Advanced Rendering Techniques

GPU Pro

Edited by Wolfgang Engel

GPU Pro

GPU Pro

Advanced Rendering Techniques

Edited by Wolfgang Engel

A K Peters, Ltd.
Natick, Massachusetts

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2010 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20140602

International Standard Book Number-13: 978-1-4398-6553-8 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

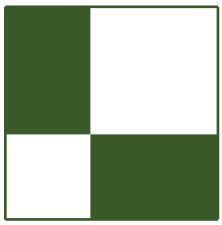
For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

For my wife, Katja, and my kids, Anna, Emma, and Lena.



Contents

Acknowledgments xxi

Web Materials xxiii

I Mathematics 1
Sam Martin, editor

1 GPU Color Quantization 3
Chi Sing Leung, Tze-Yui Ho, and Yi Xiao

1.1 Introduction . . . 3
1.2 The Self-Organizing Map . . . 4
1.3 Color Quantization with SOM . . . 6
1.4 Implementation . . . 7
1.5 Results and Discussion . . . 10
1.6 Conclusion . . . 12
Bibliography . . . 13

2 Visualize Your Shadow Map Techniques 15
Fan Zhang, Chong Zhao, and Adrian Egli

2.1 Introduction . . . 15
2.2 Article Overview . . . 15
2.3 Scope of Analysis . . . 16
2.4 Aliasing Revisited . . . 17
2.5 A Representation of Aliasing Errors . . . 19
2.6 Visualization . . . 21

2.7 Mismatch of View Direction and Warping Direction 25
2.8 Conclusion 27
2.9 Supplementary Materials 27
2.10 Acknowledgments 28
Bibliography 28

II Geometry Manipulation 31
Natalya Tatarchuk, editor

1 As Simple as Possible Tessellation for Interactive Applications 33
Tamy Boubekeur

1.1 Basic Phong Tessellation Operator 34
1.2 Extension to Quads 36
1.3 Results and Discussion 38
Bibliography 38

2 Rule-Based Geometry Synthesis in Real-Time 41
Milán Magdics and Gergely Klár

2.1 Introduction 41
2.2 Building Up the Scene with Procedures 42
2.3 L-systems and the PGI 42
2.4 Model Details and Implementation 44
2.5 Interaction with the Procedural Scene 59
2.6 Results 62
2.7 Conclusion 65
Bibliography 65

3 GPU-Based NURBS Geometry Evaluation and Rendering 67
Graham Hemingway

3.1 A Bit of NURBS Background 67
3.2 Related Work 71
3.3 NURBS Surface and Curve Evaluation 72
3.4 Trimmed NURBS Surface Evaluation 77
3.5 Results and Conclusion 82
Bibliography 84

4 Polygonal-Functional Hybrids for Computer Animation and Games 87
D. Kravtsov, O. Fryazinov, V. Adzhiev, A. Pasko, and P. Comninos

4.1 Introduction 87
4.2 Background 88

4.3 Working with FRep Models Using the GPU 94
4.4 Applications 103
4.5 Tools 110
4.6 Limitations 111
4.7 Conclusion 112
4.8 Source Code 112
Bibliography 113

III Rendering Techniques 115
Wessam Bahnassi, editor

1 Quadtree Displacement Mapping with Height Blending 117
Michal Drobot

1.1 Overview 117
1.2 Introduction 119
1.3 Overview of Ray-Tracing Algorithms 120
1.4 Quadtree Displacement Mapping 125
1.5 Self-Shadowing 134
1.6 Ambient Occlusion 138
1.7 Surface Blending 139
1.8 General Advice 146
1.9 Conclusion 147
Bibliography 148

2 NPR Effects Using the Geometry Shader 149
Pedro Hermosilla and Pere-Pau Vázquez

2.1 Introduction 149
2.2 Previous Work 149
2.3 Silhouette Rendering 151
2.4 Pencil Rendering 159
2.5 Acknowledgments 164
Bibliography 164

3 Alpha Blending as a Post-Process 167
Benjamin Hathaway

3.1 Introduction 167
3.2 The Alternatives 168
3.3 The Source Artwork 169
3.4 Initial Attempts 170
3.5 The Screen-Space Alpha Mask 170

3.6 Alpha Reference Issues . . . 179
3.7 Rendering Pipeline Integration . . . 181
3.8 Conclusion . . . 181
3.9 Demo . . . 182
3.10 Acknowledgments . . . 183
3.11 Source Code . . . 183
Bibliography . . . 183

4 Virtual Texture Mapping 101 185
Matthäus G. Chajdas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre

4.1 Introduction . . . 185
4.2 Virtual Texture Mapping . . . 185
4.3 Implementation Details . . . 189
4.4 Conclusion . . . 193
4.5 Shader Code . . . 193
4.6 Acknowledgments . . . 194
Bibliography . . . 194

IV Global Illumination 197
Carsten Dachsbacher, editor

1 Fast, Stencil-Based Multiresolution Splatting for Indirect Illumination 199
Chris Wyman, Greg Nichols, and Jeremy Shopf

1.1 Quick Review: Instant Radiosity . . . 200
1.2 Quick Review: Reflective Shadow Maps . . . 201
1.3 Multiresolution Splatting . . . 202
1.4 Fast Stencil-Based Multiresolution Splatting . . . 206
1.5 Results and Analysis . . . 210
1.6 Conclusion . . . 212
1.7 Demo and Source . . . 212
Bibliography . . . 213

2 Screen-Space Directional Occlusion 215
Thorsten Grosch and Tobias Ritschel

2.1 Introduction . . . 215
2.2 Screen-Space Ambient Occlusion . . . 216
2.3 Screen-Space Directional Occlusion . . . 218
2.4 Interleaved Sampling . . . 227
2.5 Discussion . . . 228

2.6 Conclusion . . . 229
Bibliography . . . 229

3 Real-Time Multi-Bounce Ray-Tracing with Geometry Impostors 231
Péter Dancsik and László Szécsi
3.1 Geometry Impostors . . . 232
3.2 Intersection Computation with an Environment Distance Impostor 234
3.3 Ray-Object Intersection Using Distance Impostors . . . 236
3.4 Ray-Object Intersection Using Height Maps . . . 236
3.5 Tracing Multiple Refractions within a Single Object . . . 238
3.6 Multiple Ray Bounces with Object Impostors Only . . . 238
3.7 Multiple Ray Bounces with Environment Distance Impostors Only 238
3.8 Combination of Environment and Object Impostors . . . 239
3.9 Caustics and Shadows . . . 240
3.10 Example Application: Glass Chess . . . 241
3.11 Example Application: Alien Pool . . . 242
3.12 Conclusion . . . 243
Bibliography . . . 243

V Image Space 245
Christopher Oat, editor

1 Anisotropic Kuwahara Filtering on the GPU 247
Jan Eric Kyprianidis, Henry Kang, and Jürgen Döllner
1.1 Introduction . . . 247
1.2 Kuwahara Filtering . . . 248
1.3 Generalized Kuwahara Filtering . . . 251
1.4 Anisotropic Kuwahara Filtering . . . 255
1.5 Conclusion . . . 263
Bibliography . . . 263

2 Edge Anti-aliasing by Post-Processing 265
Hugh Malan
2.1 Introduction . . . 265
2.2 Finding the Nearest Silhouette Edge . . . 268
2.3 The Post-Process . . . 271
2.4 Refinements . . . 273
2.5 GPU Implementation and Results . . . 275
2.6 Results . . . 278
2.7 Other Applications . . . 280
2.8 Conclusion . . . 288
Bibliography . . . 289

3 Environment Mapping with Floyd-Steinberg Halftoning 291
László Szirmay-Kalos, László Szécsi, and Anton Penzov
3.1 Parametrization of the Environment Map . . . 292
3.2 Importance Sampling . . . 294
3.3 Proposed Solution . . . 296
3.4 Conclusion . . . 303
Bibliography . . . 303

4 Hierarchical Item Buffers for Granular Occlusion Culling 305
Thomas Engelhardt and Carsten Dachsbacher
4.1 Introduction . . . 305
4.2 Hierarchical Item Buffers . . . 306
4.3 Application . . . 311
4.4 Results . . . 312
4.5 Conclusion . . . 314
Bibliography . . . 314

5 Realistic Depth of Field in Postproduction 315
David Illes and Peter Horvath
5.1 Depth-of-Field Equations . . . 315
5.2 Camera Lens Simulation . . . 317
5.3 Exposure Simulation . . . 317
5.4 CUDA-Accelerated Computation . . . 321
5.5 Results . . . 323
5.6 Conclusion . . . 325
Bibliography . . . 325

6 Real-Time Screen Space Cloud Lighting 327
Kaori Kubota
6.1 Introduction . . . 327
6.2 Implementation . . . 328
6.3 Results . . . 331
6.4 Extensions . . . 332
6.5 Conclusion . . . 333
Bibliography . . . 333

7 Screen-Space Subsurface Scattering 335
Jorge Jimenez and Diego Gutierrez
7.1 Introduction . . . 335
7.2 The Texture-Space Approach . . . 336
7.3 The Screen-Space Approach . . . 339

7.4 Model Preparation 347
7.5 Discussion of Results 347
7.6 Conclusion 350
7.7 Acknowledgments 351
Bibliography 351

VI Handheld Devices 353
Kristof Beets, editor

1 Migration to OpenGL ES 2.0 355
Ken Catterall

1.1 Introduction 355
1.2 Key API Differences 356
1.3 The Shader Language 360
1.4 Initialization 362
1.5 Basic Shaders: Implementing the Fixed-Function Pipeline 363
1.6 Advanced Effects Made Easy 368
1.7 Conclusion 373
1.8 Acknowledgments 373

2 Touchscreen-Based User Interaction 375
Andrea Bizzotto

2.1 Introduction 375
2.2 Motion Estimation 375
2.3 Position Prediction 378
2.4 Application: Controlling a Camera in a Spherical Coordinate System 379
2.5 Algorithm Overview 381
2.6 Conclusion 383
2.7 Acknowledgments 383

3 iPhone 3GS Graphics Development and Optimization Strategies 385
Andrew Senior

3.1 Software Development Kits 385
3.2 General Optimization Strategies 389
3.3 Conclusion 395
Bibliography 395

4 Optimizing a 3D UI Engine for Mobile Devices 397
Hyunwoo Ki
4.1 Overview . . . 398
4.2 Optimization Methods . . . 399
4.3 Conclusion . . . 410
Bibliography . . . 410

VII Shadows 413
Wolfgang Engel, editor

1 Fast Conventional Shadow Filtering 415
Holger Gruen
1.1 Overview . . . 415
1.2 Introduction . . . 415
1.3 Uniform Shadow Filtering . . . 416
1.4 Separable Shadow Filters . . . 420
1.5 Nonseparable Unique Weights per PCF Result . . . 428
1.6 Advanced Shadow Filtering Techniques . . . 430
Bibliography . . . 445

2 Hybrid Min/Max Plane-Based Shadow Maps 447
Holger Gruen
2.1 Overview . . . 447
2.2 Introduction . . . 447
2.3 Construction of an HPSM . . . 448
2.4 Using an HPSM . . . 453
2.5 Other Uses for the HPSM . . . 454
Bibliography . . . 454

3 Shadow Mapping for Omnidirectional Light Using Tetrahedron Mapping 455
Hung-Chien Liao
3.1 Shadow Mapping . . . 455
3.2 Tetrahedron Shadow Mapping . . . 455
3.3 Optimization . . . 464
3.4 Lookup Map . . . 465
3.5 Conclusion . . . 470
Bibliography . . . 475

4 Screen Space Soft Shadows 477
Jesus Gumbau, Miguel Chover, and Mateu Sbert
4.1 Introduction . . . 477
4.2 Previous Work . . . 479
4.3 Screen Space Soft Shadows . . . 480
4.4 Results . . . 485
4.5 Conclusion . . . 488
Bibliography . . . 490

VIII 3D Engine Design 493
Wolfgang Engel, editor

1 Multi-Fragment Effects on the GPU Using Bucket Sort 495
Meng-Cheng Huang, Fang Liu, Xue-Hui Liu, and En-Hua Wu
1.1 Introduction . . . 495
1.2 Design of Bucket Array . . . 496
1.3 The Algorithm . . . 497
1.4 Multi-pass Approach . . . 498
1.5 The Adaptive Scheme . . . 498
1.6 Applications . . . 503
1.7 Conclusions . . . 507
Bibliography . . . 507

2 Parallelized Light Pre-Pass Rendering with the Cell Broadband Engine 509
Steven Tovey and Stephen McAuley
2.1 Introduction . . . 509
2.2 Light Pre-Pass Rendering . . . 510
2.3 The PLAYSTATION3 and the CBE . . . 512
2.4 GPU/SPE Synchronization . . . 513
2.5 The Pre-pass . . . 515
2.6 The Lighting SPE Program . . . 516
2.7 The Main Pass . . . 523
2.8 Conclusion . . . 524
2.9 Further Work . . . 524
2.10 Acknowledgments . . . 526
Bibliography . . . 526

3 Porting Code between Direct3D9 and OpenGL 2.0 529
Wojciech Sterna
3.1 Introduction . . . 529
3.2 General Issues . . . 529
3.3 Resources Management . . . 533
3.4 Notes on Debugging . . . 538
3.5 A Word on Sample Application . . . 539
3.6 Conclusion . . . 539
Bibliography . . . 540

4 Practical Thread Rendering for DirectX 9 541
David Pangerl
4.1 Abstract . . . 541
4.2 Introduction . . . 541
4.3 Solution . . . 542
4.4 Conclusion . . . 546
4.5 Future Work . . . 546

IX Game Postmortems 547
Matthias Wloka, editor

1 Stylized Rendering in *Spore* 549
Shalin Shodhan and Andrew Willmott
1.1 Filter Chain System . . . 549
1.2 Fun with Filters . . . 554
1.3 Cheats . . . 560

2 Rendering Techniques in *Call of Juarez: Bound in Blood* 561
Pawel Rohleder and Maciej Jamrozik
2.1 Introduction . . . 561
2.2 Deferred Shading . . . 561
2.3 Single Frame of CoJ:BiB, The Black Magic . . . 563
2.4 What Else Do We Have? . . . 567
2.5 Conclusion . . . 568
2.6 Acknowledgments . . . 569
Bibliography . . . 569

3 Making it Large, Beautiful, Fast, and Consistent: Lessons Learned Developing *Just Cause 2* 571
Emil Persson

3.1 Introduction . . . 571
3.2 Making it Large and Beautiful . . . 571
3.3 Making it Consistent . . . 585
3.4 Making it Fast . . . 591
3.5 Conclusion and Future Work . . . 595
3.6 Acknowledgments . . . 595
Bibliography . . . 596

4 Destructible Volumetric Terrain 597
Marek Rosa

4.1 Introduction . . . 597
4.2 Converting Custom 3D Models into Voxel Representations . . . 598
4.3 Memory Requirements for Large Voxel Maps . . . 600
4.4 Polygonization from Voxels to Triangles . . . 601
4.5 Seamless Texture-Mapping for Free-Form Surfaces . . . 602
4.6 Reducing Repeated Texture Patterns . . . 603
4.7 Tangent Space Calculation in the Pixel Shader . . . 604
4.8 Multi-material Voxel Maps . . . 604
4.9 Level of Detail . . . 606
4.10 Ambient Light . . . 607
4.11 Conclusion . . . 608
Bibliography . . . 608

