182)
	63x63x63	250047	3.70%	19 (77)	169 (676)	50 (199)	116 (464)	152 (607)	33 (132)	136 (544)
	31x31x31	29791	6.80%	8.7 (292)	21 (715)	16 (545)	17 (584)	25 (830)	26 (857)	24 (789)
Cayley	255x255x255	16581375	0.93%	—	1135 (68)	245 (15)	695 (42)	700 (42)	—	563 (34)
	127x127x127	2048383	1.89%	31 (15)	534 (261)	118 (58)	405 (198)	438 (214)	—	377 (184)
	63x63x63	250047	3.87%	18 (72)	174 (695)	52 (206)	116 (465)	151 (606)	32 (129)	133 (530)
	31x31x31	29791	8.10%	7.3 (246)	22 (736)	17 (574)	18 (589)	25 (828)	25 (828)	23 (774)

Table 7.1: The performance of extraction and rendering of iso-surfaces, measured in million MC cells processed per second, with frames per second given in parentheses. The implementations are described in Section 7.3.6.

The 6600GT performs quite consistently at half the speed of the 7800GT, indicating that HistoPyramid buildup speeds are highly dependent on memory bandwidth, as the 7800GT has twice the memory bandwidth of the 6600GT. The 8800GT performs at around 90–100% the speed of the 8800GTX, which is slightly faster than expected, given it only has 70% of the memory bandwidth. This might be explained by the architecture improvements carried out along with the improved fabrication process that differentiates the GT from the GTX. However, the HP-VS algorithm on the 8800GTX is 10–30 times faster than on the 7800GT, peaking at over 1000 million MC cells processed per second. This difference cannot be explained by larger caches and improved memory bandwidth alone, and shows the benefits of the unified Nvidia 8 architecture, enabling radically higher performance in the vertex shader-intensive extraction phase.

The CUDA implementations are not quite as efficient as the GLHP-VS, running at only 70–80% of its speed. However, if geometry must not only be rendered but also stored (GLHP-VS uses transform feedback in this case), the picture changes. There, CUHP-CU is at least as fast as GLHP-VS, and up to 60% faster for dense datasets than our reference. CUHP-VS using transform feedback, however, is consistently slower than the GLHP-VS with transform feedback, indicating that the 1D chunk/layer-layout is not as optimal as the MipMap-like 2D layout.

The geometry shader approach, GLHP-GS, has the theoretical advantage of reducing the number of HistoPyramid traversals to roughly one sixth of the vertex shader traversal in GLHP-VS. Surprisingly, in practice we observed a throughput that is four to eight times less than for GLHPVS, implying that the data amplification rate of the geometry shader cannot compete with the HistoPyramid, at least not in this application. It seems as if the overhead of this additional GPU pipeline stage is still considerably larger than the partially redundant HistoPyramid traversals. Similarly, the NVSDK10-approach shows a relatively mediocre performance compared to GLPHP-VS. But this picture is likely to change with improved geometry shaders of future hardware generations.

The CUDA1.1 approach uses two successive passes of scan. The first pass is a pure stream compaction pass, culling all MC cells that will not produce geometry. The second pass expands the stream of remaining MC cells. The advantage of this two-pass approach is that it enables direct iteration over the geometry-producing voxels, and this avoids a lot of redundant fetches from the scalar field and calculations of edge intersections. The geometry-producing voxels are processed homogenously until the final step where the output geometry is built using scatter write. This approach has approximately the same performance as our CUDA implementation for moderatly dense datasets, and slightly worse for sparse datasets, where the HistoPyramid excels.

We also experimented with various detail changes in the algorithm. One was to position the vertices at the edge midpoints, removing the need for sampling the scalar field in the extraction pass, as mentioned in Section 7.3.4.. In theory, this should increase performance, but experiments show that the speedup is marginal and visual quality drops drastically, see Figure 7.26. In addition, we benchmarked performance with different texture storage formats, including the new integer storage format of SM4. However, it turned out that the storage type still has relatively little impact in this hardware generation. We therefore omitted these results to improve readability.

7.3.9. Future Work

Future work might concentrate on out-of-core applications, which also benefit greatly from high-speed MC implementations. Multiple Rendering Targets will allow us to generate multiple iso-surfaces or to accelerate HistoPyramid processing (and thus geometry generation) even further. For example, a view-dependent layering of volume data could allow for immediate output of transparency sorted isosurface geometry. We also consider introducing indexed triangle mesh output in our framework, as they preserve mesh connectivity. For that purpose, we would experiment with algorithmic approaches that avoid the two passes and scattering that the straight-forward solution would require.We have further received notice that this approach should fit to isosurface extraction from unstructured grids, since the whole approach is independent of the input’s data structure: It only requires a stream of MC cells.

7.4. Conclusion

In this chapter, we have shown how the HistoPyramid-algorithm, originally designed for data compaction, can be expanded to handle data expansion: All elements of an input data array are processed such that the output array contains the desired number of copies for each given input element. Additionally, each of the input elements' copies can be modified with a user defined function. In section 7.1 we detailed the basic modifications to the HistoPyramid algorithm in order to handle data expansion. We also compared our algorithm to a popular CUDA-based approach to data expansion, Scan, and find that its complexity relates only to the number of input elements, which is disadvantageous if only few of the input elements are actually required in the output.

In Section 7.2., we presented a light wavefront simulator that leads to a fast and versatile method to render a variety of sophisticated lighting effects in and around refractive objects in realtime. Both wavefront simulator and real-time renderer are based on a sophisticated model of light transport in volumetric scene representations that accounts for a variety of effects, including refraction, reflection, anisotropic scattering, emission and attenuation. Both employ a fast particle tracing method derived from the eikonal equation that enables us to efficiently simulate nonlinear viewing rays and complex propagating light wavefronts on graphics hardware. To implement adaptive wavefront tracing on the GPU, we have developed new data structures to perform adaptive wavefront tesselation in the light wavefront simulator, and thus adaptively adjust the complexity of ODE computation.

In Section 7.3. we have presented a fast and general method to extract isosurfaces from volume data, running completely on OpenGL 2.0 graphics hardware. It combines the well-known Marching Cubes algorithm with novel HistoPyramid algorithms to handle geometry generation via data expansion. We have described a basic variant using the HistoPyramid for data compaction and expansion, which works on almost any graphics hardware. Since our approach does not require any pre-processing and constantly re-generates the mesh from the raw data source, it can be applied to all kinds of dynamic datasources, be it analytical, volume dataset streaming, or a mixture of both (e.g. an analytical transfer function applied to a static volume, or output from volume processing). In addition to the basic algorithm from section 7.1., we introduced a Marching Cubes implementation using the geometry shader for data expansion, and one version implemented fully in CUDA.

We also experimented with CUDA-based data expansion, using a variant of the HistoPyramid that divides it into chunks/layers. But while the CUDA API is an excellent GPGPU tool, seemingly more fitting to data compaction and expansion, we still feel that a pure OpenGL implementation is of considerable interest. First of all, the OpenGL implementation still outperforms all other implementations. Further, it is *completely platform-independent*, and can thus be implemented on non-NVIDIA graphics hardware or even mobile graphics platforms, requiring only minor changes.

The data expansion mechanisms presented here are *not* specific to geometry generation, or even Marching Cubes in particular. Instead, totally different areas, such as particle generation in computer games, or advanced mobile image processing can benefit from the same principles. For graphics hardware lacking geometry shaders, such as mobile GPUs, the concept of HistoPyramid data expansion is easy to port. For example, a port of Marching Cubes to OpenGL ES, the mobile version of OpenGL, would be a good demonstration for general data compaction and expansion on mobile graphics hardware. This would open new application domains in mobile visual computing.

For Shader Model 4.0 generation hardware, we could show that HistoPyramid-based geometry generation outperforms the geometry shader functionality for the Marching Cubes algorithm. But our approach can also yield competitive advantages to other algorithms that depend on real-time geometry processing: For example, Dyken et al. have successfully implemented adaptive tesselation on the GPU, using a HistoPyramid algorithm to set up instancing batches, an approach yielding excellent performance [DRS07]. Of course, it can be expected that geometry shader performance will increase in forthcoming generations of graphics hardware. However, since general GPU improvements will benefit HistoPyramid performance as well, we are curious if the performance advantage of HistoPyramids to geometry shaders will actually close completely.

8. QuadPyramids

In Ziegler el al [ZTTS06], an interesting future application for HistoPyramids was sketched, based on the fact that many simulation problems deal with processing data of varying sample density. Two prominent examples of such varying sample density are data compression and encoding, which often require clustering of similar data regions, and fluid simuations, where data density has to adapt to the local complexity of the simulation [Har04, KW03]. Ziegler elaborates that HistoPyramid buildup can be modified to first cluster regions of common cell values, to mark at which level they are found in the hierarchy, and then to count these regions. With these modifications, HistoPyramid traversal can output a quadtree, because leafs terminate at the level where only identical values remain. This way, computation and storage adapts to sample density, in much the same spirit as sparse matrix computations do not waste resources on empty regions.

Following up on this early observation, it was later found that the pyramid´s inherent quadtree structure and an extended buildup and traversal actually allows *GPU-based construction of region quadtrees and octrees*.

Region quadtrees are convenient tools for hierarchical image analysis. Like the related Haar wavelets, they are simple to generate within a fixed calculation time. While the region quadtree partitioning is very rigid, it can be rapidly computed from arbitrary image input.

To achieve this, the original HistoPyramid buildup is extended with a two-mode reduction operation which can cluster cells with alike features, and thus build a specially-tagged quadtree HistoPyramid, the *QuadPyramid*. During traversal, this QuadPyramid is used as an implicit indexing data structure, which creates lists of the clustered image features directly in GPU memory. Note that for QuadPyramids, just as for HistoPyramids, *no scattering is required*. This avoids write cache conflicts on data-parallel architectures, and allows implementation in OpenGL and DirectX, even on basic graphics hardware.

The work presented in this chapter was published in [ZDTS07]. In Chapter 9, a similar set of algorithms, now extended to the 3D domain, will be used to create OcPyramids, and thus allow the creation of octrees, with a corresponding set of applications.

8.1. Overview

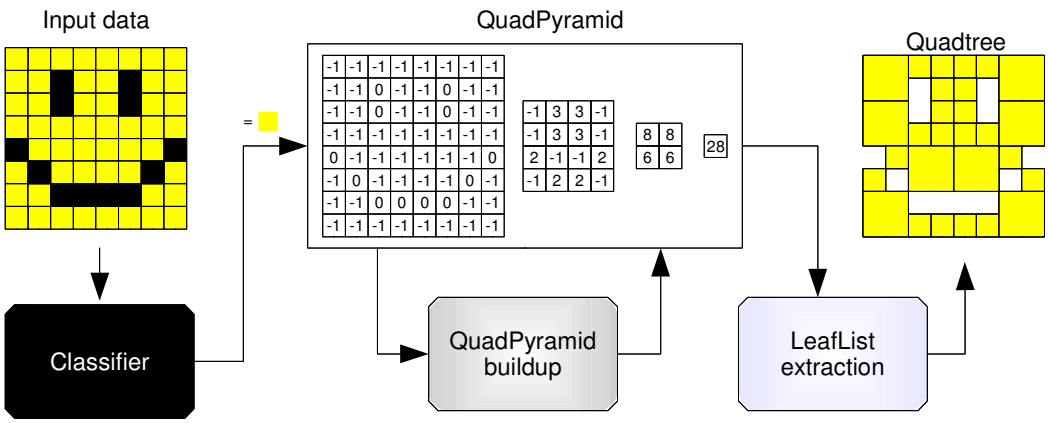


Figure 8.1: Data flow for a simple quadtree analysis.

Figure 8.1 illustrates the workflow between the different computation steps. For ease of understanding, we demonstrate a simple quadtree system, one that is only able to extract a color quad list, i.e. a list of quads of a certain color (which the discriminator identifies, here: yellow). All data is processed on the GPU – the CPU is only handling data if the quad list shall be downloaded, in case a non-GPU application requires so.

The **Classifier** decides if a cell's content is regarded as clusterable with other content (-1) or not (0). The input cells may be of arbitrary type, as long as the Classifier is able to make a clusterability decision on them. Section 8.3 explains more details.

QuadPyramid buildup receives the classifier output, and first clusters identical features as long as possible. When it cannot combine similar cells anymore, it switches to count mode and builds the pyramid of partial histograms. Its reduction operator repeatedly reduces four input cells into one, starting at the resolution level of the original input image. It finishes when only one output cell remains. We describe its GPU implementation in Section 8.4.

The **QuadList Builder** takes the prepared HistoPyramid, creates a 2D array to hold the quad leaflist, and fills every list entry via a hierarchical top-down traversal of the HistoPyramid. Section 8.5 describes details of the traversal in diagrams, and presents a log of the traversal decisions. Further, it is discussed which GPU restrictions hamper performance for the 3D case, and how their removal might improve future implementations.

The very basic nature of our proposed concepts can make it hard to understand the full range of new applications that an implementation on graphics hardware opens. Therefore, Section 8.7 outlines several real-time applications that become feasible with this GPU algorithm.

With the concept of QuadPyramid buildup and traversal, region quadtree analysis becomes a light-weight preprocessing step, e.g.for feature clustering in vision tasks. In section 8.8, we show how quadtrees can be applied in real-time video processing on standard PC hardware. The section summarizes the current performance results that we obtained by running the algorithm's variants on state-of-the-art graphics hardware. It describes the surrounding test setup, and analyzes the algorithm's runtime behaviour.

8.2. Related Work

Image pyramids have been used in Binary Tree Predictive Coding [Han85, Rob97]. For example, a quad tree leaf can signal if all of its descendants are identical, and hence skip the transmission of its descendants. Our algorithm uses this idea twice: First, it is used in QuadPyramid construction to combine features as long as they are identical, creating an as-large as possible quad leaf. Second, a zero is used to skip empty regions during the top-down QuadPyramid traversal which builds up the quad list.

The build process for the mentioned QuadPyramid is adopting the well-known parallel "reduction operation". It is applied in custom mipmapping [BP04], and processes n^2 elements in $\log_2(n)$ passes. Our operator switches modes during the process; first, it acts as a feature combiner – then, in higher levels, when features can not be combined anymore, it starts to sum them, just like in the original point list algorithm [ZTTS06].

Pyramidal image processing has been used to generate new kinds of high-speed image filters and real-time texture completion [SKE06]. While the used push-pull principle makes use of HistoPyramid-like data structures, it never explicitly generates a list, but instead conducts its operations by forwarding data between the pyramid levels. It can thus not be used for list generation.

Other GPU-based spatial data structures include indirection tables, N^3 trees and octrees [Lef06]. Those data structures are more memory-efficient than our solution, but none of them can be generated directly by the graphics hardware and thus need preprocessing.

Recently, an approach to data compaction was introduced, i.e. filtering of unwanted data elements from a given data stream [Hor05]. It does this by successively producing a running sum, which describes where to skip unwanted elements to obtain a packed result. The algorithm needs log(n) iterations to produce this running sum, and keeps the number of output elements constant during these iterations, which puts a considerable burden on memory transfers.

This can be improved by utilizing all intermediate data levels in the point list reconstruction, a reduction from log2(n)·n to 2·n data elements in the intermediate data output [Sen06].

Our algorithm uses a similar reduction approach, but in 2D space (cell coverage: 2x2:1), in contrast to the 1D nature (2:1) of both previously introduced algorithms. Besides mapping closer to the GPU's 2D image storage and texture caching mechanisms, it can smoothly integrate with the feature combiner (which is 2D in nature in the case of quadtrees). Our approach further omits the assumption of a certain number of processors in order to keep the algorithm general for future GPU generations.

In the early 80ies, quadtrees were occasionally used as a memory-efficient representation of large 2D images. Several hardware algorithms were available to generate views of the quadtree-compacted images directly on display hardware [Oli86].

Similar uses for data compaction are applied to depth maps by Parajola et al. [PSM04] and Bogomjakov et al. [BG04], who use quadtrees to create triangle meshes from depth maps before rendering them to the screen. Bogomjakov also has an early application of quadtree analysis on graphics hardware: He utilizes the GPU in a multi-pass, bottom-up approach to mark relevant triangles in the triangle mesh generation.

8.3. Classifier

Before a quadtree can be constructed from input data, we first need to determine which input cells are *clusterable* at the base level. Clusterability implies that input cells have content relevant to the quadtree, and that adjacent cells will be combined into larger and larger quadtree leafs, as long as the corresponding input regions have been completely clusterable.

In practice, the actual meaning of input clusterability can vary greatly: in a sparse matrix, all non-zero entries might be of interest for clustering, while in a vision application, it might be important to cluster all occurences of a certain color range (see Figure 8.2 for an example). In PDE applications, a list over all areas which exceed a certain maximum error threshold can be of relevance.

The Classifier's task is to conceal all these data modalities from the subsequent stages. It transforms the input into a simple decision: a quadtree leaf that is clusterable (-1) or no relevant input (0). While the Classifier output is binary, Classifier parameters for input processing can of coruse still be modified by the user. In our example application ffkvadrat, the user can modify the Classifier´s color gradient threshold during video processing, which causes a more or less rigid clustering of similar colors. Section 8.6 explains this in more detail.

The classifying function does not need to restrict itself to its associated input cell– it can also take the local neighborhood into consideration. We use this in Section 8.6.2 to perform gradient-based edge detection in the classifier. Further, more advanced examples can easily be adapted from our previous report on GPU point list generation [ZTTS06].

8.4. QuadPyramid Builder

In many graphics applications, a reduction operator is used for a simple task, namely to apply a global operation to all elements, while still making use of stream parallelization (e.g. to calculate a global minimum). In the two previous Chapters, the reduction operator was used to add up cells, generating partial sums of base level cell values. Our two-mode reduction operator is slightly more advanced: It changes its mode of operation while it builds the reduction pyramid, and thus combines feature matching with the data structure necessary for GPU list construction.

Initially, the reduction operator conducts *hierarchical feature clustering*. For every new cell it outputs, it checks the input cells' features for a match in order to create a new, larger quad leaf. In simple versions, this is a check if all incoming cells are leaf cells themselves. In more advanced operators, additional data influences if the contributing leaf cells actually may be clustered.

If the cells do match, then the leaf marker value is written to designate the single, combined quad tree leaf. In our case, the leaf marker value is a -1. Figure 8.2 shows a simple example where only clustering is employed, throughout the pyramid. Cells are repeatedly merged into larger quads, and the leaf marker -1 thus propagates all the way to the top of the pyramid. Thus, in the following leaf list extraction (to be described later), the resulting quad leaflist would only contain one quad, one that covers the whole input image. In more advanced versions, the clustering mode of the reduction acts as a recursive feature clustering. Section 7.6.1 explains this in more detail.

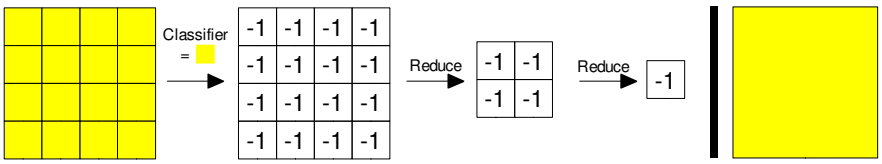


Figure 8.2: A simple example to demonstrate clustering mode. Right: Resulting single quad.

If the cells do *not* match, then the reduction operator becomes a summation operator, and effectively starts to count incoming elements. All present quad leafs are now being treated as single elements in the forthcoming reduction. Leaf markers (-1) need special treatment in the counting operation, making it seemingly expensive to calculate. But an abs( )-function call before the summation operation suffices to count a quad leaf as one single element in the reduction process, should leaf merger not be possible. Figure 8.3 shows an example where this happens, for the first time in the upper right corner at level 1 of the pyramid. Here, the three leaf markers cannot be combined into a new leaf marker, and thus must be treated as separate elements.

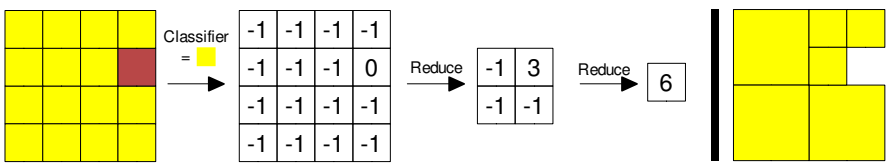


Figure 8.3: A switch to add-up mode in L1, some leaf markers cannot be clustered. Right: Result.

