LUND UNIVERSITY

Performance Improvements for the Rasterization Pipeline

Hasselgren, Jon

2009

Link to publication

Citation for published version (APA):
Hasselgren, J. (2009). *Performance Improvements for the Rasterization Pipeline*. [Doctoral Thesis (compilation), Department of Computer Science]. Department of Computer Science, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

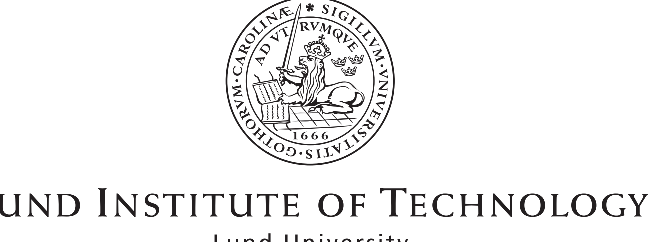
Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Performance Improvements for the Rasterization Pipeline

Jon Hasselgren
Department of Computer Science
Lund University



ISBN 978-91-976939-2-9
ISSN 1404-1219
Dissertation 32, 2009
LU-CS-DISS:2009-3

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: jon@cs.lth.se
WWW: http://www.cs.lth.se/home/Jon_Hasselgren

Typeset using $\LaTeX 2\varepsilon$
Printed in Sweden by Tryckeriet i E-huset, Lund, 2009
© 2009 Jon Hasselgren

Abstract

Performance improvements are always needed in computer graphics. Better performance frees up computational resources which can be used to increase the level of realism in the rendered images. Despite a very rapid development of graphics hardware, we are still far from the point where photo-realistic rendering can be done in real time. Therefore, better algorithms must be developed in order to advance the field.

In this thesis, we present several new algorithms targeted for the rasterization pipeline, which is, at the time of writing, the de-facto standard rendering algorithm used in graphics hardware. We focus on three areas in the pipeline. The first is the rasterization step where we present efficient sampling strategies which can improve performance, or enable new algorithms to be implemented. This includes an investigation of very inexpensive, but relatively high quality, sampling schemes, an algorithm for conservative rasterization through area sampling, and an algorithm for efficiently rasterizing multiple views which can be useful for holographic displays, and other multi-view displays.

The second area of focus is on compression algorithms. Compression is frequently used in graphics hardware in order to minimize memory traffic and thereby increase performance. We present efficient algorithms for high dynamic range texture compression, depth buffer compression, and color buffer compression. The inner workings of buffer compression have been kept secret by hardware vendors, and the only available information is available through patents. Therefore, we also present surveys of the prior art in buffer compression.

The final area we focus on is programmable culling. Culling is an old concept in computer graphics, and it is often necessary to include some form of culling to make an algorithm truly efficient. However, current graphics hardware only has a few fixed function culling algorithms implemented. This often makes it impossible, or impractical, to implement culling in new rendering algorithms that rely on graphics hardware. We present a methodology for automatically deriving culling algorithms from shader programs, and we show how this can be used to perform culling both on a pixel level, and on a geometry or vertex level. In both cases, we show that our automatic culling can greatly improve rendering performance for a wide variety of scenes.

Acknowledgements

First of all I would like to thank my main supervisor Tomas Akenine-Möller for always supporting and encouraging me during this work, and for the many rewarding discussions. I also wish to thank my assistant supervisor Lennart Ohlsson for his support when Tomas was not available.

I would also like to thank my colleagues in the computer graphics group for all great discussions and brainstorming (Petrik Clarberg, Calle Lejdfors, Jacob Munkberg, and Jim Rasmusson), and all the masters students I have worked with during their thesis projects.

The work presented in this thesis was carried out within the Computer Graphics group (LUGG) at the Department of Computer Science, Lund University. It was partly funded by Vetenskapsrådet. Also, thanks to ATI for their fellowship program funding.

Finally, I would like to thank my family and my friends for always being there for me.

Preface

This thesis summarizes my research on performance improvements for the rasterization pipeline. The following papers are included:

I. Jon Hasselgren and Tomas Akenine-Möller, “A Family of Inexpensive Sampling Schemes”, in *Computer Graphics Forum* 24(4):843–848, 2005.

II. Jon Hasselgren, Tomas Akenine-Möller and Lennart Ohlsson, “Conservative Rasterization”, in *GPU Gems 2*, pages 677–690. Addison-Wesley Professional, 2005.

III. Jon Hasselgren and Tomas Akenine-Möller, “An Efficient Multi-View Rasterization Architecture”, in *Proceedings of EGSR*, pages 61–72, 2006.

IV. Jacob Munkberg, Petrik Clarberg, Jon Hasselgren and Tomas Akenine-Möller, “High Dynamic Range Texture Compression for Graphics Hardware”, in *ACM Transactions on Graphics*, 25(3):698–706, 2006.

V. Jon Hasselgren and Tomas Akenine-Möller, “Efficient Depth Buffer Compression”, in *Graphics Hardware*, pages 102–110, 2006.

VI. Jim Rasmusson, Jon Hasselgren and Tomas Akenine-Möller, “Exact and Error-bounded Approximate Color Buffer Compression and Decompression”, in *Graphics Hardware*, pages 41–48, 2007.

VII. Jon Hasselgren and Tomas Akenine-Möller, “PCU: The Programmable Culling Unit”, in *ACM Transactions on Graphics*, 26(3):92, 2007.

VIII. Jon Hasselgren, Jacob Munkberg and Tomas Akenine-Möller, “Automatic Pre-Tessellation Culling”, *ACM Transactions on Graphics*, 28(2):19, 2009.

The following papers are also published but are not included in this thesis:

- Jon Hasselgren and Tomas Akenine-Möller, “Textured Shadow Volumes”, in *Journal of Graphic Tools*, 12(4):59–72, 2007.

- Tomas Akenine-Möller, Jacob Munkberg and Jon Hasselgren, “Stochastic Rasterization using Time-Continuous Triangles”, in *Graphics Hardware*, pages 7–16, 2007.

- Jacob Munkberg, Petrik Clarberg, Jon Hasselgren and Tomas Akenine-Möller, “Practical HDR Texture Compression”, in *Computer Graphics Forum*, 27(6):1664–1676, 2008.

- Jacob Munkberg, Jon Hasselgren and Tomas Akenine-Möller, “Non-Uniform Fractional Tessellation”, in *Graphics Hardware*, pages 41–45, 2008.

- Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren, Jacob Munkberg, Petrik Clarberg and Tomas Akenine-Möller, "Floating-Point Buffer Compression in a Unified Codec Architecture", in *Graphics Hardware*, pages 75–84, 2008.
- Henrik Malm, Magnus Oskarsson, Eric Warrant, Petrik Clarberg, Jon Hasselgren, Calle Lejdfors, "Adaptive enhancement and noise reduction in very low light-level video", in *Proceedings of ICCV'07*, pages 1–8, 2007.

Contents

1	Introduction	1
2	The Rasterization Pipeline	3
	2.1 Discussion	6
3	Rasterization	8
	3.1 Sampling	9
	3.2 Multisampling	10
	3.3 Conservative Rasterization	11
	3.4 Multi-viewpoint rasterization	13
4	Compression	16
	4.1 Texture Compression	17
	4.2 Buffer Compression	19
5	Culling	21
	5.1 Programmable Pixel Culling	23
	5.2 Programmable Vertex Culling	24
6	Conclusions and Future Work	26
Bibliography		29
Paper I: A Family of Inexpensive Sampling Schemes		**33**
1	Introduction	35
2	Previous Work	35
3	Notation	37
4	The FLIPTRI Sampling Scheme	38
5	Sampling patterns	39
	5.1 Initial Pattern Generation	39
	5.2 Pattern Ranking and Optimization	39
6	Results	39
	6.1 Evaluation	40

7 Conclusion and Future Work 42
Bibliography 45

Paper II: Conservative Rasterization **47**
1 Introduction 49
2 Problem Definition 49
3 Two Conservative Algorithms 52
3.1 Clip Space 52
3.2 The First Algorithm 52
3.3 The Second Algorithm 54
3.4 Underestimated Conservative Rasterization 57
4 Robustness Issues 57
5 Conservative Depth 58
6 Results and Conclusions 59
Bibliography 61

Paper III: An Efficient Multi-View Rasterization Architecture **63**
1 Introduction 65
2 Motivation 66
3 Background: Multi-View Rasterization 68
3.1 Brute-Force Multi-View Rasterization 68
3.2 Multi-View Projection 69
4 New Multi-View Rendering Algorithms 69
4.1 Scanline-Based Multi-View Traversal 71
4.2 Tiled Multi-View Traversal 72
4.3 Approximate Pixel Shader Evaluation 73
4.4 Accumulative Color Rendering 77
5 Implementation 78
6 Results 79
6.1 Accumulative Color Rendering 83
6.2 Small triangles 84
7 Discussion 84
8 Conclusion and Future Work 85
Bibliography 87

Paper IV: High Dynamic Range Texture Compression for Graphics Hardware **91**

1 Introduction . . . 93
2 Related Work . . . 94
3 Color Spaces and Error Measures . . . 96
3.1 Color Spaces . . . 96
3.2 Error Measures . . . 97
4 HDR S3 Texture Compression . . . 99
5 New HDR Texture Compression Scheme . . . 100
5.1 Luminance Encoding . . . 100
5.2 Chrominance Line . . . 102
5.3 Chroma Shape Transforms . . . 103
6 Hardware Decompressor . . . 105
7 Results . . . 107
8 Conclusions . . . 110
Bibliography . . . 113

Paper V: Efficient Depth Buffer Compression **117**
1 Introduction . . . 119
2 Architecture Overview . . . 119
3 Depth Buffer Compression - State of the Art . . . 121
3.1 Fast z-clears . . . 121
3.2 Differential Differential Pulse Code Modulation . . . 122
3.3 Anchor encoding . . . 123
3.4 Plane Encoding . . . 124
3.5 Depth Offset Compression . . . 125
4 New Compression Algorithms . . . 126
4.1 One plane mode . . . 127
4.2 Two plane mode . . . 129
5 Evaluation . . . 132
6 Conclusions . . . 133
Bibliography . . . 135

Paper VI: Exact and Error-bounded Approximate Color Buffer Compression and Decompression **137**
1 Introduction . . . 139
2 State-of-the-art Color Buffer Compression . . . 139
2.1 Multi-Sampling Compression . . . 140
2.2 Color Plane Compression . . . 140

2.3 Offset Compression 141
2.4 Entropy Coded Pixel Differences 142
3 A New Exact Color Buffer Compression Algorithm 143
3.1 Reversible Color Transforms 143
3.2 The Algorithm 144
4 Error-bounded Approximate Compression 146
4.1 The Error Control Mechanisms 147
4.2 A Lossy Algorithm 147
5 Results 148
5.1 Exact Compression 149
5.2 Approximate Compression 151
5.3 Structural Similarity Index - SSIM 153
6 Conclusions and Future Work 153
Bibliography 155

Paper VII: PCU: The Programmable Culling Unit **157**
1 Introduction 159
2 Previous Work 160
3 Programmable Culling Unit Overview 162
4 PCU Interaction 164
4.1 Driver 164
4.2 Hardware 166
5 Implementation - Combined Shader Unit 168
6 Results 170
6.1 Small Triangles 173
7 Discussion 173
8 Conclusion 174
A Interval Arithmetic Instruction Set 175
A.1 Arithmetic and Conditional Instructions 175
A.2 N-Dimensional Texture Lookups 175
A.3 Cube Map Lookups 178
Bibliography 181

Paper VIII: Automatic Pre-Tessellation Culling **183**
1 Introduction 185
2 Tessellation Culling 187
2.1 Overview 187

2.2 Taylor Arithmetic 188
2.3 Tight Polynomial Bounds 190
2.4 Program Analysis and Generation 192
2.5 Selective Execution 195
2.6 Culling 195
3 Implementation 198
4 Results 199
5 Conclusion and Future Work 202
Bibliography 205

1 Introduction

Computer graphics is the science of generating, or *rendering*, images on a computer. In essence, a screen has a limited number of picture elements, or *pixels*, which can be set to any given color, and a graphics algorithm is merely a way to determine which color to display at each pixel. Even though we may not think about it, computer graphics has become part of our everyday life. The most evident use is in computer games and entertainment, but it is also heavily used for special effects in the movies we watch, as well as in the graphic overlays on news and sports TV-broadcasts. Another example of widespread use is mobile phones and portable music and video players. In addition, computer graphics is used to visualize measurements from medical equipment in order to help doctors make the right diagnosis, for example.

A subfield of computer graphics that has been given particular attention is that of three-dimensional graphics. Here, a two-dimensional image is rendered from a three-dimensional model of a virtual scene or world, at a given viewpoint. Such scene models are typically constructed from several objects, where each object is built from triangles that approximate the object's actual surface as shown in Figure 1. The triangles are given material properties that describe how light interacts with the surface, and the task of the rendering algorithm is to determine how light is transferred throughout the scene. The algorithm follows the light from a virtual light source, such as the sun, and registers each surface the light hits before it reaches the viewpoint, thereby giving color to a certain pixel. This is called the light transport problem, and it turns out to be a very complex problem to solve.

The complexity of the light transport problem has divided three-dimensional computer graphics into two branches: *photo-realistic* and *real-time* rendering. Photo-realistic rendering focuses on image quality and spends a lot of resources on solving the light transport problem accurately. The rendering algorithms are often

Figure 1: *Left:* A three-dimensional model is constructed from vertices connected to form triangles. *Middle:* When the triangles are rendered, the surface of the model appears. *Right:* Finally, the triangles are given material properties in order to create a more realistic image. These images are based on a free model from www.artist-3d.com.

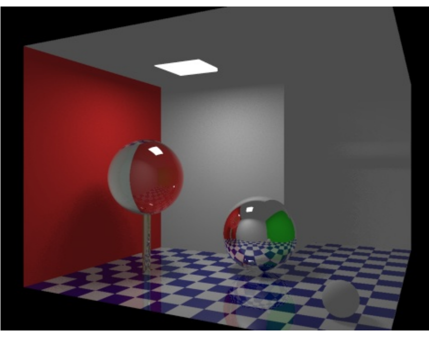
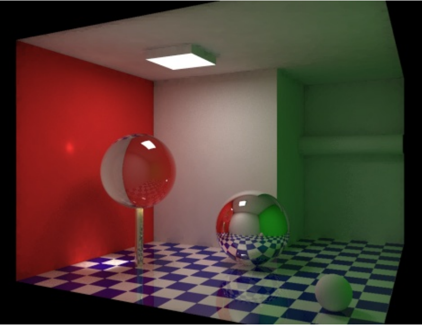
 

Figure 2: *Left:* An image rendered with a simple light model, not including any indirect lighting. *Right:* The same image rendered with a physically based lighting model. Note that a great deal of the light in this scene actually comes from indirect lighting. As a result, the rendered image to the right looks more realistic, and it also makes it easier to perceive the depth and shape of the scene.

based on physics, and simulate how light interacts in the real world. They capture effects such as indirect lighting, which occurs when light bounce off several surfaces before reaching the eye. Although this may sound like a very subtle effect, it contributes greatly to the perceived realism, as can be seen in the images in Figure 2. In an ideal world, photo-realistic rendering would be used for all purposes, but due to its complexity, it may take a single computer minutes, hours, or even days to render a single image. Photo-realistic rendering is therefore used primarily in the film industry, where a film is rendered during the course of months on big clusters of computers.

The most typical example of the other subfield, real-time graphics, is games, which have two requirements that make accurate photo-realism impractical. First, the player is able to interact with the virtual world as the game is played, so we cannot precompute a movie clip. Second, the images must be rendered in a high enough pace so they are perceived as a fluid animation rather than separate images. This typically requires around 60-100 images to be rendered each second. Therefore, rendering performance always comes first in real-time graphics and quality comes second. The rendering quality is made as high as possible without breaking the performance requirements.

Real-time graphics has received enormous attention during the last few years, and with the introduction of dedicated graphics hardware, performance has increased thousandfold in the last decade. Each time computational power is increased, the extra performance is used to improve image quality. However, consumers quickly grow accustomed to that level of image quality, and we once again need to improve performance in order to improve quality. This is a seemingly never ending cycle, where we continue to improve on graphics performance with the long term goal of bridging the gap between photo-realistic and real-time graphics. There is a general

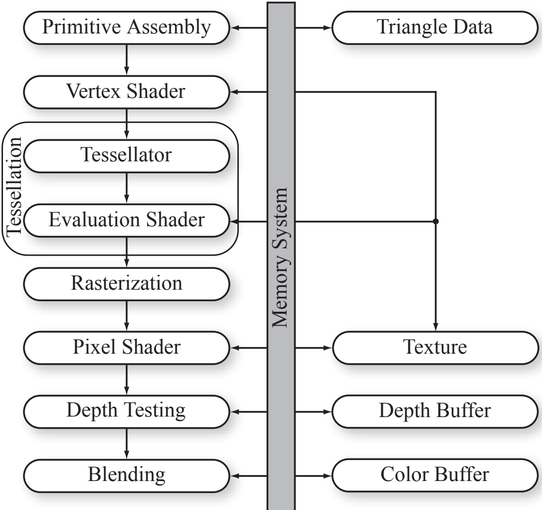


Figure 3: Schematic overview of a modern rasterization pipeline, such as the ones found in current GPUs. The boxes on the left side show which operations are performed in order to render a triangle. As can be seen in the illustration, some of these functional units access storage buffers through the memory system.

yearly performance growth that comes from better manufacturing technology of the hardware silicon. However, this growth alone will not be enough to reach photo-realistic quality in a foreseeable future, for complex scenes. In order to reach that goal, we need to improve on current real-time rendering algorithms, and that is the focus of this thesis.

In this thesis, we propose optimizations and improvements for the rasterization algorithm, which is, at the time of writing, by far the most popular rendering algorithm for real-time graphics. The improvements aim to increase performance with no, or very little, visual loss in image quality. The primary focus is graphics hardware, but our algorithms can also be useful for software rasterization frameworks.

2 The Rasterization Pipeline

Today, rasterization is the de-facto rendering algorithm used for real-time graphics applications, e.g., games, but it is also frequently used in high-quality rendering in the feature film industry. Rasterization is a very simple algorithm that can be parallelized very efficiently, and this has made it a perfect candidate for hardware implementation, which has in turn greatly contributed to its popularity. Another desirable feature of the rasterization algorithm is that triangles are processed one

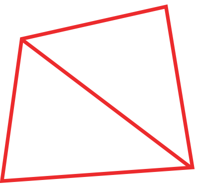
at a time, independently of each other. Therefore, scene data can be streamed in from removable media (such as, for example, a hard drive) and need not be stored in memory. This is particularly important in high-quality film rendering, where scenes often are very complex.

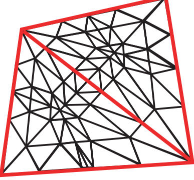
Figure 3 shows a schematic overview of a modern rasterization pipeline, as implemented by most modern graphics hardware architectures. The left part of the figure shows the functional units a triangle pass through in order to be rendered. Some of the functional units access external memory buffers through the memory system, also shown in the figure. A memory access is typically an expensive operation [34] when compared to computations or logic operations, and therefore it is important to minimize the memory traffic as much as possible.

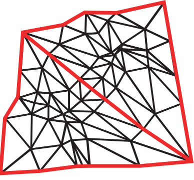
Primitive Assembly In the first step, the primitive assembly, a triangle is created by fetching the three corresponding *vertices* from memory. A vertex is specified by its *attributes*, which are typically position, normal, color, and texture coordinates. However, due to the programmability of modern *GPUs* (graphics processing units), the programmer is free to specify any attributes she or he like. In programming terms, one could say that the attributes can be seen as function parameters sent from the CPU to the GPU.