X Beyond Pixels and Triangles 611
Sebastien St-Laurent, editor

1 Parallelized Implementation of Universal Visual Computer 613
Tze-Yui Ho, Ping-Man Lam, and Chi-Sing Leung

1.1 Introduction . . . 613
1.2 Background . . . 614
1.3 CNN Implementation . . . 615
1.4 Results . . . 620
1.5 Conclusion . . . 622
1.6 Acknowledgments . . . 622
Bibliography . . . 622

2 Accelerating Virtual Texturing Using CUDA 623
Charles-Frederik Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle

2.1 Introduction . . . 624
2.2 Implementing Virtual Texturing . . . 626
2.3 Rendering with Virtual Texturing . . . 628
2.4 GPU-Based Acceleration . . . 631
2.5 Results . . . 640
2.6 Conclusion . . . 641
2.7 Acknowledgments . . . 641
Bibliography . . . 641

3 Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels 643
Cyril Crassin, Fabrice Neyret, Miguel Sainz, and Elmar Eisemann

3.1 Introduction . . . 643
3.2 Voxel Representations . . . 644
3.3 The GigaVoxels Approach . . . 645
3.4 Data Structure . . . 646
3.5 Rendering . . . 650
3.6 Out-of-Core Data Management . . . 656
3.7 Octree-Based Synthesis . . . 670
3.8 Voxel Object Instancing . . . 672
3.9 MipMap-Based Blur Effects . . . 673
3.10 Conclusion . . . 676
Bibliography . . . 677

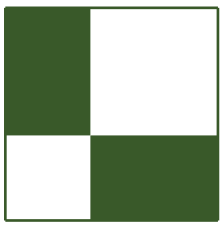
4 Spatial Binning on the GPU 679
Christopher Oat, Joshua Barczak, and Jeremy Shopf

4.1 Introduction . . . 679
4.2 Binning . . . 680
4.3 Applications . . . 684
4.4 Results . . . 687
4.5 Conclusion . . . 690
4.6 Acknowledgments . . . 691
Bibliography . . . 691

5 Real-Time Interaction between Particles and the Dynamic Mesh on the GPU 693
Vlad Alexandrov
5.1 Introduction . . . 693
5.2 Process Overview . . . 693
5.3 Sorting the Triangles . . . 694
5.4 Building the Buffer Map . . . 699
5.5 Addressing the Buffer . . . 702
5.6 Performance . . . 705
5.7 Conclusion . . . 705
Bibliography . . . 706

Section Editors 707

Contributors 709

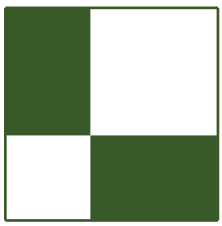


Acknowledgments

The GPU Pro: Advanced Rendering Techniques book series covers ready-to-use ideas and procedures that can solve many of your daily graphics-programming challenges. The first book in the series wouldn't have been possible without the help of many people. First of all I would like to thank the section editors for the fantastic job they did. The work of Wessam Bahnassi, Sebastien St. Laurent, Natalya Tatarchuk, Carsten Dachsbacher, Matthias Wloka, Christopher Oat, and Sam Martin ensures that the quality of the series stands up to the expectations of our readers. The great cover screenshots were taken from *Call of Juarez: Bound in Blood*. I would like to thank Techland for allowing us to use those shots. You will also find a postmortem of this game in the book. The team at A K Peters made the whole project happen. I want to thank Alice and Klaus Peters, Charlotte Henderson, and the whole production team, who took the articles and made them into a book. Special thanks go out to our families and friends, who spent many evenings and weekends during the long book production cycle without us. I hope you have as much fun reading the book as we had creating it.

—Wolfgang Engel

P.S. Plans for an upcoming *GPU Pro 2* are already in progress. Any comments, proposals, or suggestions are highly welcome (wolfgang.engel@gmail.com).



Web Materials

Example programs and source code to accompany some of the chapters are available at http://www.akpeters.com/gpupro. The directory structure closely follows the book structure by using the chapter number as the name of the subdirectory. You will need to download the DirectX August 2009 SDK.

General System Requirements

To use all of the files, you will need:

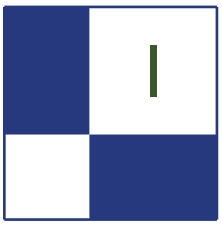
- The DirectX August 2009 SDK
- OpenGL 1.5-compatible graphics card
- A DirectX 9.0 or 10-compatible graphics card
- Windows XP with the latest service pack; some require VISTA or Windows 7
- Visual C++ .NET 2008
- 2048 MB RAM
- The latest graphics card drivers

Updates

Updates of the example programs will be periodically posted.

Comments and Suggestions

Please send any comments or suggestions to wolf@shaderx.com.



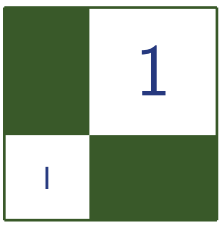
I Mathematics

In order to drive the limited capabilities of an early GPU, a programmer may have required knowledge of linear algebra, basic lighting and animation techniques, and the occasional Bézier spline. Modern GPUs are significantly more general purpose than their predecessors, yet the mathematical models they are able to run grow increasingly complex. As our techniques become less specialized, we benefit from studying generalized solutions; and it is these techniques that are the focus of the Mathematics section.

In this edition, Chi Sing Leung et al. present a unique approach to quantization in "GPU Color Quantization." They demonstrate how the high quality vector quantization algorithm, self-organizing maps (SOM), can be parallelized and run on a GPU. They show how the unique properties of SOM can provide a significant quality improvement over the more traditional octree algorithm. The quantization technique is very general and can be easily applied to other vector quantization problems.

Fan Zhang et al. explore the delicate nature of aliasing errors in "Visualize Your Shadow Map Techniques." They use a scene-independent model for measuring the all-too-familiar aliasing errors that can be expected from any shadow mapping technique. Their results provide insight into the behavior of several common shadowing techniques, and explain why warping techniques can sometimes produce unexpected results.

—Sam Martin



GPU Color Quantization

Chi Sing Leung, Tze-Yui Ho, and Yi Xiao

1.1 Introduction

Color quantization [Verevka 95, Kruger 94, Clark 96] is an example of the more general problem of vector quantization (VQ). It maps 24-bit or high dynamic range colors to a finite set of colors, which we call the color palette. The simplest way to quantize a color image is to pick a set of uniformly distributed colors for the pallet and replace each pixel by its nearest pallet entry. One such pallet is the well known set of "web safe" colors, and "nearest" is often measured by Euclidean distance. Although simple, this approach is known to produce poor quality results. Much better results can be obtained by carefully constructing the color pallet using information obtained from the image to the quantized.

There are several algorithms to create a useful color palette, including training-based methods such as *Self-Organizing Maps* (SOM) [Kohonen 01, Chang et al. 05], *Linde-Buzo-Gray* (LBG) [Verevka 95, Linde et al. 80], and the *octree method* [Clark 96]. In terms of mean square distortion, the training based methods usually give a much better quantization result. However, training based methods can also be time consuming. While the mean square distortion of LBG and SOM are similar [Leung and Chan 97], the SOM-generated pallet has a very interesting ordering property. To obtain the ordering property, we impose a neighborhood structure among the palette entries before training. After training, when two palette entries are neighbors of each other, they will have a similar color.

This article presents an efficient GPU implementation for the SOM training algorithm. We apply our approach to the problem of color quantization, and expect our results to be applicable to other vector quantization problems. Our implementation relies on the random write ability of the vertex shader and uses the GPU as the main computing processor. We review the training algorithm of SOM in Section 1.2. Color quantization through SOM is presented in

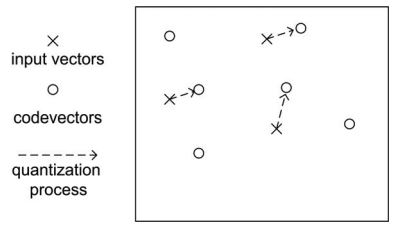


Figure 1.1. The quantization process in two-dimensional space.

Section 1.3. Section 1.4 presents our GPU implementation and the corresponding CG scripts. Section 1.5 discusses the results, the ordering property of SOM, and the applications of the ordering property. Section 1.6 presents our conclusions and summarizes the article.

1.2 The Self-Organizing Map

In vector quantization, we aim to map an input vector in a k-dimensional space to one of a finite set of codevectors in a codebook, $\mathcal{Y} = \{\vec{c}_1, \cdots, \vec{c}_M\}$. The codebook $\mathcal{Y}$ will be used to partition the k-D space into M regions $\Omega = \{\Omega_1, \cdots, \Omega_M\}$. To perform the quantization, we approximate each input vector by its nearest codevector as shown in Figure 1.1, thereby allowing us to store the index of the closest codevector in place of the original vector. In the SOM learning scheme [Leung and Chan 97], prior to training, a neighborhood structure represented by a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, is imposed on the codebook, where $\mathcal{V} = \{v_1, \cdots, v_M\}$ is a set of vertices and $\mathcal{E}$ is the set of edges in this graph. In this representation, each vertex v_i is associated with a codevector $\vec{c}_i$. We will use the following definitions when we discuss the graph:

Definition 1.1. If the codevector $\vec{c}_i$ is defined to be a neighbor of codevector $\vec{c}_{i'}$, the two corresponding vertices, v_i and $v_{i'}$, in the graph are joined by an edge with a weighted value of 1.

Definition 1.2. The neighborhood distance between $\vec{c}_i$ and $\vec{c}_{i'}$ is the length of the shortest path between v_i and $v_{i'}$ in the graph $\mathcal{G}$.

Definition 1.3. A codevector $\vec{c}_i$ is a neighbor of $\vec{c}_{i'}$ in a u-sized neighborhood, if the neighborhood distance between the two codevectors is less than or equal to u.

Definition 1.4. For a codevector $\vec{c}_i$, the collection of indices of its neighbors in a neighborhood of size u is denoted as $I_i(u)$.

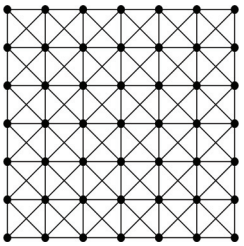


Figure 1.2. The regular grid neighborhood structure of the vertices in a two-dimensional SOM. Note that the graph does not reflect any geometric information of the codevectors in the data space.

Figure 1.2 shows the two-dimensional neighborhood structure used in our implementation of SOM. This structure can be used for mapping a high-dimensional data vector to two discrete indices. Since the sequential SOM learning algorithm [Kohonen 01] is not suitable for parallelization, this article considers the batch mode SOM [Kohonen 01]. Given a dataset $\mathcal{D} = \{\vec{x}_1, \cdots, \vec{x}_N\}$ and the initial codebook $\mathcal{Y}(0) = \{\vec{c}_1(0), \cdots, \vec{c}_M(0)\}$, the batch mode SOM is summarized as follows:[1]

1. A maximum number of iterations T is assigned, and the iteration index t is set to zero. The size of neighborhood u_t is a value decreasing along with t, which is given by
$$u_t = \begin{cases} 2, & \text{if } 0 \le t < T/8, \\ 1, & \text{if } T/8 \le t < T/2, \\ 0, & \text{otherwise.} \end{cases}$$

2. Create M empty sets, $\Psi_{i'}$, *i.e.*, $\Psi_i = \varnothing$, for all $i = 1, \cdots, M$.

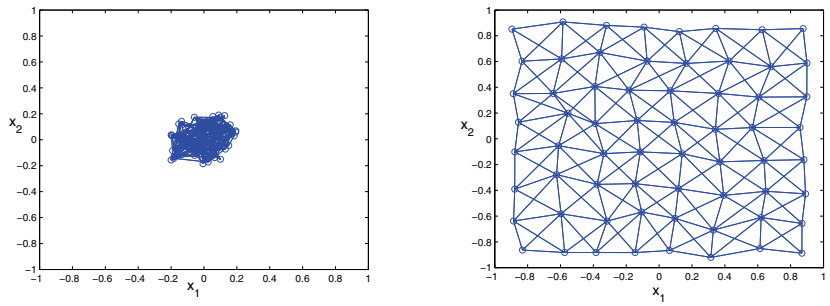
3. For each training vector $\vec{x}_j$, find its nearest codevector $\vec{c}_{i*}(t)$.

   Put $\vec{x}_j$ into the sets $\Psi_{i'}$'s, where $i' \in I_{i*}(u_t)$ as described in Definition 4. This will collect the training vectors to the sets whose corresponding codevectors have neighborhood distances to the nearest codevector less than u_t.

4. Update the codebook:
$$\vec{c}_i(t+1) = \frac{1}{|\Psi_i|} \sum_{\vec{x}_j \in \Psi_i} \vec{x}_j \,.$$

5. Set t=t+1 and go to Step 2 unless t has reached T.

[1]The initial codebook is initialized with random values.



(a) The initial codevectors. (b) The trained codevectors.

Figure 1.3. A typical run of a batch model SOM. The figure shows the codevectors in the data space. The neighborhood structure of the SOM is an 8×8 two-dimensional grid structure. To indicate the neighborhood structure of the SOM, we draw edges between codevectors if they are defined as neighbors to each other. In (a), the positions of the initial codevectors do not follow the layout of the neighborhood. However, as shown in (b), after training the codevectors are organized in the data space. Two trained codevectors are very close in the data space if they are neighbors to each other in the graph.

Due to the ordering property of the resulting codebook, the Euclidean distance between two trained codevectors is usually small if they are neighbors of each other in the graph $\mathcal{G}$. As demonstrated in Figure 1.3, while the initial codevectors do not form any ordering in the data space as shown in Figure 1.3(a), the trained codevectors form a helpful ordering as shown in Figure 1.3(b).

1.3 Color Quantization with SOM

In this application of vector quantization, we use a two-dimensional regular structured SOM to quantize a full color image. The training vectors, $\mathcal{X} = \{\vec{x}_1, \cdots, \vec{x}_N\}$, are the RGB values of the pixels in the image. Hence, the dimension k of the data space is 3. The codebook will become our image color pallet and is a finite set of RGB values $\mathcal{Y} = \{\vec{c}_1, \cdots, \vec{c}_M\}$. The neighborhood structure of the SOM is a two-dimensional regular grid structure as shown in Figure 1.2. With this arrangement, the trained color palette can be organized as a two-dimensional texture. After quantization, the color of each pixel $\vec{x}_j$ will be replaced by two ordered indices $i*$ whose corresponding palette entry $\vec{c}_{i*}$ is the closest palette entry to the original pixel $\vec{x}$. Note that the use of two indices, and the two-dimensional layout of the color pallet, provides additional structure at no extra cost that would be lost in an equivalent 1D color palette.

1.4 Implementation

We use the client server model to describe our implementation. In a training iteration, each pixel in the image can be considered as a client, and each palette entry can be considered as a service counter. Each pixel determines its nearest palette entry and generates a request to the nearest palette entry, as well as a number of requests to the palette entries neighborhood. When a pixel arrives at a palette entry, the palette entry performs an accumulation and records the number of arrived pixels. When all pixels are served, each entry holds the sum of the arrived pixel colors and the number of arrived pixels. Finally, we calculate the new color palette from the sums.