For N pixels in a quadratic arrangement, the GPU needs 2N read-write operations to create the QuadPyramid, just like for mip-mapping.

8.5. QuadList Extraction

Given the QuadPyramid as input, it is now possible to determine the number of list entries. QuadList Builder accesses the top level of the pyramid to retrieve this value, and allocates the quad leaflist, a 2D array sized large enough to hold the number of entries (see Figure 8.4 for an example

2D QuadList). The reason for choosing a 2D layout is that the GPU currently only can handle 4096 entries at maximum in a 1D image. The GPU treats this array as a 2D texture, and we use render-to-texture functionality to write to it.

Now, actual quad leaflist extraction commences. In our example in Figure 8.4, the QuadList extraction, an OpenGL shader, will be executed for all six required list entries in the 2D array.

Each running thread of the shader first determines its own list index (the key index) from its 2D coordinate in the array. Then it reads the top level, to know the total number of entries (the *quad count*, here: 6). It immediately terminates if its key index exceeds the list count (which does not happen in this example, but is possible if the 2D array is larger than the number of list entries).

Then, a HistoPyramid-like traversal starts; the algorithm descends if the key index lies within the index range of a QuadPyramid cell. Intuitively, the index range of a QuadPyramid cell describes the covered range of key indices that quad leaves can receive in this part of the 2D input image. The top level's single cell is a good example: its range covers all quad leaves' indices.

But there is one exception to the old traversal: if a leaf marker (-1) is encountered, and the key index matches the index of this marker, then no descend occurs. Instead, traversal is terminated immediately to write the output (see e.g. key index 4 in the example). The current position is scaled up to correspond to base level coordinates, and the size of the quad leaf is determined from the pyramid level at which the traversal terminated. In the example case of key index 4, traversal terminates at L1, the coordinate is scaled from L1's (0,1) to L2's (0,2), and the output size is 2x2, since the traversal terminated one level above the base level. Figure 8.5 shows all leaf markers which cause traversal to stop in our example.

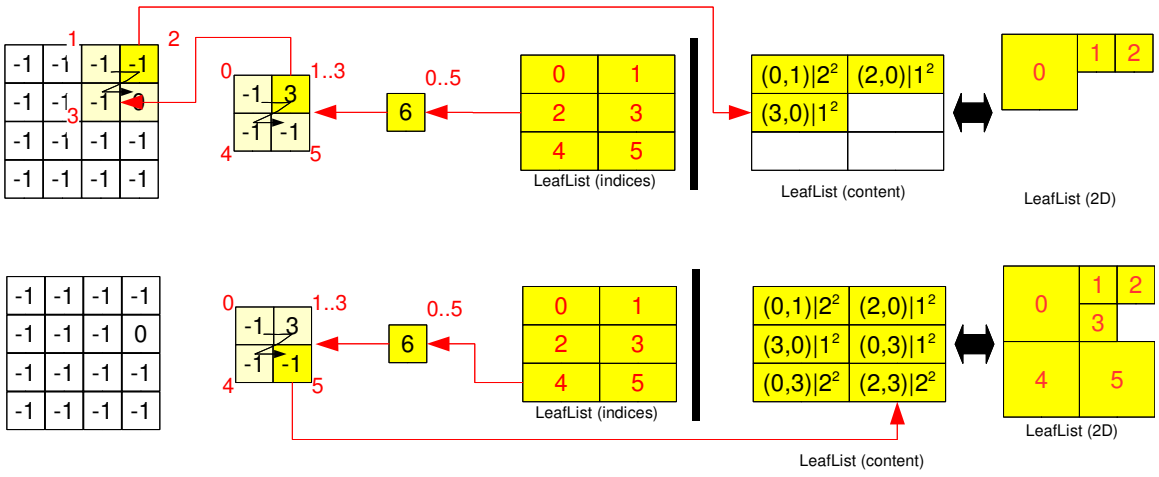


Figure 8.4: Example traversal for two LeafList entries, key indices 2 and 4.QuadPyramid Traversal starts at its top, and moves down until a quad leaf is found. The leaf entry is then placed in the LeafList (to the right).

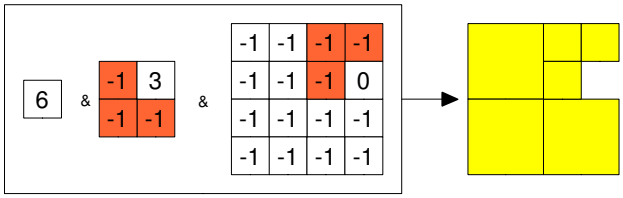


Figure 8.5: Left: Red marks where traversal ends in the quadlist construction. Right: Result.

Note that the traversal order is irrelevant as no sorting is enforced. All parallel traversal only needs to have the same order for indexing of quad list entries to avoid doublettes. That is certainly

fulfilled as all quad leaflist shader threads execute the same shader sourcecode. QuadList extraction thus assigns a unique list entry to each quadtree leaf.

The final result is a 2D array containing combined origin/size entries of all quad leafs in the image, the quad leaflist. For a square image of N pixels, the GPU thus needs at most *m log(sqrt(N))* texture accesses in the associated QuadPyramid to generate all *m* quadtree leaflist entries, depending on where the quads' leaf markers were encountered. The indexing order is somewhat unintuitive (based on the Morton code [Mor66], which generates a self-similar space-filling curve), but warrants very cache-efficient traversal.

8.6. Algorithmic Variants

Up to here, the Classifier has identified clusterable cells, and discarded all other information thereafter. Since clusterability is binary information according to its definition in section 8.3, it is not possible to create a single QuadPyramid that clusters multiple, but different feature groups. Instead, multiple QuadPyramids have to be created, every time with the Classifier set to identify a new feature. This is rather costly, and unnecessary if input cells can only belong exclusively to one of the multiple feature groups. We have therefore put some thought into how the QuadPyramid Builder can be extended to process multi-featured input data in one pass of QuadPyramid buildup, especially for cases where the exact feature clustering criterion is not known beforehand.

8.6.1. Feature Clustering Variant

In this variant, the Classifier becomes unnecessary. Instead, we let the QuadPyramid Builder calculate the feature average over the four input cells, and then calculate the deviation of the input features to the average feature. A user threshold on the deviation defines if the input cells are considered for potential merging.

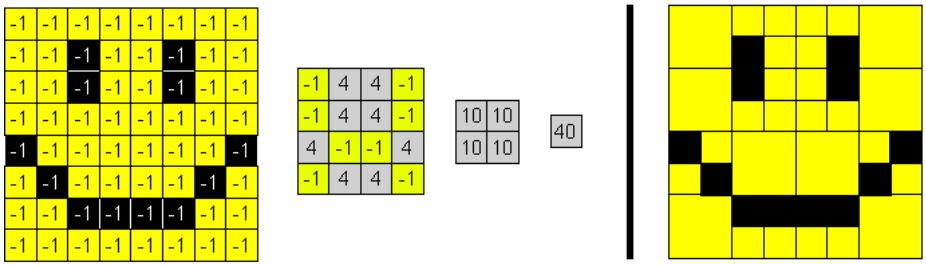


Figure 8.6: Common feature clustering uses the average of the contributing cell colors. Right: Result.

Since the deviation thresholding is baked into the reduction operator and thus is recursive, we store the calculated averages in the QuadPyramid. When no merger is possible, the average becomes uninteresting as the reduction operator has switched modes to counting leafs (grey cell background in Figure 8.6).

The advantage of this algorithm lies in its simplicity: No pre-processing is required to acquire multi-feature clustering. The feature grouping need not be known in advance. Feature grouping works accurately down to base level.

However, there are two disadvantages: First, the QuadPyramid requires considerably more memory, as it needs to accommodate the averaged feature vectors. Secondly, and more importantly, maximum deviation will differ for cells in one quad, depending on which level they have been combined. The reason for this is that each level only receives the feature averages from one level below, not the actual cell values. Thus, extreme values at the base level might create a feature

average that, if clustered by itself with other averages, does not take into account the original extreme feature deviations anymore. As a thought example, imagine what would happen if pixels of varying grey values would be grouped together, and how much cumulated deviation would be allowed at different QuadPyramid.

Still, the algorithm delivers fast results if the deviation value is chosen conservatively, and especially if only feature-identical cells shall be grouped (and deviation can thus be set to zero).

8.6.2. Edge Thresholding Variant

In this variant, we make use of the observation that input with similar features exhibits a low variation of features, i.e. the local feature gradient is low, or even zero. Therefore, the local feature gradient between adjacent input cells can be used to determine if cells shall be regarded clusterable or not. First, the local feature gradient is calculated, e.g. by simple forward differencing of feature values. A simple upper threshold of this gradient can now be used to mark areas of low feature variance as clusterable. For larger areas, it is of course possible that a certain drift in feature values occurs. But at least in human perception, the impression of continuity is based on contrast sensitivity, which depends in its turn on local comparisons, as experiments on just-noticeable differences in contrast show [Bar99]. Differently put: an input area is regarded as clusterable *as long as* no noticeable feature edges are present. In Figure 8.7, we exemplify how this general concept can be applied to cluster color features. Here, the Classifier makes its clusterability decisions by thresholding the sum of its three color channel gradients. Pixels with high color gradients are not considered clusterable (= 0), and will thus break up any grouping attempts at the base level.

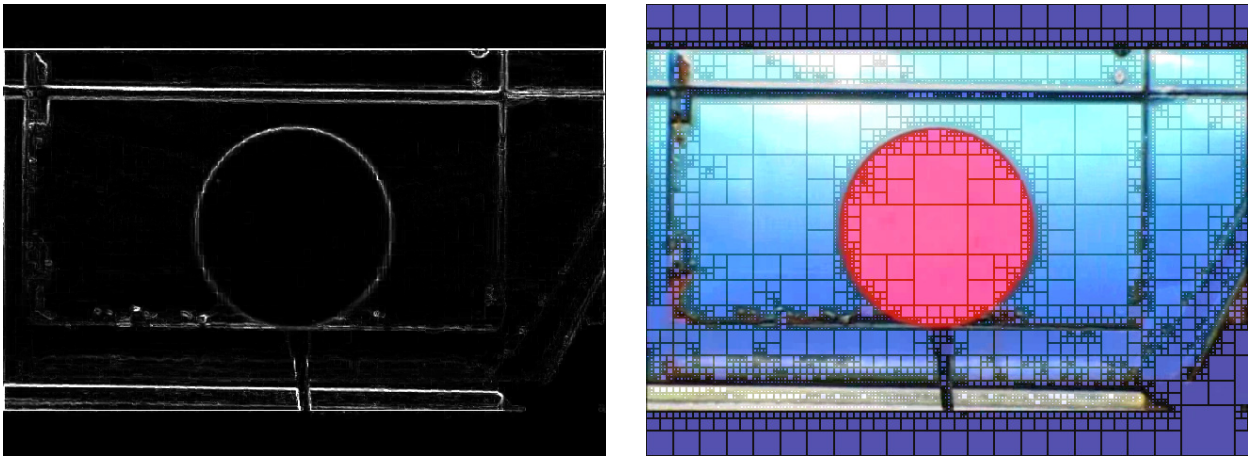


Figure 8.7: Left: Edge detection indicates non-mergeable areas. Right: Resulting QuadList.

The advantage of this method is that its memory consumption remains low, and that the clustering decisions are very intuitive for humans (low local feature variation leads to feature grouping). One disadvantage is that grouping is based on gradient information and thus not cell-exact (cells with high feature variation in their neighborhood will not be grouped at all, see the red disc's outline in Figure 8.7).

8.7. Applications

8.7.1. Feature Clustering in Computer Vision

One of the basic problems of computer vision is local feature clustering, as e.g. clustering of color, motion vectors or depth. While numerous algorithms exist for this issue, many of them fail to achieve real-time speed in video processing, and are thus only of limited use to interactive vision applications. Region Quadtree extraction can of course not provide a complete connected component analysis, but it does retrieve a quick, light-weight clustering (as shown in e.g. Figure 8.7), and can thus be used to detect objects of a certain minimum size, or allow to focus calculations on regions of interest, e.g. ones with high variance (by inverting the classifier behaviour of the Edge thresholding variant in section 8.6.2.)

8.7.2. Rendering on Demand

In Chalmers et al. [CDS*06], a system for rendering on demand is presented; it uses psycho-visual models to predict a saliency map, i.e. regions of increased viewer attention. Then, a computationally expensive physically-based lighting simulation is directed to mainly update those regions of high attention, and thus to considerably improve the perceived visual quality.

As Figure 4 in Chalmers et al. [CDS*06] shows, cells of increased attention can already be calculated on graphics hardware; however, for an efficient orchestration of lighting simulation, it is of advantage to know about complete regions of saliency, before renewed rendering is initiated. Quadtree based saliency clustering can assist here in reducing individual rendering runs, and thus further improve visual quality by reducing computational workload.

8.7.3. QuadTrees in Data Compression

Since data compression can take advantage of quadtrees [Han85] and higher-dimensional versions, like octrees [ZO04], it is probable that QuadPyramids can be used in compression applications for stream processors, especially for GPU-preprocessed volume and image data. Higher compression for the quadtree can be achieved by storing the actual traversal path for each quadtree leaf list entry, also known as the locational key, whose digits reflect successive quadrant subdivision.

Actually, this poses no larger problem as each shader thread knows about the full traversal path from the pyramid top down to the quad leaf marker, and thus can store this sequence of 2-bit cell descend decisions, the locational key, in one or more extra output variables.

Leaf list extraction applies a traversal pattern in extraction that lays out the leaves similar to a Morton Code [Mor66]. During traversal, a location code can be calculated according to e.g. [Mor66]. If this location code is stored together with the leaf data, then the QuadList has become a *linear quadtree*. Even without actual 2D coordinates for each leaf, such a linear quadtree is sufficient for point location in $O(\log_2(\text{num\_leaves}))$ [Kno06, Gar82]. QuadList extraction from QuadPyramids is thus highly suitable for compressed storage, provided a location code is calculated during extraction.

8.7.4. Mesh Reconstruction from Depth Maps

In Dimitrov et al. [Dim07], an approach for the reconstruction of 3D geometry from depth maps acquired by a robot´s laser range scanner is described. Here, feature clustering serves to determine if depth samples may be merged to form a larger quad, which leads to a more compact depthmap representation for subsequent depth processing stages.

8.7.5. Light-weight Preclustering of Linearly Behaving Regions

It is typical for PDE-based applications to operate on a specific spatial discretization and to require calculations of varying intensity for different locations ([BFGS03][LGR04]). A simple solution to stay below a certain error measure is to increase spatial resolution (that is, increased the resolution of discretization). Unfortunately, this is usually wasteful in many areas except for the hotspots, compare also the adaptive wavefront simulation of Chapter 7.2. By using the algorithm in this article and a specially adapted discriminator (specific for each PDE problem, and thus not described in this article), it is possible to identify regions where more fine-grained calculation is necessary.

Differently put: If a function f() is locally linearly interpolatable, as in Equation 1,

(1) $f(t.a + (1-t).b) = t.f(a) + (1-t).f(b)$

then a quadtree or octree analysis of the input's gradient can find regions where such function result interpolation could replace sample-by-sample calculations (for complex f()), and it can isolate regions where an increased step width in PDE solvers would not do any harm in terms of simulation stability, and thus benefit general purpose GPU computation [Lef06]. For the adaptive wavefront simulation of Chapter 7.2, this would actually make it possible to re-merge wavefront patches, should a previous high tesselation have become unnecessary.

8.8. Results

Finally, we present the results of some performance tests. The tests were conducted on a Dell Precision M70 laptop with Nvidia Quadro FX Go 1400 and 256 MB video memory, connected over PCI Express. It contained an Intel Pentium M (2.13 Ghz) and 2 GB of main memory. We could not test the algorithm on ATI graphics hardware without algorithmic redesign due to restrictions of the OpenGL driver software from ATI.

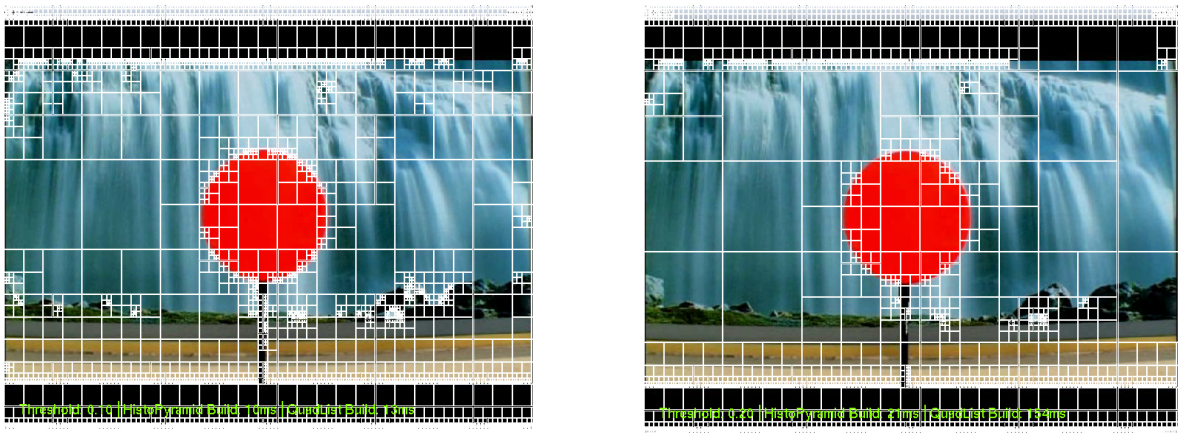


Figure 8.8: Two quadlist visualizations for different thresholds. Left: 0.20, Right: 0.10.

Our first tests were run with kvadrat, a Linux application which decomposew an image based on the algorithm variant of feature clustering, compare Section 8.6.1. Its clustering decision is based on the sum of squared deviation from the four input cells' common RGB color average. Figure 8.8 shows an example analysis of freely downloadable video footage [Gus99]. At threshold 0.20, the result seems to be intuitive: large quadtree leafs only occur in places where similar areas span a larger area. The quadtree is subdivided where two feature areas must be divided from each other. But as

the threshold changes to 0.10, it quickly becomes obvious that a human would not segment the input this way: For example, parts of the red disc are still correctly separated from the background, while other parts have been clustered together with the background.

We measured actual GPU timings with the GL_EXT_timer_query extension [Gre05b], instead of taking the time consumed by the OpenGL call delays on the CPU side. See Table 8.1 for timing measurements.

Threshold value	**0.1**	**0.2**
QuadPyramid Builder	10 ms	10 ms
QuadList Builder	12.4 ms	5.7 ms
Resulting quads	13261	7981

Table 8.1: Timings for different thresholds, feature combiner algorithm.

We further compared this algorithm with a CPU implementation. Aggressive compiler optimization accelerates the CPU based analysis. No SIMD techniques were used. CPU timings were taken as virtual process time by the getitimer() function of Linux systems. The CPU performs one pass at around 50 ms, which seems fast, considering that no SIMD techniques were involved. It has to be considered, though, that this performance was due to the fact that input data already resided in CPU memory. Most of the time, in real-time video applications, the GPU has been used in preprocessing or generation of data, which would require a bus transfer to download the data to CPU memory before it can be analyzed. If this is the case, then it ends all possible performance competition, as the bus transfers take 3-10 times the time of the actual computation, compare Ziegler et al. [ZTTS06].

In order to test the real-time behaviour of the edge thresholding algorithm from Section 8.6.2., we implemented ffkvadrat, a Linux player which analyzes video while it is being played back from GPU texture memory. After the video has been decoded as YUV 4:1:1 subsampled data planes, it is uploaded into GPU memory as three separate textures.