Vertex Shader After primitive assembly, the vertices of a triangle are sent to the vertex shader unit [24] (also known as the vertex program unit). This unit is essentially a microprocessor that executes a vertex shader program for all vertices of the triangle. The vertex shader program is written by the application programmer and can be customized to implement visual effects such as skinning, cloth simulation, and collision detection. The only strict constraint on the vertex program is that it must compute the (unprojected) screen position of each vertex of the triangle. However, the vertex program can also be used to compute additional vertex attributes. If that is the case, the attributes will continue down the pipeline, and can be used in the later pixel shader unit.

Tessellation Tessellation is a recent addition to the graphics hardware rasterization pipeline [13, 14, 46], although it has been used in software-based high-quality rasterizers [9] for a very long time. The tessellator takes a *base triangle*, which is simply the triangle we are currently rendering, and subdivides it into a number of smaller *micro triangles* as shown in Figure 4. The tessellator then executes an *evaluation shader* (sometimes also called a domain shader), which is very similar to the vertex shader in that it is used to alter the position of the newly created vertices. A typical application of tessellation is shown in Figure 4. Two input triangles are tessellated, and a height map is used to displace the position of the new vertices, thus creating a landscape.







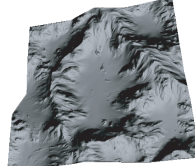


Figure 4: When tessellation is enabled, each base triangle is subdivided and several new micro triangles are created. The evaluation shader then alters the position of all the vertices which displaces the newly created triangles, and finally the triangles are rasterized. In this example, a detailed landscape is created from only two input triangles

Rasterization Once the vertices' screen space positions have been determined, the triangle is processed in the rasterization unit [26, 33, 35] which identifies the pixels it overlaps. When an overlapping pixel is identified, a *fragment* is generated. A fragment is essentially a container for all data needed through the pipeline in order to compute the final color of a point in the scene. The rasterizer computes fragment attributes by interpolating the corresponding attributes of the three vertices, and sends the fragment further down the pipeline. It should be noted that the rasterization unit acts as a data amplifier as it takes triangles as input and outputs fragments. Typically, a triangle covers more than one pixel thereby generating several fragments, and in the extreme case a single triangle may cover the entire screen.

Pixel Shader This unit is also sometimes referred to as the fragment shader or fragment program unit, as it operates on fragments rather than pixels. However, we will refer to it as the pixel shader in the remainder of this text since this is most commonly used. Like the vertex shader unit, this is a microprocessor that executes a user-defined program for every fragment which is generated in the rasterization step. The pixel shader program's primary task is to compute the final color of each fragment as they are drawn to the screen, and it will thereby affect the visual appearance of the object. Some examples of effects that can be implemented in a pixel shading program are lighting models, shadowing, reflection, volumetric lighting effects and custom clipping. In order to render visually rich images, the pixel shader algorithms often need to access texture maps containing surface structures, reflection parameters, and similar data. The texture maps are stored as images in external memory, and therefore the pixel shader unit is one of the main consumers of memory bus bandwidth.

The vertex and pixel shader units are very similar in the way they operate. A vertex or a fragment can be treated equally as they are both defined by a set of attributes. This has led to the *unified shader* unit which, although it still operates as a separate

vertex and pixel shader from the application programmer's point of view, uses the same physical hardware resource to execute both vertex and pixel shaders. This approach uses hardware resources more efficiently, and also have the benefit that it makes load balancing easier. If a scene has more vertices than fragments the unified shader unit will spend more of its time executing vertex shader programs, while in a system with separate shader units, the pixel shader unit would be idle, waiting for the results from the vertex shader unit.

Depth Test and Blending Once the fragment has passed the pixel shader unit, its final color and depth have been computed by the pixel shader program. Before the pixel is colored, a depth test [16, 45] is performed to ensure that occluded triangles never are drawn on top of, or bleed through, the triangles that occlude them. This test requires a depth buffer, which is a memory buffer where the depth of each pixel is stored. Whenever a fragment comes to the depth test, the fragment depth is compared to the depth in the depth buffer. If the fragment depth is greater than, or further away from the viewer than the value stored in the depth buffer then the fragment is discarded since it is occluded by something. On the other hand, if the fragment depth is less than (that is, closer to) the viewer than the value stored in the depth buffer, the value in the depth buffer is updated with the depth of the current fragment and the fragment color written to the color buffer in the following blending stage. As the depth test relies on memory accesses, it also consumes a relatively high amount of bandwidth. It should be noted that most graphics hardware allows the exact behavior of the depth test to be configured to allow a wide variety of algorithms to be implemented. However, the "less than" test described here is by far the most commonly used.

Finally, fragments that pass the depth test are written to the color buffer. Instead of simply overwriting the previous color, the application programmer may configure the color blending [4] unit to mix the fragment color with the color previously stored in the color buffer. The primary use of this unit is to render transparent triangles, which is used, for instance, when rendering transparent objects or particle systems. From a memory perspective, blending is very similar to depth testing as both are read-modify-write operations. These steps therefore consume roughly equal amounts of memory bandwidth. However, there are some exceptions to that rule as high dynamic range rendering [37] increases color bandwidth, and shadow rendering algorithms typically rely heavily on depth buffer bandwidth.

2.1 Discussion

Recent developments of graphics hardware have introduced more and more programmable steps into the pipeline, and therefore the graphics cards are starting to look more and more like many-core systems. However, as the rasterization pipeline is a fundamental part of the hardware design, only algorithms that fit into this model can be efficiently implemented. A rather different approach is taken by Intel with their currently unreleased product Larrabee [40], which is a

pure many-core processor on which rendering algorithms will be implemented in software with some fixed-function hardware for special case operations such as texture filtering. This has the advantage that rasterization software can continually be improved and customized as new algorithms or optimizations emerge, and it also allows automatic load balancing since there are very little hardware resources allocated for fixed functionality. Whether or not this new product will prove efficient enough despite its programmability remains to be seen. However, there is an apparent convergence between many-core systems and graphics hardware. This makes the algorithms presented in this thesis more attractive as many of them can be implemented entirely in software in a very near future.

So far, we have focused on the desirable features of the rasterization algorithm. However, as previously mentioned, rasterization is really a brute force algorithm. Although it is perhaps this aspect that has made it such a good candidate for hardware implementation, it does not mean that a naive implementation will perform well. Many improvements and optimizations have already been adopted by modern graphics hardware. They focus primarily on reducing *computational complexity* and *memory bandwidth usage*. Reducing computational complexity is a pretty straightforward optimization. With a given hardware budget there is only so many computational units you can realize, and therefore you will be bound by some theoretical computational throughput, typically measured in *FLOPS* (floating point operations per second). If you can reformulate your algorithms to use fewer computations, you gain performance. An example is tiled rasterization [27] where you first test if a *tile*, which is a block of for example 8×8 pixels, overlaps the triangle being rasterized. If the tile overlaps the triangle, you need perform an overlap test for each of the 64 pixels, but if it does not overlap you can proceed to the next tile. Since it is less expensive to test if a tile overlaps a triangle than it is to test 64 pixels, you will gain computational performance on average. Another example is to optimize the pixel shader program. Since each instruction in the pixel shader program takes some time to execute, we can very easily gain performance if we can remove instructions or substitute complex instructions for simpler alternatives.

The other target of optimizations is to reduce memory bandwidth usage and this is typically done by improving coherency in memory lookups, by using caches more efficiently, and by compressing memory contents. The general trend of computer hardware is that computational throughput grows at a much faster rate than memory bandwidth [34]. Fortunately, graphics-specialized hardware is much more forgiving than normal processors in this regard since wider memory buses can be used more efficiently, and memory latency can be hidden by processing several pixels in parallel. However, memory throughput is still a great problem which has received much attention in graphics hardware based research.

This thesis covers several novel algorithms targeted for the rasterization pipeline. The focus is mainly on performance improvements both targeting computational complexity and memory bandwidth. We also try to keep modifications to the rasterization pipeline as small as possible in order to make our algorithms easy to implement in existing hardware solutions. With the recent advance of more

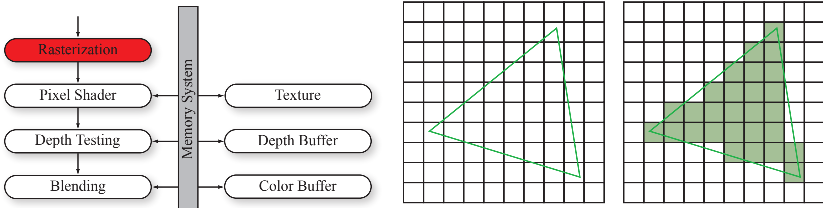


Figure 5: *Left:* Modifications to the rasterization algorithms focus on that particular unit. However, they can also indirectly affect units further down the pipeline. *Right:* Rasterization finds all pixels that overlaps a given triangle. For each such pixel, the rasterizer generates a fragment which contains all data necessary to process that pixel, and sends the fragment further down the pipeline.

programmable hardware, we believe that the algorithms are of practical importance and could be implemented on next generation graphics cards with good results. The algorithms focus mainly on three areas: rasterization, compression, and culling

3 Rasterization

As previously mentioned, rasterization determines which pixels overlap a triangle, as shown in the right part of Figure 5. In the simplest form of rasterization, one sample point is chosen to represent each pixel. Determining if a pixel overlaps the triangle or not is done by testing if the sample point lies inside the triangle. This test can be done using *edge functions* [35]. If a pixel overlaps the triangle, a fragment is generated by interpolating all vertex attributes to compute their values at the sample point. The fragment is then passed down the pipeline as shown to the left in Figure 5. It should be noted that changes to the rasterizer may change the data structure of a fragment. This may lead to indirect changes in the fragment processing parts of the pipeline.

In a naive implementation, rasterization would perform the overlap test for all pixels on the screen each time a triangle is rendered. This is not efficient, and consequently many algorithms limit the number of pixels visited either by using some overestimation of the triangle, such as for example its bounding box, or by starting from a pixel overlapping the triangle and visiting all neighboring pixels that also overlap the triangle. Some algorithms, such as recursive descent, work by starting with the whole screen as an overestimation of the pixels the triangle overlaps. The screen is then recursively subdivided into smaller and smaller boxes until a box is not overlapping the triangle, or the box is the same size as a pixel.

Traditionally, the rasterization step is preceded by a clipping [32] and projection step in order to clip the triangle by the view frustum and then project the clipped

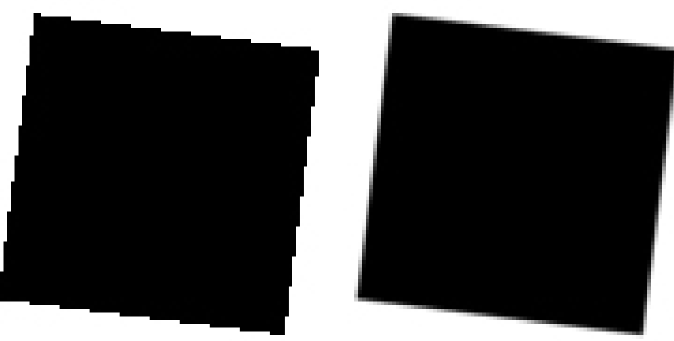


Figure 6: The right box is rendered with a single sample per pixel and the right box is rendered using supersampling and 64 samples per pixel.

triangle on the screen. However, clipping algorithms are complex to realize in hardware, and therefore a popular alternative is to use homogeneous rasterization algorithms [26, 33] which directly rasterize the unprojected triangle.

3.1 Sampling

The sampling process is the core of any rasterization algorithm, and just like any other sampling process it is subject to aliasing if the sampling frequency is not high enough. Unfortunately, models built from triangles will have discrete silhouette edges, which have infinite frequency content, so we can never avoid aliasing completely with point sampling strategies. However, by using a high enough sampling rate, we can create a result that is more or less visually indistinguishable from the correct image. To do that, we over-sample the image by using more samples than pixels in the final image. The image is then reconstructed through *filtering* where the color of each pixel is computed as a weighted sum of the colors of the samples within the filter kernel. This is called *supersampling* [41], and the effect is illustrated in Figure 6. High-quality rendering typically use complex filter kernels that covers more than one pixel, such as Gaussian filters. In real-time rendering however, it is still very common to use simple box filters where the color of a pixel is computed as the average of the samples in that pixel.

A disadvantage of supersampling is that it requires huge amounts of pixel processing power. Processing n samples per pixel results in n times more computations in the fragment processing part of the pipeline. Memory bandwidth is also greatly increased, although caches will make the increase slightly sub-linear. An alternative sampling approach that eases this burden is *multisampling*. It is based on the assumption that the majority of the high-frequency content in an image comes from geometry (i.e., triangle edges) and not from shading. With that in mind, several sampling points can be used to determine if a pixel lies within a triangle,

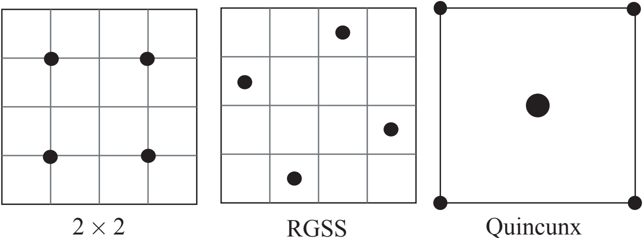


Figure 7: Sample placement is important for visual quality. The RGSS pattern is for instance known to give higher visual quality than the 2×2 pattern. Another trick to increase quality is to share samples between pixels. In the Quincunx pattern, the corner samples can be shared by four pixels. Therefore, the cost is two samples per pixel rather than five as one might think.

but the pixel shader program only needs to be evaluated in a single point. The color computed by the pixel shader can then be propagated to all samples that were determined to overlap the triangle. The effect of multisampling is that pixels completely within one triangle will effectively only use a single sample while pixels on triangle edges will get samples from multiple triangles thus smoothing the edge. From a computational and memory bandwidth standpoint, the only difference between multisampling and supersamling is that the pixel shader program is executed only once per pixel for multisampling. However, this is a huge advantage as the pixel shader execution is the most costly part of the fragment pipeline. Furthermore, the pixel shader program contains all texture lookups, which is often the main consumer of memory bandwidth.

In the following, we describe three papers on rasterization (Papers **I-III**). Although they cover very different areas, they can all be seen as different strategies for sampling tailored for their respective contexts.

3.2 Multisampling

Even though multisampling helps reduce the most costly part of the supersampling algorithm, it is still a very costly process as rasterization, depth testing, and blending stages must process many more samples. Therefore, it is important to place the sample points strategically to achieve the highest possible visual quality with as few samples as possible. For instance, the RGSS (rotated grid super sampling) [41] pattern shown in Figure 7 has long been known to produce higher quality images than the naive 2×2 pattern, also shown in that figure.

The reason why some patterns are visually more appealing than others have been investigated by Naiman [31], who studied how the human eye perceives aliasing of edges at different orientations. Perhaps not entirely surprisingly, it turns out that humans are considerably more sensitive to aliasing of near horizontal and vertical

edges. In between those extremities the sensitivity was relatively low with a peak around 45 degrees. The result of Nainan's study motivates why the 2×2 pattern from Figure 7 is considered a bad sampling pattern. For near-horizontal or vertical edges, two sample points will trigger at almost the same time, and therefore neighboring pixels will go from being completely outside a triangle to having two sample points inside the triangle. Since RGSS is a rotated version of the 2×2 pattern this effect will happen at edge orientations where we have considerably lower visual sensitivity, and this explains why RGSS is a better pattern in practice. Laine and Aila [23] have later quantified sample quality by using an error metric based on the results of Naiman's studies. It should be noted that this quality metric goes in contrast to other sample placement strategies used in high quality rendering [21, 41], which primarily focuses on reducing the variance of a sampled signal. However, it is our belief that it is well suited for real-time rendering as we can only afford very few samples, in which case reducing the perceptual aspects will be more important than variance reduction. Furthermore, the edge studies of Naiman goes hand in hand with the multisampling concept as it exclusively focuses on reducing edge aliasing.

Besides sample placement, sample sharing is also a technique that can be used to improve quality. The Quincunx pattern [10] in Figure 7 is a typical example of sample sharing. When this pattern is replicated, the corner samples will share the exact same positions for neighboring pixels and can therefore be reused.

In Paper **I**, we explore a family of very inexpensive sampling patterns that use between 1.25 and 2 samples per pixel, and present the configurations that will perform best according to the error metric defined by Laine and Aila. We consider sampling patterns that are mirrored horizontally or vertically each time we move one pixel in the horizontal and vertical direction, respectively. Such patterns can share all sample points in pixel corners between four pixels, and all sample points on pixel edges can be shared by two pixels. We performed an exhaustive search of the possible configurations and manually selected the ones that we believed to work well according to Naiman's findings. We then ran an optimization process based on Laine and Aila's error metric to compute final positions and filter weights for the samples. The optimization process is not exhaustive, which means that the presented patterns may not be optimal. However, they perform well relative to their cost as we show in our evaluation, and we believe that these patterns could be very useful for low end-graphics, e.g., for use in mobile devices.

3.3 Conservative Rasterization

Conservative rasterization can be seen as a special case of sampling. Normally when we rasterize, we only want each sample point to belong to a single triangle, and therefore normal rasterization algorithms have consistent rules to assign sample points that lie on triangle borders to one of the neighboring triangles. All this is necessary to avoid flickering in edges where two triangles meet. Conservative rasterization, however, considers the whole *pixel region* as the sampling

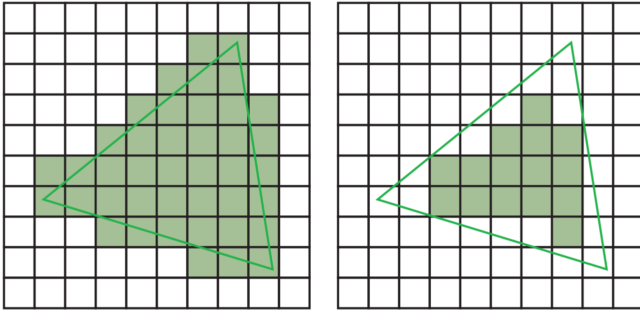


Figure 8: Two forms of conservative rasterization. Overestimated rasterization, shown to the left, visits all pixels that partly overlap the triangle. Underestimated rasterization, shown to the right, visits only the pixels that lie completely within the triangle.

area. As shown in Figure 8, there are two types of conservative rasterization. An overestimating variant that visits all pixels regions touched by the triangle, and an underestimating variant that only visits pixel regions completely within the triangle. In this thesis, we focus primarily on the overestimated variant since it is most practically useful.

Although conservative rasterization has little value for normal rendering, it has the desirable feature that we can accurately detect and handle overlapping polygons. This is particularly important in advanced shadowing algorithms [5, 42], culling algorithms [25], and collision detection algorithms [15, 30]. Since the utility of conservative rasterization has increased, Akenine-Möller and Aila [2] proposed simple extensions to the rasterization unit to support conservative rasterization. However, so far no graphics hardware vendors have thought this feature important enough to implement in hardware. Therefore many algorithms have had to rely on normal rasterization, thereby making it impossible to guarantee their correctness.