Suppose there are N training pixels and M colors in the color palette. All the training pixels are uploaded to GPU as a display list of points where pixels are represented by their vertex positions. The initial color palette is assigned with random values and then uploaded to GPU as uniform variables. A four-channel floating-point texture of resolution $\sqrt{M} \times \sqrt{M}$ is created to hold the accumulation of training pixel colors, as shown in Figure 1.4. We call this texture the accumulation color palette texture.

The user then needs to set an initial point size. In OpenGL, we can use `glPointsize()` to set the point size. For SOM, the point size should be set to $2u_t + 1$. If the point size is assigned to 1 for all training iterations, the resulting color palette would be a LBG color palette. In this case, the ordering property in the trained color palette would be lost. The flow diagram of our GPU implementation is shown in Figure 1.5. Within an iteration, for each training pixel, the vertex shader finds the texel position of the nearest palette entry. A point of size $2u_t + 1$ is rasterized at the texel position. Each vertex will generate a number of updates on the accumulation color palette texture as shown in Figure 1.6. In the

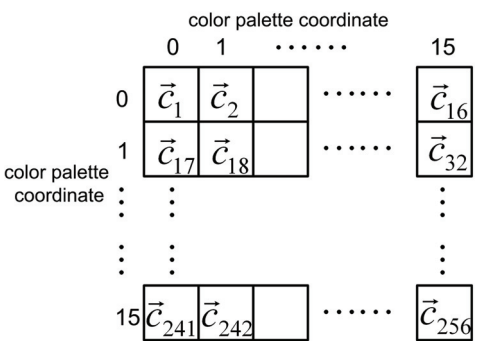


Figure 1.4. The accumulation color palette texture.

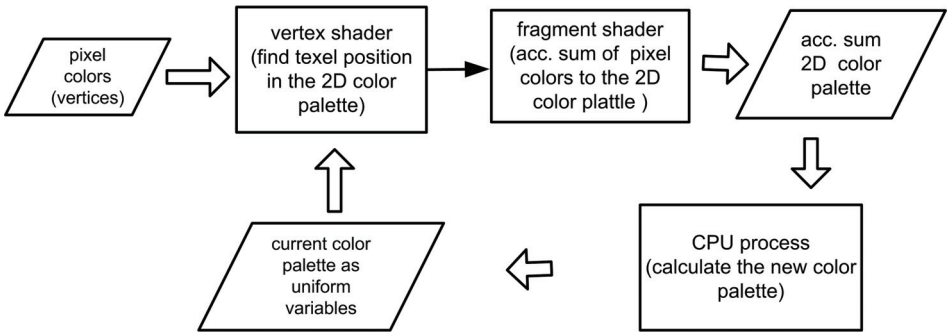


Figure 1.5. The flow of our shader implementation.

fragment shader, the training pixel and a constant "1" are passed to the graphics pipeline directly for accumulation. The constant "1" is there for counting the arrived pixels. This process generates all the accumulation data. Finally, we export the accumulation color palette texture to CPU for updating the color palette.

Our training is an iterative process. Prior to each iteration, the codebook texture is zero-initialized. The color palette is passed to the vertex shader as uniform variables. During an iteration, the training pixels are drawn as a set of points using the display list. In the vertex shader, shown in Listing 1.1, the pixel color is compared with all representative colors to find the index of the nearest representative color. We assign the output position of vertex shader depending on this index.

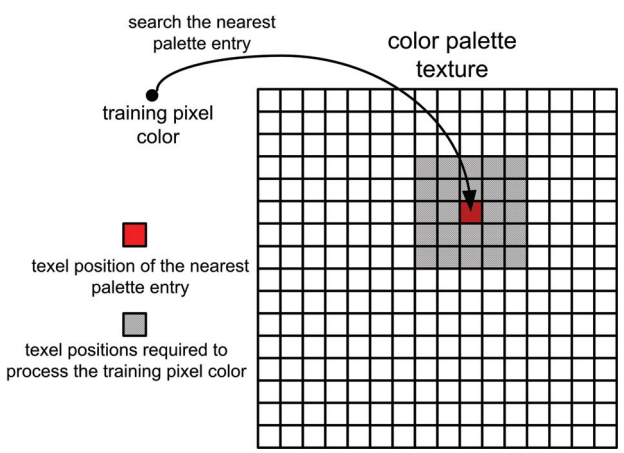


Figure 1.6. A training pixel will generate a number of updates on the accumulation color palette texture. In this example, the point size is 5.

```
void cgfl_vp_som(
  float3 ac : POSITION,
  out float4 hPosition : POSITION,
  out float3 bc : COLOR,
  uniform float3 codebook[codebook_h][codebook_w]
){
  int i, j;
  int2 imin;
  float nd, dd;
  float3 dv;

  nd = FLT_MAX;
  for ( i=0; i<codebook_h; i++ )
  for ( j=0; j<codebook_w; j++ )
  {
    dv = ac-codebook[i][j];
    dd = dot(dv,dv);
    if( nd>dd )
    {
      nd=dd;
      imin=int2( j, i );
    }
  }

  hPosition = float4( float(2*imin.x)/codebook_w-1,
  float(2*imin.y)/codebook_h-1, 0,1);

  bc=ac;
}
```

Listing 1.1. The vertex shader of SOM training for locating the nearest representative color index.

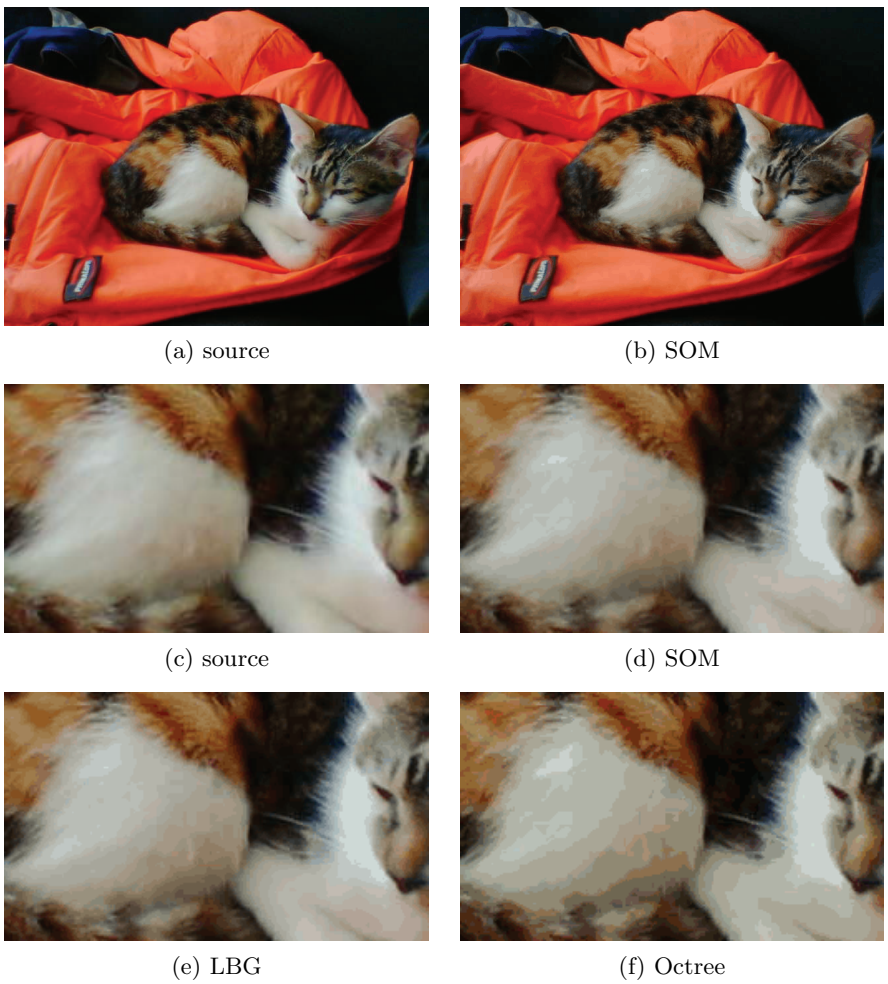
```
  float3 bc    : COLOR,
  out float4 color : COLOR
){
  color = float4(bc,1);
}
```

Listing 1.2. The fragment shader of SOM training for the accumulation and the counting of training pixel colors.

Afterward, the output position along with the training pixel color are passed to fragment shader to perform the accumulation. By using the hardware blending functionality (available since Geforce 8 or equivalent), the implementation of the accumulation in the fragment shader is simple. This is shown in Listing 1.2.

1.5 Results and Discussion

We have tested various quantization schemes, including SOM, LBG, and Octree. The Octree implementation used was taken from [Priester 04]. The source image is a 24-bit color image of resolution 640×480 as shown in Figure 1.7(a). It is quantized to 8-bit color using the various schemes. The quantized images are shown in Figure 1.7(d)(e)(f). Table 1.1 summarizes their peak signal to noise ratios (PSNRs).



(a) source (b) SOM

(c) source (d) SOM

(e) LBG (f) Octree

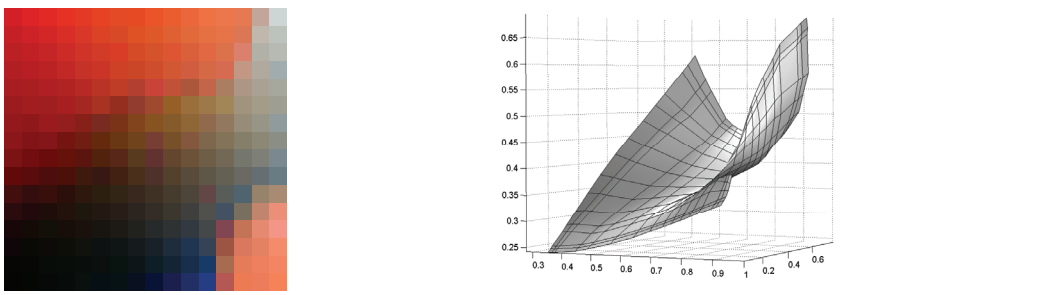
Figure 1.7. Visual comparisons of quantized images with different methods. The image resolution is 640×480.

	SOM	LBG	Octree
PSNR	37.09 dB	37.23 dB	33.0 dB

Table 1.1. The PSNR of quantized images.

As illustrated in Table 1.1, the LBG scheme and SOM scheme give a much better PSNR than the Octree scheme. The differences in PSNR can also be appreciated by comparing the quantized images in Figure 1.7. Note that, the image quantized using Octree scheme, Figure 1.7(f), has more contour artifacts than the others.

As mentioned earlier, the SOM color palette has an ordering property analogous to Figure 1.8. By considering this property, we can generate a "super color palette" based on a trained SOM color palette. As shown in Figure 1.9, given a



(a) SOM color palette. (b) SOM color palette plotted in the three-dimensional space.

Figure 1.8. The SOM color palette. In (a) we plot the palette entries according to the two-dimensional SOM neighborhood structure.

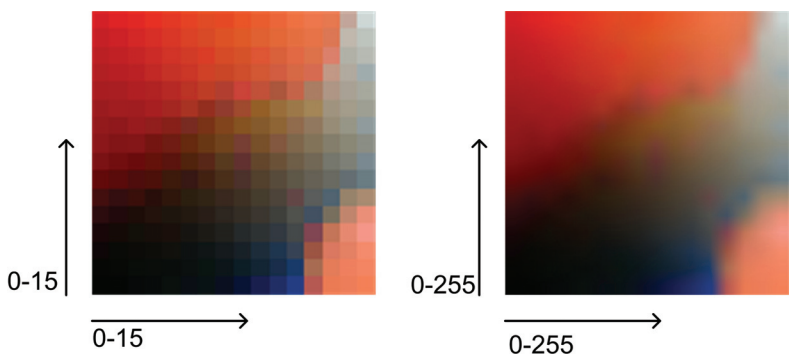


Figure 1.9. Create a high resolution color palette.

	SOM(GPU)	SOM(CPU)
Speed	10.38 ms	724.64 ms

Table 1.2. The speed is measured in time taken per iteration in milliseconds. The image is of dimension 640×480.

trained SOM color palette with 256 colors, we can plot the color palette according to the two-dimensional neighborhood structure and assign two indices, from 0 to 15, to the horizontal and vertical directions. To produce a higher resolution color palette, we can simply increase the resolution of the horizontal and vertical directions to 256×256. The entries of the higher resolution color palette are interpolated from the original color palette. Notice that in practice, we do not need to store the higher resolution color palette because it can be regenerated from the original color palette by interpolation.

We tested the speed of our implementation on a personal computer equipped with a Intel Core2 Duo 3.0 GHz CPU, 3.5GB RAM, and a NVIDIA GeForce 8800 GTX display card. The resulting performance is summarized in Table 1.2. As shown in Table 1.2, the SOM training speed accelerated by the GPU is significantly faster than that of the CPU.

1.6 Conclusion

This article describes a GPU implementation of the SOM training algorithm for color quantization. By using the vertex shader to search for the nearest palette entry and to perform random writes, the proposed scheme improves the training speed of the color palette significantly. With a slight modification, our approach can also perform the LBG training. The ordering property of SOM allows for several interesting extensions, including generation of "super color palettes" and reducing the dimension of input patterns [Togneri et al. 90]. Although our discussion focuses on the quantization of three-dimensional data, our implementation can be easily generalized to handle high-dimensional data.

Acknowledgments

The work was supported by a research grant from General Research Fund, HKSAR (Project No.: CityU 116508).

Bibliography

[Chang et al. 05] Chip-Hong Chang, Pengfei Xu, Rui Xiao, and T. Srikanthan. "New Adaptive Color Quantization Method Based on Self-Organizing Maps." *IEEE Transactions on Neural Networks*, pp. 237–249.

[Clark 96] D. Clark. "Color Quantization Using Octrees." *Dr. Dobb's Journal*, pp. 54–57.

[Kohonen 01] T. Kohonen. *Self-Organizing Maps*. Berlin: Springer, 2001.

[Kruger 94] A. Kruger. "Median-cut Color Quantization." *Dr. Dobb's Journal*, pp. 46–54.

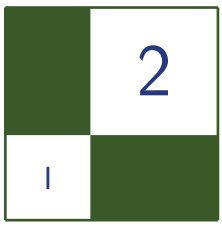
[Leung and Chan 97] C.S. Leung and L.W. Chan. "Transmission of Vector Quantized Data over a Noisy Channel." *IEEE Transactions on Neural Networks* 8:1 (1997), 582–589.

[Linde et al. 80] Y. Linde, A. Buzo, and R.M. Gray. "An Algorithm for Vector Quantizer Design." *IEEE Transactions on Communications* 28:1 (1980), 84–95.

[Priester 04] Sjaak Priester. "Better GIFs with Octrees." Available online (http://www.codeguru.com/Cpp/G-M/gdi/gdi/print.php/c3677/).

[Togneri et al. 90] R. Togneri, E Lai, and Y. Attikiouzel. "Kohonen's Algorithm for the Numerical Parametrisation of Manifolds." *Pattern Recognition Letters* 11:5 (1990), 313–319.