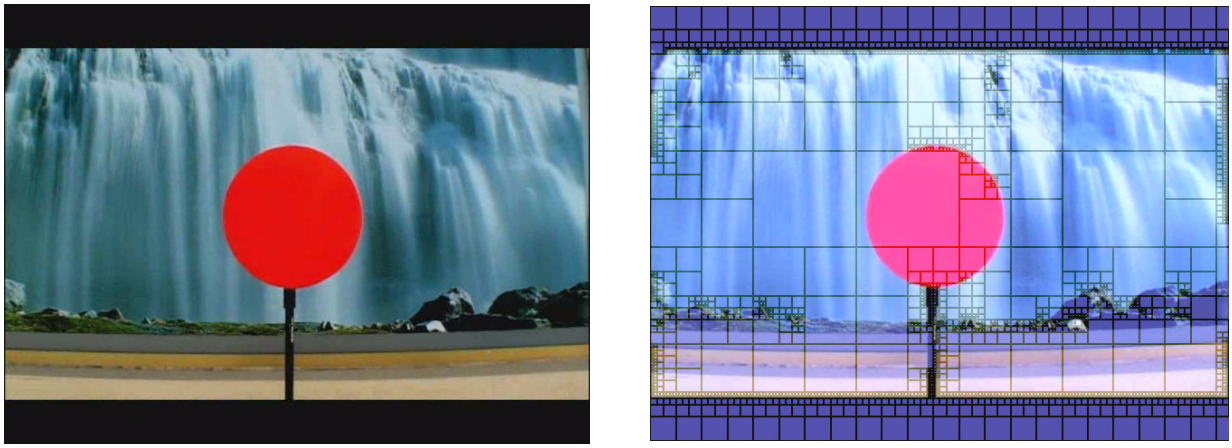


Figure 8.9: Left: Input frame. Right: Threshold value 0.92.

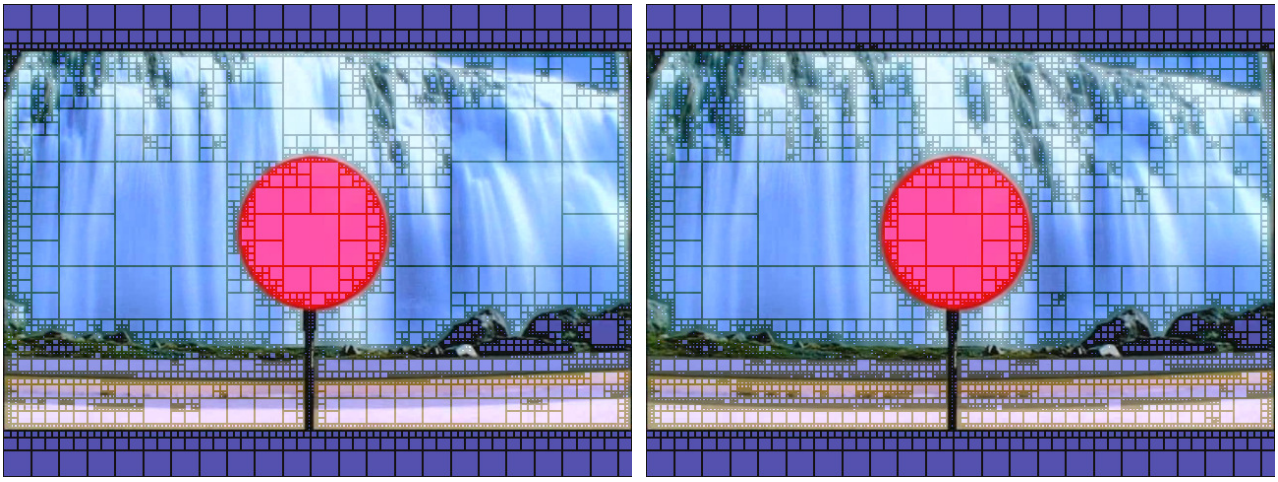


Figure 8.10: Left: Threshold value 0.22. Right: Threshold value 0.10.

Threshold value	0.10	0.22	0.92
QuadPyramid Builder	4 ms	4 ms	4 ms
QuadList Builder	13 ms	12 ms	11 ms
Resulting quads	22809	15375	4474

Table 8.2: Timings for different thresholds, edge thresholding algorithm.

Before running a simple fragment shader which converts the Y, U and V textures into RGB color values, we pre-process the video frame with gradient analysis, using forward differencing in horizontal and vertical directio. Now, we use this 2D gradient array as the input for a region quadtree analysis, as described in Section 8.4., where a quad leaf marker (= -1) is output if the square sum of the YUV gradient components remains below a certain threshold (i.e. it doesn't contain any edge information to speak of, and pixels may thus be clustered). Again, you find timings in Table 8.2.

Figure 8.9 and Figure 8.10 show the more intuitive results of the edge thresholding approach. While a high threshold does not consistently isolate features, it definitely starts to separate out the disc from the background at value 0.22, and even separates the street's different elements at value 0.10. We can therefore recommend this algorithmical variant for color segmentation, while more investigations are of course required for other feature data types.

8.9. Conclusion

We presented how quadtree based image partitioning can be done at unprecedented speed on stream processors. To achieve this, we have modified HistoPyramid algorithms to create region quadtrees from video within a dozen of milliseconds. The method is novel, fast and easy-to-use, and has a multitude of applications in image and video processing, as it makes region quadtree analysis a light-weight preprocessing step. Beyond image and video processing, applications in other fields arise as well - among them motion vector analysis, PDE calculations, and data compression.

Through experiments we have shown that our purely GPU-based implementation is significantly faster than a hybrid GPU/CPU implementation. The algorithm is highly cache-friendly, and scales innately with increasing parallelization. It will be highly interesting to see how it maps into image analysis, data compression and general purpose computation.

9. OcPyramids

In the previous Chapter, we already hinted at the fact that a QuadPyramid's inherent clustering could even be extended to three dimensions. Here, we present a set of 3D domain algorithms that create feature clustering pyramids in 3D, in short: *OcPyramids*. By traversing the OcPyramid, the GPU is enabled to generate octrees, a linked data structure which describes 3D regions with common features, completely on its own.

Octrees are highly useful in volume analysis (e.g. of medical data), raytracing applications and three dimensional PDE solvers. Due to the GPU's high memory bandwidth, even dynamic volumes such as time-dependent scalar fields or 3D volume streams can be analyzed.

Similar to the previous Chapter, we present a set of algorithms that produce leaf lists, in this case for octree leaves. In addition to this, we also present algorithms that produce *node lists*, that is, linked data structures that reflect the octree's subdivision into leaves, more widely known as pointer octrees. Pointer octrees can substantially accelerate raytracing and raycasting algorithms, and can also represent compressed high-resolution volume data in GPU memory. Just as for leaf lists, scattering capabilities are *not necessary* to create node lists.

Node lists can as well be created for dynamic volume data. To underline that node lists can be efficiently and directly created for dynamic volumetric data, we implemented real-time voxelization and octree creation of dynamic meshes. A natural continuation of this research would lead toward a classic octree application, raytracing acceleration for 3D meshes and volumes.

If explicit octree leaf and node lists are not needed, then octree leaf and node retrieval from the OcPyramid can be adapted directly to application needs – the following explanations should provide enough background to create customized OcPyramid traversals.

We further elaborate on several application areas, such as how the spatial clustering aspect of OcPyramids proves useful to GPU-based 3D PDE and ODE solvers.

The OcPyramid algorithm and data structure, as presented in this chapter, has not been published previously.

9.1. Related Work

Since OcPyramid algorithms build on the QuadPyramid algorithms, we refer to Section 8.2 for related work common to both.

A compact octree *representation* in OpenGL textures allows octree traversal without CPU assistance [LHN05]. Octree traversal is thus well-known for graphics hardware applications – but up to now, it has not been possible to *create* octrees on the GPU, which made their use in dynamic GPU applications rather restricted. Interestingly, a complementary website to this approach mentions that "*creating an octree (or any hierarchical structure) directly on the GPU is very difficult, mainly because of memory allocation and pointer creation. (It would require a scatter operation, not available on today's GPUs).*" As we will show in the following, such a scattering operation is unnecessary.

The build process for the OcPyramid is adopting the well-known parallel "reduction operation". It is applied in custom mipmapping [BP04], and processes n^3 elements in $\log_2(n)$ passes. In the Leaf OcPyramid creation, our reduction operator either clusters input cells or adds up previously allocated leaf nodes. In the Node/Leaf OcPyramid creation, it creates new nodes by observing mode

switches in the parallel leaf OcPyramid buildup, and promotes previously allocated nodes through the reduction where necessary.

Other related GPU-based spatial data structures include indirection tables, N^3 trees and octrees [Lef06]. Spatial hashing has also been mapped onto graphics hardware [Gre05b]. While such data structures are more memory-efficient than our octree solution, none of them can be generated by graphics hardware – a restriction which also applies to random access trees [LH07]. Further, the data structure creation of the mentioned spatial hashing requires an optimization step for data compaction, many times more time-consuming than our approach.

Our octree generation algorithm uses reduction in 3D space (cell coverage: 2x2x2, or 2^3), not one-dimensional reduction used in other data compaction approaches [Hor05, Sen06]. Further, we only start as many traversal threads as there are *output* elements, not input elements. Our algorithms map closer to texture caching mechanisms by using the 3D texture mipmap hierarchy in one implementation, and a flattened 2D slice representation of the 3D tiling in the other, as internal texture representation is optimized for 2D locality. While reduction in the upper pyramid levels can be serialized if the number of shader processors is known [Sen06], our approach omits this assumption to keep the algorithm simple and general for future GPU generations. For gathering the relevant input elements during output generation, binary search can be applied [Hor05]. Another solution is to use the scattering functionality of CUDA to create output elements, after all target output indices for elements in the input array have been generated via a prefix sum-scan [Har07]. This is seemingly more trivial, but (a) not available on more basic graphics hardware, and (b) requires considerably more bandwidth during its preceding output index generation phase, requiring to write at least once to all input cells, even irrelevant ones. We apply an 8-split traversal, which is gather-only, texture cache-friendly, and reduces traversal complexity from $\log_2(N)$ to $\log_8(N)$.

The generation and retrieval of leaves in OcPyramids is closely related to QuadPyramid buildup and traversal [ZDTS07]. While OcPyramids in many ways resemble QuadPyramids, neither a real-world performance measurement of an implementation, nor a way to extract node lists mirroring quadtree/octree subdivision has been presented before.

Octrees, being general spatial data structures, have been used in numerous applications. Film industry has experimented with octrees to resolve the well-known parameterization issues of 2D textures for 3D mesh surfaces [BD02]. In related work, an octree containing local 2D parameterizations was introduced [DL07]. It avoided many parameterization issues by decomposing any 3D mesh into locally planar surfaces. Especially in the 80ies, octrees were used to efficiently generate isometric 3D views of volumes, since hardware still was computationally weak [Oli86, LH91]. Before 1995, octrees were used in CPU-based raytracing and raycasting algorithms [Kno06, Sam89, Sun91]. Later, they grew out of fashion, mainly due to reduced traversal efficiency in comparison to kd-trees [Kno06]. But while kd-tree creation is no problem on a serial processor, such as the CPU, the situation is different on graphics hardware: There, kd-tree balancing creates data evaluation bottlenecks in parallelization. Since dynamic scenes constantly require a re-build of their acceleration structures, kd-tree build-up algorithms have recently come into focus for speed optimization [Pop07, ZHW*08]. Region octree creation, on the other hand, does not perform tree-balancing and can thus be data-parallelized much more easily.

GPU volume raycasting is dependent on an efficient ray-octree volume intersection. In ray-octree intersections, the ray's parameterization can be exploited to quickly find all hit octree cells [RUL02]. A GPU-based octree leaf and node list space query can similarly be accelerated with this technique.

PDE and ODE solvers often adapt resolution locally to reduce the sheer amounts of calculation [LGR04]. On the CPU, this is usually handled by sophisticated tree data structures, tesselating and

merging on the fly. The GPU, in contrary, did traditionally not perform well in dynamic list administration, but via an exhaustive data representation, memory bandwidth and idle ALU time can often be traded in for speed gains. In this particular case, the mipmap hierarchy of OcPyramids can help determine the local ODE/PDE solver resolution.

9.1.1. Tutorial Application

To introduce the algorithm's inner working, we start with a tutorial application. In this tutorial, we will use a classifier to detect yellow colored voxels in a 4x4x4 volume, and then use our algorithms to create clustered octree regions of yellow voxels, forming an OcPyramid. Then, we let another two algorithms traverse the OcPyramid to extract an octree leaf list, and a node list which links the leaves together. Focus is thus put on the underlying octree creation and traversal process. More complex application cases are of course possible, and will be sketched later.

9.2. Overview

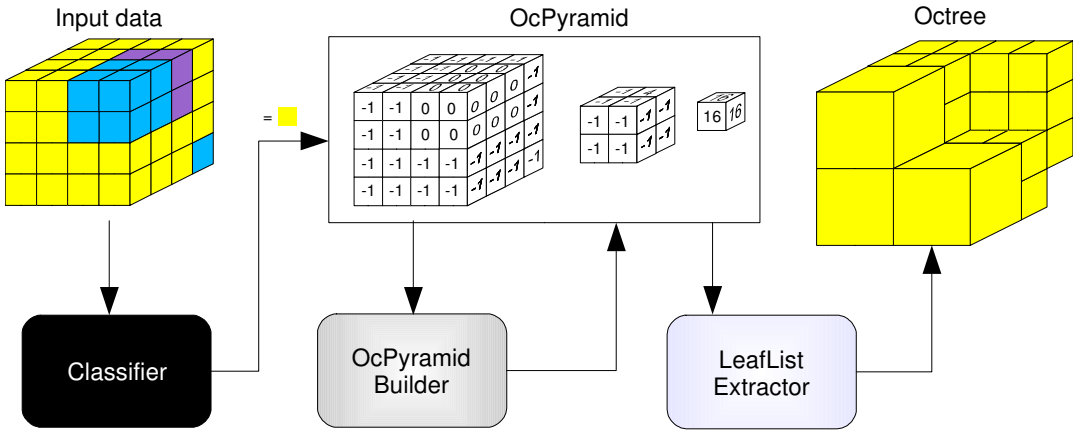


Figure 9.1: Overview over octree creation via OcPyramids.

Figure 9.1 illustrates the workflow between the different computation steps. All data is being processed on the GPU – the CPU only handles data if the final octree shall be downloaded for further processing in a non-GPU application.

We assume that the input volume data is stored in GPU memory. This data may be of arbitrary type and modality. The Classifier stage´s task is to determine if the cell content is clusterable, i.e., become a small octree leaf that can be clustered further. The Classifier outputs its decisions into a volume of integer values.

This volumetric output from the Classifier is used as the base of a 3D mipmap-like data structure, the OcPyramid. During pyramid buildup, the reduction operator clusters and counts octree leaves, successively reducing the volume size until a single cell remains.

The OcPyramid thus contains clustering information already, and can be used in various applications, acting as a pointerless region octree [Kno06]. But the pyramid has the same resolution as the input volume, and is thus wasteful in memory consumption. More compact representations are required. We therefore introduce two data structures to represent an octree in graphics memory: *leaf lists* and *node lists*.

A *leaf list* contains all clustered regions in the input volume. It is closely related to the quadtree leaf list, see Chapter 8. Beyond position and size, an octree leaf can contain auxiliary data such as color, gradients, etc.

The *node list* describes the subdivision of the octree. It answers space queries, i.e. it can locate an octree leaf that covers a given 3D position in the input volume. Nodes can refer into leaf lists.

Comparing with common terminology from octree literature, leaf and node list together comprise a pointer octree. There, *parent nodes* are mentioned – these correspond to the term *node* which is used here. Pure children nodes correspond to our notion of an octree *leaf*.

While leaf and node lists are no novelty in themselves, their creation on graphics hardware strongly differs from conventional CPU methods. This is because dynamic list generation is not possible in stream processing, as already mentioned in the introduction of Chapter 6.

Therefore, besides clustering, the OcPyramid´s *other* purpose is to provide for output allocation, and act as implicit indexing structure during OcPyramid traversal, which creates the output. For each list entry to be generated, be it octree node or leaf, the OcPyramid is traversed in a top-down fashion by individual GPU threads. On the way down the pyramid, the aftersought index is compared to implicitly generated index ranges, until the desired node or leaf has been located.

We now describe the components in detail.

The Classifier decides if a cell can pose as octree leaf (-1) or not (0). The cell content may be arbitrary, provided the classifier is able to determine if cells are clusterable. Section 9.3 explains the details.

After buildup of the Leaf OcPyramid, the GPU can extract a list of octree leaves, or short: a leaf list. Every list entry is the result of a hierarchical traversal through the OcPyramid. Section 9.4 describes Leaf OcPyramid buildup first, then the subsequent leaf list creation and the involved OcPyramid traversals.

Node list creation works similarly, but takes a two-channelled OcPyramid as input instead. From this Node/Leaf OcPyramid, a node list can be created, representing the octree subdivision. Section 9.5 describes how this two-channeled OcPyramid is constructed. It also explains the more advanced traversal that extracts node lists from the OcPyramid.

Parallelization and the limitations of graphics hardware programming have an impact on algorithm design. Section 9.6.1 holds comments on algorithmic complexity and other relevant issues.

After construction, node and leaf lists can be used for octree space queries. Section 9.7.1. explains how point queries and ray tracing can utilize a present node list. In Section 9.8, we describe algorithmic variations to better fit application needs. Section 9.10 lists several applications that benefit from rapid octree creation on graphics hardware.

Section 9.9 summarizes the current performance results that we obtained using state-of-the-art graphics hardware. It describes the surrounding test setup, and analyzes runtime behaviour.

9.3. Classifier

The purpose of octree creation is to cluster cells into larger regions, so called octree leaves. It is this clusterability property that the Classifier has to decide. This is a binary decision, resulting in a simple value: each cell is either clusterable (-1) or not (0). But the input data is usually of a more complex data modality.

Therefore, the first step is to preprocess the input volume into this simple representation. In general, any operator that maps voxel data into such a binary decision can be utilized here, but a more extensive survey of applicable classifiers (there: “discriminators”) is available [ZTTS06].

In Figure 9.2, we regard color as an example criterion for clusterability. Yellow cells are regarded as clusterable, which will later result in an octree of all yellow voxels.

In the following, we visualize the 3D content as 2D slices, with z values inscribed on the side. The classifier converts the input volume into a 3D integer array. All yellow cells are regarded as clusterable, and thus marked with a **-1**.

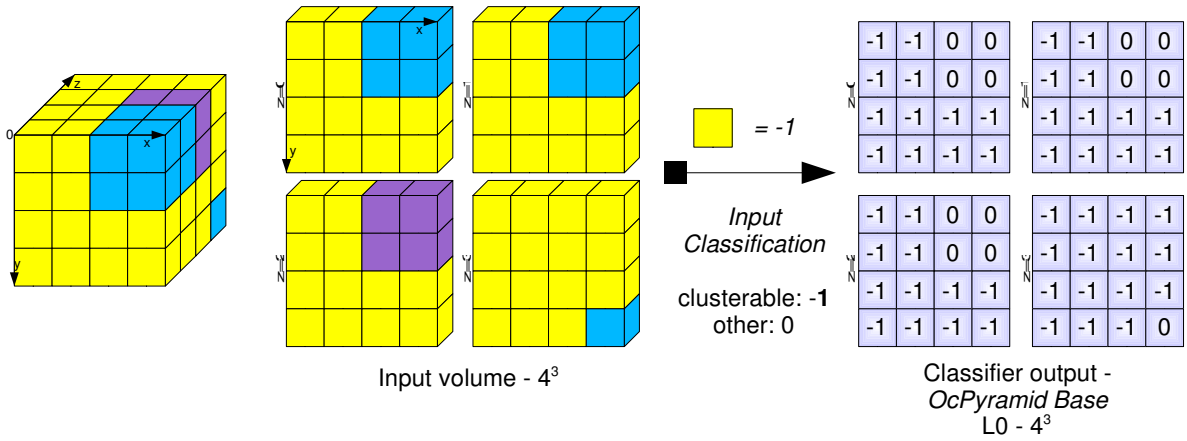


Figure 9.2: Classifier choosing clusterable cells (here: cells of yellow color).
(Volume shown as Flat3D layout, z values inscribed on the left side of slices.)

The classifier´s output now form the base for the internal OcPyramid data structures, which allow graphics hardware to extract octree leaves and nodes.

9.4. Leaf Lists

9.4.1. Leaf OcPyramid Buildup

When clusterability of all voxels are known, the actual clustering can commence. This will be a bottom-up process, similar to 3D mipmapping. The Leaf OcPyramid, short for octree leaf pyramid, is a pyramid of 3D volumes. Each volume is called a pyramid level, and filled with integer values. The pyramid's base level is provided by the Classifier's output volume.