In Paper **II**, we present two novel algorithms for conservative rasterization. They are designed to work on existing graphics hardware, and can be implemented entirely in vertex and pixel shader programs. The first algorithm was inspired by Akenine-Möller and Aila's algorithm, but modified to fit into the current graphics pipeline. In the vertex shader, we extend the vertices of each triangle so it overlaps the sample point of at least all the pixels that conservative rasterization would traverse. The extended triangle may overlap more pixels than is needed since the shape traversed for overestimated rasterization is not necessarily triangular. We discard the fragments generated for these excessive pixels by using an inexpensive test that is implemented in the pixel shader program. This algorithm has the advantage that it does not increase the vertex or triangle count of the rendered scene. However, for sliver triangles, a large amount of excessive fragments may be generated only to be discarded in the pixel shader program. It can therefore put a

heavy load on the rasterization and fragment processing part of the pipeline. Our evaluation suggests that lower end hardware favors this algorithm. However, this depends on rendering resolution and how the hardware is load balanced for vertex and fragment processing, respectively.

For the second algorithm, we use a different approach and directly generate polygons that cover the centroid sample points of all the pixels that would be traversed by a conservative rasterizer. Since graphics hardware can typically only render triangles, we supply polygons in form of pre-tessellated triangles, where every triangle vertex is duplicated three times. The vertex shader program can then compute the polygons simply by moving the vertices into place. This algorithm has the advantage of generating no unnecessary fragments during rasterization, and therefore do not require a fragment program at all. However, it puts a greater strain on the vertex processing part of the pipeline, since three times as many vertices must be processed. Our evaluation showed that this algorithm is likely to perform better on high-end and modern hardware. Furthermore, the recent addition of geometry programs have made it possible to execute the full algorithm on graphics hardware, without having to send duplicated geometry from the CPU. However, even though at least one recent project [42] has adopted this algorithm with that modification, there are, as of this date, no concrete numbers on how much performance improvement one can expect from using geometry programs.

3.4 Multi-viewpoint rasterization

Our final contribution to the family of rasterization algorithms is in the field of multi-viewpoint rasterization. The most simple form of multi-viewpoint rendering is stereoscopic rendering. Our brain uses the images of both eyes as an important cue to perceive depth in a scene, and therefore it is only natural to render one view for each eye as shown in Figure 9. The viewpoints in the virtual scene should be separated by approximately the distance between the viewer's eyes. Taking the concept of stereoscopic rendering one step further, there are multi-viewpoint displays which show different images from different viewing directions [12, 20] without the need of peripheral equipment such as glasses. We do not only get a stereoscopic effect from our eyes seeing different images, but can also move the head to see the scene from different angles. The screen goes from being a flat surface to being a window into a virtual three-dimensional world.

The technology for multi-view displays exists already today. However, a great problem that remains is how to efficiently render multiple views. The naive approach is to render one view at a time, but this causes performance to scale linearly with the number of views. For high-quality multi-view images, we may need between 32 and 64 views, translating to a performance loss of $32-64\times$. However, as can be seen in Figure 9, there is a lot of coherence between multi-view images, as the different views look very similar. This coherency can be exploited in the rendering algorithm in order to improve performance. Although there has been some research on this topic [1, 36, 43], there has been very few sugges-

Right Eye

Left Eye

Figure 9: A cross-eyes stereoscopic photography of an exhibition car. Note that the two images look very similar, which indicates that they are strongly correlated. An important key to efficient multi-view rendering is to use exploit this coherency in order to improve performance. Original image from Wikimedia Commons.

tions that could be implemented in a standard rasterization pipeline without large changes. One exception is the research of Halle [17], who presents an algorithm for multi-viewpoint rasterization using existing graphics hardware. Unfortunately, since current hardware is not well suited for multi-viewpoint rasterization, this algorithm works best when rendering a very large number of views. Due to the lack of efficient algorithms suited for a moderate or low number of views, the most used solution at this time is to use the naive approach and render one view at a time, resulting in a big performance reduction.

In Paper **III**, we present a novel multi-viewpoint rasterization algorithm that can reduce both memory bandwidth usage and computational complexity. Furthermore, the algorithm is based on the current rasterization pipeline, and we therefore believe that it could be adopted with relatively modest changes in hardware. The algorithm exploits the coherency in multi-view images, and is based on the observation that a specific point on a specific triangle is likely to result in the same texture lookups and computations for all viewpoints. This assumption holds true for all surfaces with materials which are view-independent, and gracefully degrades the more view-dependent terms are added to the material. A wrinkled mirroring surface, such as the ocean scene presented in our results, is more or less a worst case scenario since it has very high frequency content which is completely view-dependent.

Given our assumption, it makes sense to process one fragment of one triangle, render that fragment to all views, and then continue. This is, however, hard to implement in practice, since the triangle is projected differently for different view-

points, which could lead to cracks or overdraw in some of the views. Instead, we rasterize the triangle from all viewpoints in parallel, and sort the rasterization order so that we always pick the view whose next fragment lies closest (on the surface of the triangle) to the previously processed fragment. This means that we will not process the exact same point for all views as in our assumption. However, the assumption also holds well for a close neighborhood around the point.

We present two versions of the rasterization algorithm: one *exact* and one *approximate*. The exact version will produce the exact same result as rendering one view at a time, and consists of the sorted rasterization briefly described above. In itself the sorting will greatly improve performance of the texture cache which is already used in all modern graphics hardware. The sorted rasterization traverses fragments which lie very close to each other on the surface of the triangle. Therefore, in the ideal case, the texture lookups will always be in the cache for all but the first of the views, and the texture bandwidth will be identical to single-view rendering. We will never quite reach the ideal case, but our evaluation shows that our algorithm gives close to constant texture bandwidth for between 2 and 16 viewpoints. The ocean scene does not perform as well, but this is only to be expected since it is a very hard case given our initial assumption.

Although our exact algorithm greatly reduces texture bandwidth, it still requires that we execute the pixel shader for every pixel and every viewpoint, and this can become very costly. In our approximate extension, we start from our exact algorithm and add a cache that stores results of pixel shader evaluations. Each time we process a fragment, we query the cache to see if we find one or more pixel shader evaluations in "close enough" vicinity of the current fragment. If we find appropriate cached results, we compute a color for the fragment by interpolation, and skip the pixel shader execution altogether. Whereas the exact algorithm only relies on our initial assumption to improve performance, this approximate extension relies on the same assumption for the correctness of the image. Therefore, as we show in our results, the performance of the approximate version will be the same for scenes with and without view-dependency. However, the approximate version will treat view-dependent effects as if we used single viewpoint rendering and therefore the rendered images may not be correct. If the viewer can perceive this incorrectness or not depends on the scene and spread of the viewpoints. The ocean scene, for example, has a highly distorted reflection, and therefore the viewer may not notice that it comes from a single viewpoint. If we use a flat mirroring surface instead, the viewer would immediately notice the loss of depth perception. Since the effects of approximation depend so heavily on surface properties, it is important to give the application programmer full control of when to use either the exact or approximate technique. The approximate algorithm can give substantial performance gains when used appropriately. In a stereoscopic rendering system, we observed that up to 95% of the fragments could be approximated for the secondary view.

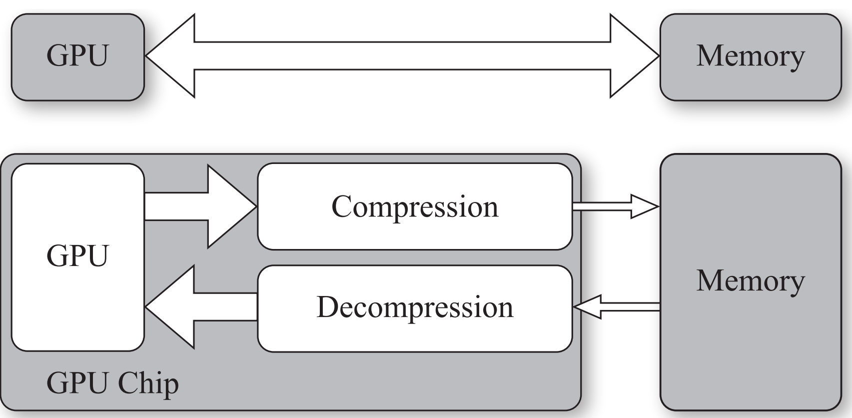


Figure 10: The top row shows a traditional GPU system without compression. Data is sent directly to memory over the bus. The bottom row shows a GPU using data compression. The GPU sends data to a compression unit and receives data from a decompression unit, both realized on the same hardware chip as the GPU. The compression unit compresses data before it is sent to memory, and the decompression unit decompresses data received from memory. Memory bandwidth usage is reduced if the data is successfully compressed.

4 Compression

Data compression is an optimization which focuses entirely on reducing memory bandwidth usage in a graphics system. An example of a GPU using data compression is shown in Figure 10. As can be seen, there are two new units, one that compresses and one that decompresses data, respectively. These units act as a communications channel between the GPU and main memory, and are typically implemented in hardware on the same chip die as the GPU. Compressing and decompressing data require computations, and one can therefore say that compression is a way of trading computational complexity for memory bandwidth.

In GPUs, the two dominating groups of compression algorithms are *texture compression* and *buffer compression*. Although these are quite similar, there are some key differences, of which the first is that buffers are continuously altered during rasterization while textures typically remain static. This has the effect that buffer compression and decompression must be a fairly symmetric operation since data must be both compressed and decompressed in real time when rendering an image. Texture compression on the other hand is an asymmetric process since textures are typically read-only during rendering. This means that the decompression algorithm must be inexpensive and realizable in real time, but the compression algorithm may be complex and slow to execute since the texture can be compressed and stored on disk prior to rendering.

Another difference that comes from the buffers being continuously altered is that buffer compression is much more sensitive to data loss. In texture compression, the image is compressed only once, and in addition, the application programmer has control over which textures she or he wish to compress. Therefore, most texture compression algorithms are *lossy*, and introduce some small error in the compressed image which is usually not visually disturbing. If the compression algorithm should produce a poor result for a particular image, the application programmer can just disable compression for that texture. Buffer compression on the other hand is much less predictable since the number of times a region of the screen is compressed and decompressed depends on the number of triangles rendered in that region. Repeated compression and decompression, so called *tandem compression*, can greatly amplify small errors in each compression step. Therefore, it is necessary to use *lossless* compression, or at least introduce some mechanism that allows the application programmer to control the total error introduced through compression. In any case, buffer compression algorithms always need a lossless fall-back for the case when the maximum allowed error is reached.

4.1 Texture Compression

As can be seen in Figure 3, both the vertex and pixel shader share access to textures. It is therefore reasonable to assume that both the vertex and pixel shader may benefit from texture compression. However, in most cases, the pixel shader is more important since it will consume the vast majority of texture lookups.

As previously mentioned, texture compression is a highly asymmetric operation where compression can be allowed to take orders of magnitude more time than the decompression process. Although there are some cases which break this assumption, such as generating a cube-map by rendering to a texture, these are usually solved by disabling compression or treating texture as a buffer and use buffer compression algorithms [44]. Another important feature for texture compression is that a fixed bit rate is often crucial for efficiency. Typically, a texture image is divided into tiles (blocks) of pixels, and each tile is given a predetermined bit budget. For instance, the S3TC [19, 22] texture compression scheme divides the texture into 4×4 pixel tiles, and each tile is stored using 64 bits of data according to a predetermined format. The reason for using a fixed bit rate is that random access to the texture must be supported. If fixed bit rate is used, it is easy to find the memory position of a tile in constant time. If variable bit rate is used instead, finding a tile requires searching from the beginning of the texture image, or looking up the memory position of the tile in some data structure [18]. These approaches require complex search algorithms to be implemented, and are typically less efficient than the simple approach used in fixed bit rate. This is the reason why variable bit rate has not been adopted in texture compression although it is known to give better image quality than fixed bit rate approaches.

Traditionally, texture images are stored in RGB (red-green-blue) color space with one value for each component. The value is typically in the range $[0,1]$, where 0

Figure 11: These images show the difference between a low dynamic range (LDR) and high dynamic range (HDR) image when the exposure is changed. To the left, the image is shown with some base exposure. When the exposure of the LDR image is changed the image looks the same, just darker overall. In the HDR image, several new details, such as the column of light reflected in the water and the structure of the clouds, appear when the exposure is changed. These images are Copyright ©2004, Industrial Light & Magic, a division of Lucasfilm Entertainment Company Ltd.

indicates that the color channel is turned off, and 1 indicates that the color channel is at the maximum intensity that the screen can display. A problem with such images is that they are based on the properties of the screen, rather than true physical properties of a scene. This is illustrated in Figure 11, which shows an image with a widely varying range of the incoming luminance. There are some relatively dark areas on the trees and the sun is an extremely bright light source. A traditional *LDR* (low dynamic range) encoding of this image fails to accurately capture the whole range of the incoming luminance, which can be seen when the exposure is changed as in the middle image. If we use an alternative image encoding, called *HDR* or high dynamic range [11], we can accurately represent the whole range of incoming luminance. When we change the exposure of this image, as shown to the right in the figure, we can see that the details in the bright parts of the images have been captured.

As HDR images can represent a wider range of luminance, it is only fair to assume that they require more memory space and bandwidth for rendering. Whereas traditional textures are encoded with 8 bits per color channel, HDR textures are typically stored using 16 or 32 bits floating-point values per color channel. Despite this, at the time of writing, there is no support for HDR texture compression from graphics hardware vendors. This has perhaps hindered the development of HDR rendering somewhat since an uncompressed HDR texture requires 12 times the storage and memory bandwidth of a compressed LDR texture.

In Paper **IV**, we present the first texture compression format targeting HDR texture compression. Simultaneously Roimela et. al [38] also developed an HDR texture compression scheme, also published at the annual SIGGRAPH conference in 2006. These compression algorithms complement each other well as Roimela et. al focused on inexpensive hardware while we used a more expensive decompression

algorithm that gives better image quality. Just like S3TC, both formats compress textures by six times. However, since the bit budget is larger for HDR images, we found that our algorithm gives less visual compression artifacts than S3TC in most cases. There have been several new algorithms on HDR texture compression [39, 47, 49], since we published our original algorithm, and although our algorithm remains rather computationally expensive, we see that it still performs very well in terms of visual quality when compared to its competitors. Recently, Microsoft published some information on their upcoming Direct3D 11 API [14], which contains a new HDR texture compression format called BC6. Unfortunately, at the time of writing, there is still no public information about how this format works, or how well it performs. We have also published an additional paper on practical HDR texture compression [29] of further details and findings which we were not able to fit into the original paper. That paper is not included in this thesis.

The paper on HDR texture compression was a joint collaboration between myself, Jacob Munkberg, Petrik Clarberg, and Tomas Akenine-Möller. I was not the primary author of this paper, but was active during the whole process. I contributed to the conceptual design of the algorithm, to some of the exhaustive search algorithms used for verification and some unreleased implementations. I also implemented and evaluated the extended versions of the S3TC compression algorithm, and co-authored in the writing and internal reviewing process of the paper.

4.2 Buffer Compression

The most distinctive feature of buffer compression, when compared to texture compression, is that the contents of the buffer will constantly be modified whereas the texture is mostly static. Furthermore, buffer compression is often transparent to the application programmer, so she or he will not know if compression is enabled or not. Therefore, most buffer compression algorithms are lossless, since lossy compression could cause an application to behave unexpectedly, or generate errors that the programmer would not be aware of.

A shared requirement of both texture and buffer compression is that efficient random access must be supported. For texture compression this is solved using fixed bit rate, but this does not work for buffer compression since lossless compression cannot be guaranteed with fixed bit rate unless the original bit rate is used. Therefore, buffer compression uses variable bit rate encoding, and this is often realized by using a number of fixed bit rate modes. In particular, there is always a fall-back mode where tiles are stored in uncompressed form if all other compression modes fail to represent the data without loss. However, there remains the problem of variable bit rates and random memory access as described in Section 4.1. Variable bit rate requires searching through data structures to find the memory position of a tile, and this is not desirable. The most simple solution to this problem is to reserve enough memory to store an uncompressed version of each tile, as shown in Figure 12. This means that the compressed buffer will use the same amount of

Tile 1	Tile 2	Tile 3	Tile 4

Uncompressed memory layout

Tile 1	Tile 2	Tile 3	Tile 4

Compressed memory layout

Figure 12: Buffer compression stores a data buffer tile by tile. For each tile sufficient memory is reserved to hold the uncompressed tile data, as this makes random accesses in the buffer efficient. A downside of this is that a compressed buffer will require as much memory as an uncompressed. However, we can still reduce memory bandwidth as we do not have to read or write more than the compressed data.

memory as an uncompressed buffer. However, less memory bandwidth is used by only writing and reading compressed data, and all the advantages of fixed bit rate compression are kept.

Buffer compression approaches have long been kept secret internally by the graphics hardware vendors, and very little information about them exists in academia. Our first paper on buffer compression, Paper **V**, is based on information we extracted from patent applications of the main graphics hardware vendors. We present in detail how a modern depth buffer architecture with compression and hierarchical depth culling (HDC) works [16], and it is reasonable to assume that color buffer architectures work in a very similar fashion. We also present a survey of the state of the art in depth buffer compression algorithms, and propose a novel depth compression algorithm of our own.

Our algorithm, just as most of the other depth compression algorithms, is based on representing a tile of pixels by one or more prediction planes, and to store small correction deltas which represent how much the actual pixel values deviate from the prediction planes. In particular, we found that previous work had missed a simple but efficient observation in the interpolation of the prediction plane, which makes it sufficient to use one-bit correction deltas instead of the two bits previously used. Although it is a simple observation, we save approximately one bit per pixel which is a significant gain. Furthermore, we also improve on previous approaches for coding multiple planes, and present robust one and two plane versions of our algorithm. The evaluation shows that our algorithm performs better than previous art, in particular for small tile sizes, such as 4×4 pixels. We therefore believe it is well suited for use in mobile graphics devices.

In our follow-up research, Paper **VI**, we similarly present a survey of existing color buffer compression algorithms, and present a compression algorithm of our own. The compression algorithm is based on the LOCO-I algorithm [48], but with modifications to make it more suitable for compressing small tiles, and on-the-fly compression. This algorithm differs somewhat from the depth compression approaches in that it is a variable bit rate entropy encoder. It is therefore not limited

to a fixed number of "trial and error" modes, but will rather produce the number of bits necessary to represent the tile. However, we still limited ourselves to a number of fixed sizes in the results. The reason for this is that the compressed size of a tile must be stored in an on-chip memory so that the GPU knows the number of bytes to read. If we allow arbitrary sizes, this memory becomes very large, and furthermore the width of the memory bus will limit which sizes that are practical to read anyway.

In this paper we also present the concept of lossy buffer compression. This is a sensitive subject since buffer compression so far has been transparent to the application programmer. However, it makes sense to give the application programmer a way to control the accepted compression loss, and we can easily solve backwards capability problems by allowing no error by default. In the paper, we propose a mechanism to track the total error of a tile, including possible errors introduced through tandem compression, and guarantee that it never exceeds a given threshold value. Although it would be nice to be able to track and control the maximum pixel error, we noted that such a system is much too conservative for practical use. We therefore settle for controlling the maximum allowed *RMSE* (root mean square error) over a tile. Our results show great bandwidth reductions for lossy compression, with an often negligible quality decrease. However, we found some hard cases in which color leakage may lead to visual artifacts.

The paper on color buffer compression was co-authored with Jim Rasmusson and Tomas Akenine-Möller. Although I was not the primary author of this paper, I contributed significantly to it. Both Jim and myself were active in all parts of writing the paper and implementation, although I was more involved in the implementation of the lossless codec, and Jim focused more on the lossy compression aspect.