[Verevka 95] Oleg Verevka. "Color Image Quantization in Windows Systems with Local K-means Algorithm." *Proceedings of the Western Computer Graphics Symposium '95*, pp. 74–79.



Visualize Your Shadow Map Techniques

Fan Zhang, Chong Zhao, and Adrian Egli

2.1 Introduction

Things that look simple on the outside may not be simple inside. *Shadow mapping* [Williams 78] is a very good example of this. One of the most popular real time shadowing techniques, shadow mapping has played a key role in realistic rendering in computer games. However, all shadowing solutions based on imperfect sampling suffer from aliasing. With the wide array of techniques available, it is not easy to compare their results. In this article, we present a quantitative analysis of several popular techniques using a novel mathematical model specifically designed for measuring the aliasing errors in shadow maps.

2.2 Article Overview

The quantitative analysis presented in this article is based on the mathematical model for measuring *aliasing errors* in shadow mapping presented in [Zhang et al. 09]. Given appropriate assumptions and simplifications, we are able to explicitly evaluate the aliasing error at an arbitrary frustum position for a general view light configuration. Our model can measure the expected aliasing error independently of scene geometry.

The remainder of this paper is organized as follows: First, we outline the problems that this article attempts to address in Section 2.3. The aliasing error in shadow mapping is discussed from a mathematical perspective in Section 2.4. In Section 2.5 we describe how we measure aliasing in our model. As a direct application of this model, Section 2.6 illustrates and analyzes the aliasing

distributions in a few representative shadow map techniques. A corollary, Section 2.7, explains why a stronger *warping strength* may not always give an improved shadow quality.

2.3 Scope of Analysis

A full analysis of aliasing in shadow maps is beyond the scope of this article. Instead, we focus on providing a graphical illustration of the aliasing errors. This section describes the scope of our analysis.

The majority of our analysis establishes a quantitative comparison of different shadowing techniques (see Section 2.6). As illustrated in Figure 2.1, given a suitable example scene we can perceive the difference in visual quality produced by alternative shadowing techniques and varying lighting configurations. Our model gives a uniform description of the aliasing error independent of an example scene. We will use it to numerically quantify how "good" the shadow quality is at every position in the frustum, for a given technique and light configuration. This enables simpler comparisons of shadowing techniques.

As illustrated in Figure 2.2, the mathematical model described in Section 2.5 builds upon previous models of aliasing along the view direction only to

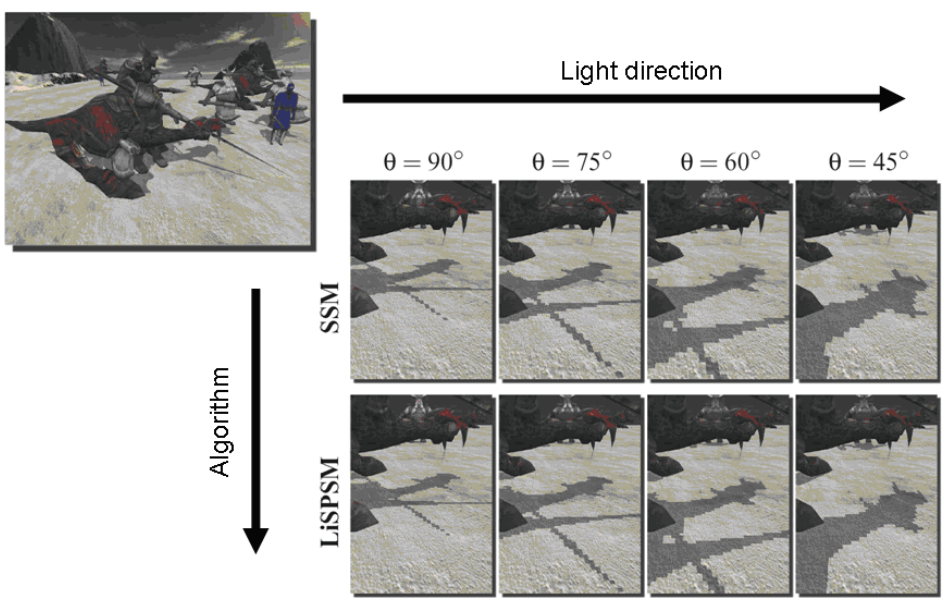


Figure 2.1. Shadowing techniques parameterized by light direction.

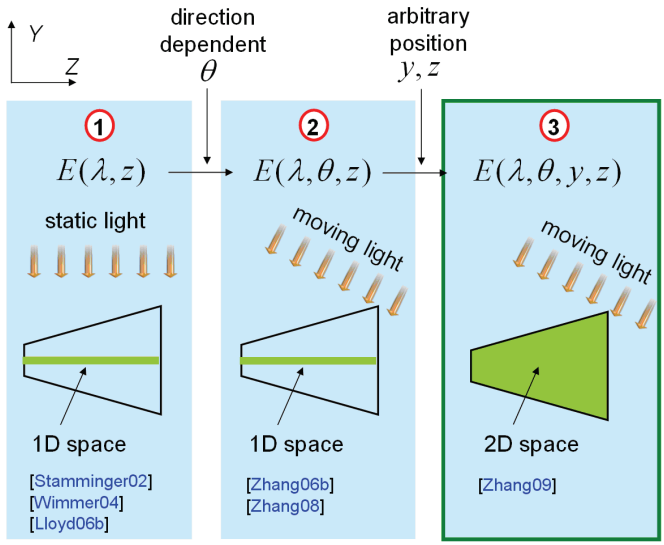


Figure 2.2. The evolution of mathematical models used for aliasing analysis. The first model [Stamminger and Drettakis 02, Wimmer et al. 04, Lloyd et al. 06b] is only valid for points along the view direction with an overhead directional light. The second model [Zhang et al. 06b] [Zhang et al. 08] extends the first to handle dynamic lights. Our model [Zhang et al. 09] works for both arbitrary light directions and frustum positions. Notice that we always consider the aliasing error in the YZ plane, so x does not appear.

include arbitrary light directions as a model parameter. Our model is applicable to most shadow mapping techniques, including *standard shadow maps* (SSM) [Williams 78], *perspective shadow maps* (PSM) [Stamminger02], *light space perspective shadow maps*(LISPSM) [Wimmer et al. 04], *trapezoidal shadow maps* (TSM) [Martin and Tan 04] and *cascaded shadow maps* (CSM) [Engel 06, Zhang et al. 06a].

The last issue we cover is an interesting corollary, which explains the unusual frustum warping behavior of TSM (see Section 2.7).

2.4 Aliasing Revisited

In this section, we review the aliasing error in shadow mapping from a mathematical perspective. The review will help us derive the aliasing representation in the next section.

As illustrated in Figure 2.3, consider a point $\mathbf{c}$ with eye-space coordinates (x, y, z), whose projections on the screen and shadow plane are $\mathbf{p}(p, q)$ and $\mathbf{s}(s, t)$,

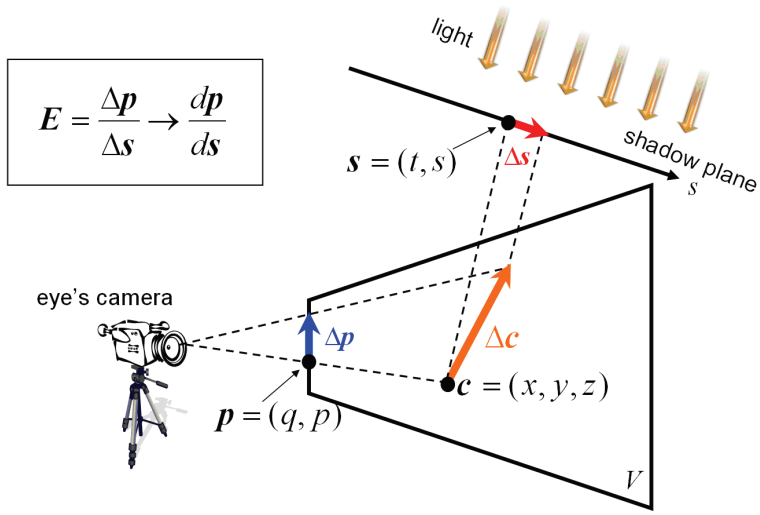


Figure 2.3. Measurement of aliasing in shadow mapping.

respectively. A small increment $d\mathbf{p} = (dx, dy, dz)$ in the view space will cause a shift $d\mathbf{p} = (dq, dp)$ on the screen, and an offset $d\mathbf{s} = (ds, dt)$ on the shadow plane. The aliasing error $\mathbf{E}$ at this position is measured as the ratio of the projected lengths on the shadow plane and image plane: i.e.,

$$\mathbf{E} = (\mathrm{E}_q,\ \mathrm{E}_p) = \left(\left| \frac{\mathrm{d}q}{\mathrm{d}t} \right|,\ \left| \frac{\mathrm{d}p}{\mathrm{d}s} \right| \right).$$

We refer to E_q and E_p as the *aliasing functions* [Zhang et al. 09]. They are used to quantify the aliasing in the vertical and horizontal dimensions of the screen, respectively. When E_q or E_p becomes small, *under sampling* occurs and we can expect shadow boundaries to appear jagged.

We can locate situations where aliasing occurs by understanding when the aliasing functions become small. Figure 2.4 illustrates how the aliasing functions can be defined on geometry. For a surface point $\mathbf{c}$, the local geometry is determined by the frame $(\mathbf{c}; \mathbf{n}, \mathbf{l}, \mathbf{n} \times \mathbf{l})$, where $\mathbf{n}$ and $\mathbf{l}$ are the normal and tangent directions, respectively. The angle between the light direction and view direction is denoted by $\mathbf{\Theta}$. As shown in Figure 2.4, if we consider the aliasing problem on the two-dimensional plane containing $\mathbf{l}$ and the light vector, then the aliasing functions can be parameterized with respect to $\mathbf{c}, \mathbf{\Theta}, \mathbf{l}$.

We do not wish to include knowledge of scene geometry, and by implication the tangent direction $\vec{\mathbf{l}}$, in our model. To facilitate an *offline aliasing analysis*, we

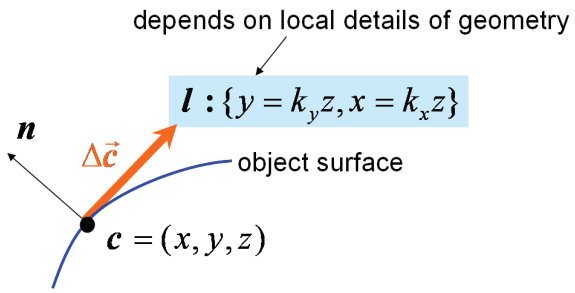


Figure 2.4. Defining aliasing functions at a surface point.

factorize the aliasing functions into two parts,

$$E(\mathbf{c}, \mathbf{\Theta}, \mathbf{l}) = E^{\text{pers}}(\mathbf{c}, \mathbf{\Theta}) \times E^{\text{proj}}(\mathbf{c}, \mathbf{\Theta}, \mathbf{l}). \tag{2.1}$$

Equation (2.1) is a model for warping algorithms in which the aliasing is divided into *perspective aliasing* E^{pers} and *projection aliasing* E^{proj}, where $E \in \{E_q, E_p\}$. The term E^{proj} is scene dependent (i.e., depending on $\mathbf{l}$), but the term E^{pers} is independent of the scene geometry. Perspective aliasing comes from the perspective foreshortening effect and can be reduced by warping the distribution of shadow-map texels. In this article we examine E^{pers} in all well known warping and traditional shadow map algorithms.

2.5 A Representation of Aliasing Errors

In this section, we present a generalized representation of aliasing errors based on the simplified computational model shown in Figure 2.5. This representation is the essence of our aliasing theory, which will be used to visualize various techniques.

V and W	View frustum and warping frustum
θ	Angle between view and light directions
n and f	Near- and far-plane values for V
λ	Near-plane value for W
ϕ	Half field of view of V
$(O; X, Y, Z)$	Eye's coordinates frame
$(q,\ p)$	Normalized screen coordinates $(-1 \leq q,\ p \leq 1)$
$(t,\ s)$	Texture coordinates $(0 \leq s,\ t \leq 1)$
$(E_q,\ E_p)$	Aliasing functions

Table 2.1. Notations.

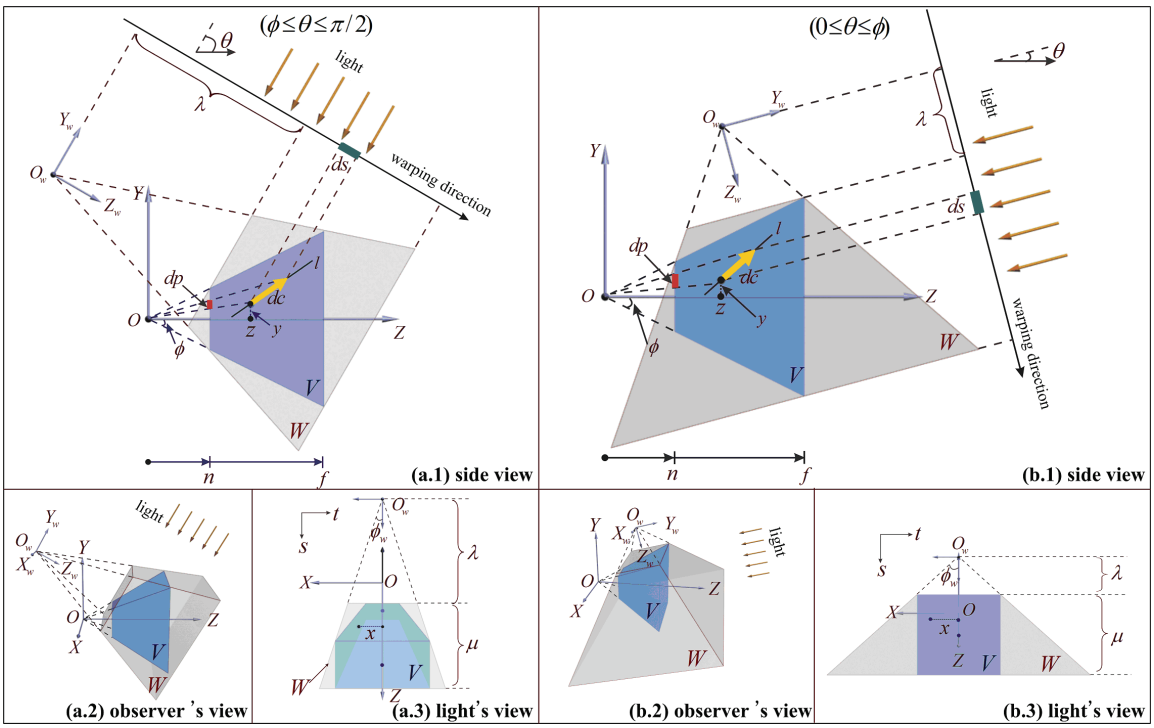


Figure 2.5. The computational model.

Frequently used notations may be found in Table 2.1. Please refer to the extra notes available at http://www.akpeters.com/gpupro for the full list of symbols used in our derivations.