Just like in 3D mipmapping, a reduction operator is applied to build the remaining levels of the OcPyramid. This is a reduction process in 3D – therefore, 2x2x2 = 8 cells are clustered into one cell for every reduction pass. In Figure 9.3, which shows a cut-up 2D atlas of the 3D volume, this is reflected by accessing 2x2 cells in two z-adjacent 2D slices.

During reduction, two rules can apply: clustering (output value **-1**, marked in green), or, the sum of all absolute cell values is generated (leaf allocation, marked in red). Successive pyramid levels are repeatedly reduced, until only one output cell remains.

While clustering itself is intuitive, the leaf allocation that happens at the same time might seem strange at first. But in essence, it resembles the principle already known from HistoPyramid-based data compaction: Individual leaf indices are allocated with a -1 (interpreted as 1), when clustering cannot continue, and previous allocations are promoted upwards, via the summing operation in the reduction.

Due to this upwards promotion of leaf allocations, the top level´s cell value will contain the total number of octree leaves. This value is used to allocate the leaf list, as shown to the right of Figure 9.3.

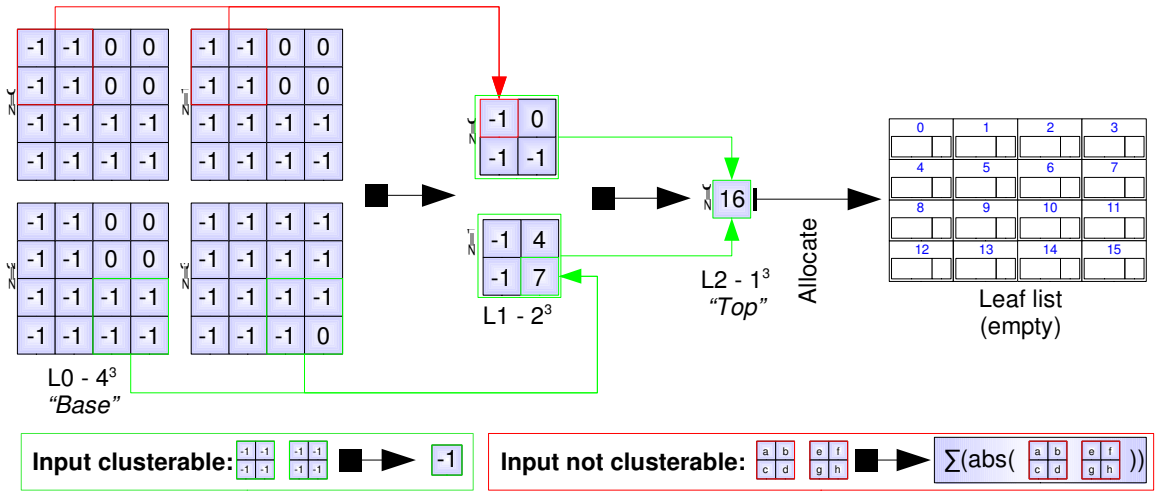


Figure 9.3: Leaf OcPyramid buildup.
2x2 cells of two z-adjacent slices are repeatedly reduced, via two reduction rules (top): Clustering (green) and, if no clustering is possible: Leaf allocation and promotion (red).

9.4.2. Leaf List Creation

When the OcPyramid has been built and the leaf list has been allocated, leaf list creation can commence. The goal is to fill the list with all octree leaves that the clustering process has created.

We fill the list by executing an OpenGL shader on it. Such shaders execute in parallel, creating one thread for each output element. In our example, one thread will be started for each of the sixteen possible list entries, see also Figure 9.4.

Each thread first determines their assigned *leaf index*, the index of the leaf which the thread shall locate. This index is deducted from the array coordinate of execution in the list. In Figure 9.4, the array coordinate in the output is (x,y)=[0,3], with `list_height=4` and `list_width=4`. The leaf index is thus `list_width*y + x=3*4 + 0=` **`12`**.

Now the thread can start traversing the OcPyramid in search of the leaf. It enters at the top level, reading the cell value there. As described before, a non-leaf cell (> 0) contains all leaf allocations that were promoted to it from the level below. Cell values are therefore re-interpreted as *index ranges*, just as in the HistoPyramid traversal of section 6.6. These index ranges cover all leaf indices that children of a given cell in lower levels have allocated.

Example: the top level's single cell covers all leaf cell indices, and is thus used for leaf list allocation. In the traversal example of Figure 9.4, the value 16 thus becomes the index range [0, 15].

In every iteration, partial index ranges will be produced for each of the eight children. Leaf markers (-1) are treated as +1 in index range calculation, using a simple `abs()` operation. In Figure 9.4, the ranges [0,1[, [1,2[, [1,1[, [2,3[, [3,4[, [4,8[, [8,9[and [9,15[are produced from the cell values.

The algorithm descends into a child if the key index lies within this child´s partial index range. In our example, index 12 lies in the index range [9,15[, and traversal descends there.

Descend progresses until a leaf has been found. This is the case when the leaf index matches the partial index range of a cell with a leaf marker. Obviously, this has to happen at the base level at latest. This is actually the case in Figure 9.4, where leaf index 12 finds its matching leaf at the base level. However, leaf markers can also happen to match an index at higher levels – in that case, traversal ends there.

After traversal has ended, the coordinates and size of the octree leaf are written to the leaf list. The leaf size is derived from the current level. In Figure 9.4, the final position is `x=3, y=3, z=2`, or `[x,y,z]=[3,3,2]`. Since the leaf marker was encountered at the base level (level 0), its size is thus 2^0=1 in all dimensions: 1^3.

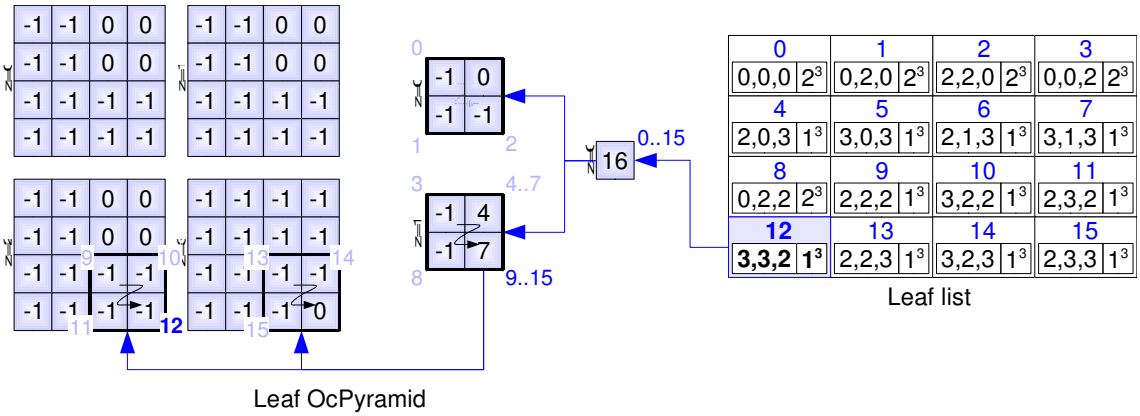


Figure 9.4: Leaf OcPyramid traversal example for an octree leaf, index 12.
Black frames mark the examined 2x2x2 cells at each OcPyramid level.

Note that the leaf list is unsorted, and leaf order is irrelevant. The traversal used in our Figures employ depth-first-search output and a 3D adaption of the Z-order or Morton-order, as described in [Mor66]. Other traversal orders can be introduced, as long as they are consistent for all shader threads.

The final result is thus an octree leaf list, stored in a 2D array. A minimal list entry contains the 3D coordinate and region size for one octree leaf. More leaf related data can be stored at the same position, e.g. a common feature shared by all the covered voxels, such as a color average.

9.5. Node Lists

In many applications, the subdivision structure of the octree itself is of interest. How is the octree sub-divided to host all its leaves? The leaf list, being an unsorted list of independent entries, cannot provide this information.

Naturally, the OcPyramid itself already tracks subdivision: A traversal from top down easily shows where subdivision ends due to first encountered leaf markers. But it is extremely wasteful to keep the space-consuming OcPyramid in memory just for that purpose. Actually, especially at high volume resolutions, it would not be possible to keep a full OcPyramid in memory at all.

Therefore, a more compact representation is desired. A pointer octree fits this purpose: It contains a tree of linked nodes, describing how subdivision into children takes place, from the undivided cube, the root, down to the individual octree leaves.

To create such a linked data structure on graphics hardware, we introduce two concepts: *in-pyramid-allocation* and *co-traversal*. We apply them in creation and traversal of a new, supplemental data channel in the OcPyramid.

9.5.1. Node/Leaf OcPyramid Buildup

The allocation of a pointer octree requires knowledge about its number of nodes. We employ a pyramid concept to determine how many nodes the pointer octree is comprised of, just like we did for Leaf OcPyramids.

Therefore, a pyramid of integer values is again built on top of the base level, the Classifier's output. This time, however, the generated pyramid contains *two* channels, as a supplemental data channel has been added, which we call the *node channel*. Figure 9.5 demonstrates this extended buildup process: the leaf channel (in blue) is created as described in the Leaf OcPyramid buildup of section 9.4.1, while the new node channel (in red) will be used to allocate entries in the node list. The node channel thus takes care of node allocation, *inside the pyramid*.

9.5.1.1. In-Pyramid Allocation

Nodes are allocated whenever input cells cannot be clustered. This can happen throughout the pyramid, and is signalled via the output value **1**, see Figure 9.5. As node allocations can happen further up in the hierarchy and in order to count all allocated nodes, node allocations need to be promoted upwards. To achieve this, the node allocations of the eight input cells are added in. But since no clustering can be performed, a new node must *also* be allocated at the same position,. Therefore, the number of previously allocated nodes is added up, and increased by one. This is reflected in the red rule at the bottom right of Figure 9.5.

Why is this a novelty? In the original HistoPyramid principle, the number of allocated elements is always present at the base level. Even for a Leaf OcPyramid, leaves were introduced only at the base level. But here, output entries, in our case nodes, are created at *arbitrary levels* of the OcPyramid.

Again, reduction is repeated until the final level consists of only one cell.

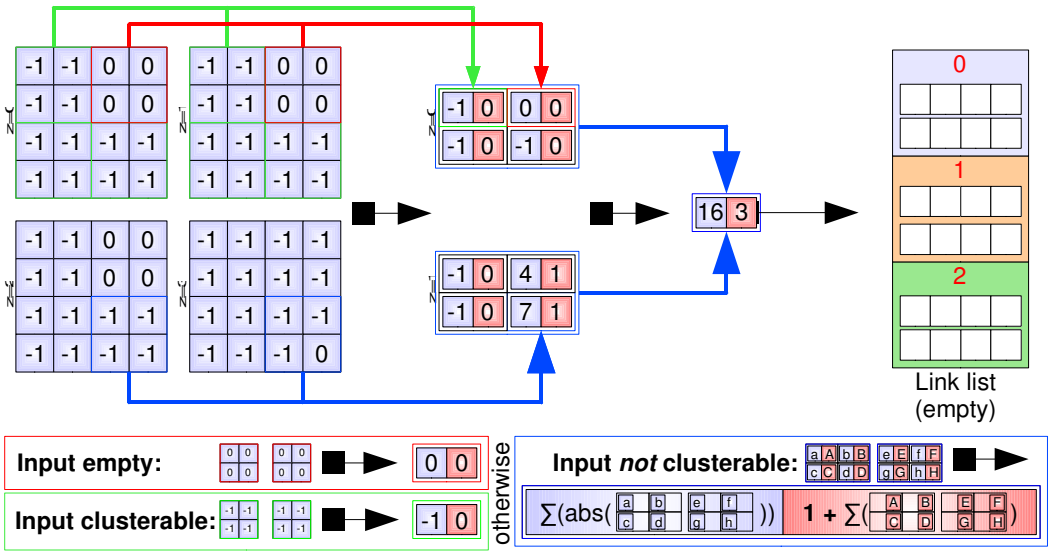


Figure 9.5: Node/Leaf OcPyramid buildup, followed by node list allocation.
Reduction rules at diagram top and bottom.
Blue: Leaf channel. Red: Node channel.

We call this two-channelled data structure a Node/Leaf-OcPyramid.

Due to this upwards promotion, the node channel of the top level cell value contains the total number of octree leaves. This value is used to allocate the node list, see the right of Figure 9.5.

9.5.2. Node List Creation

Now, the node list is filled by running one shader thread for each list entry. Traversal is very similar to the one used in leaf list creation, compare section 9.4.2 - Again, traversal starts at the

OcPyramid's top level, and progresses downwards. Partial index ranges decide over which child to descend into. But, as already mentioned, nodes have been created *within* the pyramid: at every OcPyramid position where clustering would not occur, the value **1** has been added, to signal that a node had to be allocated. Traversal needs to respect this while searching for a given node index.

We eliminate this level's allocated node from the index range by adding **1** to the index range start, right before every descent. In other words, every descent *consumes* the first index of the range. As an example, we take Node 0: It is always the root node, situated at the top level, and traversal reflects this by consuming it immediately, that is, after descent below top level, no other node can receive index 0. Therefore, for all traversals below node 0, the index range start is modified to [1, ..[after the first descent from the top level. The modified node index ranges are shown above the solid and dashed red arrows in Figures 9.6 and 9.7.

In contrast to leaf list traversal, the exit criterion differs: Every time the node index matches the index range start after a descent, traversal ends. Since there are no nodes at the base level, it is thus the leaf list´s early exit criterion that is the *standard* exit for node traversal, compare section 9.4.2. In Figure 9.7, traversal for node 1 ends at level 1. The exit criterion is even applied before the root node is descended into, since the root node *can also be a leaf* (compare Figure 8.4 of the previous Chapter). This is the case for node 0 in Figure 9.6.

9.5.2.1. Co-traversal / Path Sharing for Node Children

When a node has been found, we need the indices of the node´s children to create links in the node list. Certainly, as the node's position is known, the *position* of its children in the pyramid is obvious, too. But since the node list shall be independent of the pyramid itself, the children's actual *indices in the node and leaf list* are required.

Incidentially, the path leading to a node´s children, must pass through the node itself. As a first example, Figure 9.6 shows the traversal path for node indices 1 and 2, both children of node 0.

Hence, we can *re-use index ranges* that were used for finding the node *itself*, and calculate the node childrens' indices from there, descending one level further down the pyramid. In Figures 9.6 and 9.7, dashed red arrows and two dashed frames indicate a node's children. The solid red arrows indicate that up to the node, children traversal had shared path with traversal to the node itself.

A node may also have octree leaves as children. An octree leaf's presence is indicated by value 0 in the node channel, and value -**1** *in the leaf channel*. Node children indices could be assigned the leaf marker value **-1** to signal an octree leaf. But aside from position and size, all other crucial information that this leaf might hold would get lost. An octree leaf might contain values common to the clustered region, and this information would only be present in the leaf list, without accessability from the node list. Therefore, a node that refers to children's leaf indices would be valuable.

Again, for octree leaves, we exploit that traversal for the node's children passes through the node itself. The new observation is that traversal path sharing even holds for octree *leaves*, with the only difference that the leaf index is determined by the *leaf channel*, not the node channel. Otherwise, the path is the same. Figure 9.7 exemplifies this for the children of node 1, the leafs 4, 5, 6 and 7 (blue).

We thus employ *co-traversal*: while tracking the node index range, we *also* track the leaf index range. This happens by additionally evaluating the leaf channel *while traversing the node channel, producing a supplemental leaf index range.* This leaf index range determines the leaf indices for a given node's leaf children. Figure 9.6 shows this calculation for node 0. The leaf indices 0, 1, 2, 3

and 8, shown in blue, are calculated while traversal progresses, even though the descend is based on node indices.

The last type of child is an empty region. It is indicated by the value 0 in both node and leaf channel. In the Figures, it is marked with a blue X.

9.5.2.2. Summary of Node Output

When traversal has retrieved all eight children, and determined their type via the mentioned deduction, the node entry can be written.

In summary, three node types can be produced: leaf indices (blue), node indices (red), or empty regions (transparent). Compare Figure 9.6 for node 0 as an example, which has 2 node children, 5 leaf children, and an empty child.

After the last node entry has been written and the leaf list is available, node 0 will link from the octree root node down to all octree leaves.

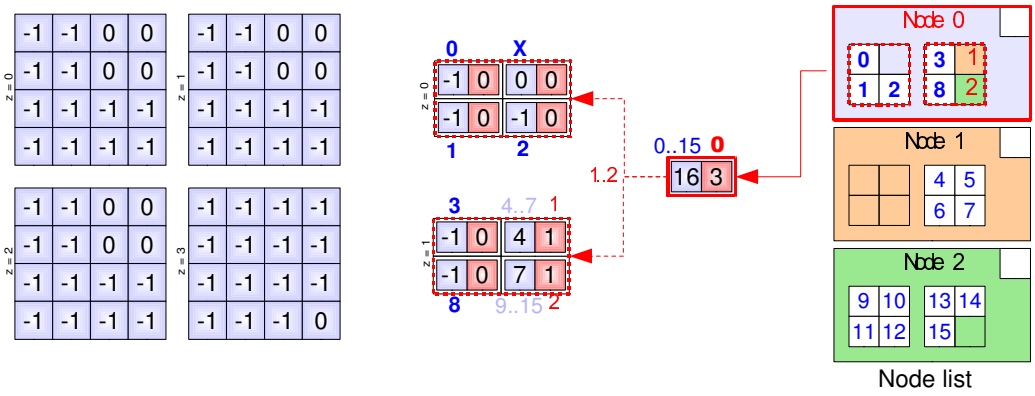


Figure 9.6: Node list creation, node 0.
Red frame: Node position. Dotted red frame: Children positions.
Red: Node indices. Blue: Leaf indices. X/Blank=NULL/empty region.

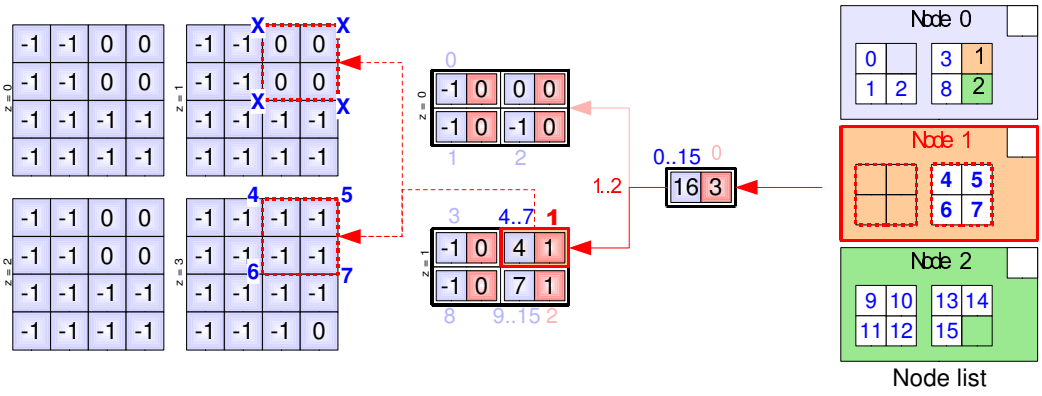


Figure 9.7: Node list creation, node 1.
Black frame: Investigated positions. Red frame: Node position. Dotted red frame: Children positions.
Red: Node indices. Blue: Leaf indices. X/Blank=NULL/empty region.

9.6. Implementation Remarks

In our implementation, the GPU represents leaf and node lists as 2D OpenGL textures, which can cause empty output cells due to the 2D packing. ShaderModel 4.0 implementations may choose 1D buffer textures to avoid this.

Node entries may refer to other nodes or leaves. Output channels are limited in OpenGL shader output, which is why we use the *floating point sign* to discern between node and leaf references in a node's list of children. Node indices are represented as positive, leaf indices as negative values. In the Figures, colors were used to ease understanding (red: nodes, blue: leaves).

Absence of clusterable content in an octree region is signalled with NULL pointers, indicated as empty squares in Figure 9.6. Exploiting the fact that no octree node will point to the octree root (node index 0), we encode NULL pointers as a positive floating point zero (`+0.0`).

9.6.1. Algorithmic Complexity

OcPyramid traversal for leaf list extraction may terminate early, and thus does not have a predictable number of dependent lookups for every list entry – this contrasts with HistoPyramid traversal, which has predictable traversal depth [ZTTS06]. But as an upper limit, with N being the number of input cells, no more than 8 $\log_8(N)$ texture accesses are needed while generating one octree leaf in the list. If M list entries shall be created and K processors are available, the total complexity would thus be 8 M/K $\log_8(N)$ or lower, depending on input clustering.