5 Culling

Generally speaking, *culling* in computer graphics is when we use a simple test to determine if we need to run a complex computation or not. If the test indicates that the computation will not contribute to the final image it can safely be skipped, and consequently performance is improved. A typical example of this is view frustum culling, shown in Figure 13. Here, a complex object is represented by a simple bounding box. If the bounding box is not visible from the camera's point of view, then the same is true for the entire object. Using this simple test, we can quickly skip rasterization of the whole object, and thereby gain performance. It should be noted that culling algorithms often are *conservative*. That is, they may give false negatives, such as the red object in Figure 13 which is incorrectly classified as visible. However, they may never incorrectly cull something which is visible. Whereas a false negative will only affect performance, incorrect culling may give errors in the rendered image.

Current graphics hardware supports a number of fixed-function culling algorithms,

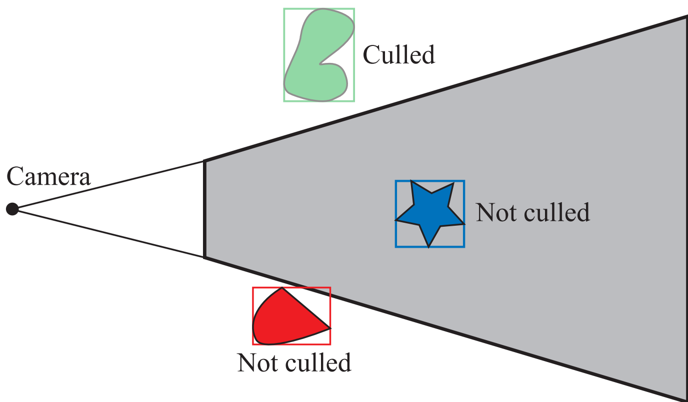


Figure 13: An illustration of view frustum culling, where a bounding box is used to represent each object in the scene. If the bounding box is not visible from the camera, the object is culled. As can be seen for the red object, the algorithm may falsely classify an invisible object as visible. However, a visible object will never be culled, which is important since this could give errors in the rendered image.

such as view frustum culling [6], back-face culling [4], and hierarchical depth culling [16]. However, a great problem is that the graphics hardware becomes more and more programmable while the culling algorithms have stayed the same. This leads to a mismatch between culling and programmability. For example, if we implement a new lighting algorithm in a pixel shader program, it would be nice to add culling to improve its performance. However, in order to do so we must to reformulate our new culling algorithm to fit into any of the aforementioned fixed function-culling algorithms. This is not always possible, or is often so complex that it leads to a decrease in performance when compared to not using culling.

In this thesis, we introduce the concept of *programmable culling*. By analyzing the pixel and vertex shader programs it is possible to automatically generate a *cull shader program* that can be executed for a set of pixels, or a set of vertices. The cull shader program determines if the set of pixels or vertices can safely be discarded without affecting the final image, and if such is the case we gain performance by not having to execute the pixel or vertex shader. There are two main benefits with our programmable culling approach which we believe makes it very attractive. First, the absence of culling will not effect the final image, only the performance. This means that if a hardware vendor chose not to implement programmable culling, or to only implement a subset of it to support some culling algorithms, it will only affect performance and will not break applications that rely on programmable culling. Second, our programmable culling fits well into the current rasterization pipeline. It can therefore be implemented with minimal- or even no changes in current APIs. This makes it both transparent to the application programmer, and makes it simple to ensure backwards compatibility.

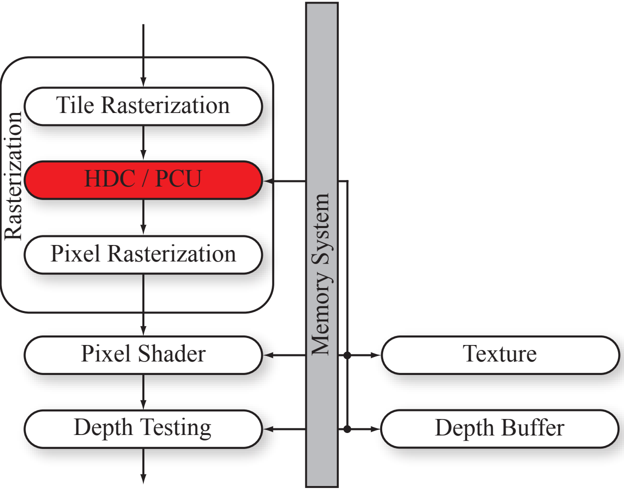


Figure 14: Hierarchical depth culling (HDC) is done in the rasterization unit of the pipeline. The hierarchical depth test is done once for every tile, and therefore the HDC unit is placed between the tile rasterization and pixel rasterization steps. Our PCU is a generalization of HDC culling, where culling is based on a program rather than a fixed algorithm.

5.1 Programmable Pixel Culling

To date, the only standard pixel culling algorithm used in graphics hardware is hierarchical depth culling (HDC). The hierarchical depth culling algorithm is designed to rapidly cull tiles of pixels which are occluded by already drawn geometry. One could say that it is an extra depth test that is performed on a per-tile level. Exactly where in the pipeline the HDC algorithm is implemented is shown in Figure 14. In order to perform this occlusion test, the HDC algorithm needs to keep track of the maximum depth, Z_{max}^{tile}, for each tile stored in the depth buffer. When a triangle is rendered, the rasterizer computes the minimum depth of the triangle, Z_{min}^{tri} over the current tile, and if $Z_{min}^{tri} > Z_{max}^{tile}$ then all the fragments in this tile will fail their depth test, and the entire tile can be discarded early. The algorithm can easily be generalized to support more types of depth test functions (such as greater than, equal to, etc.) by keeping track of both the minimum and maximum depth value for a tile.

In Paper **VII**, we present our programmable (pixel) culling unit, the *PCU*, which can in some ways be seen as a generalization of hierarchical depth culling. Our PCU essentially takes the place of the HDC unit in Figure 14, but instead of being limited to a fixed-function occlusion culling algorithm, we analyze the pixel shader program for culling opportunities. In the pixel shader, there is a special type of instruction called the *kill* instruction, which causes the current fragment

to be discarded and not used in the final image. What we do is to find these kill instructions and all instructions they depend on, and we can then automatically generate a cull shader program which is executed once for each tile in the PCU. The cull shader program is based on interval arithmetic [28], which is a form of conservatively bounded arithmetic. This means that if you perform, for example, an addition between two intervals $\hat{x}$ and $\hat{y}$, then the result of $\hat{x}+\hat{y}$ is a new interval which is guaranteed to contain all $x+y$ where $x \in \hat{x}$ and $y \in \hat{y}$. Running the cull shader using interval arithmetic will tell if the kill instruction is going to trigger for all pixels in the tile, and in such cases we can discard the tile at an early stage which avoids many unnecessary pixel shader evaluations. Although the PCU is modeled as a separate unit in the pipeline, it is in fact just another program being executed and it can therefore be included in a unified shader to make better use of hardware resources.

In the paper, we present several example applications where culling has previously not been practical, including the highly optimized game Quake 4. Our evaluations show that significantly fewer pixel shader instructions have to be executed to render an image when using the PCU, and this translates directly into improved computational performance. Furthermore, we found that the memory bandwidth usage also decreases slightly. This is because we cull tiles early on, and may therefore avoid texture, depth, and color buffer memory accesses. Although reducing memory bandwidth was not our primary goal with the PCU, it is definitely a desired bonus. On a variety of test scenes, we observed between a $1.5\times$ and $2\times$ performance improvement in the pixel processing part of the pipeline, as well as a memory bandwidth reduction of about 15%.

5.2 Programmable Vertex Culling

As mentioned in Section 2, a recent addition to the hardware accelerated rasterization pipeline is the tessellation step. Although this still remains largely unsupported at the time of writing, Direct3D 11 has announced support for tessellation [14]. Hence, it is only a matter of time before mainstream hardware will support it. The tessellation process works as follows. A base triangle is sent to the tessellation unit, and split into several micro triangles. An evaluation shader program is then executed for each vertex of every micro triangle in order to compute its final position. This program can be used to modify the appearance of the tessellated surface, for instance by applying a displacement map [8]. Once the positions of all vertices of a micro triangle have been computed, the micro triangle is passed further down the pipeline, where it is subject to view frustum and back-face culling before being rasterized.

In Paper **VIII**, we present our solution for culling vertices in a pipeline using tessellation. Note that the tessellation process is conceptually similar to rasterization. Instead of splitting a tile into pixels, a triangle is split into micro triangles. However, we can still use the same basic approach as we used in our pixel culling paper. Our goal is to be able to cull an entire base triangle without having to subdivide it

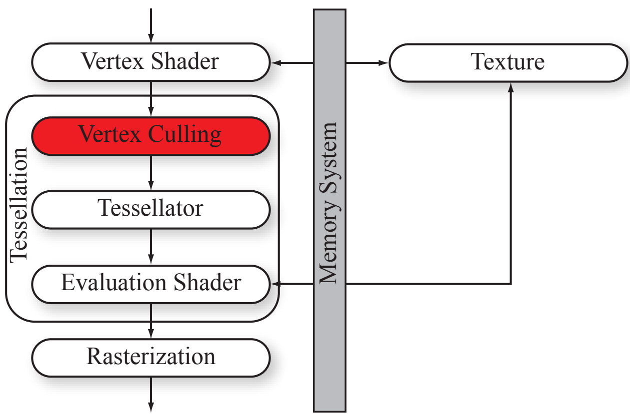


Figure 15: Our programmable vertex culling is placed first in the tessellation unit. If we determine if a base triangle can be culled, we can skip both tessellation of all the micro triangles as well as the execution of the evaluation shader for all generated vertices.

into micro triangles, and without executing the evaluation shader. This is done by adding a new culling step to the pipeline as illustrated in Figure 15. If the culling is successful, we can both skip the subdivision step, and the succeeding step of executing the evaluation shader program for every generated vertex. Using the same approach as in our pixel culling paper, we analyze the evaluation shader program, since it is this program that will determine the position of each vertex in the final image. One thing that differs between the evaluation shader and pixel shader is that there is no kill instruction for an evaluation shader (it does not make sense to discard a single vertex). However, we do know that the tessellator is succeeded by view frustum and back-face culling. This means that if we, for example, can compute the bounding box of all micro triangles that will be generated through tessellation, we can use that bounding box for view frustum culling. If the bounding box is outside, we can ensure that all micro triangles will be outside the view frustum as well, so we do not need to generate them.

Computing the bounding box of a tessellated triangle, which is further displaced through an arbitrary evaluation shader, can be done using the same approach as we used in our pixel culling paper. However, interval arithmetic makes the bounds grow too much to be of any practical use in this context. Instead, we use a more modern form of bounded arithmetic, called Taylor models [7]. Whereas interval arithmetic has problems with higher order polynomials and surfaces, Taylor models handle them very well. Furthermore, Taylor models can track and eliminate redundant or correlated terms in an equation, while interval arithmetic cannot.

In our evaluation, we note that it can be very beneficial to perform vertex culling for highly tessellated scenes. Typically, we can avoid executing enough evalua-

tion shader programs to indicate a speedup of a factor $1-5\times$ of this stage of the pipeline. It should be noted that these figures do not include the possible performance improvements that may arise due to that we can avoid tessellation and micro triangle setup for culled base triangles. We were not able to estimate such performance gains since those depend on how the tessellation and triangle setup units are designed, as well as how the entire hardware system has been load-balanced.

This paper was co-authored with Jacob Munkberg and Tomas Akenine-Möller, and most of the ideas arose from our joint discussions. Both Jacob and I worked on all aspects of the paper. However, I was more involved in the implementation of Taylor models, bounding shaders, and the culling system. Jacob was more involved in implementing the fractional tessellation system, and the test scenes such as the subdivision surface scenes. Both were involved in writing the paper, benchmarking, and conceptual algorithm design.

6 Conclusions and Future Work

The algorithms presented in this thesis are mostly targeting improved rendering performance, and in that category there is always more work to be done. In the rasterization field, it would be interesting to see how our conservative rasterization approach benefits from geometry shaders. Although there is already an implementation [42] based on geometry shaders, there are still no figures on how this affects performance. We would also like to continue to improve on the multi-view rasterization work. It would be interesting to see if our multi-view rasterization approach could be adapted to work in a tiling architecture (often referred to as "binning") used in some graphics hardware such as the upcoming Larrabee [40], Mali 200 [4] and KYRO II [3]. Also, a limitation of the algorithm is that it only takes horizontal view parallax into account. It would be interesting to investigate if one can create a similar rasterization scheme that can handle full view parallax. This may sound like a simple extension, but it is in practice very hard to go from our simple one-dimensional sorting scheme to full two-dimensional sorting.

Although texture and buffer compression are fairly mature fields, the compression algorithms can always be improved upon. Upcoming hardware will always change the prerequisites for a compression algorithm, and in a near future we can implement our own compression algorithms as software programs. The general trend, however, seems to be that more and more complex compression algorithms can be allowed since computational performance generally grows faster than memory bandwidth. The lossy buffer compression is a track that I think is worth looking further into. In particular, it would be interesting to see if one can use a better error control mechanism than we suggested in Paper **VI**, since this mechanism cannot give any guarantees as to what the maximum error is. It is also interesting to see how HDR compression schemes can be introduced to buffer compression with or without lossy compression, and how we can unify the treatment of textures and buffers in a graphics hardware system. Although these topics have been covered

by Ström et al. [44] there is much work that remains to be done.

Finally, for our programmable culling approaches, there is still much work to be done. As we have no access to the internals of high-end graphics hardware pipelines, we have used a simulation framework written in software, and therefore there may be several questions or issues raised if the algorithms were to be implemented in an extremely parallel hardware system. Furthermore, when we wrote the paper on vertex culling, we noted that interval arithmetic was not accurate enough to represent bounds of displaced surfaces. This forced us to learn more about bounded arithmetic, such as Taylor models, and we realized that it might not be best to use a fixed choice of interval arithmetic for pixel culling and Taylor models for vertex culling. Instead, it would be very interesting to design a "smart" program analyzer that could use appropriate bounded arithmetic to ensure fairly tight bounds while still generating cull shader programs with good performance. Also, in our paper on pixel culling (Paper **VII**), we assumed that the actual hardware was designed to execute interval instructions. It might be more realistic to assume that the hardware operates on scalar instructions, and that the cull shader generator creates the code to implement interval arithmetic. In this case, there may be several interval arithmetic specific optimizations that can be made by analyzing the original pixel shader program.

Hence, it is quite obvious that there still remains a lot of interesting graphics research to be done in the future.

Bibliography

[1] S. J. Adelson and C. D. Hansen. Fast Stereoscopic Images with Ray-Traced Volume Rendering. In *Symposium on Volume Visualization*, pages 3–9, 1994.

[2] Tomas Akenine-Möller and Timo Aila. Conservative Tiled Rasterization Using a Modififed Triangle Setup. *Journal of Graphics Tools*, 10(2):1–8, 2005.

[3] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering 2nd Edition*. A. K. Peters Ltd., June 2002.

[4] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters Ltd., 2008.

[5] Jukka Arvo. Alias-Free Shadow Maps using Graphics Hardware. *Journal of Graphics Tools*, 12(1):47–59, 2007.

[6] Ulf Assarsson and Tomas Möller. Optimized View Frustum Algorithms for Bounding Boxes. *Journal of Graphics Tools*, 5(1):9–22, 2000.

[7] Martin Berz and Georg Hoffstätter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds. *Reliable Computing,*, 4(1):83–97, 1998.

[8] Robert L. Cook. Shade trees. *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 18(3):223–231, 1984.

[9] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, 21(4):95–102, 1987.

[10] NVIDIA Corp. *HRAA: High-Resolution Antialiasing Through Multisampling*. Technical Brief, 2001.

[11] Paul E. Debevec and Jitendra Malik. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH*, pages 369–378, 1997.

[12] Neil A. Dodgson. Autostereoscopic 3D Displays. *IEEE Computer*, 38(8):31–36, 2005.

[13] Michael Doggett. Xenos: XBOX 360 GPU. Eurographics presentation, September 2005.

[14] Kevin Gee. Introduction to the Direct3D 11 Graphics Pipeline. nVision conference course, 2008.

[15] Naga K. Govindaraju, Redon Stephane, Ming C. Lin, and Dinesh Manocha. CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware. In *Graphics Hardware*, pages 25–32, 2003.

[16] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer Visibility. In *Proceedings of ACM SIGGRAPH*, pages 231–238, 1993.

[17] Michael Halle. Multiple Viewpoint Rendering. In *Proceedings of ACM SIGGRAPH*, pages 243–254, 1998.

[18] Tetsugo Inada and Michael D. McCool. Compressed lossless texture representation and caching. In *Graphics Hardware*, pages 111–120, 2006.

[19] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999.

[20] Bahram Javidi and Fumio Okano. *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

[21] Alexander Keller. Strictly Deterministic Sampling Methods in Computer Graphics. Technical report, Mental Images, 2001.

[22] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for superior texture performance. *Computers & Graphics*, 20(4):475–481, 1996.

[23] Samuli Laine and Timo Aila. A Weighted Error Metric and Optimization Method for Antialiasing Patterns. *Computer Graphics Forum*, 25(1):83–94, 2006.

[24] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A User-Programmable Vertex Engine. In *Proceedings of ACM SIGGRAPH*, pages 149–158, 2001.

[25] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. CC Shadow Volumes. In *Eurographics Symposium on Rendering*, pages 197–205, 2004.

[26] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[27] Joel McCormack and Robert McNamara. Tiled Polygon Traversal Using Half-plane Edge Functions. In *Graphics Hardware*, pages 15–21, 2000.

[28] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[29] Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. Practical HDR Texture Compression. *Computer Graphics Forum*, 27(6):1664–1676, 2008.

[30] Karol Myzskowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast Collision Detection Between Complex Solids Using Rasterizing Graphics Hardware. *The Visual Computer*, 11(9):497–512, 1995.

[31] Avi C. Naiman. Jagged Edges: When is Filtering Needed? *ACM Transactions on Graphics*, 17(4):238–258, 1998.

[32] W. Newman and R. Sproull. *Principles of Interactive Computer Graphics*. New York: McGraw-Hill, 2nd edition, 1979.

[33] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Graphics Hardware*, pages 89–96, 1997.

[34] John Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470, 2005.

[35] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of ACM SIGGRAPH*, pages 17–20, 1988.

[36] Dennis R. Proffitt and Mary Kaiser. Hi-Lo Stereo Fusion. In *ACM SIGGRAPH 96 Visual Proceedings*, page 146, 1996.

[37] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 1st edition, 2005.

[38] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. High Dynamic Range Texture Compression. *ACM Transactions on Graphics*, 25(3):707–712, 2006.

[39] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. Efficient High Dynamic Range Texture Compression. In *Symposium on Interactive 3D graphics and games*, pages 207–214, 2008.

[40] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. *ACM Transactions of Graphics*, 27(3):18, 2008.

[41] Peter S. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois, 1991.

[42] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample Based Visibility for Soft Shadows using Alias-free Shadow Maps. *Computer Graphics Forum (Proceedings of EGSR)*, 27(4):1285–1292, 2008.

[43] Stanislav L. Stoev, Tobias Hüttner, and Wolfgang Strasser. Accelerated Rendering in Stereo-Based Projections. In *Third International Conference on Collaborative Virtual Environments*, pages 213–214, 2000.

[44] Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren, Jacob Munkberg, Petrik Clarberg, and Tomas Akenine-Möller. Floating-point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware*, pages 75–84, 2008.