Our model makes the following two simplifications:

1. *We assume directional light sources, confined to the YZ plane:* For point lights, the mapping from view coordinates (x, y, z) to texture coordinates (s, t) is not intuitive. Even though a preliminary result was presented for point lights [Lloyd et al. 06a], a comprehensive analysis in the general case remains challenging. Confining the movement of the light to the YZ plane allows us to restrict our study to the side view of the view frustum.

2. *We confine warping to be parallel to the shadow plane:* The choice of warping direction strongly influences the implementation complexity, because the type of light may switch between direction and point in the post-transformed space [Stamminger and Drettakis 02]. Wimmer et al. [Wimmer et al. 04] propose using a warping direction parallel with the shadow plane, and construct the warping frustum in the light space. An advantage is the direction

of the light source will not change in the post-transformed space, such that no mapping singularities result. Our computational model adopts the same parallel warping direction.

With the above simplifications, the warping frustum W is constructed as shown in Figure 2.5. Notice that we only need to consider the light directions $0 \leq \theta \leq \pi/2$ due to symmetry. In Equations (2.2) and (2.3), E_p and E_q quantify the aliasing in the vertical and horizontal dimensions of the screen independently. Here we simply state the result of our model. Detailed derivations for this model can be found in the extra notes available at http://www.akpeters.com/gpupro:

$$E_q \approx E_q^{\text{pers}} = F_q(\lambda, \theta) \frac{z + G(y, \lambda, \theta)}{z} \tag{2.2}$$

$$E_p \approx E_p^{\text{pers}} = F_p(\lambda, \theta) \frac{\Big(z + G(y, \lambda, \theta)\Big)^2}{z}. \tag{2.3}$$

We make use of an additional aliasing metric $E_{p\times q} = E_p \times E_q$ in this article. $E_{p\times q}$ is an aggregate measure of the aliasing errors in both dimensions. Remember that projection aliasing is ignored in all terms.

With the generalized representation of aliasing errors (Equations (2.2) and (2.3)), we are now ready to evaluate aliasing distributions in various techniques with respect to the metrics E_p, E_q and $E_{p\times q}$ in the next section.

2.6 Visualization

In this section, we perform three measurements to visually compare representative shadow map techniques.

The first measurement visualizes the λ distributions against θ for alternative techniques. For a fixed light view configuration, the main difference between the warping algorithms is the selection of the only free parameter, λ.

Given the λ distributions from the first measurement, the aliasing distributions within the view frustum can be directly visualized by using Equations (2.2) and (2.3). This gives us our second measurement.

Finally, we measure the aliasing distribution as it varies with view direction (i.e., $y = 0$).

2.6.1 Measurement 1: How the Warping Strength Varies as the Light Direction Changes

The purpose of this measurement is to plot λ against θ. This is shown in Figure 2.6, where we assume $n = 1$, $f = 400$ and $\phi = 30°$. For a detailed description of these λ values refer to the extra notes available at http://www.akpeters.com/gpupro.

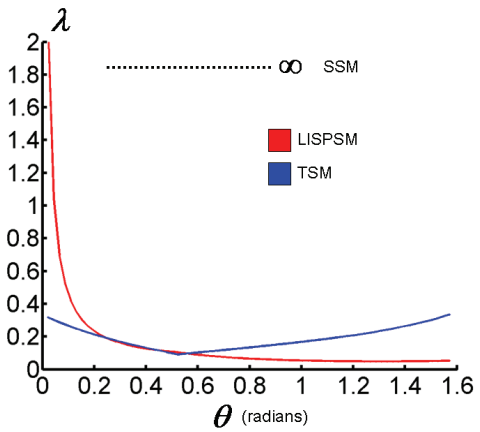


Figure 2.6. Warping strength varies as light direction changes.

Let λ_{SSM}, λ_{PSM}, λ_{LISPSM} and λ_{TSM} be the values in SSM, PSM, LISPSM, and TSM, respectively. We review the behavior of each algorithm in turn:

- λ_{SSM} does not employ any warping and therefore can be considered to have an infinite λ. The distribution of aliasing error in the camera frustum is inversely proportional to the view depth.

- λ_{PSM} produces a *linear* error distribution with respect to view depth [Wimmer et al. 04] [Zhang et al. 06b]. This linear behavior provides better shadow quality at the near camera frustum plane. Notice that typical PSM implementations in general cases ($\theta \neq \pi/2$) do not use the computational model shown in Figure 2.5, thus we do not visualize PSM in this article.

- λ_{LISPSM} produces an error distribution that is minimal with respect to the L^{∞} *norm* [Wimmer et al. 04] [Zhang et al. 08], which empirically works better than PSM in both practice and theory.

- The *focus-driven* behavior of λ_{TSM} requires the user to specify the region for the majority of the shadow map resolution.

As we can tell from Figure 2.6, $\lambda_{\text{LISPSM}} < \lambda_{\text{TSM}} < \lambda_{\text{SSM}}$ such that the warping strength becomes weaker from LISPSM to SSM. In the ideal case $\theta = \pi/2$, the stronger the warping strength is, the lower the aliasing errors are at the near region.

2.6.2 Measurement 2: How Aliasing Errors are Distributed Within the View Frustum

The purpose of this measurement is to visualize the distributions of aliasing error within the view frustum for different light directions across techniques. The result is shown in Figure 2.7.

For each combination of light direction and technique, alongside the colored error distributions, a simple scene comprised of 25 evenly spaced dragon models is used to illustrate the associated shadow map and shadow details.

In this article, the term *degenerate case* is used to describe the case when $\theta \leq \phi$. The term *dueling frusta* is often used to describe this case but in this article the term *dueling frusta* only refers to the extreme case $\theta = 0$.

To simplify our discussion further, we will mainly use the aggregate aliasing metric $E_{p\times q}$ to analyze Figure 2.7 in the following three cases:

1. *Ideal case.* $\theta = \pi/\mathbf{2}$ *(illustrated in the first column in Figure 2.7)*: In this case, the warping direction and the view direction are identical. The aliasing is constant in any vertical line for all techniques such that we only need to consider the aliasing distribution along the view direction. When examining SSM, we can see the aliasing errors in the near camera region are more intense (shown as red) than other warping algorithms. This illustrates why shadow quality can suffer on near objects in SSM. The algorithm places greater importance on the distant region which may not be as crucial to visual quality. Larger errors can be seen in LISPSM and TSM for distant regions, which are depicted in a yellow color. However, we see the aliasing for near objects is significantly alleviated in LISPSM and TSM due to warping rebalancing the sampling rate. This explains why shadow quality in warping algorithms can be significantly improved, especially in the ideal case.

2. *Normal case.* $\phi \leq \theta \leq \pi/\mathbf{2}$ *(illustrated in the second column in Figure 2.7)*: In theory, to minimize the maximum errors of $E_{p\times q}$ (also E_p and E_q), you need to choose an appropriate λ value to satisfy the condition $\min E_{p\times q} = \max E_{p\times q}$. The λ values satisfying this condition can be computed from Equations (2.2) and (2.3). Such λ values normally provide a very strong warping strength. It is possible to prove that it is not always possible to produce a positive (valid) λ value for some small θ values. Although stronger warping should provide higher quality at near regions, the warping direction is not identical with the view direction in normal cases (Figure 2.4). In this case the warping allocates more shadow map resolution to the region near the warping frustum's projection point (denoted by O_w in Figure 2.4), rather than the eye's position.

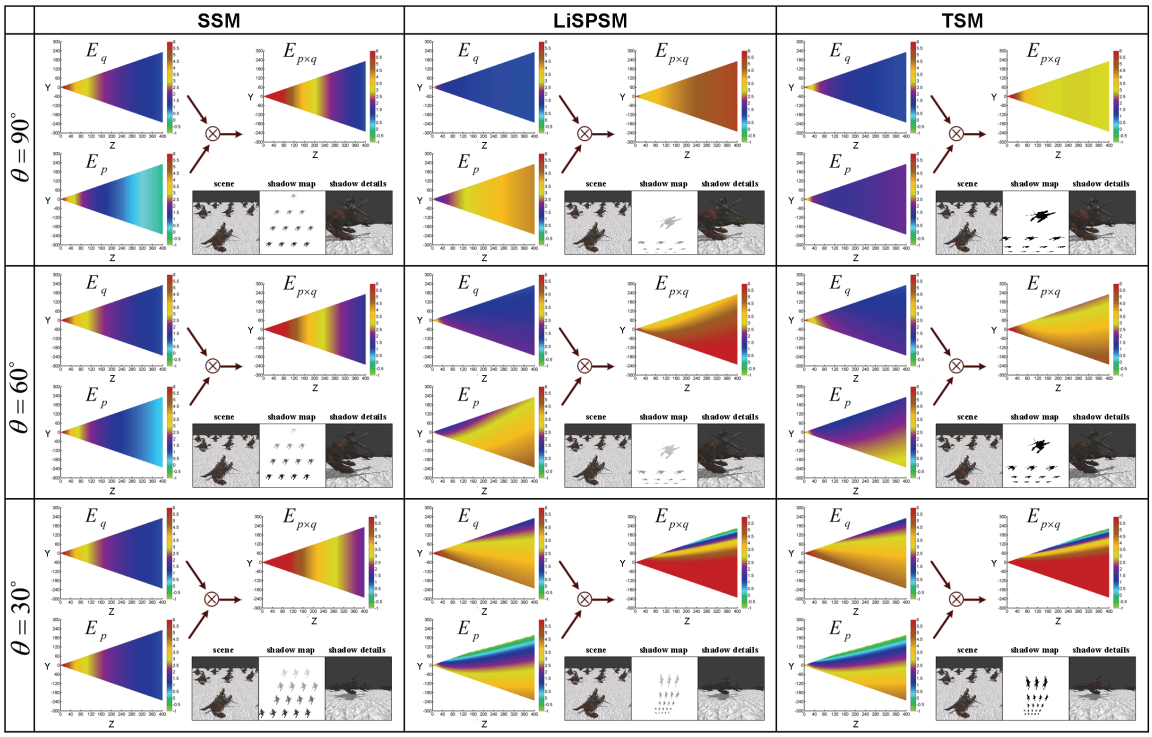


Figure 2.7. From left to right: the side views of aliasing distributions within the view frustum in SSM, LISPSM, and TSM, respectively. From top to bottom: the side views of aliasing distributions within the view frustum at different θ values. A sample scene is used to illustrate the corresponding shadow map and shadow quality for each. The values of the aliasing errors are colorized using the same color-bar in which the red color corresponds to the highest errors. The plots use a log_2 scale with $n = 1$, $f = 400$ and $\phi = \pi/6$.

3. *Degenerate case.* $\mathbf{0} \leq \theta \leq \phi$ *(illustrated in the last column in Figure 2.7)*: As we can see from Figure 2.4, in degenerate cases the view frustum is a quad from the light's point of view such that the warping is applied to the vertical direction. As we detailed earlier, perspective aliasing mainly results from the eye's projection transforming along the view direction (horizontal in the view space). Therefore, the effectiveness of redistributing perspective aliasing by using the warping transform decreases as θ goes to zero. For the dueling frusta case in particular, all warping algorithms fail to reduce E_p, and redistribute E_q and $E_{p\times q}$ in the vertical direction only. This is frequently undesirable because it will cause vertical stretch in shadow quality on the image plane from top to bottom. Therefore, all warping algorithms do not have an advantage over SSM in the degenerate case. As illustrated

by Figure 2.7, redistributing $E_{p\times q}$ is more effective in the vertical direction in both LISPSM and TSM. Subsequently, errors are smaller in the upper half frustum and larger in the lower half frustum. This phenomenon is particularly noticeable in $E_{p\times q}$ for TSM.

2.6.3 Measurement 3: How Aliasing Errors are Distributed along the View Direction

The purpose of this measurement is to visualize the aliasing distribution on the line $y = 0$ when the θ value changes. Figure 2.8 plots $E(z, \theta)$ for λ_{SSM}, λ_{LISPSM} and λ_{TSM} on this line at different θ values, where E stands for E_p, E_q and $E_{p\times q}$. We use the aggregated aliasing metric $E_{p\times q}$ to compare different techniques.

Our measurements are illustrative of how aliasing error in several techniques behaves in practice. However, it is a simplification and different shadowing techniques may suit different scenes. When implementing warping algorithms, there are no fixed criteria for determining an appropriate warping strength, because no criteria can suit all cases. The warping strength needs to be carefully tuned according to the scene model in practice.

2.7 Mismatch of View Direction and Warping Direction

In this section, we use our measurements to explain why a stronger warping strength does not always produce better shadow quality, due to the mismatch of the view direction and light direction in general cases.

In the general case $\theta \neq \pi/2$, the warping direction and view direction are not identical in the computational model (see Figure 2.5). As shown in Figure 2.9, due to the nature of the warping transform, a stronger warp will result in less resolution allocated to the "High Error" region. On the other hand, you can easily prove that the maximum errors of aliasing functions are located at the view frustum's left bottom or right bottom corners, which are covered by the "High Error" region. Therefore a strong warping strength will actually increase the maximum errors in the "High Error" region.

In many practical cases (e.g., a typical driving games), objects located in the lower half of the view frustum are the most important to a user's perception. However, the lower half of the view frustum receives less resolution from a typical sunlight as the warping strength increases. This phenomenon can be confusing as a weaker warp will produce a better shadow quality for objects in this region. Since different techniques produce different warping strengths this can make comparisons between techniques more complicated. For example, in Figure 2.7 we can see the errors in the lower half of the view frustum for TSM are lower than that for LISPSM.

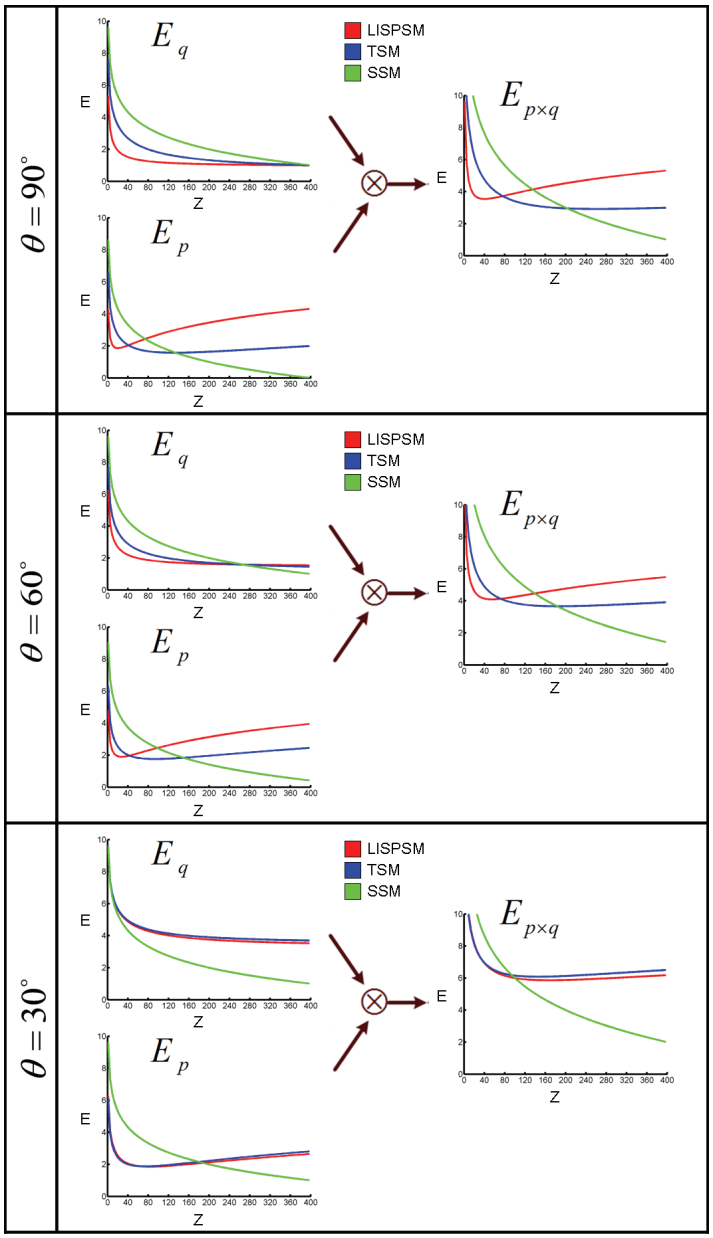


Figure 2.8. The aliasing distributions along the line $y = 0$ at different θ values. The plots use a log_2 scale with $n = 1$, $f = 400$ and $\phi = \pi/6$.