For node list extraction, similar assumptions hold. However, for every node to be created, its children have to be retrieved, which adds 8 more texture accesses for every entry. Incidentially, nodes never exist at the base level, which is why the maximum bound of 8 $\log_8(N)$ holds here, too.

OcPyramid buildup itself is a classic 3D mipmapping process. Classifier output first creates the base level, with N elements. Then, a 3D mipmap is built on top of it. It requires approximately $1/8 + 1/(8 \cdot 8)$ cells. Thus, the overall buildup is bounded by 1/16. Each new channel entry requires 8 texture accesses in itself. Buildup complexity is thus bounded by 17/16 * 8 * N ~ 8.9 * N texture accesses for Leaf OcPyramids, slightly more for Node/Leaf OcPyramids.

Overall, list creation seems most costly in terms of complexity, as a $\log_8(N)$ factor is involved. But in the application, M is often considerably smaller than N, with caching alleviating the redundant texture accesses of the concurrent OcPyramid traversals. List creation is thus a lot less relevant in GPU execution time than the OcPyramid buildup process: In our test application *Achterl*, runtimes are dominated by OcPyramid buildup, while list extraction only uses 10% or less of the overall time.

Therefore, it is important to optimize the OcPyramid buildup process first. This can be achieved by keeping the relevant shaders for classification and buildup simple, and, as already described in Node/Leaf OcPyramid buildup, by fusing node and leaf channel buildup into one common shader. But in general, memory bandwidth determines OcPyramid buildup speeds. Therefore, omitting the base level via a *base level up-shift*, described in section 9.8.3, can help to reduce buildup times.

9.7. Octree Queries with Node/Leaf Lists

Since the OcPyramid children's serialization is known from node list creation, we can track space subdivision during node list traversal. Thus, we know our current position and subdivision size as soon as a new node or leaf is accessed in node list lookups.

At the same time, it is possible to find all node list entries that cover a certain spatial position, making the node list pivotal in Octree space and ray queries. Figure 9.8 shows how a node list describe the octree subdivision from the root down (in Font serif, varying colors to show octree cubes that will be divided further), and how it references into the leaf list (shown in sans-serif font, yellow).

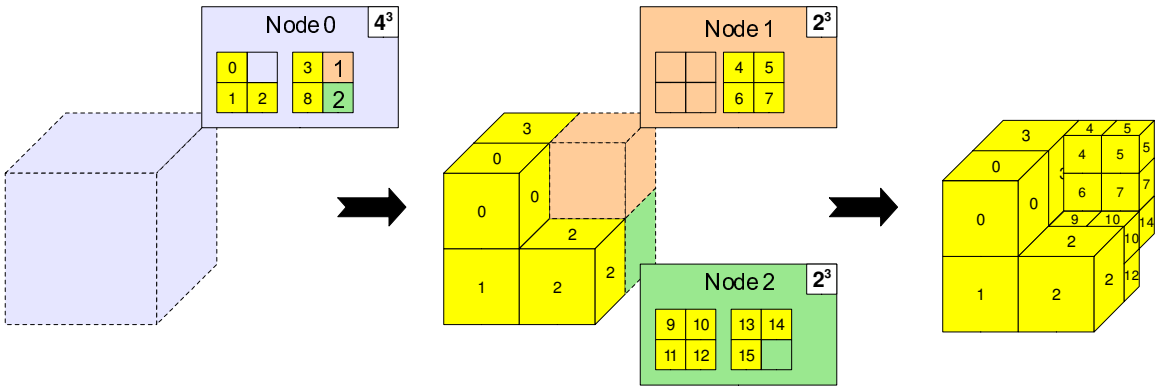


Figure 9.8: Node list traversal. Node 0 is always the octree root node.
From there, references to octree leaves (yellow)
or other nodes (non-yellow) can be followed.
On the way, subdivision increases.

9.7.1. Point Query

One basic application of octrees is leaf lookup for a given 3D volume position. Figure 9.9 shows how this is accomplished: First, the point coordinates are mapped into the octree's domain. Then, the root node is accessed and its partitioning in terms of coordinate interval calculated. The point will then fall into one of the eight subdivisions of this node. From there, traversal either continues to another node and thus descends into a childs, or it ends because empty space or a leaf has been encountered. In our example, a point with `x=3.9, y=2.7, z=3.5` falls into coordinate intervals `z=[2,4[, y=[2,4[, x=[2,4[` of Node 0, which leads on to Node 2 of the node list. Here, the cube with intervals `x=[3,4[, y=[2,3[, z=[3,4[` matches the point coordinates, and thus let traversal terminate at Leaf 14, which is a cube of size 1 with origin `x=3, y=2, z=3`.

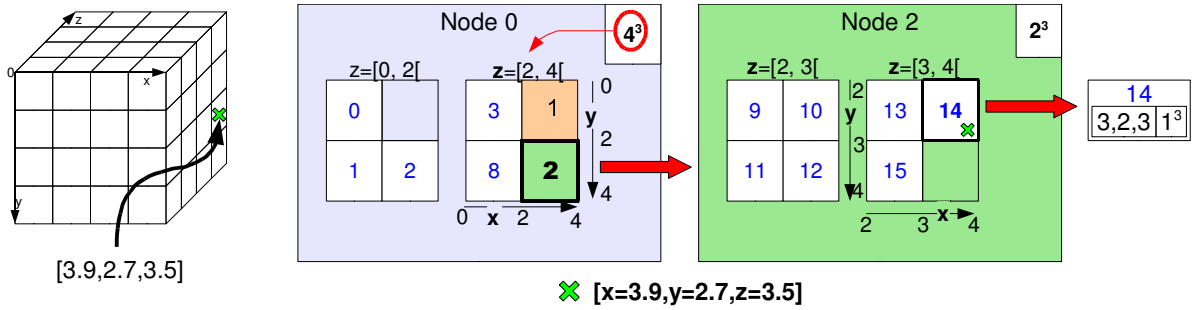


Figure 9.9: Point location (example subdivision scheme from Figure 9.8).

9.7.2. Ray Intersection

Octree data structures have traditionally been considered as too inefficient for ray tracing applications. For every ray query, the octree has to be traversed from the root down. But better suggestions for computationally efficient ray traversals through an octree volume are available

[RUL02]. These can basically also be employed in GPU-based ray traversals, but recursive implementations are not very efficient on graphics hardware. Instead, we propose to integrate ray intersection with octree traversal.

Given the ray equation and the octree position and size in world coordinates, intersection points with the octree's root node can easily be computed. From there, a sorted list of children intersections is produced, starting at the ray entry point. Note that due to the simple subdivision scheme, this sorted intersection list can even be produced by a hard-coded intersection algorithm. The resulting list of intersection candidates is now investigated one by one. Empty space can be skipped immediately. If a child is a leaf node and only the first hit is aftersought, ray intersection can exit directly. For every intersection with a node, further intersections, descends into the octree, are necessary.

The advantage to classic algorithms is here that octree descend only happens on demand. If the first ray intersection with any octree leaf is required, then octree descend will only have the complexity of one point query.

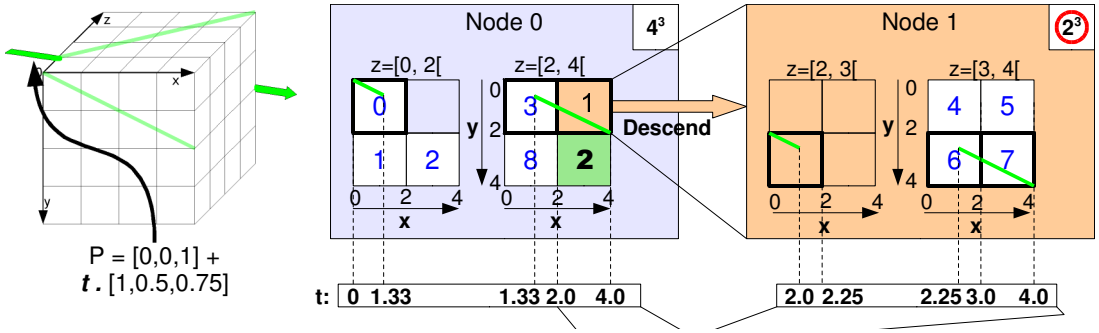


Figure 9.10: Tracing rays through the octree volume (example subdivision scheme from Figure 9.8).

In summary, from evaluating ray passage through the root node, more detailed intersections are found, but only on demand. In our example in Figure 9.10, we see how a ray, defined by P=[0,0,1]+t.[1, 0.5, 0.75] first encounters leaf 0 and 1, before the ray continues into node 1. There, empty space is spanned, before leaves 6 and 7 are hit. From there, the ray exits the octree structure. Thus, from evaluating the root node 0, observations can be made on which regions of the octree would be hit, even without investigating node 1 in detail.

9.8. Algorithmic Variants

9.8.1. Octree Capping

If space queries in the octree are too costly, a simple trade-off of traversal depth for memory is possible: Traversal depth can easily be reduced by *capping the top* of the OcPyramid, replacing its root node with independent octree entry points. This works because it is very improbable that large space regions are clustered. Even if such large regions can be clustered, the complexity of such octrees would be low, and the performance hit for not clustering close to the octree root thus negligible.

9.8.2. Node OcPyramids

If leaves have no additional information, e.g. while clustering empty space, then a pure pointer octree based on a node list can suffice. In that case, the leaf channel can be omitted. But since node

allocation depends on leaf markers, and clustering is usually part of the leaf channel´s reduction process, it must be done in the node channel's reduction instead.

9.8.3. Base Level Upshift

In some cases, classification can be a rather trivial operation: for example, the isolation of the color yellow is very simple to compute. Such operations are often faster to re-compute than to store their outcome in memory. If re-classification thus is considerably cheaper than writing the classification outcome, then the base-level can be omitted. In this case, classification is integrated into the first stage of OcPyramid *buildup*.

In our tutorial application *Achterl*, alpha value thresholding is used to separate empty space from solid material. This is such a local and inexpensive operation, and we therefore omitted explicit Classifier output and a base-level. This reduces both memory usage and bandwidth, and thus accelerates computation.

Note that leaf and node extraction in OcPyramid traversal have to *re-do* Classifier operations *every time* the non-existing base level would be accessed. Therefore, base-level upshift should only be used if Classifier's calculations are negligible in comparison to writing their outcome.

9.8.4. Feature Clustering

In some applications, clusterability is based on feature relations, not on a particular feature itself. If e.g. alike features shall be clustered instead, then a more memory-intensive variant of the algorithm, *feature clustering*, can be applied. This works by storing features in the OcPyramid during build-up, and comparing them during clustering. Such features could be colors or, as in our application *Achterl*, counters that tell empty space from solid interior. Features of already clustered regions will be propagated up the OcPyramid to assist further clustering decisions. For more details, please refer to section 8.6.1 of the previous Chapter on quadtree creation for the corresponding 2D algorithm. The principle is very similar, expect for the added dimension.

Note that in feature clustering, the reduction´s clustering operation must include two new calculations: feature comparison, and, if clustering shall commence, feature averaging. This increases shader complexity and may thus lead to slowdowns in OcPyramid buildup. Note also that feature clustering drastically increases the OcPyramid's memory consumption.

9.8.5. Tiling/Partial Processing

Due to GPU memory restrictions, volumes exceeding sizes of 256^3 are currently impossible to process at once. But OcPyramid creation is a bottom up and totally parallel process, and thus separable. We can thus create an OcPyramid for this large volume by partitioning *the input*, e.g. into portions of 256^3. For example, it would be possible to stream 2048^3 resolution volume data to the GPU in tiles, and have them compressed into one large node and leaf list, thereby creating a compressed representation that can be held in GPU memory. The final merger of the partial octrees, i.e. creating final leaf and node lists of the octree from the partial 256^3 "diced octrees", can either be conducted on the CPU, or in a subsequent pass on the GPU. But often, e.g. for applications in raytracing, this is not even necessary, see section 9.8.1 on octree capping.

9.8.6. Hi-Res Processing (Sub-OcPyramids, Hierarchical Processing)

Due to the simple structure of the node and leaf traversals in OcPyramids, it is possible to embed one OcPyramid into another with little modification. Take, for example, that certain regions of an input volume need to be sampled at a much higher resolution, e.g. 2048^{-1}, whereas most of the

volume can be sampled at merely 256^{-1}. To solve this, sub-OcPyramids are built from these hi-res regions, but not completely - only to a resolution level of 256^{-1}. There, the top levels of these sub-OcPyramids (hence, the number of elements that this hi-res region requires) are entered at the base level of the main OcPyramid, together with an appropiate pointer to where the sub-OcPyramid can be found. The main OcPyramid is now built as usual. During list extraction, traversal will repeatedly arrive at the base level of the main OcPyramid; there, the pointer will redirect traversal to the sub-OcPyramid to find the corresponding list elements.

Inversely, a sub-OcPyramid can also be created *on demand*, a method that mimics the "split-on-demand" process that many CPU-based top-down octree generators apply, e.g. for structuring triangle meshes. Assume that a triangle mesh shall be subdivided in such a way that each Octree cell shall at maximum hold one triangle (assuming the triangle mesh is not self-intersecting). Again, we create a volume of a certain sampling rate, e.g. 256^{-1} , acting as base level for the main OcPyramid. All voxels are initialized with value 0. During voxelization, each triangle rasterization into the volume increases the touched voxels by one. In effect, it marks that this triangle occupies this voxel. At the end of rasterization, all voxels are examined. If voxels have a value higher than 1, obviously they need to be subdivided further. Here, sub-OcPyramids are created – the GPU can create a list of all sub-Ocpyramids through 3D Data Compaction, as described in Chapter 6. After all sub-OcPyramids have been created, the final Octree can be produced.

9.8.7. Large Volume Extension

The OcPyramid data structure described here has a certain volume in space as its domain. From there, a region octree with the same domain is constructed. Sometimes, input data *outside* of the octree´s space domain shall be added to the octree. This does not require a re-build of the OcPyramid itself, neither is there a need to adapt subdivision to maintain a similar subsampling. Instead, new OcPyramids of similar structure can be set up besides the original cube domain, thus extending the octree coverage in any direction desired. Then, these new OcPyramids are built the same way. Finally, a new level is introduced on top of the OcPyramid tiles, until all of them can be reduced to one cell. This is the new octree root from which node and leaf lists can be extracted.

In effect, all the OcPyramids that tile input space have thus become *sub-OcPyramids*. Therefore, this problem is actually a dual of hi-res processing, see section 9.8.6.

9.9. Results

In order to test the real-time behaviour of our algorithm, we created a Linux OpenGL application called *Achterl*. It converts static or animated 3D models into a solid voxelization, and creates octrees for either the model interior or the empty space around them.

9.9.1. Voxelization

The 3D models, a teapot and an animated ball, are stored in a vertex buffer object, maximizing geometry throughput during voxelization. The software voxelizes the 3D mesh by rendering it into 256 slices of 256x256 each, spanning a volume of [-1,-1,-1] to [1,1,1] in world space. Since our octree analysis shall be exemplified on solid volumes, we use a solid voxelization approach for the 3D meshes, similar to shadow volumes [Cro77]. Figure 9.11 describes this process. First, we render the 3D model repeatedly into a volume texture, expanding the viewport volume step by step to describe the various z planes. During rasterization, forward facing triangles output +1, while backfacing triangles generate -1. With blending, output increases the more objects we “enter”, i.e. render their forward faces (blue), and decreases when they are left again, i.e. rendering their back faces (red). Now, values over zero indicates solid interior. We can therefore handle double forward

facing triangles, and avoid assuming prematurely that the solid interior has already been left. Figure 9.11, to the left, shows how XOR-based approaches such as [ED06] run into issues with such a problem setting. Note that a non-watertight mesh is a degenerate case that is not compatible with any approach in voxelization, as non-closed meshes don't properly define a volume.

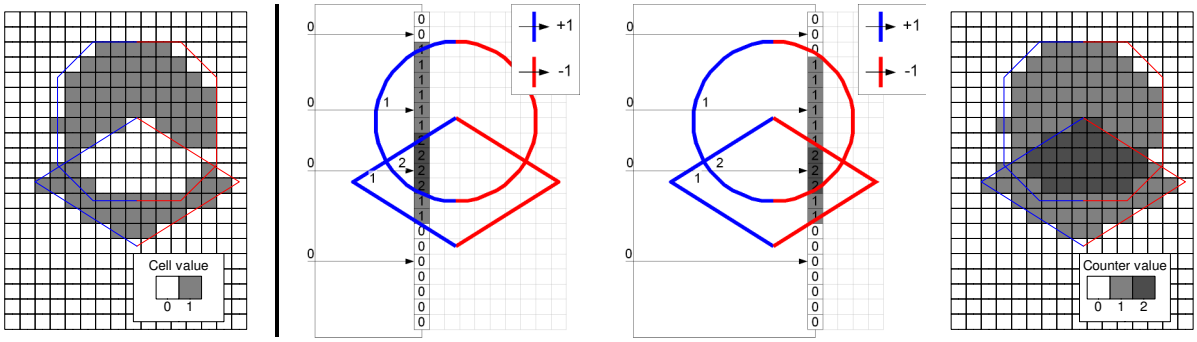


Figure 9.11: Solid voxelization of overlapping triangle geometry.
Left to right: Incorrect XOR-based voxelization.
Two example viewport ranges and their output.
Complete voxelization after 16 rasterization passes.

9.9.2. Solid vs. Empty Space Clustering

After voxelization, classification can commence. We examine two cases: clustering empty space and the solid interior itself. If empty space shall be converted into an octree, then -1 is output for all voxels that have values equal to 0. If an octree shall be created from the solid 3D model interior instead, we mark all values > 0 as clusterable instead. As a remark, values below 0 would indicate a mesh with inverse normals. This could easily be handled by the classifier, too.

Afterwards, OcPyramid buildup and clustering commences. Typically, leaf list extraction extracts hundreds to thousands of octree leaves. They can be visualized in various ways, e.g. as wireframe cubes, such as shown in Figure 9.13.

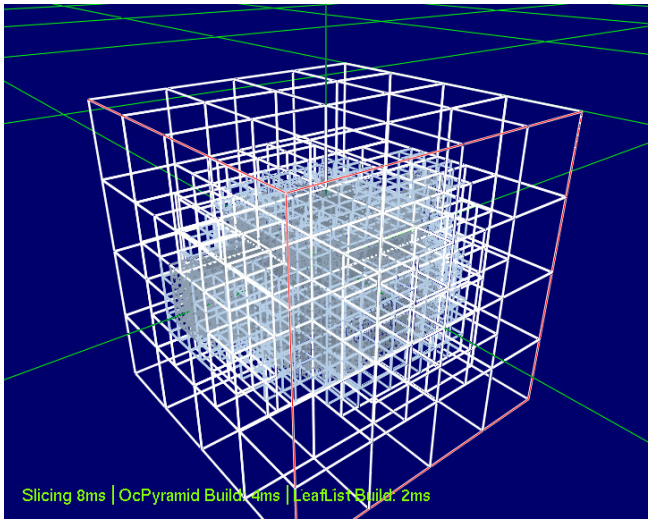
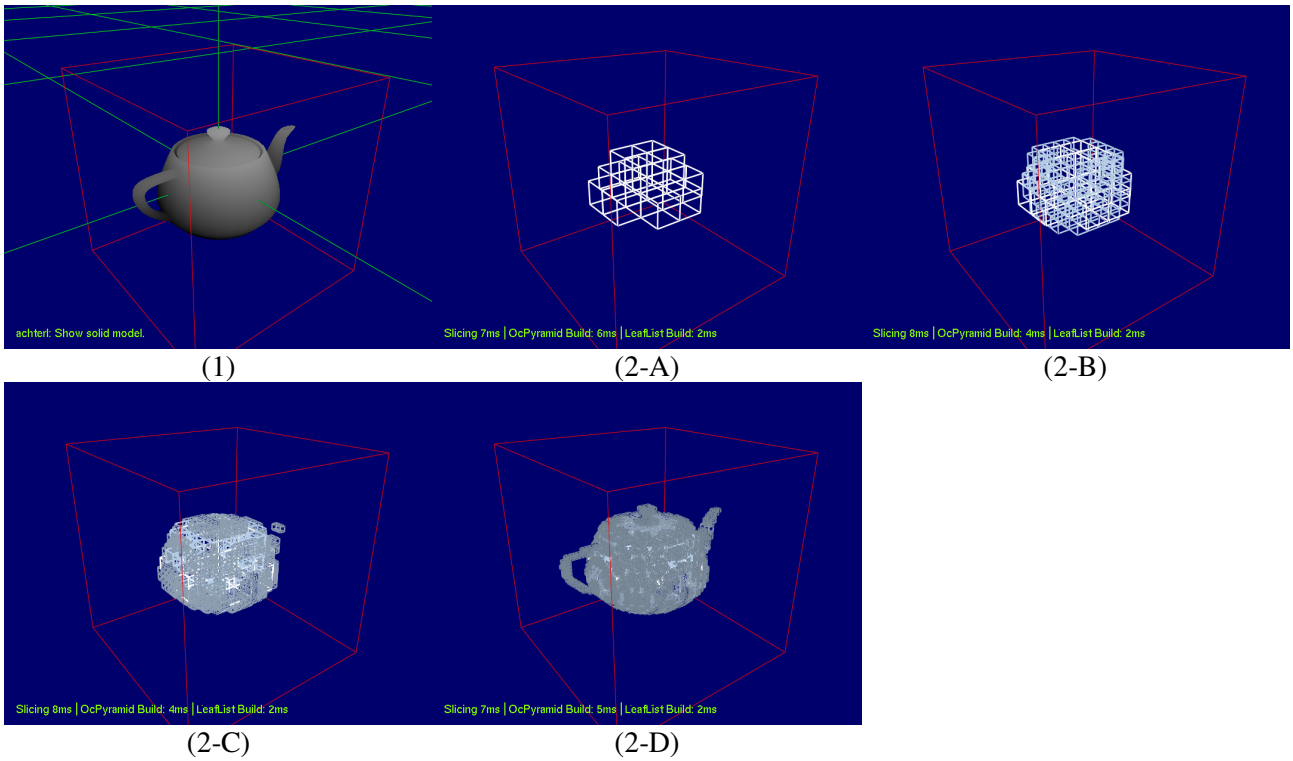


Figure 9.12: Octree decomposition of empty space surrounding a teapot model.
Gray levels mark varying granularity (octree leaf size).