[45] Evan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[46] Natalya Tatarchuk, Christopher Oat, Jason L. Mitchell, Chris Green, Johan Andersson, Martin Mittring, Shanon Drone, and Nico Galoppo. Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH course, 2007.

[47] Lvdi Wang, Xi Wang, Peter-Pike Sloan, Li-Yi Wei, Xin Tong, and Baining Guo. Rendering from Compressed High Dynamic Range Textures on Programmable Graphics Hardware. In *Symposium on Interactive 3D Graphics and Games*, pages 17–24, 2007.

[48] Marcelo J. Weinberger, Gadiel Seroussi, and Guiellermo Sapiro. LOCO-I: A low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conference*, pages 140–149, 1996.

[49] Sun Wen, Lu Yan, Wu Feng, and Li Shipeng. DHTC: An Effective DXTC-based HDR Texture Compression Scheme. In *Graphics hardware*, pages 85–94, 2008.

Paper I

A Family of Inexpensive Sampling Schemes

Jon Hasselgren[‡] Tomas Akenine-Möller[‡] Samuli Laine[*†]

[‡]Lund University,
[*]Helsinki University of Technology/TML,
[†]Hybrid Graphics, Ltd.

ABSTRACT

To improve image quality in computer graphics, antialiasing techniques such as supersampling and multisampling are used. We explore a family of inexpensive sampling schemes that cost as little as 1.25 samples per pixel and up to 2.0 samples per pixel. By placing sample points in the corners or on the edges of the pixels, sharing can occur between pixels, and this makes it possible to create inexpensive sampling schemes. Using an evaluation and optimization framework, we present optimized sampling patterns costing 1.25, 1.5, 1.75, and 2.0 samples per pixel.

Computer Graphics forum 24(4):843–848, 2005.

1 Introduction

For computer generated imagery, it is most often desirable to use antialiasing algorithms to improve the image quality. Aliasing effects occur due to undersampling of a signal, where a high frequency signal appears in disguise as a lower frequency signal. Antialiasing algorithms in screen space reduce these image artifacts by raising the sampling rate and computing the color of a pixel as a weighted sum of a number of associated sample points' colors. Such algorithms are often divided into two categories: *supersampling* and *multisampling*.

Supersampling includes all algorithms where the scene is sampled in more than one point per pixel and the final image is computed from the samples. In multisampling techniques, the scene is also sampled in more than one point per pixel, but the results of fragment shader computations (e.g. texture color) is shared between the samples in a pixel.

When using multisampling or supersampling, the positions and weights of the different sample points play a major role in the final image quality. Work has already been done to find the best sampling schemes for certain numbers of samples per pixel but so far no one has studied those that only cost between 1.25 and 2 samples per pixel. This family of sampling schemes is particularly interesting when designing hardware with strict performance and memory limitations, such as graphics hardware for mobile platforms [3].

The second author of this paper has written a technical report [1], where he presented a sampling scheme, called *FLIPTRI*, that costs only 1.25 samples per pixel on average. The purpose of this paper is to present that scheme to a broader audience, as well as taking the idea one step further and explore all sampling schemes that cost between 1.25 and 2.0 samples per pixel. We use the optimization and evaluation framework by Laine and Aila [6] to evaluate the quality of the sampling patterns and to compute optimized sample coordinates and weights.

2 Previous Work

The first attempts to produce inexpensive antialiasing in graphics hardware were simple solutions that performed poorly in many cases. This includes, for example, the Kyro hardware described by Akeine-Möller and Haines [2] that uses completely *horizontal* or *vertical* sampling schemes (Figures 1b and 1c) to perform supersampling with two samples per pixel. Placing the samples diagonally instead (Figure 1d) gives a slight visual improvement. The 2×2 *box pattern* (Figure 1e) was also implemented on Kyro, as well as on GeForce 3, 4 and FX [5] and probably on most of the other consumer level hardware. The popularity of 2×2 box pattern is obviously due to its simple implementation. However, it should be noted that the *Rotated Grid Supersampling* (RGSS) pattern (Figure 1f) delivers much better quality with equal cost.

RGSS is a scheme of the more general *N-rooks* family presented by Shirley [9].

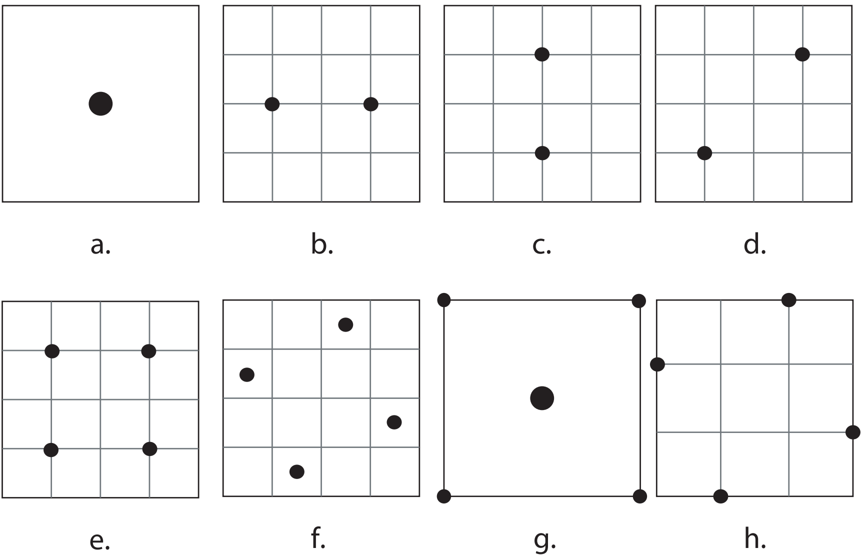


Figure 1: A collection of sampling schemes. From top left to bottom right: a) Centroid sampling, b) Kyro horizontal, c) Kyro vertical, d) Diagonal, e) 2×2 box pattern, f) Rotated Grid Supersampling (RGSS), g) Quincunx, h) FLIPQUAD.

Assuming that the pixel is divided into an $n \times n$ uniform grid, an N-rooks pattern satisfies the criterion that no two sample points are placed on same row or column. This is analogous to placing n rooks on an $n \times n$ chessboard without letting any two rooks capture each other. Sampling schemes fulfilling the N-rooks criterion perform well for near-horizontal and near-vertical edges.

An inexpensive multisampling scheme exploiting sample sharing between adjacent pixels is the *Quincunx* scheme [4] used in NVIDIA graphics hardware (Figure 1g). It resembles the five on a six-sided die and is horizontally and vertically symmetric. Therefore, it can be repeated for every pixel and the sampled colors from the sample points in the corners can be shared by four pixels resulting in a total cost of two samples per pixel. A weakness of the Quincunx pattern is that it does not fulfill the N-rooks criterion. The weights are 0.125 for the corner samples, and 0.5 for the center sample.

The *FLIPQUAD* [3] scheme, shown in Figure 1h, is an example of a multisampling scheme based on the N-rooks criterion. All of the sample points can be shared with neighboring pixels when the pattern is horizontally and vertically reflected for different pixels as shown in Figure 2a. Therefore the cost is only two samples per pixel.

Naiman [8] has presented a study where several test subjects were instructed to identify the more jagged edge from a set with two edges rendered at different resolutions. The result of this study can be interpreted as an estimate of the im-

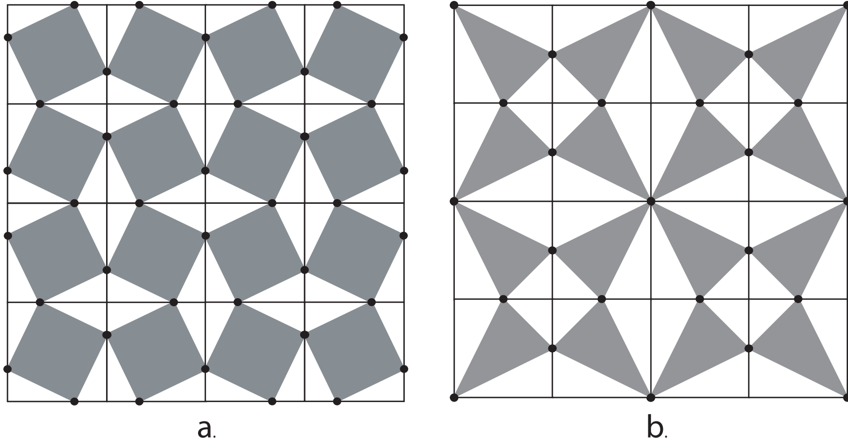


Figure 2: The a) FLIPQUAD and b) FLIPTRI sampling patterns repeated and reflected over 4×4 pixels. For illustration purposes, shaded regions are added to emphasize the sample points that are used for reconstructing the pixels.

portance of antialiasing for lines with different slopes. It suggests that people are most sensitive to lines nearly horizontal or vertical and to lines with a slope near 45°. To that end, we suggest that one uses the *N-queens* sample point positioning strategy, where the sample point positions fulfill the N-rooks criteria with an additional requirement that no two sample points are allowed on the same diagonal. Both RGSS and FLIPQUAD fulfill this condition, which indicates that they should perform well on edges near 45° as well.

Laine and Aila [6] define an error metric, E_2, for evaluating and optimizing sampling patterns. The metric takes into account the slope-specific acuity factors in the study by Naiman [8] and uses a high-quality reconstruction filter [7] for computing the reference value for the final color of each pixel. The error is evaluated by sweeping a set of lines with different slopes over the sampling pattern and comparing the values given by a sampling pattern to the exact reference value. To correct for human perceptual ability, the slope-specific supersampling errors are weighed based on the acuity measurements in Naiman's study. After this, the maximum error among the different slopes is chosen. This error metric allows us to perform various computer driven searches for best patterns fulfilling a set of restrictions.

3 Notation

We use the same $P(s, r, n)$-notation as Laine and Aila [6] to describe a specific class of sampling patterns. The s parameter represents the number of pixel-sized sample point sets that form the periodically repeating pattern, r is the number of

Sample point position	Cost
Corner	0.25
Edge	0.5
Pixel	1

Table 1: Cost of differently placed sample points in the reflected $P(4, 1^+, n)$ class

pixels used by the reconstruction filter, and n is the average number of samples taken for each pixel. As a special case, notation r^+ means that the reconstruction filter may also use samples from sample points located on the boundary of the reconstruction region. For instance, if r is 1^+, the reconstruction filter uses samples from sample points in and on the border of a single pixel. In this study we are only concerned with the $P(4, 1^+, n)$ family of sampling patterns where $n \in \{1.25, 1.5, 1.75, 2.0\}$. Note that the FLIPQUAD scheme belongs to this class.

4 The FLIPTRI Sampling Scheme

In this section, we present the FLIPTRI sampling scheme, which previously only has been described in a technical report [1]. The FLIPQUAD scheme assumed that the sampling pattern for a pixel is reflected through the pixel edges in order to ensure that sample sharing between pixels can occur. We continue to use that approach, and note that if a sample point is placed in the corner of a pixel, then the cost of that sample point is 0.25 samples per pixel, since the corner is shared by four pixels. Sample points on pixel edges are, in general, shared by two pixels, and so the cost is 0.5 samples per pixel. Finally, a sample point placed inside the pixel costs one sample per pixel. Table 1 summarizes these costs. At this point, it is simple to verify the costs for a given scheme. For FLIPQUAD, the cost is $4 \times 0.5 = 2$, and for Quincunx it is $1 + 4 \times 0.25 = 2$ samples per pixel.

To the best of our knowledge, all previously presented supersampling schemes cost at least two samples per pixel. In that sense, the FLIPTRI scheme is quite radical since it costs less than that. This is achieved by placing one sample point at a corner, and two sample points on different edges, as shown in Figure 2b, giving the scheme a total cost of $0.25 + 2 \times 0.5 = 1.25$ samples per pixel. As can be seen, by placing the edge sample points on the edges that do not share the corner sample point, the N-rooks property is fulfilled. Due to Naiman's study [8], one can expect that a similar and slightly better scheme can be obtained by offsetting the sample points positioned on the edges. This way, the error can be moved to angles for which the eye is less sensitive.

5 Sampling patterns

Here, we further explore the *design space* of inexpensive sampling patterns in order to verify the efficiency of the FLIPTRI and FLIPQUAD patterns, but also to produce and evaluate new sampling schemes that cost 1.5 and 1.75 samples per pixel.

5.1 Initial Pattern Generation

Given the costs for different sample point placements (corner, edge or center) in Table 1, it is quite straightforward to write a computer program that generates all possible unique configurations of placements for $P(4, 1^+, n)$ sampling patterns. At this point, we are not interested in the actual coordinates and weights of the sample points but only in deciding if the sample point is placed in a specific corner, on a specific edge or somewhere inside the pixel.

The set of unique configurations of sample point placements is small when $n \leq 2$. Therefore we could use a brute-force algorithm that generates all possible placement configurations with the specified cost and discards a configuration if it is a rotated (90°, 180° or 270°) and/or reflected version of an already generated candidate.

5.2 Pattern Ranking and Optimization

The number of placement configurations with unique properties, in our target families, are only 94 in total and were therefore quite easily manageable. Once generated, we manually examined each configuration and removed those that would clearly not perform well. For example, patterns with all sample points placed along a single edge could be safely removed from further consideration.

The patterns passing this preliminary culling phase were processed by the error metric-based optimizer by Laine and Aila [6] to find an optimal set of weights and coordinates for the sample points and also an estimation of the E_2 error for a given pattern. The error was used to rank the patterns in each category.

6 Results

We present a summary of the most interesting sampling schemes, that we found, in Figure 3, and more exact pattern descriptions in Appendix A. In the cases where two patterns in the same class had similar errors, but different reconstruction costs (number of samples used to compute the final color), both alternatives are presented.

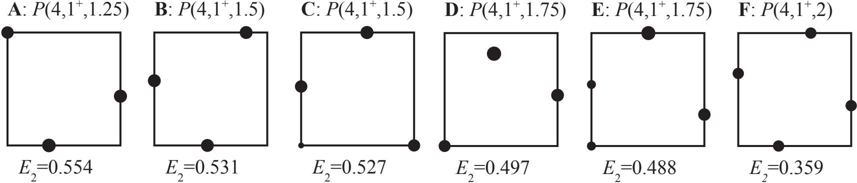


Figure 3: The figure shows the best schemes in terms of the error metric E_2 in each class. The area of each circle is proportional to the weight of the corresponding sample point. For $n = 1.5$ and $n = 1.75$, two patterns were found with similar error measures, but with different number of samples used during reconstruction. Schemes A and F are versions of the FLIPTRI and FLIPQUAD schemes respectively, that have been optimized using the E_2 error metric (see section 5.2)

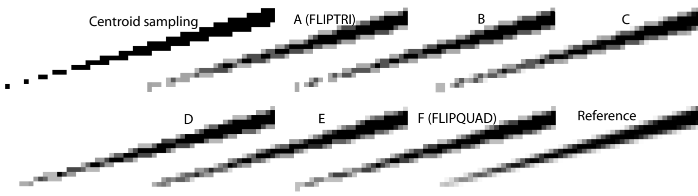


Figure 4: A thin polygon rendered using the sampling schemes, A through F, from Figure 3. The upper left polygon uses a single sample point per pixel while the reference polygon is supersampled with 1024 jittered grid sample points and uses a 4×4 pixel Mitchell-Netravali reconstruction filter.

6.1 Evaluation

We have evaluated our results both visually and experimentally. For the visual evaluation we simply implemented supersampling using the sampling patterns in Figure 3 and rasterized a number of test images. Figure 4 shows the results of drawing a thin polygon with the respective schemes as well as using centroid sampling and a high quality supersampling pattern for reference. Figure 5 shows a more complex test image.

We have also performed experimental evaluation by rasterizing two synthetic animations. One showing a rotating triangle and the other showing a translating circle. The animations were rasterized using each of our schemes and we compute the per-pixel deviation, as compared to a reference animation. The reference filter was identical to the references used in Figure 4 and 5 but with 256 sample points for performance reasons. Since the evaluation result of the two animations were almost identical, we only present figures for the translation animation in this paper.

It should be noted that this measurement does not take any psychovisual aspects into account, but can still be used as a rough measurement of quality. We can use

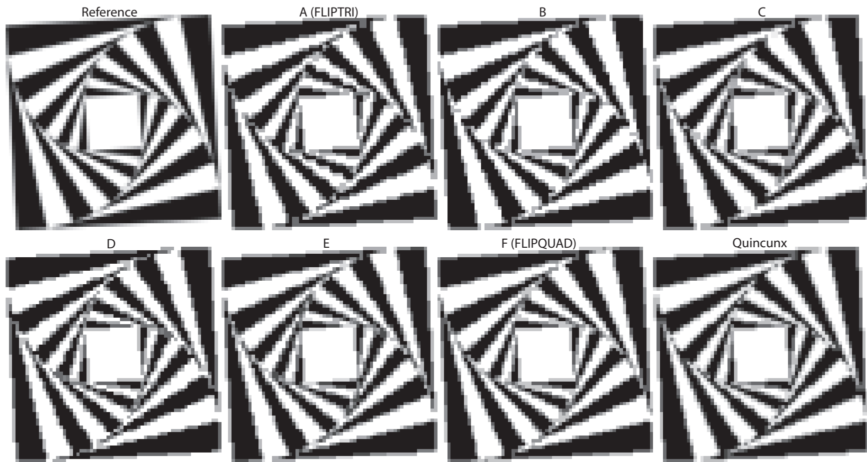


Figure 5: A spiral test image rendered using the sampling schemes, A through F, from Figure 3. The reference image is supersampled with 1024 jittered grid sample points and uses a 4×4 pixel Michell-Netravali reconstruction filter. An image rendered using the Quincunx pattern is also included for comparison.

the deviations to compute the Root Mean Square Error (RMSE) as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{255} C(i)i^2} \tag{1}$$

Where $C(i)$ is the number of pixels with a given deviation, i, from the reference animation. The normalization constant n is the total number of pixels in the animation. In Figure 6 and 7, we present plots of the $C(i)i^2/n$ values of different patterns. The plots can be seen as the scheme's penalized histogram of the pixel deviations, and the area below each curve is approximately equal to the Mean Square Error (MSE).

For Figure 6, we settle for comparing our own sampling schemes, since there are no other sampling schemes with similar costs. It should be noted that FLIPTRI (scheme A) actually has a worst case deviation higher than that of the centroid sampling scheme. This is because of the placement of sample points in the scheme. Looking at the very center of Figure 2b, we see that the sample point in the corner will be further away from its nearest neighbors than in the case of centroid sampling. It is this distance that causes the greater maximum deviations.

In Figure 7, we compare FLIPQUAD (scheme F) with other sample patterns costing 2 samples per pixel. The behavior of FLIPQUAD is clearly preferable to both Quincunx and normal super sampling using two diagonal samples. The Quincunx plot has a behavior that is very similar to the centroid sampling curve, but the four extra "low-pass" sampling points removes a substantial part of the larger errors.