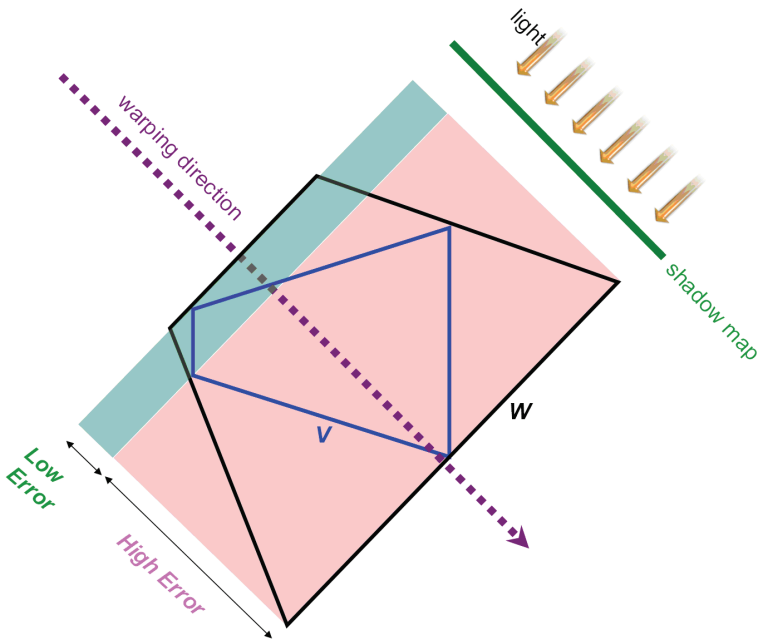


Figure 2.9. The mismatch of the warping direction and view direction.

2.8 Conclusion

In this article, we have used a model for measuring aliasing errors in shadow maps to examine the distribution of error independently of geometry. This has allowed us to visually compare the behavior of several popular techniques. We hope the results presented in this paper will help developers achieve a deeper understanding of shadow mapping techniques, and thus motivate further novel ideas in this field.

2.9 Supplementary Materials

The supplementary materials for this article include a set of videos and high quality pictures generated by Matlab that plot the aliasing functions in several techniques. An extra note is provided to provide a more detailed derivation of the underlying model.

2.10 Acknowledgments

Thanks to Sam Martin for his valuable review comments and patient proofreading. Thanks are also given to David Lam for his early review.

Bibliography

[Engel 06] Wolfgang Engel. "Cascaded Shadow Maps." In *ShaderX 5*, pp. 197–206. Charles River Media, 2006.

[Lloyd et al. 06a] Brandon Lloyd, Naga K. Govindaraju, David Tuft, Steve Molnar, and Dinesh Manocha. "Practical Logarithmic Shadow Maps." In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, p. 103. New York: ACM Press, 2006.

[Lloyd et al. 06b] Brandon Lloyd, David Tuft, Sung-eui Yoon, and Dinesh Manocha. "Warping and Partitioning for Low Error Shadow Maps." In *Proceedings of the Eurographics Symposium on Rendering 2006*, pp. 215–226. Eurographics Association, 2006.

[Martin and Tan 04] Tobias Martin and Tiow-Seng Tan. "Anti-aliasing and Continuity with Trapezoidal Shadow Maps." In *The Eurographics Symposium on Rendering 2004*. Eurographics, Eurographics Association, 2004.

[Stamminger and Drettakis 02] Marc Stamminger and George Drettakis. "Perspective Shadow Maps." In *Proceedings of SIGGRAPH '02*, pp. 557–562. New York: ACM Press, 2002.

[Williams 78] Lance Williams. "Casting Curved Shadows on Curved Surfaces." In *Proceedings of SIGGRAPH '78*, pp. 270–274. New York: ACM Press, 1978.

[Wimmer et al. 04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. "Light Space Perspective Shadow Maps." In *The Eurographics Symposium on Rendering 2004*. Eurographics, Eurographics Association, 2004.

[Zhang et al. 06a] Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. "Parallel-Split Shadow Maps for Large-Scale Virtual Environments." *Virtual Reality Continuum and Its Applications*, pp. 311–318.

[Zhang et al. 06b] Fan Zhang, Leilei Xu, Chenjun Tao, and Hanqiu Sun. "Generalized Linear Perspective Shadow Map Reparameterization." In *VRCIA '06: Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and its Applications*, pp. 339–342. New York: ACM Press, 2006.

[Zhang et al. 08] Fan Zhang, Hanqiu Sun, Chong Zhao, and Lifeng Wang. "Generalized Minimum-norm Perspective Shadow Maps." *Computer Animation and Virtual Worlds* 19:5 (2008), 553–567.

[Zhang et al. 09] Fan Zhang, Chong Zhao, and Hanqiu Sun. "An Aliasing Theory of Shadow Mapping." In *TPCG '09: Proceedings of the Seventh Theory and Practice of Computer Graphics 2009 Conference*. Eurographics Association, 2009.

II Geometry Manipulation

The "Geometry Manipulation" section of the book focuses on the ability of *graphics processor units* (GPUs) to process and generate geometry in exciting and interesting ways.

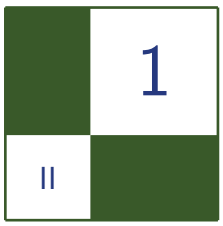
Tamy Boubekeur covers Phong Tesselation in the article "As Simple as Possible Tessellation for Interactive Applications." This operator is simpler than GPU subdivision surfaces and their approximations but succeeds at hiding standard "polygonization" artifacts often encountered on mesh silhouettes and interior contours. P*hong Tessellation* can be implemented on today's GPU using either uniform or adaptive instanced tessellation (vertex shader), or on the geometry shader for low tessellation rates.

The article "Rule-Based Geometry Synthesis in Real-Time" by Milan Magdics and Gergely Klar presents a framework for synthesizing and rendering geometry described by a rule-based representation in real time. The representation is evaluated completely on the GPU; thus, the geometry synthesis can be very fast and there is no need to copy data between the CPU and the graphics card. By applying frustum culling and rule selection based on the distance from the camera during the synthesis, only what is required for rendering with dynamic level of detail is generated.

Graham Hemingway describes in the article "GPU-based NURBS Geometry Evaluation and Rendering" a method for using the GPU to calculate NURBS geometry. Compared to evaluation on the CPU, this method yields significant performance improvements without drawbacks in precision or flexibility.

The last article of the section "Polygonal-Functional Hybrids for Computer Animation and Games," by Denis Kravtsov et al., describes how to represent geometry with functions to overcome some of the challenges with polygons like produce animations involving dramatic change of the shape of the model and creating complex shapes with changing topology. They also cover the integration of existing polygonal models and functional representations.

—Natalya Tatarchuk



As Simple as Possible Tessellation for Interactive Applications

Tamy Boubekeur

A *tessellation operator* increases the number of nodes in a mesh and can be understood as an upsampling method for polygonal surfaces.

Using real-time tessellation allows the application developer to specify a coarse mesh at the application level, but dynamically add vertices as needed at GPU level. This high resolution mesh may then be used to sample a (visually) smoother surface, eventually displaced using scalar or vector maps.

We describe (Section 1.1) and extend (Section 1.2) *Phong tessellation*, a simple operator generating curved surfaces over piecewise linear surface meshes. This operator is simpler than GPU subdivision surfaces [Shiue et al. 05] and their approximations [Vlachos et al. 01] [Boubekeur and Schlick 07b] [Loop and Schaefer 08]. However, for a minimal computational cost, it succeeds at hiding standard "polygonization" artifacts often encountered on mesh silhouettes and interior contours (see Figure 1.1). This operator is purely local (per-polygon computation, without neighborhood queries) and reproduces quadratic geometry variations. It naturally completes Phong Normal Interpolation.

Phong tessellation can be implemented on today's GPU using either *uniform* or *adaptive instanced tessellation* (vertex shader), or on the geometry shader for low tessellation rates. Upcoming graphics hardware and APIs such as Microsoft Direct3D 11 [Gee 08] will feature a standard tessellator unit with two new programmable stages: the *Hull Shader*, which defines the control network of the polygon to tessellate (in our case, this is the polygon itself together with its normal vectors and without any additional control points), and the *Domain Shader*, which evaluates the curved geometry (in our case, a code similar to Listing 1.1).

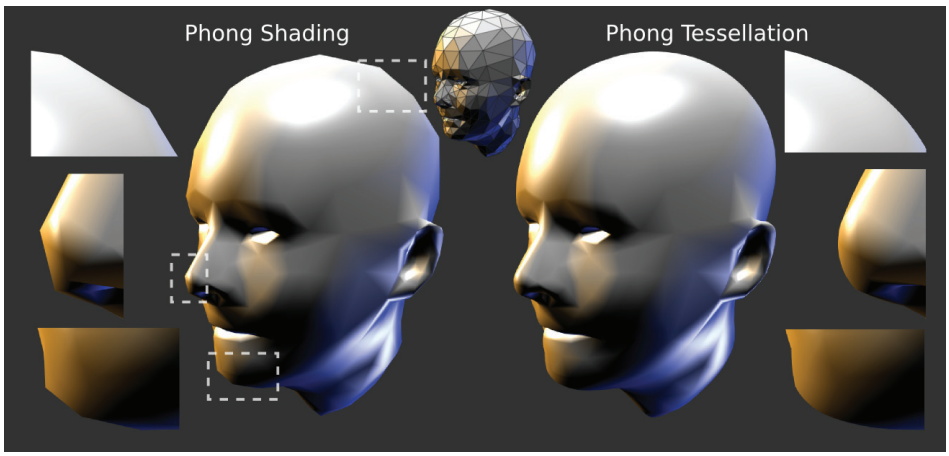


Figure 1.1. Phong tessellation (right) completes Phong shading (left).

1.1 Basic Phong Tessellation Operator

Consider a triangle $\mathbf{t}$ indexing three vertices $\{\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k\}$, with $\mathbf{v} = \{\mathbf{p}, \mathbf{n}\}$, $\mathbf{p} \in \mathbb{R}^3$ the position and $\mathbf{n} \in S^2$ the normal vector. Generally, the tessellation of $\mathbf{t}$ generates a set of new vertices lying on a surface defined over $\mathbf{t}$. Linear tessellation simply generates vertices on the plane defined by $\mathbf{t}$. Each generated vertex $\mathbf{p}(u, v)$ in the linear tessellation corresponds to a barycentric coordinate $(u, v, w), u, v \in [0, 1], w = 1 - u - v$ defined by

$$\mathbf{p}(u, v) = (u, v, w)(\mathbf{p}_i, \mathbf{p}_j, \mathbf{p}_k)^{\mathsf{T}}.$$

Phong normal interpolation [Phong 75] uses the same process but normalizes the result in the end:

$$\mathbf{n}'(u, v) = (u, v, w)(\mathbf{n}_i, \mathbf{n}_j, \mathbf{n}_k)^{\mathsf{T}}, \quad \mathbf{n}(u, v) = \mathbf{n}'/\|\mathbf{n}'\|.$$

Phong tessellation [Boubekeur and Alexa 08] generalizes this idea to the generation of curved geometry (see Figure 1.2). Each vertex of the triangle defines a tangent plane (vertex position and normal), and the Phong tessellation interpolates between three points, which are the projections on each plane of a linear interpolation over $\mathbf{t}$:

$$\mathbf{p^*}_{\alpha}(u, v) = (1 - \alpha)\mathbf{p}(u, v) + \alpha(u, v, w) \begin{pmatrix} \pi_i(\mathbf{p}(u, v)) \\ \pi_j(\mathbf{p}(u, v)) \\ \pi_k(\mathbf{p}(u, v)) \end{pmatrix}.$$

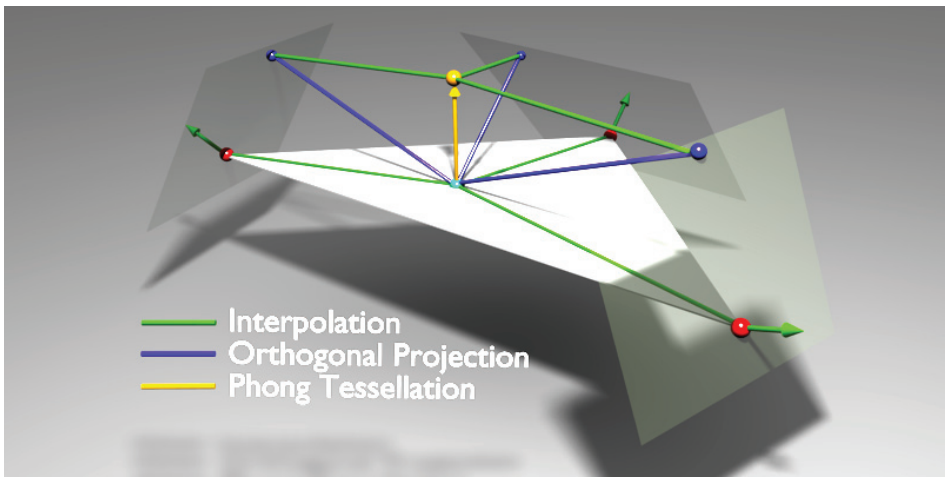


Figure 1.2. Phong tessellation performs in three steps: (1) Flat tessellation: vertices positions (red) are interpolated at a point (light blue); (2) Orthogonal projection: this point is orthogonally projected onto the three tangent planes defined at vertices (blue lines); and (3) Final displacement: these three points are interpolated again, defining the Phong tessellation displacement vector (in yellow). The initial interpolation (1) accounts for dynamic base mesh and standard tessellation methods, which starts from local coordinates expressed relatively to the current polygon.