(1) (2-A) (2-B)

(2-C) (2-D)

Figure 9.13: Octree decomposition of a teapot's solid interior.
(1) Input 3D mesh, before voxelization. (2-A) to (2-D): Octree leaves of increasing granularity.

9.9.3. Performance

The tests were conducted on a desktop and a notebook. The desktop machine was an AMD Athlon64 3500+, equipped with a GeForce GTX 260 with 1 GB video memory (GT200 architecture), running Linux 64 bit. The OpenGL version was 3.2.0, with GLSL 1.5, provided by the NVIDIA driver 195.13. The notebook machine was a Lenovo T61p with Nvidia Quadro FX 570M and 256 MB video memory (G80 architecture), running Linux 32 bit. The OpenGL version was 3.2.0, with GLSL 1.5, provided by the NVIDIA driver 195.14.

We compare two variants of the algorithm: `achterl` uses real 3D textures for both the voxelization and the ocpyramid, and is thus the most intuitive implementation. It uses the OpenGL texture format GL_TEXTURE_3D and according mipmaps and render-to-3Dtexture functionality via FBOs. Also, it uses 3D texture fetches, which have worse performance than 2D texture lookups, even if nearest interpolation has been chosen. Table 9.1 lists the timings for `achterl`.

`achterl_rect` is more intricate and uses `GL_TEXTURE_RECTANGLE`, a texture format without mipmaps and only two dimensional addressing modes. However, this format uses less memory inside graphics memory, and thus allows us to run tests with 256^3 volume resolution. Since this texture format is two-dimensional, we had to map the 3D volume and its mipmaps into a 2D domain. Therefore, the floating point texture consists of 16 x 16 tiles of 256^2 each to represent a 3D Volume of 256^3 and its mipmaps in a texture of 4096x4096 resolution. We generate the 3D volume mipmaps in one texture because node and leaf list traversal needs to have all mipmap levels of the OcPyramid available. Since we both read from and write to the same texture during mipmap generation, we violate the OpenGL specification in theory, but never encountered problems, since

we never read from locations that have been written in the very same pass. Table 9.3 lists the timings for `achterl_rect`.

Instead of measuring the time consumed by the OpenGL call delays on the CPU side, we measure actual GPU timings with the OpenGL extension `GL_EXT_timer_query` [Gre05b]. Due to unstable OpenGL driver support in Linux, we could not test the algorithm on AMD/ATI graphics hardware.

As input, we use two 3D meshes as input models, and analyze the empty space around them. The bouncing ball animation has 264 triangles, and the teapot model uses 3136 triangles. We measure octree analysis for the ball animation, while the ball's center is at [x,y,z] = [0.19, 0.114, 0.884].

The 128^3 octree analysis of the empty space around the ball generates 147 nodes and 643 leaves. The empty space around the teapot creates 5094 nodes and 18824 leafs.

	GTX 260 (Desktop GPU)		Quadro FX 570M (Mobile GPU)	
	Bouncing Ball	Teapot	Bouncing Ball	Teapot
Input mesh	264 triangles	3136 triangles	264 triangles	3136 triangles
Voxelization	3.0-3.3 ms	3.5 ms	2.6 ms	5 ms
OcPyramid buildup	3.2 ms	2-3 ms	5 ms	4.5 ms
LeafList generation	0.5 ms	0.86 ms	0.6 ms	3.5 ms
NodeList generation	0.5 ms	0.5 ms	0.8 ms	1. 8 ms

Table 9.1: Timings for `achterl`, with an octree analysis at 128x128x128 resolution.

	GTX 260 (Desktop GPU)		Quadro FX 570M (Mobile GPU)	
	Bouncing Ball	Teapot	Bouncing Ball	Teapot
Input mesh	264 triangles	3136 triangles	264 triangles	3136 triangles
Voxelization	2.2 ms	2.2 ms	2.6 ms	5 ms
OcPyramid buildup	0.9 ms	0.9 ms	3 ms	3.5 ms
LeafList generation	0.42 ms	0.4 ms	0.6 ms	2.5 ms
NodeList generation	0.4 ms	0.4 ms	0.7 ms	1.2 ms

Table 9.2: Timings for `achterl_rect`, with an octree analysis at 128x128x128 resolution.

Table 9.1 provides the timings for `achterl` on two GPUs. The more optimized version `achterl_rect` is evaluated with the same tasks in Table 9.2. Voxelization times for ball and teapot show that the consumed time scales less than linearly with the number of triangles. This is good, as the same triangles have to be processed many times to fill the slices of this volume – of course, they are also often clipped before rasterization. OcPyramid buildup times are more or less independent of model size, as expected.

	GTX 260 (Desktop GPU)		Quadro FX 570M (Mobile GPU)	
	Bouncing Ball	Teapot	Bouncing Ball	Teapot
Input mesh	264 triangles	3136 triangles	264 triangles	3136 triangles
Voxelization	3.7-4.1 ms	6.8 ms	11.6 ms	19.5 ms
OcPyramid buildup	4.6 ms	4.6 ms	20 ms	21.8 ms
LeafList generation	0.4 ms	1.6 ms	0.8 ms	10.2 ms
NodeList generation	0.4 ms	0.7 ms	0.7 ms	3.3 ms

Table 9.3: Timings for `achterl_rect`, using two different GPUs.

Due to driver restrictions, higher volume resolutions could only be handled by `achterl_rect`. The 256^3 octree analysis of the empty space around the ball generates 661 nodes and 2357 leaves, with results shown in Table 9.3. The empty space around the teapot creates 20816 nodes and 75275 leafs.

Results from both measurements show that `achterl_rect` is currently the best choice for performance, and the only implementation that can handle larger input volumes. `achterl`, on the other side, is easier to learn from because of its more straight-forward implementation.

In the future, we aim to combine the advantages of both approaches with 2D texture arrays. These do not require the 2D mapping of volume textures of `achterl_rect`, while still retaining faster 2D texture lookups and thus higher performance. However, they require geometry shaders, and can thus only be used on SM4 class graphics hardware.

9.9.4. Dynamic Octree Generation

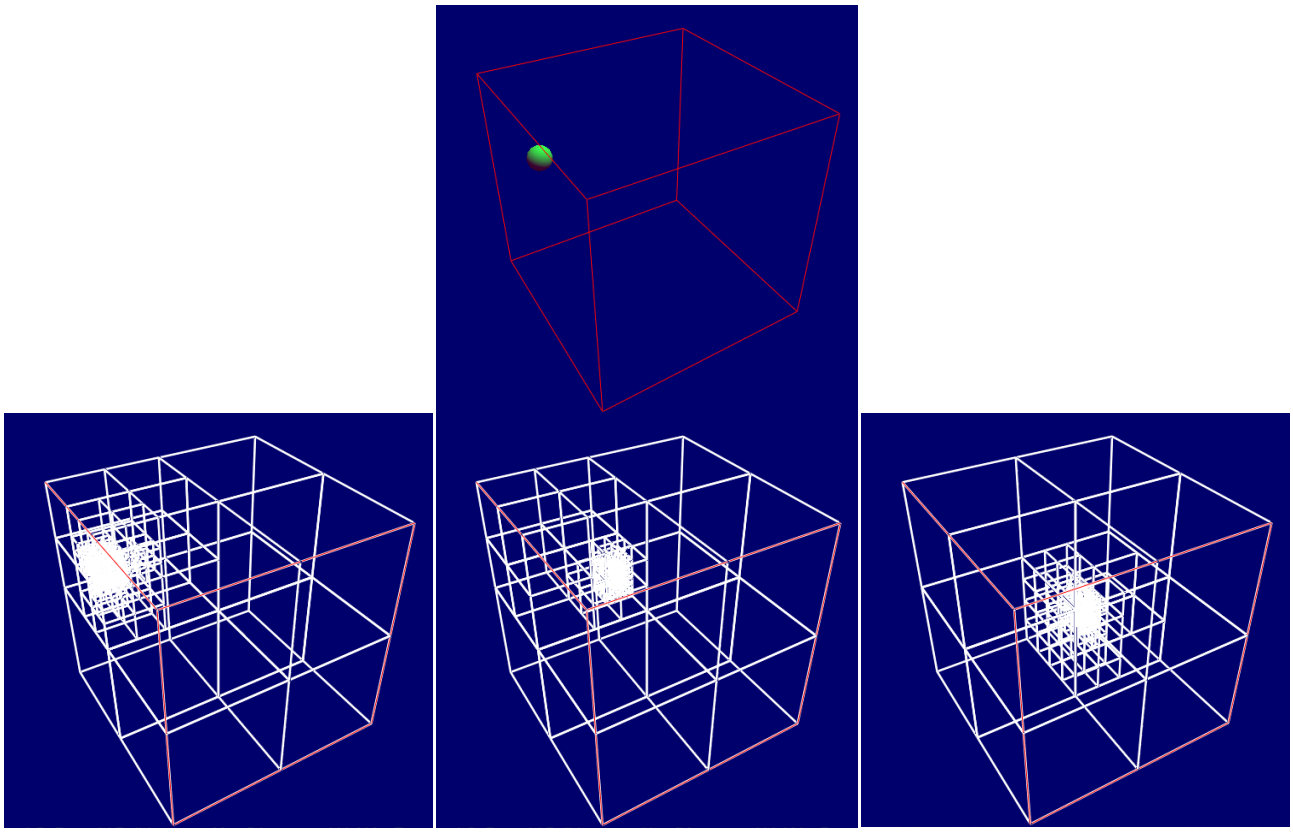


Figure 9.14: Continous octree analysis of a bouncing ball animation.

During our masurement, we also investigated how to deliver octrees from dynamically changing volumes. For this purpose, we animated the mentioned ball to bound around the input volume, with some screenshots shown in Figure 9.14. This demonstrates that acceleration structures for dynamic 3D models, such as the ones needed in interactive raytracing, can now be updated continuously as the 3D model changes position or form.

9.10. Applications

As already noted in Section 9.1.1., we kept the tutorial application simple to focus on the algorithm basics during explanation, and did similarly with `achterl` to deliver generally valid time measurements. But as with all octree implementations, the application realm goes far beyond the generation of octrees from 3D mesh voxelization - especially if such an octree can be created in real-time.

Please note that while the presented algorithm creates octrees from the *voxelization* of triangle meshes, it does *not* include an octree construction from unsorted triangle soups. Such triangle soups first need to be *sorted* before they can be structured into a octree, an operation that goes beyond the scope of this chapter.

9.10.1. Ocxelization

The presented 3D mesh voxelization and the resulting octree are surely not of high enough resolution to replace 3D meshes in scene rendering. But we can generate a compressed octree representation of a much higher resolution than a single voxelization. O*xcelization* alternates voxelization with octree generation by voxelizing parts of the scene and immediately converting the voxelization to a node/leaf list representation, that continously grows until the whole scene has been converted.

This method can e.g. applied to finely detailed scene meshes, yielding high-resolution octrees. Then, any octree-aware ray-casting or iso-surface extraction algorithm can be used to interactively visualize the complete volume. Similar methods have already been used for isosurface extraction from large volumes [WKE99].

9.10.2. Volume compression

Since an octree node/leaf list losslessly represents the original input volume, it can be used as a memory-efficient representation for volume data in graphics memory, or as rapid compression method for persistent volume data storage (e.g. for volume streams). For example, it would now be possible to stream 2048^3 resolution volume data in tiles to the GPU, and have them compressed into node and leaf lists on the fly, creating a compressed representation that can be completely held in GPU memory. Note that due to Morton-order traversal, the leaf list entry only needs to be equipped with location codes, and can still allow queries.

9.10.3. Volume analysis

Octrees are, unlike quadtrees, rarely applied in classic computer vision. But they are of use in areas of visual computing which merge computer graphics and image processing, such as 3D reconstruction from multi-view video or medical volume processing.

Real-time octree generation is also relevant to GPU-based PDE solver implementations. There, it is often of crucial importance to adapt scarce computational resources to local PDE complexity. A

quick Octree analysis on the GPU can organize the volume into regions of varying size, but equal PDE complexity, and thus adapt simulation granularity to the task at hand.

Another unusual application of octree-based volume analysis was demonstrated by Gervautz and Purgathofer [GP90]. Given an image and its pixels' color values, they place the color values into a cube of 256x256x256, exploiting the fact that colors are usually given in 8bit RGB triples. Then, hierarchical clustering takes place, registering all the occupied volume positions (i.e. actually used colors), and hierarchically averaging them. Given a maximum number of bits available, e.g. four bits for red and two bits for green and red, color entries can be extracted from the octree's higher levels. This way, the palette is exploited to the maximum.

Using the presented algorithm of this chapter, this octree-based palette creation can now happen in real-time. While palette reduction is rarely used nowadays, a similar technique might e.g. prove useful for the pre-clustering of motion vectors before data compression.

9.11. Conclusions and Outlook

We have presented a novel GPU algorithm for the rapid generation of octrees from 3D volume data. Through experiments we have shown that real-time octree generation can be achieved for volumes up to 256^3 on recent graphics hardware. An even larger compact octree can be created piece by piece from large input volumes or finely detailed 3D meshes.

For the analysis of 3D meshes, we introduced a voxelization algorithm that is even stable for closed, but nested meshes. This is accomplished through concepts from shadow volume approaches: repeated rasterization with iteratively growing viewports along the z-axis. While this seemingly impairs performance should the mesh become too large, in reality, this will not happen as volume grid resolution sets resolution limits for the mesh details.

Numerous extensions are possible. For example, the mentioned pyramid buildup extensions for node lists can also be backported to the QuadPyramids introduced in [ZDTS07]. The concepts also prove useful to run-length-encode the output of HistoPyramid-based data compaction.

The algorithms can act as a major building stone in advanced volume processing on graphics hardware, such as GPU-based machine learning applications, 3D volume data compression to adaptive approaches in GPU PDE/ODE calculation. Hi-speed GPU octree generation also paves the way for new approaches to GPU raytracing and future, non-triangle based graphics applications.

10. Discussion and Conclusions

In this thesis, we have demonstrated that graphics hardware can utilize its data-parallel capabilities to deliver new levels of performance for complex vision and graphics processing. Conveniently, many of the algorithms even proved useful in *general data processing* on graphics hardware. In this chapter, we will first summarize the overall contributions gained from the individual chapters in this thesis. The summary is divided into two parts: Section 10.1 covers Chapter 1-5 which used real-time graphics techniques to accelerate applications on the boundary between computer vision and graphics. Section 10.2 covers Chapters 6-9, in which data structures and algorithms had become so general that they can be used outside computer vision and graphics. These algorithms widen the use of graphics hardware to a wider field of computing, also known as GPGPU (General Purpose Computing on graphics hardware). We continue with a discussion in the next sections. Section 10.3 argues that a restriction of the data-parallel programming model to stream processing enforces more thorough algorithm design, because read-write hazards are not even possible due to the strict thread I/O model. Section 10.4. brings forward the insight that data-parallel programming is not merely a switch to a new software or hardware API, but that it really requires a new way of thinking from the programmer. Section 10.5 discusses the weaknesses and strengths of upcoming generic programming languages for graphics hardware. Section 10.6. concludes this thesis, with an outlook on the future of many-core processing..

10.1. GPU Techniques for Image Processing

In Chapter 3, we investigated the compression of Multi View Video (MVV) with known scene geometry. With a transformation from camera views to object surface texture, the redundance in the multiple camera views can be exploited much better for compression than with 2D motion compensation (MVV texture), but such a transformation would be prohibitively expensive on the CPU. Therefore, we used the GPU's texturing and vertex transformation capabilities to transform the 3D mesh of the object of interest and its projected-upon camera images into the texture domain, where spatial and temporal redundancies are removed, and wavelet compression can commence. Additionally, GPUshadow mapping was used to determine the object surface's exact exposure to the different cameras, which further assisted in wavelet compression by masking parts of the texture.

Chapter 4 ventured deeper into aspects of the camera/projector-dualism on graphics hardware. First, we demonstrated how manual multi-camera calibration can be simplified if a proxy geometry of the depicted scene is present, and graphics hardware enables interactive adjustment of the camera parameters by continous reprojection of the camera images onto the scene. Afterwards, we demonstrated how not only color video, but also depth video can be re-projected by the use of a fine-tesselated mesh, and explained the OpenGL concepts that allow for a camera/projector representation in a file format. As free viewpoint video often is composed from several camera views, we presented a novel, CSG-based method for the composition of multiple depth video streams on graphics hardware.

Chapter 5 moved to more non-typical uses of graphics hardware, and exploited the recently introduced programmability more thoroughly for image processing. First, several approaches to depth reconstruction from camera images are presented. We re-used the aspects of camera/projector dualism again, and implemented a plane-sweep approach with a coarse-to-fine depth approximation based on mip-maps, which is more resilient to local noise and saves on computation. We applied this approach both on stereo images and lightfield data. Later, we traded performance for improved depth accuracy, and generated image stacks of varying filter kernel sizes before coarse-to-fine depth

reconstruction via plane sweep commenced. Second, we utilized data-parallel reduction for noise suppression in object tracking. By joining feature regions that are connected into one and suppressing other regions that are disjoint and smaller, we introduce a notion of connectivity, while retaining the data-parallelism of reduction. The tracking algorithm's implementation accurately tracked a laser pointer or a head-light in high-resolution video in real-time without disturbances by image noise. Thirdly and finally, we show how histograms of local image features can be computed through data-parallel reduction in multiple channels. We utilized this fast GPU histogram computation in an application that automatically maximizes the variance of local image gradient directions in camera images. By providing the image of a checker calibration image, optimal parameters to compensate lens cushion distorsion were retrieved within seconds.

10.2. General Data Processing

In Chapter 6, we moved to a problem that is not only found in image processing: The selection of several elements from a larger input array in graphics memory. We introduced the HistoPyramid data structure and associated, data-parallel buildup and traversal algorithms as a solution to this problem, and demonstrated its practicality in two applications from volume processing, representative for the large number of applications: point list generation from 3D meshes, and the volume visualization technique of vector field contours, which relies on the quick generation of streamlines from a large input volume.