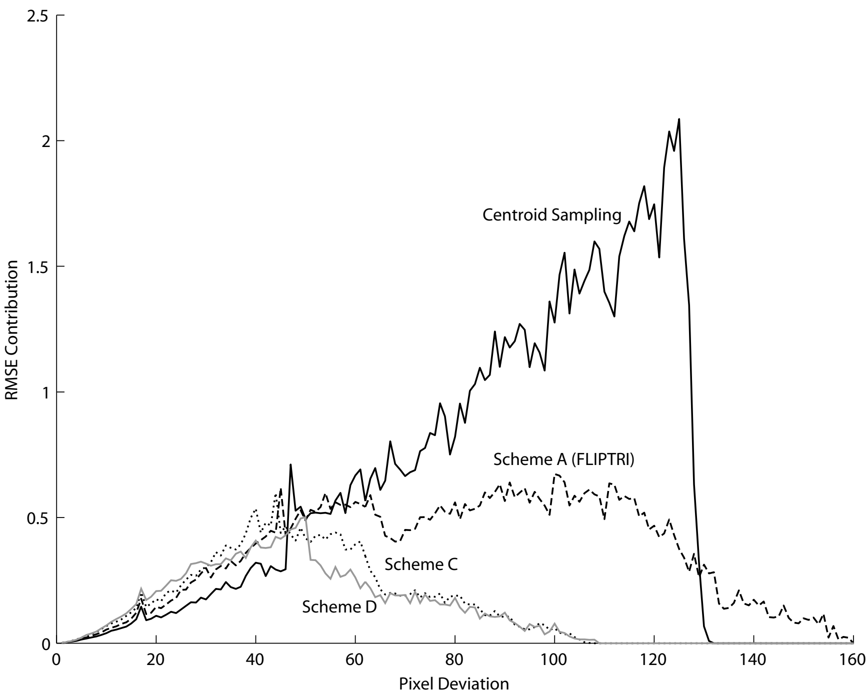


Figure 6: Histogram of the squared pixel error. This figure shows a comparison between the 1.25, 1.5 and 1.75 sample shemes. Sampling with a single centroid sample is also included for reference. The RMSE contribution is computed as $C(i)i^2/n$

Finally, we present a summary of the errors in Table 2. We see that even though the E_2 metric does not match the RMSE metric, we still get similar performance with the exception of FLIPTRI and scheme C. We have already motivated the behavior of FLIPTRI. Scheme C is given an unusually low RMSE when we compare it to schemes B and D, but this is not very surprising if we look at the design of the sample schemes. Scheme C has an additional sample point which lowers the RMSE but does not significantly affect the perceptually important nearly horizontal and nearly vertical edges.

7 Conclusion and Future Work

In this paper, we have generated a family of inexpensive sampling schemes. It is difficult to compare and evaluate sampling schemes, so we present visual results along with comparisons using the RMSE metric.

Our evaluation shows that the modified FLIPQUAD pattern clearly stands out in

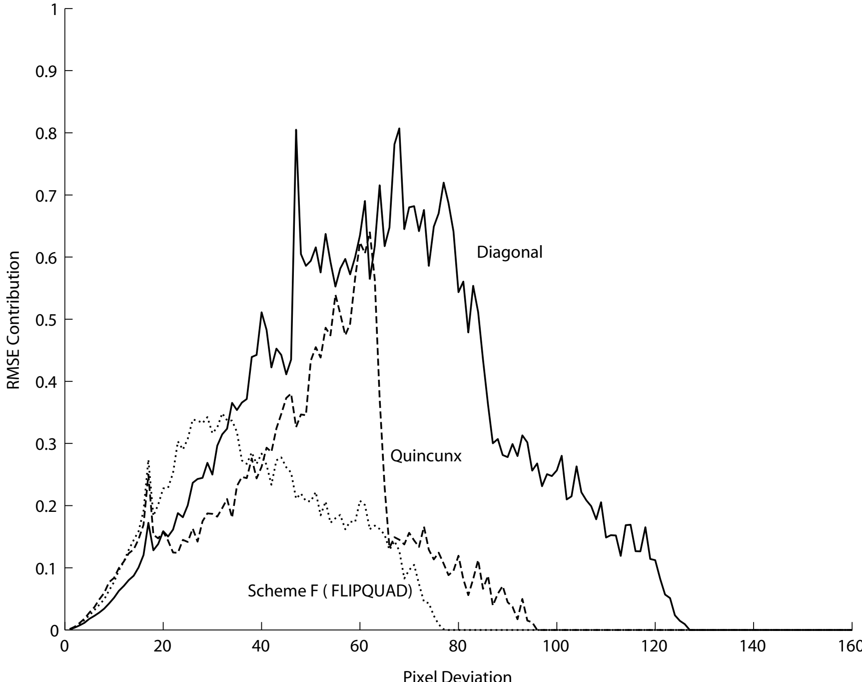


Figure 7: Histogram of the squared pixel error. This figure shows how scheme f (FLIPQUAD) compares to Quincunx and the Diagonal sample scheme from Figure 1d.

terms of quality. It is also along with FLIPTRI the most cost efficient filter, in terms of *quality*/*cost*, as far as the E_2 metric is concerned. However, in the case of FLIPTRI, we have to sacrifice the performance of certain edges yielding relatively bad worst case performance. Our other schemes, costing 1.5 and 1.75, samples per pixel also have very low cost and eliminates any chance of a worse performance than normal centroid sampling. In fact they are very close to Quincunx in terms of performance, with respect to both the E_2 and RMSE metrics.

It should also be noted that extra degrees of freedom, in the sample patterns, help in cases not taken into account by the E_2 model. An example of such a case is shown in Figure 4 where the tip of the triangle is thinner than one pixel and therefore do not fall within the assumptions of the E_2 metric. The tip is broken into a few disjoint pieces for the more inexpensive schemes, A-C, while the more expensive schemes, D-F, gives a better visual result.

An open issue with the suggested sampling schemes is where to locate the shader sample point to allow sharing data between sample points in order to avoid texture blurring. For FLIPQUAD this problem has already been addressed [3], but the placement for the other schemes, such as FLIPTRI, are less obvious. It would also

	E_2	RMSE	max dev.
Centroid sampling	1.256	9.810	131
Diagonal	0.692	6.500	126
Scheme A (FLIPTRI)	0.554	7.548	159
Scheme B	0.531	5.139	114
Scheme C	0.527	4.830	107
Scheme D	0.497	5.000	107
Scheme E	0.488	4.574	108
Quincunx	0.484	4.400	95
Scheme F (FLIPQUAD)	0.359	3.759	76

Table 2: The errors of our sampling schemes, using different metrics. The "Diagonal" scheme is the diagonal super sampling scheme from Figure 1d.

be interesting to explore the $P(4,4^+,n)$ family for small values of n since they potentially provide higher quality.

A Sample positions and weights

The following table lists the sample point coordinates and weights for the sampling patterns in Figure 3. Each row contains the description of one sample point: $x, y, weight$. The origin $(x = 0, y = 0)$ is located at the center of the pixel. The width and height of a pixel are both 1.0, i.e., the borders are at $x = \pm 0.5$ and $y = \pm 0.5$.

A: $P(4, 1^+, 1.25)$	**B**: $P(4, 1^+, 1.5)$
-0.500, 0.500, 0.299	-0.500, 0.073, 0.335
-0.133, -0.500, 0.360	-0.030, -0.500, 0.331
0.500, -0.064, 0.341	0.313, 0.500, 0.334

C: $P(4, 1^+, 1.5)$	**D**: $P(4, 1^+, 1.75)$
-0.500, 0.022, 0.306	-0.500, -0.500, 0.280
-0.500, -0.500, 0.068	-0.063, 0.318, 0.397
0.083, 0.500, 0.338	0.500, -0.050, 0.323
0.500, -0.500, 0.288	

E: $P(4, 1^+, 1.75)$	**F**: $P(4, 1^+, 2)$
-0.500, -0.500, 0.158	-0.500, 0.143, 0.250
-0.500, 0.045, 0.156	0.500, -0.143, 0.250
0.004, 0.500, 0.380	0.143, 0.500, 0.250
0.500, -0.222, 0.306	-0.143, -0.500, 0.250

Bibliography

[1] Tomas Akenine-Möller. *An Extremely Inexpensive Multisampling Scheme*. Technical Report 03–14, Chalmers University of Technology, 2003.

[2] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A.K. Peters, 2 edition, 2002.

[3] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics*, 22(3):801–808, 2003.

[4] NVIDIA Corp. *HRAA: High-Resolution Antialiasing Through Multisampling*. Technical Brief, 2001.

[5] NVIDIA Corp. *The GeForce 6 Series of GPUs: High Performance and Quality for Complex Image Effects*. Technical Brief, 2004.

[6] Samuli Laine and Timo Aila. A weighted error metric and optimization method for antialiasing patterns. *Computer Graphics Forum*, 25(1):83–94, 2006.

[7] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computergraphics. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 221–228, 1988.

[8] Avi C. Naiman. Jagged edges: when is filtering needed? *ACM Trans. Graph.*, 17(4):238–258, 1998.

[9] Peter S. Shirley. *Physically based lighting calculations for computer graphics*. PhD thesis, 1991.

Paper II

Conservative Rasterization

Jon Hasselgren Tomas Akenine-Möller Lennart Ohlsson

Lund University

{jon|tam|lennart.ohlsson}@cs.lth.se

GPU Gems 2, pages 677–690. Addison-Wesley Professional, 2005.

1 Introduction

Over the past few years, general-purpose computation using GPUs has received much attention in the research community. The stream-based rasterization architecture provides for much faster performance growth than that of CPUs, and therein lies the attraction in implementing an algorithm on the GPU: if not now, then at some point in time, your algorithm is likely to run faster on the GPU than on a CPU.

However, some algorithms do not return exact results when the GPU's standard rasterization is used. Examples include algorithms for collision detection [2, 5], occlusion culling [3], and visibility testing for shadow acceleration [4]. The accuracy of these algorithms can be improved by increasing rendering resolution. However, one can never guarantee a fully correct result. This is similar to the antialiasing problem: you can never avoid sampling artifacts by just increasing the number of samples; you can only reduce the problems at the cost of performance.

A simple example of when *standard rasterization* does not compute the desired result is shown in Figure 1a, where one green and one blue triangle have been rasterized. These triangles overlap geometrically, but the standard rasterization process does not detect this fact. With *conservative rasterization*, the overlap is always properly detected, no matter what resolution is used. This property can enable load balancing between the CPU and the GPU. For example, for collision detection, a lower resolution would resultin less work for the GPU and more work for the CPU, which performs exact intersection tests. See Figure 1b for the results of using conservative rasterization.

There already exists a simple algorithm for conservative rasterization [1]. However, this algorithm is designed for hardware implementation. In this chapter we present an alternative that is adapted for implementation using vertex and fragment programs on the GPU.

2 Problem Definition

In this section, we define what we mean by conservative rasterization of a polygon. First, we define a *pixel cell* as the rectangular region around a pixel in a pixel grid. There are two variants of conservative rasterization, namely, *overestimated* and *underestimated* [1]:

- An overestimated conservative rasterization of a polygon includes all pixels for which the intersection between the pixel cell and the polygon is nonempty.

- An underestimated conservative rasterization of a polygon includes only the pixels whose pixel cell lies completely inside the polygon.

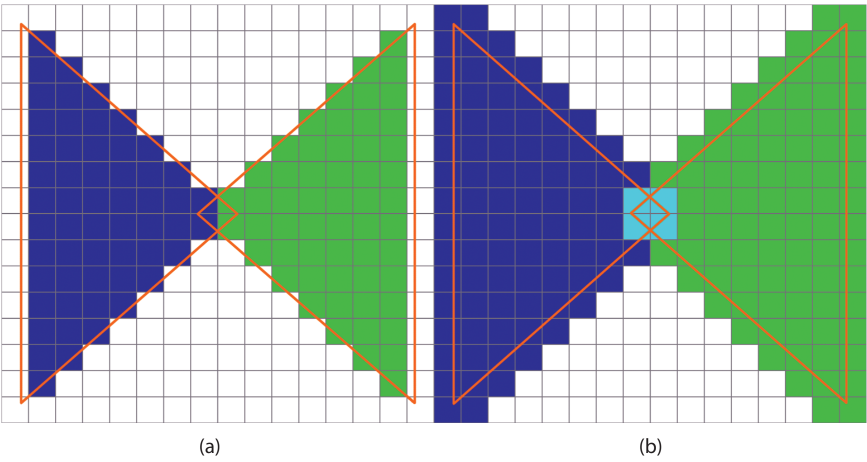


Figure 1: Comparing Standard and Conservative Rasterization: A blue and a green triangle rasterized at 16×16 pixels resolution using additive blending. (a) Standard rasterization. (b) Overestimating conservative rasterization. Note that the small overlap of the triangles is missed with standard rasterization and correctly detected using conservative rasterization.

Here, we are mainly concerned with the overestimated variant, because that one is usually the most useful. If not specified further, we mean the overestimated variant when writing "onservative rasterization."

As can be seen, the definitions are based only on the pixel cell and are therefore independent of the number of sample points for a pixel. To that end, we choose not to consider render targets that use multisampling, because this would be just a waste of resources. It would also further complicate our implementation.

The solution to both of these problems can be seen as a modification of the polygon before the rasterization process. Overestimated conservative rasterization can be seen as the image-processing operation *dilation* of the polygon by the pixel cell. Similarly, underestimated conservative rasterization is the *erosion* of the polygon by the pixel cell. Therefore, we transform the rectangle-polygon overlap test of conservative rasterization into a point-in-region test.

The dilation is obtained by locking the center of a pixel-cell-size rectangle to the polygon edges and sweeping it along them while adding all the pixel cells it touches to the dilated triangle. Alternatively, for a convex polygon, the dilation can be computed as the convex hull of a set of pixel cells positioned at each of the vertices of the polygon. This is the equivalent of moving the vertices of the polygon in each of the four possible directions from the center of a pixel cell to its corners and then computing the convex hull. Because the four vectors from the center of a pixel cell to each corner are important in any algorithm for conserva-

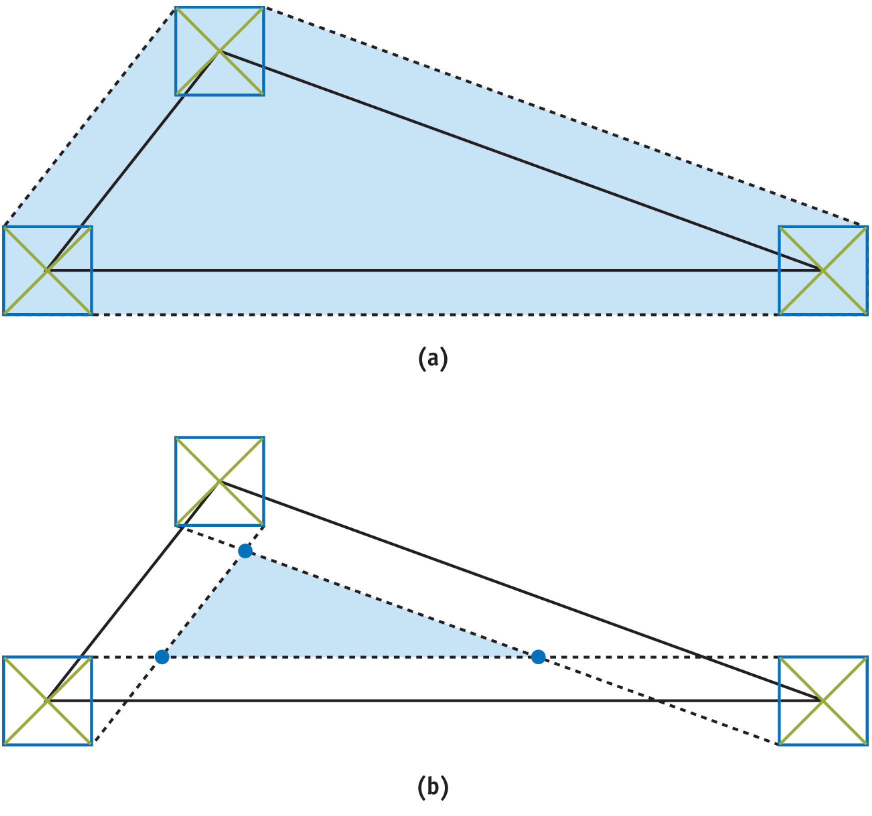


Figure 2: Overestimated and Underestimated Rasterization: (a) Overestimated rasterization. (b) Underestimated rasterization. Solid lines show the input triangle. The dashed lines, dots, and shaded area indicate the bounding polygon's edges, vertices, and area.

tive rasterization, we call them the *semidiagonals*. This type of vertex movement is shown by the red lines in Figure 2a, which also shows the *bounding polygon* for a triangle.

The erosion is obtained by sweeping the pixel-cell-size rectangle in the same manner as for dilation, but instead we erase all the pixel cells it touches. We note that for a general convex polygon, the number of vertices may be decreased due to this operation. In the case of a triangle, the result will always be a smaller triangle or an "empty" polygon. The erosion of a triangle is illustrated in Figure 2b.

3 Two Conservative Algorithms

We present two algorithms for conservative rasterization that have different performance characteristics. The first algorithm computes the optimal bounding polygon in a vertex program. It is therefore optimal in terms of fill rate, but it is also costly in terms of geometry processing and data setup, because every vertex must be replicated. The second algorithm computes a bounding triangle – a bounding polygon explicitly limited to only three vertices – in a vertex program and then trims it in a fragment program. This makes it less expensive in terms of geometry processing, because constructing the bounding triangle can be seen as repositioning each of the vertices of the input triangle. However, the bounding triangle is a poor fit for triangles with acute angles; therefore, the algorithm is more costly in terms of fill rate.

To simplify the implementation of both algorithms, we assume that no edges resulting from clipping by the near or far clip planes lie inside the view frustum. Edges resulting from such clipping operations are troublesome to detect, and they are very rarely used for any important purpose in this context.

3.1 Clip Space

We describe both algorithms in *window space*, for clarity, but in practice it is impossible to work in window space, because the vertex program is executed before the clipping and perspective projection. Fortunately, our reasoning maps very simply to *clip space*. For the moment, let us ignore the z component of the vertices (which is used only to interpolate a depth-buffer value). Doing so allows us to describe a line through each edge of the input triangle as a plane in homogeneous (x_c, y_c, w_c)-space. The plane is defined by the two vertices on the edge of the input triangle, as well as the position of the viewer, which is the origin, $(0,0,0)$. Because all of the planes pass through the origin, we get plane equations of the form

$$a \cdot x_c + b \cdot y_c + c \cdot w_c = 0 \Leftrightarrow a(x \cdot w_c) + b(y \cdot w_c) + c \cdot w_c = 0 \Rightarrow a \cdot x + b \cdot y + c = 0.$$

The planes are equivalent to lines in two dimensions. In many of our computations, we use the normal of an edge, which is defined by (a,b) from the plane equation.

3.2 The First Algorithm

In this algorithm we compute the optimal bounding polygon for conservative rasterization, shown in Figure 2a. Computing the convex hull, from the problem definition section, sounds like a complex task to perform in a vertex program. However, it comes down to three simple cases, as illustrated in Figure 3. Given two edges e_1 and e_2 connected in a vertex v, the three cases are the following:

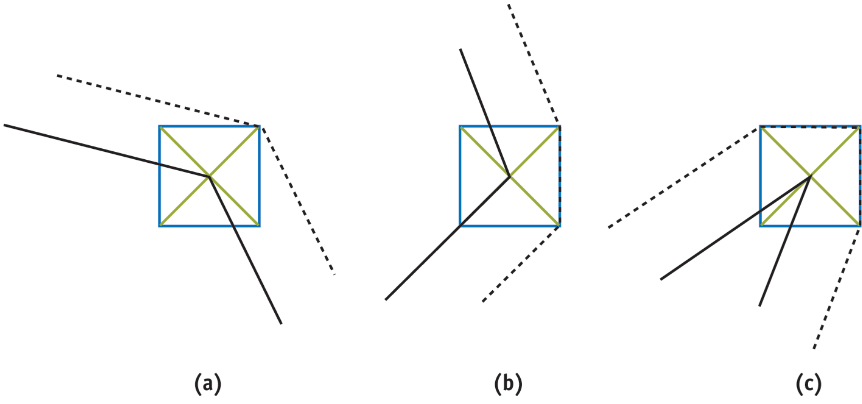


Figure 3: Computing an Optimal Bounding Polygon

- If the normals of e_1 and e_2 lie in the same quadrant, the convex hull is defined by the point found by moving the vertex v by the semidiagonal in that quadrant (Figure 3a).