Here $\pi_i(\mathbf{x})$ is the orthogonal projection of $\mathbf{x}$ on the plane defined by $\mathbf{v_i}$ and α acts as a simple shape factor interpolating between flat and curved triangle patch. We usually set it to $3/4$. Note that this operator defines a quadratic patch implicitly. The coarse mesh to tessellate can simply be set on the GPU using extra vertex attributes, textures or, for the sake of simplicity, *uniform* variables in the vertex shader (on a per-triangle basis):

```
uniform vec3 p0, p1, p2;
```

The Phong tessellation vertex shader is given in Listing 1.1. At rendering time, when a triangle has to be tessellated, we use *Instanced Tessellation* [Boubekeur and Schlick 05] for generating a large number of vertices for each triangle and evaluate the Phong tessellation vertex shader on each of them. The Phong normal interpolation already offers a visually smooth shading inside shapes, so there is no need to apply the Phong tessellation operator everywhere on the input mesh. It is needed only on locations where Phong normal interpolation cannot hide the lack of high order continuity of the geometry, namely, on contours and silhouettes. Therefore, we make use of *Adaptive Instanced Tessellation* [Boubekeur and Schlick 07a, Boubekeur and Schlick 08], where the adaptivity factor (i.e., level

```
vec3 p0, p1, p2; // Current coarse triangle vertex positions
vec3 n0, n1, n2; // Current coarse triangle vertex normals
float alpha = 3/4; // Shape factor
vec3 project (vec3 p, vec3 c, vec3 n) {
  return p - dot (p - c, n)*n;
}
// Curved geometry at p={w,u,v} (barycentric coord.).
vec3 PhongGeometry (float u, float v, float w) {
  vec3 p = w*p0 + u*p1 + v*p2;
  vec3 c0 = project (p, p0, n0);
  vec3 c1 = project (p, p1, n1);
  vec3 c2 = project (p, p2, n2);
  vec3 q = w*c0 + u*c1 + v*c2;
  vec3 r = mix (p, q, alpha);
  return r;
}
// Continuous normal field at p
vec3 PhongNormal (float u, float v, float w) {
  vec3 n = normalize (w*n0 + u*n1 + v*n2);
  return n;
}
```

Listing 1.1. Basic GLSL vertex shader for Phong Tessellation. The `PhongGeometry` function defines the displacement to apply on any point of the tessellated polygon. Although watertight (C^0 continuity), we use a different function for interpolating normals (`PhongNormal`) in order to prevent any visual continuity artifact.

of tessellation) is defined on the coarse mesh vertices and edges using a simple *silhouetteness* measure based on vertex normals:

$$d_i = \left(1 - \left\| \mathbf{n}_i^{\mathsf{T}} \frac{\mathbf{c} - \mathbf{p}_i}{||\mathbf{c} - \mathbf{p}_i||} \right\| \right) m,$$

with $\mathbf{c}$ the position of the camera and m the maximum refinement depth. This value can be mapped on the shape factor to modulate a progressive transition between flat and curved geometry. We refer the reader to our paper [Boubekeur and Alexa 08] for additional analysis on performances and visual quality. Note that other tessellation environments may be used such as NVIDIA SDK's version of Instanced Tessellation [NVIDIA 08] or the ADM Tessellation Library [Tatarchuk et al. 09].

1.2 Extension to Quads

We extend Phong tessellation to quad and tri-quad meshes by redefining the operator over a quadangular polygon $\mathbf{q} = \{\mathbf{v_i}, \mathbf{v_j}, \mathbf{v_k}, \mathbf{v_l}\}$, a primitive frequently

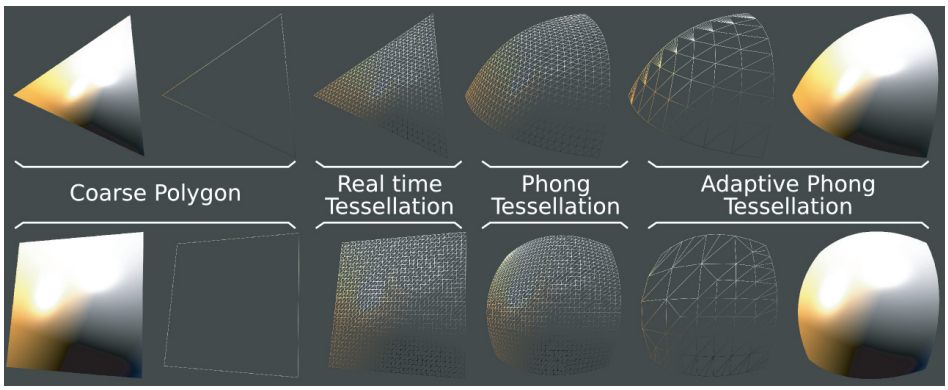


Figure 1.3. Phong tessellation for triangles and quads. Note that, while sampled with triangles, Phong geometry is defined at every point by the four vertices of the input quad.

used in production. We keep the basic idea of interpolation between tangent plane projections and simply replace the barycentric interpolation by a bilinear one (see Figure 1.3). This interpolation involves the four vertices of the quad and is substituted to the barycentric one wherever it appears in the triangle case, including the normal interpolation for shading:

$$\mathbf{p}^*{}_\alpha(u,v) = (1-\alpha)\mathbf{p}(u,v) + \begin{pmatrix} v(u\pi_i(\mathbf{p}(u,v)) + (1-u)\pi_i(\mathbf{p}(u,v))) + \\ (1-v)(u\pi_l(\mathbf{p}(u,v)) + (1-u)\pi_k(\mathbf{p}(u,v))) \end{pmatrix}.$$

Triangle and quad Phong tessellation reproduce the same geometry on polygon edges, as only edge vertices are involved on this parameter domain (linear interpolation of orthogonal projections in both cases). Therefore we obtain a

Figure 1.4. Phong tessellation extended to quad and tri-quad meshes.

crack-free tessellation scheme for triangle, tri-quad, and quad meshes. By doing this we avoid the typical trick consisting in triangulating quads, which often results in poor shape quality, even under (triangle) Phong tessellation. Note that while we define a curved geometry over a quad, this imposes no restriction on the way we sample the quad and triangle-based instanced tessellation can still be used safely with two triangular patches for sampling a single quad under quad Phong tessellation projection. Figure 1.4 illustrates the use of this extended operator.

1.3 Results and Discussion

Considering a typical scenario involving a dynamically deforming input coarse mesh (few thousands polygons) tessellated at level 6 (64×64)—a consistent 40% frame rate improvement is measured compared to Curved PN Triangle [Vlachos et al. 01] as well as a lower bandwidth usage for a similarly convincing rendering. Note that the aim of such an operator is to "fix" the strong polygonization artifacts appearing on silhouettes and interior contours, where Phong normal interpolation cannot hide the lack of continuity. Consequently, Phong tessellation differs significantly, for instance, from *subdivision surfaces approximations* [Boubekeur and Schlick 07b, Loop and Schaefer 08], which alterate significantly the perceived shapes in an attempt to generate globally smoother surfaces. Last, Phong tessellation is about 10% slower than flat tessellation when fragment shading is disabled. This number boils down to 2% when lighting and texturing are enabled and used, making it the cheapest way to generate visually smooth and curved geometry out of a dynamic polygonal mesh.

Bibliography

[Boubekeur and Alexa 08] Tamy Boubekeur and Marc Alexa. "Phong Tessellation." *ACM Transaction on Graphics - Special Issue on ACM SIGGRAPH Asia 2008* 27:5 (2008), 1–5.

[Boubekeur and Schlick 05] Tamy Boubekeur and Christophe Schlick. "Generic Mesh Refinement on GPU." In *Proceedings of ACM SIGGRAPH/Eurographics Graphics Hardware*, pp. 99–104, 2005.

[Boubekeur and Schlick 07a] Tamy Boubekeur and Christophe Schlick. *GPU Gems 3*, Chapter Generic Adaptive Mesh Refinement. Addison-Wesley, 2007.

[Boubekeur and Schlick 07b] Tamy Boubekeur and Christophe Schlick. "QAS: Real-time Quadratic Approximation of Subdivision Surfaces." In *Proceedings of Pacific Graphics 2007*, pp. 453–456, 2007.

[Boubekeur and Schlick 08] Tamy Boubekeur and Christophe Schlick. "A Flexible Kernel for Adaptive Mesh Refinement on GPU." *Computer Graphics Forum* 27:1 (2008), 102–114.

[Gee 08] K. Gee. "Direct3D 11 Tessellation." Presentation at Gamefest 2008, 2008.

[Loop and Schaefer 08] Charle Loop and Scott Schaefer. "Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches." *ACM Transaction on Graphics* 27:1 (2008), 1–8.

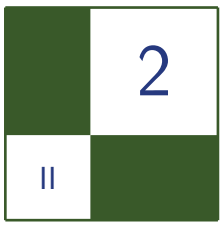
[NVIDIA 08] NVIDIA. "NVIDIA OpenGL SDK 10." Technical report, NVIDIA Corp., 2008.

[Phong 75] Bui Tuong Phong. "Illumination for Computer Generated Pictures." *Communications of the ACM* 18:6 (1975), 311–317.

[Shiue et al. 05] Le-Jeng Shiue, Ian Jones, and Jorg Peters. "A Realtime GPU Subdivision Kernel." In *Proceedings of ACM SIGGRAPH*, pp. 1010–1015, 2005.

[Tatarchuk et al. 09] Natalya Tatarchuk, Joshua Barczak, and Bill Bilodeau. "Programming for Real-Time Tessellation on GPU." Technical report, AMD Inc., 2009.

[Vlachos et al. 01] Alex Vlachos, Jorg Peters, Chas Boyd, and Jason Mitchell. "Curved PN Triangles." In *Proceedings of ACM Symposium on Interactive 3D*, pp. 159–166, 2001.



Rule-Based Geometry Synthesis in Real-Time

Milán Magdics and Gergely Klár

2.1 Introduction

Procedural modeling is a very popular modeling technique, since it allows us to create complex models with a small amount of work. Furthermore, algorithmic scene and object representations are often very compact and have low storage costs. Using operations that repeat elements and add random variations, we can represent arbitrarily large and detailed scenes without increasing the storage needs, while the generated objects remain diverse.

Evaluating the algorithmic description on the fly or, in other words, generating only those objects that we see through the camera, results in a runtime storage cost proportional to the complexity of the potentially visible objects, allowing us to store and visualize potentially infinite virtual worlds.

Rule-based modeling, whether using different kinds of grammars like L-systems, or split grammars, is a good trade-off between simplicity, usability, expressiveness, and formalism and has been widely used to generate different kinds of objects such as plants, buildings, road systems, fractals, and subdivision surfaces. This article presents a framework for synthesizing and rendering geometry described by a rule-based representation in real time. The representation is evaluated completely on the GPU, thus, the geometry synthesis can be very fast, and there is no need to copy data between the CPU and the graphics card. By applying frustum culling and rule selection based on the distance from the camera during the synthesis, we can generate only what is required for rendering with dynamic level-of-detail.

Interaction with the procedural scene is also an important problem to solve. The article discusses a technique for discrete collision detection based on a very simple observation: if we use an object's bounding box instead of the viewing

frustum for culling, we can get the intersection of the object with the procedural scene the same way as generating objects inside the camera frustum. Thus, the GPU can speed up the calculation of these intersections when the number of objects is large, which is useful for controlling a large number of AI players, having particle systems colliding with the procedural scene, or in physical simulations.

Synthesizing the geometry on the GPU requires shader programs that depend on the current rules. The article shows how to automate shader and client code generation from an arbitrary (but well-defined) grammar description that is easier to create and understand.

2.2 Building Up the Scene with Procedures

The model we use is built on *Procedural Geometric Instancing* (PGI), which was introduced by Hart in [Hart 92,Ebert et al. 02]. PGI is a scene graph model, where nodes are extended with a procedure that executes at the time of instantiation. This procedure can set the parameters of the node based on the parent node or global parameters, and it can create new nodes. The whole scene is built up by instantiating the root node(s).

Since the procedure of a node depends only on the parent node, if two nodes are not descendants of each other, they can be evaluated independently. For example, we can build up the scene graph level by level (which corresponds to the breadth-first traversal of the graph), and evaluate the nodes of a level in parallel, resulting in better performance. We use this kind of geometry generation in our work.

Another simple but important observation is that we get the same result if we start the generation from an interior node instead of the root nodes. This allows us to buffer some parts of the scene graph, and regenerate only a smaller part of the scene in subsequent frames.

2.3 L-systems and the PGI

An *L-system* [Prusinkiewicz and Lindenmayer 91] is a parallel rewriting system defined as a 4-tuple $L = (N, T, \omega, P)$, where T and N is the set of *terminal* and *nonterminal* symbols, ω is the *initial string*, or *axiom* of the system, and P is the set of *production rules*, defining the way nonterminal symbols can be *replaced* with combinations of terminals and other nonterminals. A rule consists of two strings: the *predecessor* and the *successor*. Similarly to formal grammars, the *production* of an L-system iteratively replaces a predecessor with a successor in the current string (this step is often called *applying* of the rule). In contrast to formal grammars, an L-system applies as many rules as possible in an iteration,

thus, it is much more capable of describing the growth what we need for fast generation of procedural geometry. The *generated language* of an L-system is a set of symbol sequences (or *strings*) that can be generated by applying zero or more rules to the axiom.

To utilize L-systems in computer graphics we assign geometric meaning to each symbol: for example, a command that changes drawing position or draws a line. Additionally, to make the modeling easier and increase the expressiveness, we can assign parameters to symbols (the concept we get is usually called a *module*). For example, let symbol F denote drawing a line. In this case, the parametrized symbol $F(l)$ may denote drawing a line with length l. This way, any kind of mesh can be described as a sequence of symbols. The graphical objects that can be generated using a specific L-system L are the geometric interpretations of the strings in L's generated language.

The traditional method to evaluate the geometric interpretation of a string is the *turtle graphics model*. In turtle graphics, drawing commands and state modifier commands (i.e. those that change the current global rendering state) are assigned to the symbols. The string is processed sequentially, and every command is evaluated relative to the current rendering state. The main problem of this method is that the evaluation cannot be parallelized, since every command that changes the rendering state has effect on every following commands in the string.

In [Hart 92, Ebert et al. 02], Hart shows how a procedural scene description given in the PGI model can be translated to a context-free L-system (that is, an L-system in which every predecessor consists of exactly one symbol) and vice versa. Thus, in terms of geometric modeling PGI and context-free L-systems are equivalent. Figure 2.1 shows the correspondence of the two models. The most important consequence of the equivalence is that the geometric interpretation of an L-system generated string can be evaluated in parallel, or in other words, a string can be considered as a set of symbols. In our work, we extend Hart's scene-graph-based L-system model to give more flexibility. The details are described in Section 2.4.

Hart also showed how the rules used with the turtle graphics model should be transformed to allow parallel evaluation. Since the turtle graphics model is frequently used in rule-based geometry generation, we will discuss this transformation with respect to our model in Section 2.4.