Chapter 7 extended the problem task to data expansion, i.e. enabling not only the selection of a number of elements, but also the generation of *several* modifiable copies of selected input elements. We explained how HistoPyramid buildup and traversal can be modified in such a way that it allows for 1-to-m generation of selected input elements, and therewith even can replace and outperform the geometry shader functionality of recent graphics hardware. To prove the extended algorithm's performance, we demonstrated two applications that require data expansion: First, an ODE-based light wavefront simulation for refractive index volumes, which requires adaptive tesselation of the wavefront to remain computationally manageable. Second, an implementation of an algorithm for the extraction of iso-surfaces from volumes, Marching Cubes, which runs all stages on graphics hardware and therefore outperforms all previous implementations.

Chapter 8 utilized the 2D texture implementation of the HistoPyramid data structure in a different way: Quadtree generation. We introduced a new marker to the HistoPyramid to signal that a certain region was a growing quadtree leaf in the buildup process, the leaf marker. Afterwards, we demonstrated in a video processing application that quadtree analysis now has become a real-time operation on graphics hardware.

Chapter 9 extended the notion of quadtree generation to the 3D case: octree generation. The resulting 3D data structure, the OcPyramid, is used to generate octree leaves from data volumes, by repeated clustering of smaller octree leaves during the OcPyramid buildup. In addition to the 3D port of the quadtree concept, this chapter is the first to explain how lists of octree leaf references, node lists, are generated. These node lists define the hierarchy of octree subdivisions from the complete volume down to individual leaves, and are crucial for applications in ray tracing and octree position queries. In the algorithm's implementation, we demonstrated a GPU implementation which is able to generate octrees from 3D volumes at real-time speeds.

10.3. Innovation from Programming Model Restrictions

In this thesis, we adhere to the stream processor model in this thesis, which is more strict than e.g. the newer CUDA programming model. But a restricted processing model can sometimes lead to *leaps in algorithmic design*: One such example is the fixed-output location, which led us to

introduce tree buildup and traversal techniques in data compaction. Interestingly, such algorithmic adaptation can provide performance gains even if the original restrictions to the programming model *have been lifted*. The reason is that while new features of a programming model or a hardware architecture often simplify the algorithm, they also come with performance sacrifices: For example, G80 hardware architecture was the first to allow random writes to memory (scattering), and was consequently a feature of the CUDA programming model. But these random writes break the coherence that memory write transactions usually had for fixed output locations. A GPGPU-style algorithm which maintains a fixed output-location is therefore still more performant in CUDA. After an algorithm has been designed and proven to work in a restricted programming model, new features can later be added to test if they provide benefits, which e.g. our CUDA-based data expansion demonstrates by using on-chip shared memory. Algorithmic development thus becomes a natural evolution, with a safe fallback if a new hardware feature should fail to provide a performance gain.

From the given restrictions, it is clear that algorithmic design for data-parallel architectures is hard. Human thought is used to serial task execution, and many serial algorithms for the CPU, often heavily optimized over the years, have *serial dependencies* throughout the code, which make them highly inefficient on graphics hardware due to the unbearable number of GPU execution passes.

But whenever the data-parallel version of a serial algorithm has been found, it will be *arbitrarily parallelizable*, i.e. it scales in performance with the number of cores. Theoretically, performance is only limited by the number of data elements, or non-parallelizable calculation bottlenecks in the algorithm.

10.4. Relevance of Programming Model

We can also safely state that the plain knowledge of an API for GPU-computation, such as CUDA, does not suffice for attaining optimal hardware performance. This is important because many developers use serial programming techniques, even while developing data-intense applications that expose a high-level of data-parallelism, and often see the underlying hardware API as a mere means to implement their algorithm..This is acceptable as long as a powerful CPU can execute the serial code at high speeds. But many-core architecture run serial code very inefficiently due to the lack of parallell threads that would keep all cores equally busy. A many-core architecture, which is otherwise extremely well suited for data-parallel tasks, is thus severely hampered by the serial program code which lacks parallelism.

Intel's Larrabee engineers had initially hoped that academia would create auto-parallelization software to convert serial code automatically into data-parallel code, an opinion also shared by Hwu [Sha08]. However, when considering the different design of some data-parallel algorithms in this thesis, it is hard to see how serial program code could be analyzed so deeply that it can be rewritten into an efficient data-parallel implementation for a many-core architecture.

Thus, even if the API provides more and more CPU features such as random scattering to memory, the developer's thought model *must* adapt to the different characteristics of many-core-architectures. With the spread of these architectures, this knowledge will prove useful in other areas such as mobile computing, where mobile and embedded processor architectures start integrating basic graphics hardware into their designs, and in CPU development, where more and more parallel threads will eventually cause a transition from task-parallel to data-parallel programming.

10.5. OpenGL, CUDA and OpenCL

By now, CUDA shows the way towards a general-purpose language for graphics hardware, and many-core architectures in general, which is underlined by the similarity of CUDA and the OpenCL standard. A general purpose programming model for the GPU was necessary because developers from non-graphics areas are, naturally, not willing to learn a graphics API for conducting general purpose computing on GPU architectures, and CUDA is very important to reduce the learning threshold.

However, performance of current CUDA-based algorithms can still lag behind OpenGL implementations – one reason is that CUDA does not completely support the available hardware features of graphics hardware, such as write-to-texture, another reason is that CUDA-OpenGL interoperability is still limited. This will change in future CUDA releases, since early OpenGL-based GPGPU applications set a reference for the actual capabilities of graphics hardware in visual computing.

For above reasons, it might still be easier to use OpenGL for general purpose computing in graphics applications that have to make use of the graphics part of the GPU in any case. After all, the developer often knows the API well, and for 3D computer vision or similar 3D-based computations (such as image based rendering) it might actually be of advantage to stick closer to the OpenGL API and its graphics-related features, instead of using a general purpose API such as CUDA. As a bonus, OpenGL concepts might actually transfer to the application design, and both simplify 3D programming and yield new ideas, such as the ones we used for GeoCast and lightfield 3D reconstruction in this thesis.

10.6. Future

In general, there should be only few computational tasks left that cannot be done on GPUs, and on many-core architectures in general. As serial-processor performance is currently improving too slowly, we expect a boom in many-core hardware usage for all branches of industry. Time will show if CPU and GPU will merge, a concept which seems advantageous from the programmer's point of view. But the penalties for performance would be noticed and have to be considered, as both CPU and GPU would have to share memory bandwidth.

We hope that the presented algorithms have contributed to improving computing performance in various areas of research by extending the application domains of graphics hardware. Vasconcelos et al. have used our QuadPyramid algorithms to quickly find spatial neighbours of quadtree leaves on graphics hardware [Vas08], and to generate quadtrees for energy minimization via graph cuts [Vas07]. HistoPyramid data compaction has recently been implemented in the OpenVIDIA software framework for computer vision by Fung et al. [FM05], and the related data expansion has accelerated a GPU-based adaptive tesselation through geometry instancing by Dyken et al beyond the performance that geometry shader functionality would provide [DRS07].

We are thus looking forward to seeing how GPU computing will soon transform computer vision and graphics into real-time research areas.

11. Bibliography

[3DIP] 3D-IP. Company homepage, archive snapshot. http://web.archive.org/web/20040322171004/http://3d-ip.com/

[3DTV] 3DTV Research Consortium. Validation Data for Camera Parameters of MVC. Submission to Call for Proposals to MPEG-3DAV (MPEG-3D Audio/Video part of MPEG committee) . https://www.3dtv-research.org/3dav_Validation_Data

[3DV] 3DVSystems. Company homepage, 2003. http://www.3dvsystems.com

[ABC*06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. EECS Department, University of California, Berkeley. Technical Report No. UCB/EECS-2006-183, Dec. 18th, 2006.

[Arn07] M. Arnold. GPU-based 3D light marker tracking. Bachelor Thesis, Universität des Saarlandes, 2007.

[ATR*08] T. Annen, H. Theisel, C. Rössl, G. Ziegler, H.-P. Seidel. Vector Field Contours. Proceedings of Graphics Interface, Volume 322, pp. 97 - 105, 2008.

[Bar99] P. G. Barten. Contrast sensitivity of the human eye and its effects on image quality. SPIE - The International Society for Optical Engineering. 1999. ISBN 0-8194-3496-5.

[BBF*97] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grün and C. Lichtenau. Building the 4 Processor SB-PRAM Prototype. Proceedings of the Hawaii 30th International Symposium on System Science HICSS-30. pp. 14-23, 1997.

[BCL06] L. Buatois, G. Caumon, B. Levy. GPU accelerated isosurface extraction on tetrahedral grids. Proceedings of International Symposium on Visual Computing 2006, 2006.

[BD02] D. Benson, J. Davis. Octree textures, SIGGRAPH 2002, ACM TOG 23, 785 - 790, 2002.

[BDL07] R. Brunner, F. Doepke, B. Laden.Object Detection by Color: Using the GPU for Real-Time Video Image Processing. GPU Gems 3, Chapter 26, Addison-Wesley, Aug. 2007. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch26.html

[Ben96] J.-D. Benamou. Big ray tracing: Multivalued travel time field computation using viscosity solutions of the eikonal equation. Journal of Computational Physics 128, issue 2, pp. 463-474, 1996.

[BFGS03] J. Bolz, I. Farmer, E. Grinspun, P. Schröder. Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics 22, pp. 917-924, 2003.

[BFH04] I. Buck, K. Fatahalian, P. Hanrahan. GPUBench: Evaluating GPU performance for numerical and scientific applications. In 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, pp. C-20, Aug. 2004. http://graphics.stanford.edu/projects/gpubench

[BG04] A. Bogomjakov, C. Gotsman. GPU-Assisted Z-Field Simplification. Proceedings of the 2nd International Symposium for 3D Data Processing, Visualization, and Transmission (3DPVT), pp. 673 – 679, 2004.

[BH95] H. Bast, T. Hagerup. Fast Parallel Space Allocation, Estimation and Integer Sorting. Journal Information and Computation, Volume 123, Issue 1, pp. 72-110, Nov. 1995.

[BIG*03] S. Bihlmaier, I. Ihrke, B. Goldluecke, M. Magnor, L. Ahrenberg. "Kung-Fu Girl" - A synthetic test sequence for multi-view reconstruction and rendering research. Digital multi-view video footage. 2003. http://www.mpi-inf.mpg.de/departments/irg3/kungfu/

[BK00] S. Buske, U. Kästner. Efficient and Accurate Computation of Seismic Traveltimes and Amplitudes. Geophysical Prospecting 52, pp. 313-322, 2000.

[BL06] P. Brown, B. Lichtenbelt. NV_geometry_shader4 (OpenGL extension specification). Nov. 2006. http://tinyurl.com/2jdf5v

[Bli82] J. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In Proc. of SIGGRAPH'82, pp. 21-29, 1982.

[BLW06] P. Brown, B. Lichtenbelt, E. Werness. NV_transform_feedback, OpenGL extension specification. Nov. 2006. http://tinyurl.com/2jdf5v

[Bou] J.-Y. Bouguet. Camera Calibration Toolbox for Matlab. Intel Corp. http://www.vision.caltech.edu/bouguetj/calib_doc

[BP04] I. Buck, T. Purcell. A Toolkit for Computation on GPUs. GPU Gems, pp. 621-636, Addison-Wesley, 2004.

[BW99] M. Born, E. Wolf. Principles of Optics. Cambridge University Press (Seventh Edition), 1999.

[Cat74] E. Catmull. A Subdivision Algorithm for Computer Display of Curved Surfaces. PhD thesis, Dept. of Computer Science, University of Utah, 1974.

[CDS*06] A. Chalmers, K. Debattista, Veronica Sundstedt, Peter Longhurst, Richard Gillibrand. Rendering on Demand. Eurographics Symposium on Parallel Graphics and Visualization, May 2006.

[CL96] B. Curless, M. Levoy. A volumetric method for building complex models from range images. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques , pp. 303 - 312, 1996.

[CMS06] H. Carr, T. Moller, J. Snoeyink. Artifacts caused by simplicial subdivision. IEEE Transactions on Visualization and Computer Graphics, 12(2):231–242, Mar. 2006.

[Col94] S. Collins. Adaptive Splatting for Specular to Diffuse Light Transport. In Proc. of EGWR, pp. 119-135, 1994.

[Col97] S. Collins. Wavefront Tracking for Global Illumination Solutions. PhD thesis, Department of Computer Science, Trinity College Dublin, 1997.

[Cro77] F.C. Crow. Shadow Algorithms for Computer Graphics. Computer Graphics (SIGGRAPH '77 Proceedings), vol. 11, no. 2, 242-248.

[CTMS03] J. Carranza, C. Theobalt, M. Magnor, H.-P. Seidel. Free-viewpoint video of human actors. Proceedings of ACM SIGGRAPH 2003, pp. 569-577, San Diego, CA.

[Deb96] P. E. Debevec, C. J. Taylor, J. Malik. Modeling and Rendering Architecture from Photographs. Proc. SIGGRAPH 1996, August 1996.

[Der92] R. Deriche. Recursively implementing the Gaussian and its derivatives. Proceedings of the 2nd International Conference on Image Processing, Singapore, p. 263–267. 1992.

[DFRS03] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, A. Santella. Suggestive contours for conveying shape. ACM Transactions on Graphics, 22(3):848–855, 2003.

[DHE*03] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, I. Buck. Merrimac: Supercomputing with Streams. Proceedings of the 2003 ACM/IEEE conference on Supercomputing, November 2003.

[Dim07] R. Dimitrov. Solid Environment Reconstruction on the GPU. Bachelor thesis / Senior research project report, Jacobs University Bremen, 2007.

[DL07] C. Dachsbacher, S. Lefebvre. TileTrees. Proc. of the 2007 symposium on Interactive 3D graphics and games, pp. 25-31, 2007.

[DM03] H. Danyali, A. Mertins. Fully scalable texture coding of arbitrarily shaped video objects. Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'03), pp. 393-396, Apr.2003.

[DMH04] H. Doleisch, P. Muigg, and H. Hauser. Interactive visual analysis of hurricane Isabel. VRVis Technical Report, 2004-058, 2004.

[DRS07] C. Dyken, M. Reimers, J. Seland. Real-time GPU Silhouette Refinement using adaptively blended Bézier patches. Computer Graphics Forum, Volume 27, number 1, pp. 1-12, 2007.

[DZTS08] C. Dyken, G. Ziegler, C. Theobalt, H.-P. Seidel. High-Speed Marching Cubes using Histogram Pyramids. Computer Graphics Forum, Issue 27, number 8, pp. 2028-2039, 2008.

[ED06] E. Eisemann, X. Décoret. Fast Scene Voxelization and Applications. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 71-78, 2006.

[Eikonal] I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt. Eikonal Rendering, video footage. http://www.mpi-inf.mpg.de/resources/EikonalRendering/index.html

[ER03] B. Enquist, O. Runborg. Computational High Frequency Wave Propagation. Acta Numerica 12, pp. 181-266, 2003.

[Eve02] C. Everitt. Projective Texture Mapping. Online Publication, NVIDIA Corp. http://developer.nvidia.com/object/Projective_Texture_Mapping.html

[Flu06] O. Fluck, S. Aharon, D. Cremers, M. Rousson. GPU Histogram Computation. Poster Session of ACM SIGGRAPH 2006.

[FM05] J. Fung, S. Mann. OpenVIDIA: parallel GPU computer vision. Proceedings of 13th annual ACM international conference on Multimedia, 2005, pp. 849 - 852. http://openvidia.sf.net

[Fow04] J. E. Fowler. Shape-adaptive coding using binary set splitting with k-d trees. Proceedings of IEEE International Conference on Image Processing, Singapore, Volume: 2, pp. 1301- 1304, 2004.

[Gar82] I. Gargantini. An Effective Way to Represent Quadtrees. Communications of the ACM, Issue 25(12), pp. 905–910, 1982.

[GB08] A. Grundhöfer and O. Bimber. Real-Time Adaptive Radiometric Compensation. IEEE Transactions on Visualization and Computer Graphics (TVCG), vol. 14, no. 1, pp. 97-108, 2008.

[GBM*04] F. Galpin, R. Balter, L. Morin, S. Pateux. Efficient and scalable video compression by automatic 3D model building using computer vision. Proceedings of Picture Coding Symposium (PCS 2004), San Francisco, USA, December 2004.

[GCS06] M. Gösele, B. Curless, S. M. Seitz. Multi-View Stereo Revisited. Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), Volume 2, pp. 2402-2409, 2006 .

[Geo04] G. Ziegler. Example GeoCast/OpenEXR sequence: Single view, Sedan. Data Repository of 3DTV Network of Excellence, 2004. https://www.3dtv-research.net/downloadlist.php?dat&s=90

[GGH02] X. Gu, S. J. Gortler, H. Hoppe. Geometry Images. ACM Transactions on Graphics (TOG), Volume 21, Issue 3, Proceedings of ACM SIGGRAPH 2002, pp. 355 – 361, 2002.

[GH95] A. Guéziec, R. Hummel. Exploiting Triangulated Surface Extraction Using Tetrahedral Decomposition. IEEE Transactions on Visualization and Computer Graphics, Volume 1, Issue 4, pp. 328-342, 1995.

[GHL*04] N. Govindaraju, M. Henson, M. Lin, D. Manocha. Computations among Geometric Primitives in Complex Environments. Proc. ACM Symposium on Interactive 3D Graphics and Games, 2005.

[GJD05] F. Goetz, T. Junklewitz, G. Domik. Real-Time Marching Cubes on the Vertex Shader. Eurographics 2005 Short Presentations, Dublin, Irland, August 2005.

[GM04] K. Gruchalla, J. Marbach. Immersive Visualization of the Hurricane Isabel Dataset.IEEE Visualization 2004 Contest. http://vis.computer.org/vis2004contest/colorado/isabel_gruchalla_marbach.pdf

[GMAS05] D. Gutierrez, A. Muñoz, O. Anson, F. J. Seron. Non-linear volume photon mapping. In Proc. of EGSR, pp. 291-300, 2005.

[GMDL05] G.Gimel'farb, J.Morris, P. Delmas, J. Liu. Noise-driven symmetric concurrent stereo matching. Proceedings of Image Vision Computing New Zealand Conference (IVCNZ), Dunedin, New Zealand, November 2005, pp. 90-95, 2005. http://ncsm.uuuq.com/ncsm_intr.html

[GP90] M. Gervautz, W. Purgathofer. A simple method for color quantization: Octree quantization. Graphics Gems I, pp. 287-293, Academic Press Professional Inc., ISBN:0-12-286169-5, 1990.

[Gre01] Simon Green, NVidia Corp. gllife: OpenGL accelerated Game of Life. http://www.geocities.com/simesgreen/gllife/

[Gre05] S. Green, NVidia Corp. OpenGL Image Processing Tricks. GDC 2005 Presentations, 2005. http://tinyurl.com/f59r4

[Gre05b] S. Green, NVidia Corp. NVidia OpenGL Update. GDC 2005 Presentations, 2005, pp. 40-42. http://tinyurl.com/pngld

[GS04] G. Ziegler, M. Schantin. GeoCast Software Repository (Blender Export Plugin, Viewer). 2004. http://geocast.sf.net

[Gsch06] M. Gschwind. The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor. IBM Research Report RC24128 (W0601-005), Computer Science, Oct 2nd, 2006.

[Gus99] GusGus. Desire. Full quality music video footage, http://www.gusgus.com/

[GWS04] J. Günther, I. Wald, P. Slusallek. Realtime caustics using distributed photon mapping. In Proc. of EGSR, pp. 111-121, 2004.

[Han85] S. Hanan. Data structures for quadtree approximation and compression. Communications of the ACM, Volume 28, Issue 9, pp. 973-993, 1985.

[Har04] M. Harris. Fast Fluid Dynamics Simulation on the GPU. GPU Gems, pp.637-665, Addison-Wesley, 2004.

[Har07] M. Harris. Parallel prefix sum (scan) with CUDA, part of CUDA Data-Parallel Primitives Library, 2007 and onwards. http://gpgpu.org/developer/cudpp

[HC04] L. Hong, G. Chen. Segment-Based Stereo Matching Using Graph Cuts. Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'04), Volume 1, pp.74-81, 2004.

[HCK*99] K.E.I. Hoff, T. Culver, J. Keyser, M. Lin, D. Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. Proceedings of SIGGRAPH 1999, pp. 277- 286, ACM / ACM Press, 1999.

[Hei07] L. Heidenreich. Real-Time Hierarchical Stereo Matching on Graphics Hardware. Diplomarbeit, Universität des Saarlandes, 2007.