- If the normals of e_1 and e_2 lie in neighboring quadrants, the convex hull is defined by two points. The points are found by moving v by the semidiagonals in those quadrants (Figure 3b).

- If the normals of e_1 and e_2 lie in opposite quadrants, the convex hull is defined by three points. Two points are found as in the previous case, and the last point is found by moving v by the semidiagonal of the quadrant between the opposite quadrants (in the winding order)(Figure 3c).

Implementation

We implement the algorithm as a vertex program that creates output vertices according to the three cases. Because we cannot create vertices in the vertex program, we must assume the worst-case scenario from Figure 3c. To create a polygon from a triangle, we must send three instances of each vertex of the input triangle to the hardware and then draw the bounding polygon as a triangle fan with a total of nine vertices. The simpler cases from Figure 3, resulting in only one or two vertices, are handled by collapsing two or three instances of a vertex to the same position and thereby generating degenerate triangles. For each instance of every vertex, we must also send the positions of the previous and next vertices in the input triangle, as well as a local index in the range $[0,2]$. The positions and indices are needed to compute which case and which semidiagonal to use when computing the new vertex position.

The core of the vertex program consists of code that selects one of the three cases and creates an appropriate output vertex, as shown in Listing II.1.

```
// semiDiagonal[0,1] = Normalized semidiagonals in the same
// quadrants as the edge's normals.
// vtxPos = The position of the current vertex.
// hPixel = dimensions of half a pixel cell

float dp = dot(semiDiagonal[0], semiDiagonal[1]);
float2 diag;
if(dp > 0) {
    // The normals are in the same quadrant -> One vertex
    diag = semiDiagonal[0];
}
else if (dp < 0) {
    // The normals are in neighboring quadrants -> Two vertices
    diag = (In.index == 0 ? semiDiagonal[0] : semiDiagonal[1]);
}
else {
    // The normals are in opposite quadrants -> Three vertices
    if (In.index == 1) {
        // Special "quadrant between two quadrants case"
        diag = float2( semiDiagonal[0].x * semiDiagonal[0].y *
                       semiDiagonal[1].x,
                       semiDiagonal[0].y * semiDiagonal[1].x *
                       semiDiagonal[1].y);
    }
    else
        diag = (In.index == 0 ? semiDiagonal[0] : semiDiagonal[1]);
}

vtxPos.xy += hPixel.xy * diag * vtxPos.w;
```

Listing II.1: Vertex Program Implementing the First Algorithm

In cases with input triangles that have vertices behind the eye, we can get projection problems that force tessellation edges out of the bounding polygon in its visible regions. To solve this problem, we perform basic near-plane clipping of the current edge. If orthographic projection is used, or if no polygon will intersect the near clip plane, we skip this operation.

3.3 The Second Algorithm

The weakness of the first algorithm is that it requires multiple output vertices for each input vertex. An approach that avoids this problem is to compute a bounding triangle for every input triangle, instead of computing the optimal bounding

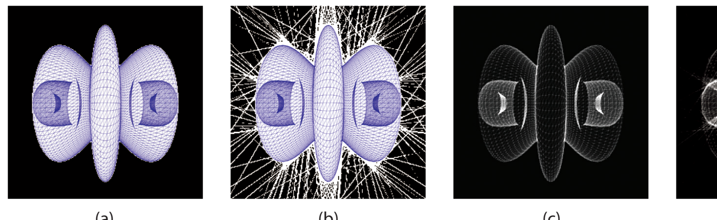


Figure 4: Objects Rasterized in Different Ways: (a) An object rasterized with the first algorithm. (b) The object rasterized using bounding triangles only. (c) Overdraw for the first algorithm. (d) Overdraw for the second algorithm.

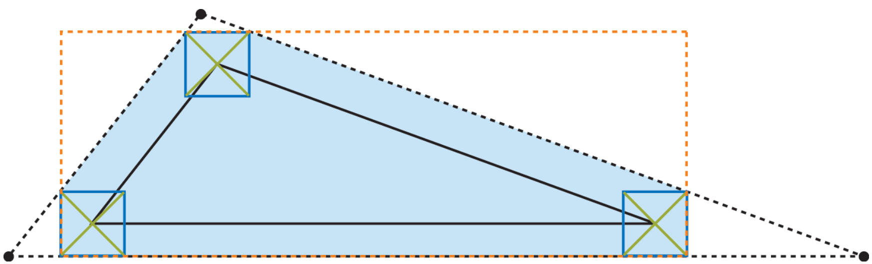


Figure 5: An Input Triangle and Its Optimal Bounding Polygon: The bounding triangle is shown by the dashed line. Its axis-aligned bounding box is shown by the orange dashed line. The shaded area is the optimal bounding polygon.

polygon, which may have as many as nine vertices. However, the bounding triangle is a bad approximation for triangles with acute angles. As a result, we get an overly conservative rasterization, as shown in Figure 4. The poor fit makes the bounding triangles practically useless. To work around this problem, we use an alternative interpretation of a simple test for conservative rasterization [1]. The bounding polygon (as used in Section 3.2) can be computed as the intersection of the bounding triangle and the axis-aligned bounding box (AABB) of itself. The AABB of the bounding polygon can easily be computed from the input triangle. Figure 5 illustrates this process.

In our second algorithm, we compute the bounding triangle in a vertex program and then use a fragment program to discard all fragments that do not overlap the AABB.

We can find the three edges of the bounding triangle by computing the normal of the line through each edge of the input triangle and then moving that line by the worst-case semidiagonal. The worst-case semidiagonal is always the one found in the same quadrant as the line normal.

After computing the translated lines, we compute the intersection points of adjacent edges to get the vertices of the bounding triangle. At this point, we can make good use of the clip-space representation. Because each line is represented as a plane through the origin, we can compute the intersection of two planes with normals $n_1 = (x_1, y_1, w_1)$ and $n_2 = (x_2, y_2, w_2)$ as $n_1 \times n_2$. Note that the result is a direction rather than a point, but all points in the given direction will project to the same point in window space. This also ensures that the w component of the computed vertex receives the correct sign.

Implementation

Because the bounding triangle moves every vertex of the input triangle to a new position, we need to send only three vertices for every input triangle. However, along with each vertex, we also send the positions of the previous and next vertices, as texture coordinates, so we can compute the edges. For the edges, we use planes rather than lines. A parametric representation of the lines, in the form of a point and a direction, would simplify the operation of moving the line at the cost of more problems when computing intersection. Because we represent our lines as equations of the form:

$$(a,b) \cdot x + c = 0,$$

we must derive how to modify the line equation to represent a movement by a vector v. It can be done by modifying the c component of the line equation as:

$$c_{moved} = c - v \cdot (a,b).$$

If we further note that the problem of finding a semidiagonal is symmetric with respect to 90-degree rotations, we can use a fixed semidiagonal in the first quadrant along with the component-wise absolute values of the line normal to move the line. The code in Listing II.2 shows the core of the bounding triangle computation.

```
// hPixel = dimensions of half a pixel cell

// Compute equations of the planes through the two edges
float3 plane[2];
plane[0] = cross(currentPos.xyw - prevPos.xyw, prevPos.xyw);
plane[1] = cross(nextPos.xyw - currentPos.xyw, currentPos.xyw);

// Move the planes by the appropriate semidiagonal
plane[0].z -= dot(hPixel.xy, abs(plane[0].xy));
plane[1].z -= dot(hPixel.xy, abs(plane[1].xy));

// Compute the intersection point of the planes.
float4 finalPos;
finalPos.xyw = cross(plane[0], plane[1]);
```

Listing II.2: Cg Code for Computing the Bounding Triangle

We compute the screen-space AABB for the optimal bounding polygon in our vertex program. We iterate over the edges of the input triangle and modify the current AABB candidate to include that edge. The result is an AABB for the input triangle. Extending its size by a pixel cell gives us the AABB for the optimal bounding polygon. To guarantee the correct result, we must clip the input triangle by the near clip plane to avoid back projection. The clipping can be removed under the same circumstances as for the previous algorithm.

We send both the AABB of the bounding polygon and the clip-space position to a fragment program, where we perform perspective division on the clip-space position and discard the fragment if it lies outside the AABB. The fragment program is implemented by the following code snippet:

```
// Compute the device coordinates of the current pixel
float2 pos = In.clipSpace.xy / In.clipSpace.w;

// Discard fragment if outside the AABB. In.AABB contains min values
// in the xy components and max values in the zw components
discard(pos.xy < In.AABB.xy || pos.xy > In.AABB.zw);
```

3.4 Underestimated Conservative Rasterization

So far, we have covered algorithms only for overestimated conservative rasterization. However, we have briefly implied that the optimal bounding polygon for underestimated rasterization is a triangle or an "empty" polygon. The bounding triangle for underestimated rasterization can be computed just like the bounding triangle for overestimated rasterization: simply swap the minus for a plus when computing a new c component for the lines. The case of the empty polygon is automatically addressed because the bounding triangle will change winding order and be culled by the graphics hardware.

4 Robustness Issues

Both our algorithms have issues that, although not equivalent, suggest the same type of solution. The first algorithm is robust in terms of floating-point errors but may generate front-facing triangles when the bounding polygon is tessellated, even though the input primitive was back-facing. The second algorithm generates bounding triangles with the correct orientation, but it suffers from precision issues in the intersection computations when near-degenerate input triangles are used. Note that degenerate triangles may be the result of projection (with a very large angle between view direction and polygon), rather than a bad input.

To solve these problems, we first assume that the input data contains no degenerate triangles. We introduce a value, ε, small enough that we consider all errors caused by ε to fall in the same category as other floating-point precision errors. If the

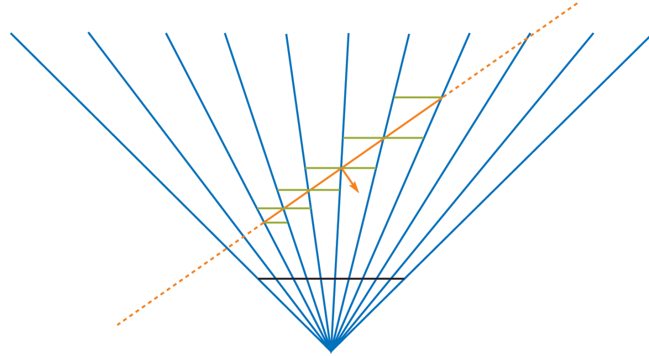


Figure 6: Finding the Farthest Depth Value: A view frustum, with pixel cells (blue lines) and a triangle (orange), as seen from above. The dashed line is the plane of the triangle, and the orange arrow indicates its normal. The range of possible depth values is also shown for the rasterized pixels. The direction of the normal can be used to find the position in a pixel cell that has the farthest depth value. In this case, the normal is pointing to the right, and so the farthest depth value is at the right side of the pixel cell.

signed distance from the plane of the triangle to the viewpoint is less than ε, we consider the input triangle to be back-facing and output the vertices expected for standard rasterization. This hides the problems because it allows the hardware-culling unit to remove the back-facing polygons.

5 Conservative Depth

When performing conservative rasterization, you often want to compute conservative depth values as well. By conservative depth, we mean either the maximum or the minimum depth values, z_{max} and z_{min}, in each pixel cell. For example, consider a simple collision- detection scenario [2] with two objects A and B. If any part of object A is occluded by object B and any part of object B is occluded by object A, we say that the objects potentially collide. To perform the first half of this test using our overestimating conservative rasterizer, we would first render object B to the depth buffer using a computed z_{min} as the depth. We would then render object A with depth writes disabled and occlusion queries enabled, and using a computed z_{max} as depth. If any fragments of object A were discarded during rasterization, the objects potentially collide. This result could be used to initiate an exact intersection point computation on the CPU.

When an attribute is interpolated over a plane covering an entire pixel cell, the extreme values will always be in one of the corners of the cell. We therefore compute z_{max} and z_{min} based on the plane of the triangle, rather than the exact triangle representation. Although this is just an approximation, it is conservatively

correct. It will always compute a z_{max} greater than or equal to the exact solution and a z_{min} less than or equal to it. This is illustrated in Figure 6.

The depth computation is implemented in our fragment program. A ray is sent from the eye through one of the corners of the current pixel cell. If z_{max} is desired, we send the ray through the corner found in the direction of the triangle normal; the z_{min} depth value can be found in the opposite corner. We compute the intersection point between the ray and the plane of the triangle and use its coordinates to get the depth value. In some cases, the ray may not intersect the plane (or have an intersection point behind the viewer). When this happens, we simply return the maximum depth value.

We can compute the depth value from an intersection point in two ways. We can compute interpolation parameters for the input triangle in the vertex program and then use the intersection point coordinates to interpolate the z (depth) component. This works, but it is computationally expensive and requires many interpolation attributes to transfer the (constant) interpolation base from the vertex program to the fragment program. If the projection matrix is simple, as produced by `glFrustum`, for instance, it will be of the form:

$$\begin{pmatrix} \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2far \times near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

In this case we can use a second, simpler way of computing the depth value. Under the assumption that the input is a normal point with $w_e = 1$ (the eye-space w component is one), we can compute the z_w (window-depth) component of an intersection point from the w_c (clip-space w) component. For a depth range $[n, f]$, we compute z_w as:

$$z_w = \left(\frac{f+n}{2}\right) + \left(\frac{f-n}{2}\right)\left(\left(\frac{far+near}{far-near}\right) - \frac{2far \times near}{(far-near)w_c}\right)$$

6 Results and Conclusions

We have implemented two GPU-accelerated algorithms for conservative rasterization. Both algorithms have strong and weak points, and it is therefore hard to pick a clear winner. As previously stated, the first algorithm is costly in terms of geometry processing, while the second algorithm requires more fill rate. However, the distinction is not that simple. The overdraw complexity of the second algorithm depends both on the acuity of the triangles in a mesh and on the rendering resolution, which controls how far an edge is moved. The extra overdraw caused by the second algorithm therefore depends on the mesh tessellation in proportion to the rendering resolution. Our initial benchmarks in Figure 7 show that the first algorithm seems to be more efficient on high-end hardware (such as GeForce 6800

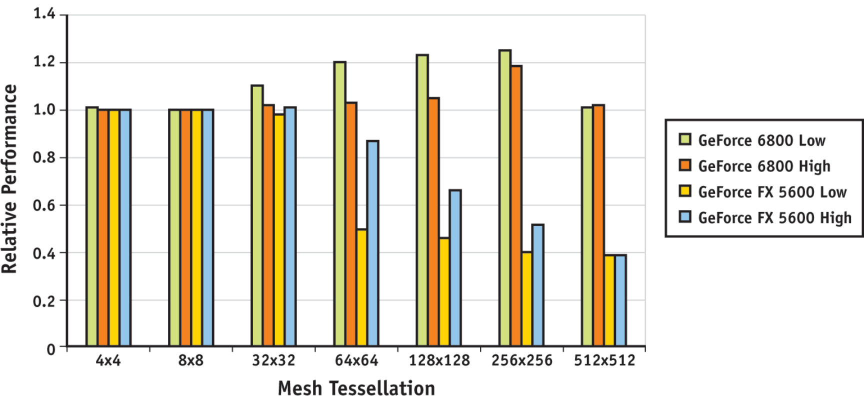


Figure 7: Benchmarked Results: Benchmarks on a GeForce 6800 Series GPU and a GeForce FX 5600. Tests were performed at low (128×128 pixels) and high (1024×1024 pixels) resolution. The graph shows the relative performance of the first algorithm compared to the second.

Series GPUs). Older GPUs (such as the GeForceFX 5600) quickly become vertex limited; therefore, the second algorithm is more suitable.

It is likely that the first algorithm will be the better choice in the future. The vertex processing power of graphics hardware is currently growing faster than the fragment processing power. And with future features such as geometry shaders, we could make a better implementation of the first algorithm.

We can probably enhance the performance of both of our algorithms by doing silhouette edge detection on the CPU and computing bounding polygons only for silhouette edges, or for triangles with at least one silhouette edge. This would benefit the GPU load of both of our algorithms, at the expense of more CPU work.

Our algorithms will always come with a performance penalty when compared to standard rasterization. However, because we can always guarantee that the result is conservatively correct, it is possible that the working resolution can be significantly lowered. In contrast, standard rasterization will sometimes generate incorrect results, which for many applications is completely undesirable.

Bibliography

[1] Tomas Akenine-Möller and Timo Aila. Conservative Tiled Rasterization Using a Modififed Triangle Setup. *Journal of graphics tools*, 10(2):1–8, 2005.

[2] Naga K. Govindaraju, Redon Stephane, Ming C. Lin, and Dinesh Manocha. CULLIDE: Interactive Collision Detection Between Complex Models in Large Environments Using Graphics Hardware. In *Graphics Hardware*, pages 25–32, 2003.

[3] Vladlen Koltun, Daniel Cohen-Or, and Yiorgos Chrysanthou. Hardware-Accelerated From-Region Visibility Using a Dual Ray Space. In *12th Eurographics Workshop on Rendering*, pages 204–214, 2001.

[4] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. CC Shadow Volumes. In *Eurographics Symposium on Rendering*, pages 197–205, 2004.

[5] Karol Myzskowski, Oleg G. Okunev, and Tosiyasu L. Kunii. Fast Collision Detection Between Complex Solids Using Rasterizing Graphics Hardware. *The Visual Computer*, 11(9):497–512, 1995.

Paper III

An Efficient Multi-View Rasterization Architecture

Jon Hasselgren Tomas Akenine-Möller

Lund University

{jon|tam}@cs.lth.se

Abstract

Autostereoscopic displays with multiple views provide a true three-dimensional experience, and full systems for 3D TV have been designed and built. However, these displays have received relatively little attention in the context of real-time computer graphics. We present a novel rasterization architecture that rasterizes each triangle to multiple views simultaneously. When determining which tile in which view to rasterize next, we use an efficiency measure that estimates which tile is expected to get the most hits in the texture cache. Once that tile has been rasterized, the efficiency measure is updated, and a new tile and view are selected. Our traversal algorithm provides significant reductions in the amount of texture fetches, and bandwidth gains on the order of a magnitude have been observed. We also present an approximate rasterization algorithm that avoids pixel shader evaluations for a substantial amount (up to 95%) of fragments and still maintains high image quality.

Proceedings of EGSR, pages 61–72, 2006.

1 Introduction

The next revolution for television is likely to be 3D TV [18], where a *multi-view autostereoscopic* display [10, 16] is used to create a greatly enhanced three-dimensional experience. Such displays can be viewed from several different view-points, and thus provide motion parallax. Furthermore, the viewers can see stereo-scopic images at each viewpoint, which means that binocular parallax is achieved. Displays capable of providing binocular parallax without the need for special glasses are called *autostereoscopic*. This is in contrast to ordinary displays, where a 3D scene is projected to a flat 2D surface.

Analogously, for real-time graphics, the next revolution might very well be the use of autostereoscopic multi-view displays for rendering. Possible uses are scientific & medical visualization, user interfaces & window managers, advertising, and games, to name a few. Another area of potential great impact is stereo displays for mobile phones, and companies such as Casio and Samsung have already announced such displays.