Modeling objects with PGI requires writing programs, which can be difficult to modelers who are not familiar with programming. Although PGI and rule-based models are very similar, the formalism is different. According to other works [Parish and Müller 01, Wonka et al. 03, Müller et al. 06, Lipp et al. 08], modelers can easily learn and use rule-based modeling methods, which was our main reason to use rules for real-time geometry generation. Please note that when

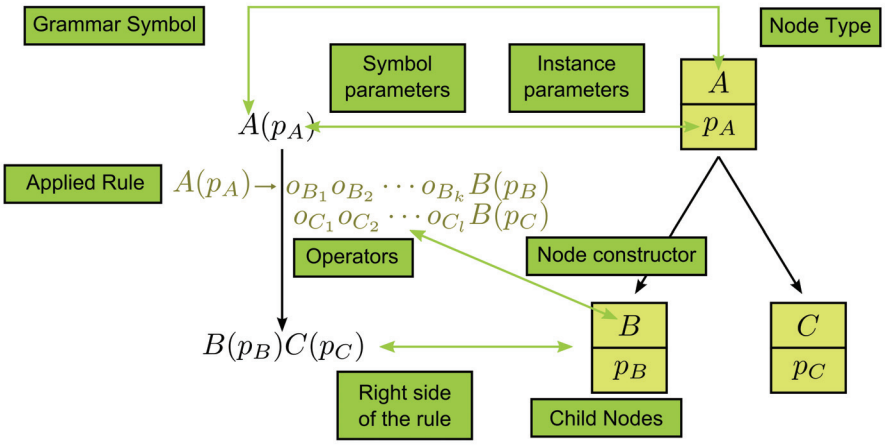


Figure 2.1. Equivalence of the PGI model with L-systems.

using only context-free rules, rules of a sequential grammar G can be evaluated in parallel. As a proof, consider the syntax tree of a string derived in G, the parallel breadth-first traversal of this tree corresponds to the L-system-style rule application. In terms of geometric modeling, when two procedural objects are independent they can always be refined independently, thus, in parallel.

Also note that this does not necessarily hold for context-sensitive rules (i.e., having more than one symbol in the predecessor), as context-sensitive rules are not allowed in our model. Thus, the expressibility of some complex conditions like connectivity and object dependency is very limited in our system.

2.4 Model Details and Implementation

In this section, we will give a detailed overview of the L-system-based model we used and we will describe its GPU implementation. Since our intention is to allow modelers to write rules instead of programs, we will also discuss automatic code generation from a grammar and describe what parts a grammar description should consist of to work with our model. The concrete, XML-based grammar descriptor format we used is included with the demo programs.

2.4.1 Symbols

Representing geometry. In the turtle graphics representation, geometry is created using symbols that represent drawing commands which are relative to the cur-

rent rendering state. Here, these symbols represent the geometry (an arbitrary mesh) with their parameters and are called *instanced symbols*. Every instanced symbol stores every parameter needed for its drawing (*instancing*), thus, instanced symbols are independent. This enables parallel instancing, and a sequence of symbols becomes a set. In contrast to the turtle graphics model, instanced symbols cannot affect the rendering state in our model. Such symbols given in the turtle graphics model can be treated by splitting them into an instanced symbol and one or more operators (see the definition of operators below). For example, let module $F(l)$ mean drawing a line having length l and then moving the actual drawing position (a global rendering state) by l to the current direction. In the scene-graph-based representation this corresponds to $\overline{F}(l, o)m(\vec{v})$, where $\overline{F}(l, o)$ represents a line with length l and orientation o, and operator $m(\vec{v})$ represents translation, with vector $\vec{v}$ having the same effect as in $F(l)$. In this case, the operator has to follow the instanced symbol to get the same result. After these splitting transformations, the rules can be rewritten as described in Section 2.4.6.

Operators. Symbols representing a command that changes module parameters are called *operators*, we denote them with lower case letters. There is no global rendering state, as every operator modifies only the parameters of the next geometric object. In other words, every operator belongs to one specific instanced symbol in a string and modifies only that symbol's parameters. This way we can easily achieve equivalence with the PGI model, and these operators can hide the programs of the nodes that execute at the time of instantiation. Operators are relative to the left side of the rule, or in terms of PGI the parent node. For example, consider the rule

$$A \rightarrow hBhC$$

where operator h halves the size. The size parameter of both B and C will be the half of A's size.

Representation. In our model, each module is represented by a vertex, which allows us to implement rules and create new modules in the geometry shader. Section 2.4.2 will discuss the benefits of this decision.

Theoretically, different symbols can have different parameters. However, we found this impractical. Having multiple vertex types would result in more passes or additional branching in the shader code. Thus, our approach sacrifices storage space for efficiency and stores generic vertices that are capable of representing any type. The corresponding vertex type (a record in both the CPU and the GPU code) is created automatically as the union of all module data types and a symbol identifier. Thus, symbols appear as identifiers, mapped into integers between 1 and the number of symbols, which is also generated automatically from

the grammar description, similarly to [Sowers 08]. To make the implementation of the code generation easier, we required the explicit specification of symbols in the grammar description, although they are implicitly given with the rules.

Since we merge all symbol types into one general record, the vertex type grows bigger. Therefore, it is easier to exceed the limit of the vertex size or the limit of the output size of the geometry shader. To overcome these problems we can use line or triangle primitives (instead of vertices) to represent modules or distribute module data between several primitives (vertex, line or triangle) and perform a single iteration step of the geometry generation in multiple passes. However, the first option alone does not solve the problem of the geometry shader's output limit, and splitting into multiple passes only works if rules do not depend on every attribute value at the same time. Fortunately, this holds in many cases; for example, position, size, and orientation are usually independent. We would like to note here that all our examples used a single vertex.

Modules are stored on the GPU in vertex buffers. In our experience 100 bytes is a reasonable upper bound for the storage requirement of one module (e.g., 25 float attributes can fit into it). Thus, the representation of one million vertices requires about 100 MB of storage, which can fit in a modern GPU's memory several times. Generating much more than one million modules with real-time rates exceeds the capabilities of our implementation, therefore, storage requirements of the algorithm should not cause any problems on high performance graphics cards.

2.4.2 Algorithm Overview

Figure 2.2 shows the overview of the complete algorithm. It consists of three main parts.

The first step is the generation of the procedural scene graph, called *generation step* (see Listing 2.1). This part is similar to the split grammar GPU implementation in [Sowers 08]. Starting with the axiom string, we perform iterations by applying rules of an L-system to generate instanced modules. These are only descriptors of different geometric objects stored as vertices, no concrete geometry is created here. Every operation is performed in the geometry shader, and modules are processed in parallel. The vertex shader simply passes its input without any modifications, and rasterization is disabled. We apply culling first for every module; this ensures that those objects that are not visible in the current frame are not generated. For modules that passed culling and were not terminated, we apply a rule. The left side of the rule is the symbol type of the module, which is stored in a symbol identifier. If there are multiple rules for that symbol, a *rule-selection algorithm* is applied to choose one of them. The chosen rule is applied by creating new modules and initializing them with the proper operators. As a last step,

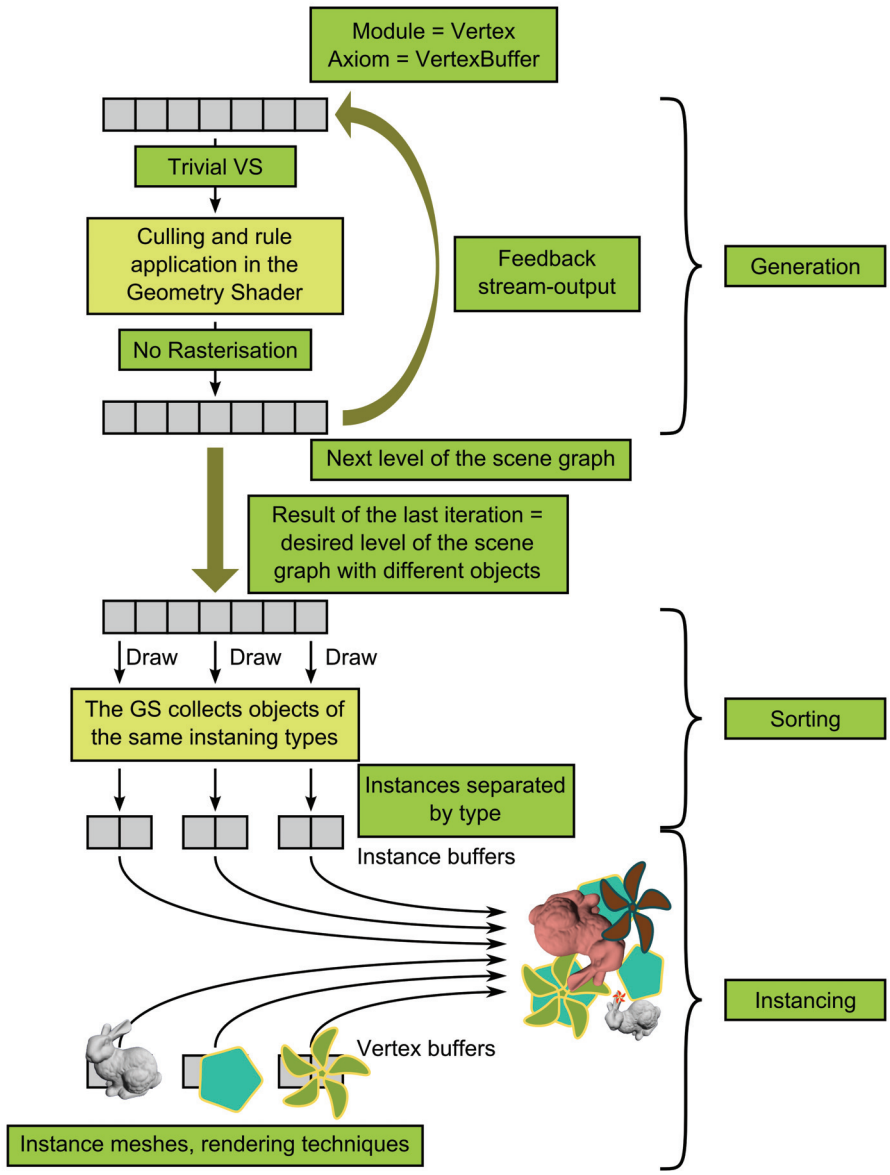


Figure 2.2. Overview of the implementation.

```
// The input of the first iteration is the axiom.
device->IASetVertexBuffers(0, 1, &axiomBuffer,
   &modulStrides, &zeroOffset);
// Apply generation shaders.
generationTechnique->GetPassByIndex(0)->Apply(0);
// Execute an iteration maxDepth times.
for (int depth = 0; depth < maxDepth; ++depth)
{
   // Set dstBuffer as stream output.
   device->SOSetTargets(1, &dstBuffer, &zeroOffset);
   // First draw call has to process the axiom.
   if (0 == depth)
      device->Draw(axiomLength, 0);
   else
      device->DrawAuto();
   // Ping pong
   ID3D10Buffer* nullBuffer = NULL;
   device->SOSetTargets(1, &nullBuffer, &zeroOffset);
   std::swap(dstBuffer, srcBuffer);

   // Set srcBuffer as input
   device->IASetVertexBuffers(0, 1, &srcBuffer,
      &modulStrides, &zeroOffset);
}
```

Listing 2.1. DX10 implementation of the CPU code of the generation step using fixed iteration number.

the modules are inserted into the new set of modules (the stream output), which is the input of the next iteration. Since we utilize the geometry shader for the creation of new modules, only the desired number of output is generated. There are no dummy symbols as in Hart's GPU L-system implementation using pixel shaders [Lacz and Hart 04].

After the generation step, modules are sorted by instancing type; that is, what kind of geometry they represent and what technique is used to render them. Sorting is essential for the efficient instancing of the generated descriptors.

Finally, we render the scene using instancing.

2.4.3 Iteration and Termination

We build up the procedural scene graph using breadth-first traversal, generating the subsequent level in every iteration step by performing a draw call and applying one rule for each non-terminated module. In most cases, the depth of the generation can be exactly specified. The concrete depth depends on the grammar as well as other conditions like dynamic level of detail. Thus, the most basic way

```
[maxvertexcount(MAX_SUCCESSOR_LENGTH)]
void gsGeneration(point Module input[1],
                  inout PointStream<Module> stream)
{
   // Perform culling here.

   // Checking termination
   if (input[0].terminated)
   {
      stream.Append(input[0]);
      return;
   }
   // Rule selection and application here.
}
```

Listing 2.2. The role of termination in the geometry shader.

to implement the CPU code is to execute the iteration `maxDepth`-times, where `maxDepth` is the desired level.

Objects usually do not need to be refined to the same level. For example, generating a building may need more iterations then a flower, or we add more details to a building closer to the camera then another building that is far away. To support this, we added a parameter (called `terminated`) to the module type that indicates whether the object is completed or not (see Listing 2.2). The ideal implementation would be to separate the completed objects from those that require further iterations, but the current shader model does not allow inserting different data into multiple stream outputs. We found it too inefficient to insert a sorting step that collects terminated modules and removes them from the working stream. Thus, we keep completed modules in the same buffer with the incomplete modules and simply pass them through the pipeline. Please note that if the shader model would allow dividing the data between multiple buffers, we could simply send the terminated modules into another stream.

Termination can be useful to implement another strategy to determine the generation depth. Sometimes it is better to perform iterations until there has been a change in the current modules. In other words, we can end the iterations if all modules are terminated. Unfortunately we cannot set any kind of global flag in the geometry shader to indicate that there is an incomplete module. However, we can extend the generation with a pixel shader stage, which discards pixels where the `terminated` parameter coming from the geometry shader is true. Thus, if any pixel is refreshed (this can be checked with a single occlusion query), that signals a change in the module set. To avoid writing the same pixels we can extend the Module struct with a module identifier. If M is the maximum number of symbols in successors, d is current generation level, and p denotes the position of the newly

```
[maxvertexcount(MAX_SUCCESSOR_LENGTH)]
void gsGeneration(point Module input[1],
                  inout PointStream<Module> stream)
{
   if ( cullModule( input[0] )
      return;
   // Check for termination. Select and apply rules.
}
```

Listing 2.3. Culling in the geometry shader.

created module in the applied rule, the following formula

$$\texttt{newModule.objectID} = M^d + \texttt{parent.objectID} * M + p$$

gives a unique identifier, which can be easily mapped to the pixels of the render target.

Note that with multiple different stream outputs this strategy is greatly simplified, the iteration has to be continued until the working buffer is not empty.

2.4.4 Culling

The goal of culling is to throw away those modules that do not effect the currently rendered frame. Formally, this corresponds to the application of the rule

$$A \rightarrow \epsilon$$

where ϵ is the empty string. However, adding this rule for every symbol would be too tedious; thus, it is not included in the grammar description. The implementation is simple—the geometry shader simply does not return anything for culled modules (see Listing 2.3).

The culling strategy can be very diverse depending on the procedural model; for example, culling based on size in pixels, alpha value, distance from the camera, or frustum culling.

We implemented spheres and *axis-aligned bounding boxes* (AABBs) as bounding geometries for frustum culling (we use it for interaction in Section 2.5 as well). These are resized to the module's size parameter (possibly three-dimensional) and centered to the module's position (see Listing 2.4). However, any kind of bounding volume can be stored or computed, possibly different for every symbol, and depending on the module parameters.

2.4.5 Rule Selection

Before applying a specific rule, we have to decide which rule to use. First, we have to identify the left side of the rule (the predecessor), which is simply the symbol