[Hen00] R.D. Henkel. Synchronization, Coherence-Detection and Three-Dimensional Vision. Technical Report of the Institute of Theoretical Neurophysics, University of Bremen, 2000.

[Hen97] R.D. Henkel. Fast Stereovision by Coherence-Detection. Proc. of the Seventh International Conference on Computer Analysis of Images and Patterns (CAIP'97), pp. 297-304, 1997.

[HG41] L.G. Henyey, J. L. Greenstein. Diffuse Radiation in the Galaxy. Astrophysical Journal 93, pp. 70-83, 1941.

[Hor05] D. Horn. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, ch. Stream Reduction Operations for GPGPU Applications, pp. 573–589, 2005.

[HS97] J. Heikkilä, O. Silvén. A Four-step Camera Calibration Procedure with Implicit Image Correction. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'97), San Juan, Puerto Rico, pp. 1106-1112, 1997.

[HSBL03] M. Harris, T. Scheuermann, W. V. Baxter III, A. Lastra. Simulation of cloud dynamics on graphics hardware. Proceedings of Graphics Hardware 2003, 2003.

[IAM04] I. Ihrke, L. Ahrenberg, M. Magnor. External camera calibration for synchronized multi-video systems. Proceedings of Workshop Computer Graphics (WSCG), 2004.

[IZT*07] I. Ihrke, G. Ziegler, A. Tevs, C. Theobalt, M. Magnor, H.-P. Seidel. Eikonal Rendering: Efficient Light Transport in Refractive Objects, ACM Transactions on Graphics (Siggraph'07), 2007. http://www.mpi-inf.mpg.de/resources/EikonalRendering/index.html

[JB94] X.Y. Jiang, H. Bunke. Fast Segmentation of Range Images into Planar Regions by Scan Line Grouping. Journal on Machine Vision and Applications, Volume 7, Number 2, June 1994. DOI 10.1007/BF01215806

[JC06] G. Johansson, H. Carr. Accelerating marching cubes with graphics hardware. In CASCON ’06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research (2006), ACM Press.

[Jen01] H. W. Jensen. Realistic Image Synthesis Using Photon Mapping. Published by AK Peters, ISBN 1568811470, 2001.

[JM92] D. G. Jones, J. Malik. A Computational Framework for Determining Stereo Correspondence from a Set of Linear Spatial Filters. Proceedings of the Second European Conference on Computer Vision, Lecture Notes in Computer Science, Vol. 588, pp. 395 – 410, 1992.

[JO04] G. James. J. O'Rorke. Real-Time Glow. GPU Gems, Chapter 21, Addison-Wesley, 2004. http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html

[JS05] J. Juliano, J. Sandmel. GL_EXT_framebuffer_object. OpenGL extension registry, 2005.

[KDR*02] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens and Brucek Khailany. The Imagine Stream Processor. Proceedings of the 2002 International Conference on Computer Design, September 16-18, 2002, Freiburg, Germany.

[Kes06] J. Kessenich. The OpenGL Shading Language. 3Dlabs Inc. Ltd., Version 1.20 edition, September 2006. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

[Kil97] M. J. Kilgard. Realizing OpenGL: two implementations of one architecture. HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, August 1997.

[KKS*05] R. Koch, K. Koeser, B. Streckel, J.-F. Evers-Senne. Markerless Image-based 3D Tracking for Real-time Augmented Reality Applications. Proceedings of International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS) 2005, 2005.

[KKT01] J. Keller, C. Keßler, J. Träff. Practical PRAM Programming. John Wiley and Sons, 2001

[KMS05] G. Krawczyk, K. Myszkowski, H.-P. Seidel. Perceptual effects in real-time tone mapping. In Proc. of Spring Conference on Computer Graphics, ACM, pp. 195-202, 2005.

[Kno06] A. Knoll. A Survey of Octree Volume Rendering Methods. GI Lecture Notes in Informatics, Proceedings of 1st IRTG Workshop, Dagstuhl, Germany, 2006.

[KSE04] T. Klein, S. Stegmaier, T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. Proc. Pacific Graphics, 2004.

[KW03] J. Krüger, R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. Proc. ACM SIGGRAPH 2003, pp. 908-916, 2003.

[KW05] P. Kipfer, R. Westermann. GPU construction and transparent rendering of iso-surface. In Proceedings Vision, Modeling and Visualization 2005, IOS Press, pp. 241–248, 2005.

[LB07] B. Lichtenbelt, P. Brown. GL_EXT_gpu_shader4. OpenGL extension registry, 2007.

[LC87] W. Lorensen, H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. Computer Graphics (SIGGRAPH 87 Proceedings) 21, issue 4, pp. 163–170, 1987.

[Lef06] A. Lefohn, J. Kniss, R. Strzodka, S. Sengupta, J. Owens. Glift: Generic, efficient, random-access GPU data structures. ACM Transactions on Graphics 25, pp. 60-99, 2006.

[LGR04] F. Losasso, F. Gibou, R. Fedkiw. Simulating Water and Smoke with an Octree Data Structure. SIGGRAPH 2004, ACM Transactions on Graphics 23, pp. 457-462, 2004.

[LH06] S. Lefebvre, H. Hoppe. Perfect Spatial Hashing. Proc. of ACM SIGGRAPH 2006.

[LH07] S. Lefebvre, H. Hoppe. Compressed Random-Access Trees for Spatially Coherent Data. Proceedings of EGSR, 2007.

[LH91] D. Laur, P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. Computer Graphics, Volume 25 (4), pp. 285-288, July 1991.

[LHN05] S. Lefebvre, S. Hornus, F. Neyret. Octree textures on the GPU. GPU Gems 2, pp. 593-613, 2005.
http://lefebvre.sylvain.free.fr/octreetex/

[Li01] M. Li. Correspondence analysis between the image formation pipelines of graphics and vision. Proceedings of the 9th Spanish Symposium on Pattern Recognition and Image Analysis, pp. 187–192, May 2001.

[LKM01] E. Lindholm, M. J. Kilgard, H. Moreton. A user-programmable vertex engine. SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, August 2001.

[LP96] M. Levoy, P. Hanrahan. Light Field Rendering. Proceedings of SIGGRAPH 1996, pp. 31-42, 1996.

[MGAK03] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, July 2003.

[MH92] D. Mitchell, P. Hanrahan. Illumination from curved reflectors. In Proc. of SIGGRAPH '92, pp. 283-291, 1992.

[Mor66] G.M. Morton. A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing. Technical Report, IBM Ltd. Ottawa, Canada, 1966.

[MRG03] M. Magnor, P. Ramanathan, B. Girod. Multi-view coding for image-based rendering using 3-D scene geometry. IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 11, pp. 1092–1106, Nov. 2003.

[MSS94] C. Montani, R. Scateni, R. Scopigno. A modified look-up table for implicit disambiguation of Marching Cubes. The Visual Computer 10, pp. 353–355, 1994.

[NSI99] K. Nishino, Y. Sato, K. Ikeuchi. Eigen-texture Method: Appearance Compression based on 3D model. Proceedings of IEEE Conf. on Computer Vision and Pattern Recognition, pp. 618–624, June 1999.

[NV04b] NVIDIA Corp. Image Histogram. SDK Code Samples - Video and Image Processing, 2004. http://tinyurl.com/dcx4rv

[NV05] C. Everitt , A. Rege, C. Cebenoyan. Hardware Shadow mapping. NVIDIA Technical Whitepaper, 2005. http://developer.nvidia.com/object/hwshadowmap_paper.html

[NV07] NVIDIA Corp. CUDA Programming Guide, Version 1.1, 2007.

[NVI99] NVIDIA Corp. 2nd Gen Transformation and Lighting (T&L). http://www.nvidia.com/object/transform_lighting.html

[OBM00] M. Oliveira, G. Bishop, D. McAllister. Relief Texture Mapping. Proceedings of SIGGRAPH 2000, pp. 359-368, 2000.

[OCK*02] S. Osher, L.-T. Cheng, M. Kang, H. Shim, Y.-H. Tsai. Geometric Optics in a Phase-Space-Based Level Set and Eulerian Framework. Journal of Computational Physics 179, issue 2, pp. 622-648, 2002.

[OGL21] M. Segal, K. Akeley. The OpenGL Graphics System: A Specification. Silicon Graphics, Inc., Version 2.1 edition, July 2006. http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf

[OLG05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T.J. Purcell. A survey of general-purpose computation on graphics hardware. In Eurographics 2005, State of the Art Reports, pp. 21–51, Aug. 2005.

[Oli86] M. A. Oliver. Display algorithms for quadtrees and octtrees and their hardware realisation. Proc. Workshop on Data Structures for raster graphics, ISBN:0-387-16310-7, pp. 9 – 37, 1986.

[OpenEXR] OpenEXR homepage (Industrial Light & Magic), 2003 and onwards. http://www.openexr.com

[Pas04] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. Joint Eurographics - IEEE TVCG Symposium on Visualization (2004), pp. 292–300, 2004.

[PDC*03] T.J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, P. Hanrahan. Photon mapping on programmable graphics hardware. In Proc. of Graphics Hardware, pp. 41-50, 2003.

[POC05] F. Policarpo, M. Oliveira, J. Comba. Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. ACM Transactions on Graphics, Volume 24, Number 3, pp. 935, July 2005.

[Pop07] S. Popov. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. Computer Graphics Forum, issue 26(3), pp. 415–424, September 2007.

[PSM04] R. Pajarola, M. Sainz, Y. Meng. DMesh: Fast Depth-Image Meshing and Warping. International Journal of Image and Graphics (IJIG), Volume 4, Issue 4, pp. 1 – 29, 2004.

[Rob97] J A Robinson. Efficient General-Purpose Image Compression with Binary Tree Predictive Coding. IEEE Transactions on Image Processing, Vol 6, No 4, pp 601-607, April 1997.

[Ros84] A. Rosenfeld. Multiresolution Image Processing and Analysis. Springer, Berlin, 1984.

[RUL02] J. Revelles, C. Urena, M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Media, 2000.

[Sam89] H. Samet. Implementing Ray Tracing with Octrees and Neighbor Finding. Computers and Graphics, Issue 13(4), pp. 445-460, 1989.

[SCG*05] P. Sen, B. Chen, G. Garg, S. Marschner, M. Horowitz, M. Levoy, H. P. A. Lensch. Dual Photography. Proceedings of SIGGRAPH'05, vol. 24, no. 3, pp. 745-755, 2005.

[Sch07] M. Schilz. High-Resolution multiple points tracking on the GPU. Bachelor Thesis, Universität des Saarlandes, 2007.

[SCS*08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. Proceedings of SIGGRAPH 2008, to be published, 2008.

[Sen06] S. Sengupta, A. E. Lefohn, J. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. Proc. 2006 Workshop on Edge Computing Using New Commodity Architectures, pp. D26-27.

[SGA*99] H.S. Sawhney, Y. Guo, J. Asmuth, R. Kumar. Multi-view 3D estimation and applications to match move. Proceedings on IEEE Workshop on Volume Multi-View Modeling and Analysis of Visual Scenes, pp. 21-28, 1999.

[Sha08] J. Shalf. Using Sequential Programming Models to Program Manycore Systems. Blog entry in View, The Landscape of Parallel Computing Research: A View From Berkeley, March 1st, 2008. http://view.eecs.berkeley.edu/blog/?postid=19

[SIM03] R. Strzodka, I. Ihrke and M. Magnor. A Graphics Hardware Implementation of the Generalized Hough Transform for fast Object Recognition, Scale, and 3D Pose Detection. Proceedings of 12th International Conference on Image Analysis and Processing (ICIAP 2003), pp. 188-193, 2003.

[SJEG05] N. Svakhine, Y. Jang, D. Ebert, K. Gaither. Illustration and photography inspired visualization of flows and volumes. In IEEE Visualization, pages 687–694, 2005.

[SKE06] M. Strengert, M. Kraus, T. Ertl. Pyramid Methods in GPU-Based Image Processing. Proc. VMV 2006, pp. 169-176, 2006.

[SP96] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. IEEE Transactions on Circuits and Systems for Video Technology, vol. 6, pp. 243-250, 1996.

[SS96] D. Stalling, T. Steinke. Visualization of vector fields in quantum chemistry. Technical report, ZIB Preprint SC-96-01, 1996.

[ST88] H. Samet, M. Tamminen Efficient Component Labeling of Images of Arbitrary Dimension Represented by Linear Bintrees. TIEEE Transactions on Pattern Analysis and Machine Intelligence, 1988. doi:10.1109/34.3918

[Sta] The Stanford Volume Data Archive. http://graphics.stanford.edu/data/voldata/

[Sta95] J. Stam. Multiple Scattering as a Diffusion Process. In Proc. of EGSR, pp. 41-50, 1995.

[Sun91] K. Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In Eurographics ´91, pp. 73-85, September 1991.

[SWG*03] P. Sander, Z. Wood, S. Gortler, J. Snyder, H. Hoppe. Multi-Chart Geometry Images. Proceedings of Symposium on Geometry Processing, pp. 146 – 155, 2003.

[TAG04] C. Theobalt, N. Ahmed, G. Ziegler. MPI Dancer - a test sequence for multi-view reconstruction and codec research. Digital multi-view video footage. 2004. http://www.mpi-inf.mpg.de/~gziegler/mpidancer/

[Tan75] S. L. Tanimoto, T. Pavlidis. A Hierarchical Data Structure for Picture Processing. Computer Graphics and Image Processing, vol. 4, no. 2, June 1975, pp. 104-119.

[TAZS07] C. Theobalt, N. Ahmed, G. Ziegler, H.-P. Seidel. High-Quality Reconstruction from Multiview Video Streams. IEEE Signal Processing Magazine, pp. 45-57, November 2007.

[Tevs07] A. Tevs. Realistic Real-time Rendering of Refractive Objects. Master's Thesis in Computer Science, Saarland University (Germany), Faculty of Natural Science and Technology I, Department of Computer Science, pages 1-89, 2007.

[THC*04] M. Tarini, K. Hormann, P. Cignoni, C. Montani. PolyCube-Maps. ACM Transactions on Graphics, vol. 23, issue 3, pp. 853-860, August 2004.

[Thi06] AMD Corp., N. Thibieroz. Direct X 10 for Techies. Presentation Slides for Game Developer Conference 2006, July 2006. http://developer.amd.com/media/gpu_assets/DirectX 10 for techies.pdf

[TWHS03] H. Theisel, T.Weinkauf, H.-C. Hege, H.-P. Seidel. Saddle connectors - an approach to visualizing the topological skeleton of complex 3D vector fields. In Proc. IEEE Visualization 2003, pp. 225–232, 2003.

[TZ06] A. Tevs, G. Ziegler. HeartBreaker, Video footage. 2006. http://www.tevs.eu/project_vmv06.html

[TZMS04] C. Theobalt, G. Ziegler, M. Magnor, H.-P. Seidel. Model-Based Free Viewpoint Video: Acquisition, Rendering, and Encoding. Proceedings of Picture Coding Symposium (PCS 2004), San Francisco, USA, December 2004.

[Ups90] S. Upstill. The Renderman Companion: A Programmer´s Guide to Realistic Computer Graphics. Published by Addison-Wesley, 1990.

[Ura06] Y. Uralsky. DX10: Practical metaballs and implicit surfaces. GameDevelopers conference, 2006.

[Var08a] Various authors. List of home computers by video hardware. Wikipedia online encyclopedia. http://en.wikipedia.org/wiki/List_of_home_computers_by_video_hardware

[Var08b] Various authors. MOS Technology VIC-II. Wikipedia online encyclopedia. http://en.wikipedia.org/wiki/MOS_Technology_VIC-II

[Var09a] Various authors. Graphics processing units and their history. Wikipedia online encyclopedia. http://en.wikipedia.org/wiki/Graphics_processing_unit

[Var09b] Various authors. MasPar. Wikipedia online encyclopedia. http://en.wikipedia.org/wiki/MasPar

[Vas07] C. N. Vasconcelos, A. Sá, P. C. Carvalho, M. Gattass. Using Quadtrees for Energy Minimization Via Graph Cuts. Proceedings of VMV - 12th Vision, Modeling, and Visualization Workshop, pp. 71-80. Saarbrücken, Germany 2007.

[Vas08] C. N. Vasconcelos, A. Sá, P. C. Carvalho, M. Gattass. QuadN4tree: A GPU-Friendly Quadtree Leaves Neighborhood Structure. Proceedings of Computer Graphics International Conference (CGI) 2008. Istanbul, Turkey 2008.

[VD08] V. Volkov, J.W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Article No. 31, 2008.

[VecCont] T. Annen. Complementary video material to Vector Field Contours. http://www.mpi-inf.mpg.de/~tannen/videos/Gi2008VFCVideos.zip

[VYV98] L. van Vliet, I. Young, P. Verbeek. Recursive Gaussian derivative filters. Proceedings of the International Conference on Pattern Recognition, Brisbane, p. 509–514. 1998.

[WAA*00] D. Wood, D. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. Salesin, W. Stuetzle. Surface light fields for 3D photography. Proceedings of ACM Conference on Computer Graphics (SIGGRAPH-2000), New Orleans, USA, pp. 287–296, July 2000.

[WJV*05] B. Wilburn, N. Joshi, V. Vaish, E. Talvala, E. Antunez, A. Barth, A. Adams, M. Levoy, M. Horowitz. High Performance Imaging Using Large Camera Arrays. Proceedings of ACM SIGGRAPH 2005.
http://www-graphics.stanford.edu/projects/array/

[WK04] J. Woetzel, R. Koch. Real-time multi-stereo depth estimation on GPU with approximative discontinuity handling. Proceedings of 1st European Conference on Visual Media Production (CVMP 2004), pp. 245-254, March 2004.

[WKE99] R. Westermann, L. Kobbelt, T. Ertl. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. The Visual Computer, Issue 15(2), pp. 100-111, 1999.

[WTS*05] T. Weinkauf, H. Theisel, K. Shi, H.-C. Hege, and H.-P. Seidel. Topological simplification of 3D vector fields by extracting higher order critical points. Proceedings of IEEE Visualization 2005, pp. 559–566, 2005.

[YKP05] X. Ye, D. Kao, A. Pang. Strategy for seeding 3D streamlines. In IEEE Visualization VIS 05, pp. 163–170, 2005.

[YP05] R. Yang, M. Pollefeys. A versatile stereo implementation on commodity graphics hardware. Real-Time Imaging, 11(1), pp. 7–18, 2005.

[ZDTS07] G. Ziegler, R. Dimitrov, C. Theobalt, H-P. Seidel. Real-time Quadtree Analysis using HistoPyramids. Proc. of SPIE Electronic Imaging conference in San Jose, USA, Jan 2007.

[ZHMS05] G. Ziegler, L. Heidenreich, M. Magnor, and H.-P. Seidel. GeoCast: Unifying depth video with camera meta-data. In 2nd Workshop on Immersive Communication and Broadcast Systems, Berlin, Germany, October 2005.

[ZHW*08] K. Zhou; Q. Hou, R. Wang, B. Guo. Real-Time KD-Tree Construction on Graphics Hardware. Technical Reports of Microsoft Research, MSR-TR-2008-52, April 2008.

[ZKHK03] C. Zach, A. Klaus, M. Hadwiger, K. Karner. Accurate dense stereo reconstruction using graphics hardware. Technical Reports of VRVis, Vienna, Austria. 2003.

[ZLA*04] G. Ziegler, H. Lensch, N. Ahmed, M. Magnor, H.-P. Seidel. Multi-Video Compression in Texture Space. In: 11th IEEE International Conference on Image Processing (ICIP 2004), Singapore, pp. 2467-2470, 2004.

[ZLM*04] G. Ziegler, H. Lensch, M. Magnor, H.-P. Seidel. Multi-Video Compression in Texture Space using 4D SPIHT. 6th IEEE Workshop on Multimedia Signal Processing, Siena, Italy, pp. 39-42, 2004.

[ZO04] J. Zhang, C. Owen. Octree-based Animated Geometry Compression. Proceedings of the Conference on Data Compression, pp. 508, 2004.

[ZTTS06] G. Ziegler, A. Tevs, C. Theobalt, H.-P. Seidel. GPU Point List Generation through Histogram Pyramids. Tech. Rep. MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.

[Vol] Volvis volume dataset archive.
http://www.volvis.org