Stereo is the simplest case of multi-view rendering, and APIs such as OpenGL [27], have had support for this since 1992. To accelerate rendering for stereo, a few approximate techniques have been suggested [25, 30]. Furthermore, efficient algorithms for stereo volume rendering [2, 13] and ray tracing [1] have been proposed. The PixelView hardware architecture [29] is used to compute and visualize a four-dimensional ray buffer, which is essentially a lumigraph or light field. The drawback is the expensive computation of the ray buffer, which makes supporting animated scenes difficult.

Surprisingly, to the best of our knowledge, only one research paper exists on rasterization for multiple viewpoints. Halle [12] presents a method for multiple viewpoint rendering on existing graphics hardware, by rendering polygons as a multitude of lines in epipolar plane images. His system and ours can be seen as complementary, since his algorithm works well for hundreds of views, but breaks down for few views (<10). Our algorithms, on the other hand, have been designed for few views (≤ 16). Another difference is that we target a new hardware implementation, instead of using existing hardware.

The inherent difficulty in rendering from multiple views is that rasterization for n views tends to cost n times as much as a single view. For example, for stereo rendering the cost is expected to be twice as expensive as rendering a single view [4]. Rendering to a display with, say, 16 views [18] would put an enormous amount of pressure on even the most powerful graphics cards of today. However, since the different viewpoints are relatively close to each other, the coherency between images from different views can potentially be exploited for much more efficient rendering, and this is the main purpose of our multi-view rasterization architecture.

Our architecture is orthogonal to a wide range of bandwidth reducing algorithms, such as texture compression [7, 17], texture caching [11, 14], prefetching [15], color compression [20], depth compression & Z-max-culling [21], Z-min-culling [6],

and delay streams [3]. In addition to using such techniques, our architecture directly exploits the inherent coherency when rendering from multiple views (including stereo) by using a novel triangle traversal algorithm. Our results show that our architecture can provide significant reductions in the amount of texture fetches when rendering exact images. We also present an algorithm that approximates pixel shader evaluations from nearby views. This algorithm generates high quality images, and avoids execution of entire pixel shaders for a large amount of the fragments.

2 Motivation

Here we will argue that texturing is very likely to be the dominating cost in terms of memory accesses, now and in the future. For this, we use some simple formulae for predicting bandwidth usage for rasterization. Given a scene with average depth complexity d, the average overdraw, $o(d)$, is estimated as [9]:

$$o(d) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{d}. \tag{1}$$

Without using any bandwidth reducing algorithms, the bandwidth required by a single pixel is [6]:

$$b = d \times Z_r + o(d) \times (Z_w + C_w + T_r), \tag{2}$$

where Z_r and Z_w are the cost, in bytes, for reading and writing a depth buffer value, respectively. Furthermore, C_w is the cost for writing to the color buffer (assuming no blending is done, since the term C_r is missing) and T_r is the total cost for accessing textures for a fragment. Standard values for these are: $Z_r = Z_w = C_w = 4$ bytes and a texel is often stored in 4 bytes. Trilinear mipmapping [33] is commonly used for reducing aliasing, and this requires eight texel accesses, which makes $T_r = 8 \times 4 = 32$ bytes for a filtered color from one texture. If a texture cache [11] with miss rate, m, is used, and we have a multi-view display system with n views, Equation 2 becomes:

$$\begin{aligned} b(n) &= n \times [d \times Z_r + o(d) \times (Z_w + C_w + m \times T_r)] \\ &= n \times [\underbrace{d \times Z_r + o(d) \times Z_w}_{\text{depth buffer, } B_d} + \underbrace{o(d) \times C_w}_{\text{color buffer, } B_c} + \underbrace{o(d) \times m \times T_r}_{\text{texture read, } B_t}] \\ &= n \times [B_d + B_c + B_t] \end{aligned} \tag{3}$$

Now, we analyze the different terms in this equation by looking at two existing example shaders (not written by the authors). First, we assume the scene has a depth complexity of $d = 4$ ($\Rightarrow o \approx 2$). This is quite reasonable, since occlusion culling algorithms and spatial data structures (e.g. portals) efficiently reduce depth

complexity to such low numbers, or even lower. For example, the recent game S.T.A.L.K.E.R [28] has a target depth complexity of $d = 2.5$. Using $d = 4$, the depth buffer term will be $B_d \approx 4 \times 4 + 2 \times 4 = 24$ bytes and the color buffer term becomes $B_c \approx 2 \times 4 = 8$ bytes. Furthermore, we assume a texture cache miss rate of 25% ($m = 0.25$). This figure comes from Hakura and Gupta [11], who found that each texel is used by an average of four fragments when trilinear mipmapping is used. We have observed roughly the same behavior in our tests scenes.

Example I: Pelzer's ocean shader [24] uses seven accesses to textures using trilinear mipmapping. The texture bandwidth alone will thus be $B_t = 2 \times 0.25 \times 7 \times (8 \times 4) \approx 112$ bytes. □

Example II: Uralsky [31] presents an adaptive soft shadow algorithm, which uses more samples in penumbra regions. Eight points are first used to sample the shadow map, and if all samples agree, the point is considered as either in umbra or fully lit. Otherwise, 56 more samples are used. With percentage-closer filtering [26] using four samples per texture lookup, we get, $B_t = 8 \times (4 \times 4) = 128$ bytes in non-penumbra regions, while in penumbra, $B_t = 64 \times (4 \times 4) = 1024$ bytes. Assuming only 10% of the pixels are in penumbra, the estimated cost becomes $B_t = 2 \times 0.25 \times (0.9 \times 128 + 0.1 \times 1024) \approx 109$ bytes. □

Compared to $B_c = 8$ and $B_d = 24$, it is apparent that texture memory bandwidth is substantially larger ($B_t = 112$ and $B_t = 109$). This term can also easily grow much larger. For example, current hardware allows for essentially unlimited texture accesses per fragment, and with anisotropic texture filtering the cost rises further.

In addition to this, both B_d and B_c can be reduced using lossless compression [21, 20]. Morein reports that depth buffer compression reduces memory accesses to the depth buffer by about 50%. B_d can also be further reduced using Z-min-culling [6] and Z-max-culling [21]. The advantage of buffer compression and culling is that they work transparently—the user do not need to do anything for this to work. For texture compression, however, the user must first compress the images, and feed them to the rasterizer. Both Example I and II contain textures that cannot be compressed, since they are created on the fly. Furthermore, none of the ordinary textures were compressed in the example code.

Texturing can easily become the largest cost in terms of memory bandwidth, as argued above. This fact is central when designing our traversal algorithm (Section 4). In computer cinematography, pixel shaders can be as long as several thousand lines of code [23], and this trend of making longer and longer pixel shaders can be seen in real-time rendering as well. This means that many applications are often pixel-shader bound. Thus, in a multi-view rasterization architecture, it may also be desirable to reduce the number of pixel shader evaluations, which is the topic of Section 4.3.

```
BRUTEFORCE-MULTIVIEWRASTERIZATION()
1  for i ← 1 to n                    // loop over all views
2     create M^i                     // create projection matrix
3     for j ← 1 to t                 // loop over all triangles
4          RASTERIZETRIANGLE(M^i, Δ_j)
5     end
6  end
```

```
NEW-MULTIVIEWRASTERIZATION()
1  create all M^i, i ∈ [1,...,n]     // create projection matrices
2  for j ← 1 to t                    // loop over all triangles
3     RASTERIZETRIANGLETOALLVIEWS(M^1,...,M^n, Δ_j)
4  end
```

Figure 1: Hi-level pseudo code for brute-force multi-view rasterization (top), and for our new multi-view rasterization algorithm (bottom). Assume rendering is done to n views, and that the scene consists of triangles, Δ_j, $j \in [1,\dots,t]$. Note that the core of our algorithm lies in the RASTERIZETRIANGLETOALLVIEWS() function.

3 Background: Multi-View Rasterization

This section briefly describes a brute-force architecture for multi-view rasterization and multi-view projection.

3.1 Brute-Force Multi-View Rasterization

Here, we describe a brute-force approach to multi-view rasterization. This is used when rendering stereo in OpenGL and DirectX, and therefore we assume that this is the norm in multi-view rasterization. The basic idea is to first render the entire scene for the left view, and save the resulting color buffer. In a second pass, the scene is rendered for the right view, using another projection matrix. The resulting color buffers form a stereo image pair. Extending this to $n \geq 2$ views is straightforward.

Pseudo code for brute-force multi-view rasterization is given in the top part of Figure 1, where it is assumed that we have a scene consisting of t triangles, Δ_j, $j \in [1,\dots,t]$.

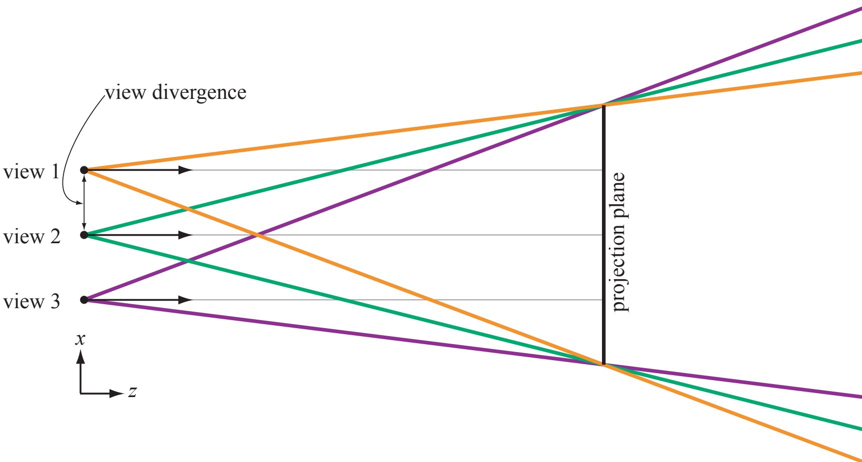


Figure 2: Off-axis projections for three views. As can be seen, all views share the same z-axis. We use the term "view divergence" for the distance between two views' viewpoints.

3.2 Multi-View Projection

We concentrate on projections with only horizontal parallax, since the great majority of existing autostereoscopic multi-view displays can only provide parallax along one axis. In this case, off-axis projection matrices are used. In Figure 2, this type of projection is illustrated for three views. Note that the distance between two views' camera viewpoints is denoted *view divergence*. In the following, we assume that n views are used, and we use an index, $i \in [1, \ldots, n]$, to identify a particular view. Furthermore, we have a vertex, $\mathbf{v}$, in object space, that should be transformed into homogeneous screen space for each view. These transformed vertices are denoted $\mathbf{p}^i$, $i \in [1, \ldots, n]$. For each view, a different object-space to homogeneous screen-space matrix, $\mathbf{M}^i$, must be created. Since parallax is limited to the x-direction, only the components of the first row of the $\mathbf{M}^i$ are different—the remaining three rows are constant across all views. Thus, the y, z, and w components of $\mathbf{p}_i = (p^i_x, p^i_y, p^i_z, p^i_w)^{\mathrm{T}} = \mathbf{M}^i\mathbf{v}$ will be exactly the same. We will use this fact when designing our traversal algorithm (next section). This could also be exploited for implementing an efficient vertex shader unit for a multi-view rendering architecture, but that is beyond the scope of this paper.

4 New Multi-View Rendering Algorithms

Since it is likely that texturing (B_t) is the largest consumer of memory bandwidth, our strategy is to devise a traversal algorithm that reduces the $n \times B_t$-term of Equation 3 as much as possible. Our hypothesis is that this should be possible if the

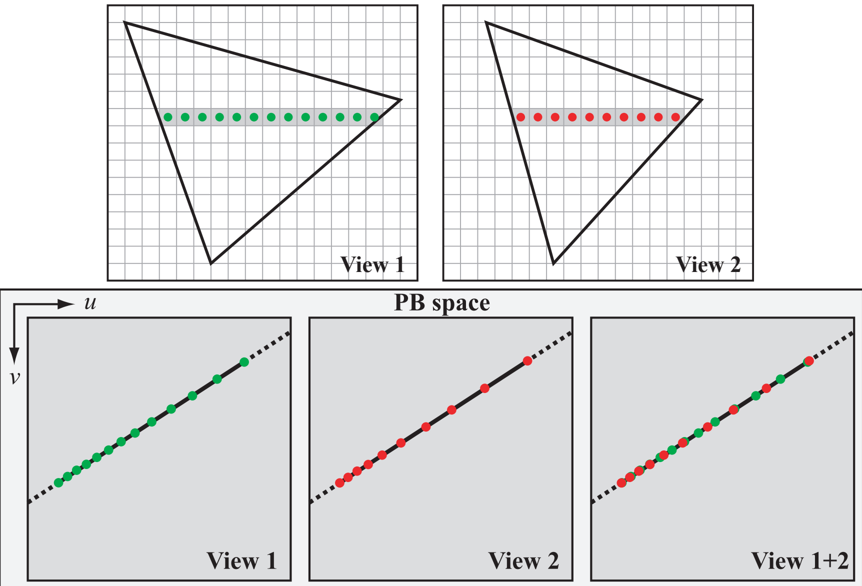


Figure 3: The two views in a stereo system. A single scanline is highlighted for both the left and right view. In the bottom row of the figure, we show the PB space for view 1 and view 2, where the green and red samples in PB space correspond to the samples along the selected scanlines. To the right, the texture sample locations from the two views are placed on top of each other. With respect to the texture cache, the best order to traverse the pixels in the two views is the order in which the samples occur along the PB traversal direction (fat line in PB space). The reason that the PB traversal directions in the two views are identical, is the multi-view projection, described in Section 3.2.

texture cache can be exploited by all views simultaneously. Ideally, all views use exactly the same texels, and that would reduce $n \times B_t$ to B_t, which is a substantial improvement. Obviously, the best case will not occur, but very good results can be obtained as we will see. To make this possible, our approach is to rasterize a triangle to all views before starting on the next triangle. Since the same texture is applied to a particular triangle for all views, one can expect to get more hits in the texture cache with this approach compared to the brute-force variant (Section 3.1).

The number of texture cache hits will also vary depending on how a triangle for the different views is traversed. Therefore, the core of our architecture lies in the function RASTERIZETRIANGLETOALLVIEWS (Figure 1), and in the following two subsections, we describe two variants of that algorithm. The general idea of our traversal algorithms is to traverse a small part (e.g., a tile or even just a pixel) of a triangle for a given view, and then determine which view to traverse next. To do this, we maintain an *efficiency measure*, E^i, for each view. With respect to the

texture cache, the efficiency measure estimates which view is the best to continue traversing. Thus, E^i guides the traversal order of the different views.

We start by describing our algorithm for scanline-based traversal, and then show how it can be generalized to tiled traversal. In Section 4.3, we extend the traversal algorithm so that pixel shader evaluations can be approximated from one view to other views. Finally, in Section 4.4, our architecture is augmented in order to generate effects, such as depth of field, which require many samples.

4.1 Scanline-Based Multi-View Traversal

As many different parameters are often interpolated in perspective over a triangle, it is beneficial to compute normalized, perspective-correct barycentric coordinates (PBs), $\mathbf{t} = (t_u, t_v)$ once per fragment, and then use these to interpolate the parameters in a subsequent step. The PBs can be expressed using rational basis functions [19], and we will refer to the coordinate space of the PBs as *PB space*. Note that each view and pixel has its own PB, $\mathbf{t}^i = (t^i_u, t^i_v)$, where i is the view number.

The goal of our algorithm is to provide for substantial optimization of texture cache performance by roughly sorting the rasterized pixels by their respective PBs, and thereby sorting all texture accesses. In order to motivate this statement, we assume that a pixel shader program is used to compute the color of a fragment. The color will be a function, $color = f(\mathbf{t}^i, S^i)$, of the PB, t^i, and some state, S^i, consisting of constants for view i. If we assume that the shader contains no view dependencies, then all states, S^i, will be equal and we can write $color = f(\mathbf{t}^i, S)$, meaning that the only varying parameter will be the PBs. Since pixel shader programs are purely deterministic, the exact same PB will yield the same texture accesses. Therefore, it is reasonable to assume that roughly sorted PBs will give roughly sorted texture accesses. This applies to all texture accesses, including nested or dependant accesses, as long as they do not depend on the view. Note in particular that a shader containing view-independent texture access followed by view-dependent shading computations will be efficiently handled by our algorithm. An example of this is a shader that applies a bump map to a surface in order to perturb the normal, and after that, specular shading is computed based on the normal.

The rationale for our traversal algorithm is illustrated in Figure 3. For simplicity, only a stereo system is shown, but the reasoning applies to any number of views. We focus on a single scanline at a time. As can be seen, evenly spaced sample points in screen space are unevenly distributed in the PB space due to perspective. Using multi-view projection, the sample points for both views are located on a straight line in PB space. The direction of this line is denoted the *PB traversal direction*.

To guide the traversal, we define a signed efficiency measure, E^i, for each view, i, as:

$$E^i = \mathbf{d} \cdot \mathbf{t}^i, \tag{4}$$

where $\mathbf{d} = (d_u, d_v)$ is the PB traversal direction, which is computed as the differ-

ence between two PBs located on the same scanline. Thus, E^i is the projection of $\mathbf{t}^i$ onto the PB traversal direction.

In order to sort the pixel traversal order, we simply chose to traverse the pixel, and view, with the smallest efficiency measure E^i. When a pixel has been visited for view, i, the $\mathbf{t}^i$ for the next pixel for that view is computed, and the efficiency measure, E^i, is updated. The next view to traverse in is selected as before, and so on, until all pixels on the scanline have been visited for all views. Then, the next scanline is processed until the entire triangle has been traversed. An optimization of Equation 4 is presented in Section 5.

Below, pseudo code for our new traversal algorithm is shown. A single scanline is only considered since every scanline is handled in the same way.

TRAVERSESCANLINE(scanlinecoord y)

1. compute coordinate, x^i, for the leftmost pixel inside triangle
   for each view on scanline y
2. compute l^i =no. of pixels on current scanline for all views i
3. compute $\mathbf{t}^i$ for all views, i, for the leftmost pixel, (x^i, y)
4. compute $\mathbf{d}$, and compute $E^i = \mathbf{d} \cdot \mathbf{t}^i$ for all views, i
5. **while** (pixels left on scanline for at least one view)
6.   find view, k, with smallest E^k and $l^k > 0$
7.   visit pixel (x^k, y) using $\mathbf{t}^k$ for view k
8. $x^k = x^k + 1$, $l^k = l^k - 1$
9.   update $\mathbf{t}^k$ and E^k
10. **end**

In the algorithm presented above, we have used only the perspective-correct barycentric coordinates (PBs) to guide the traversal order. This does not take mipmapping into account. It would be very hard to optimize for mipmapping as it depends on a view-dependent level-of-detail parameter, λ^i, which is usually computed from the pixel shader program state using finite differences. It may be possible to optimize for mipmapping in the simple case of linear texture mapping, but it is next to impossible to generalize this to dependent accesses. Furthermore, we believe it is not worth complicating our algorithm further for an expectedly slight increase in performance.

4.2 Tiled Multi-View Traversal

While scanline-based traversal works fine, there are many advantages of using a tiled traversal algorithm, where a tile of $w \times h$ pixels is visited before moving on to the next tile. For example, it has been shown that texture caching works even better [11], that simple forms of culling [6, 21] can be implemented, and that depth buffer and color buffer compression [20, 21] can be employed. These types of algorithms are difficult or impossible to use with scanline-based traversal.

Fortunately, our scanline-based traversal algorithm can be extended to work on a per tile basis. This is done by imagining that the pixels in Figure 3 are tiles

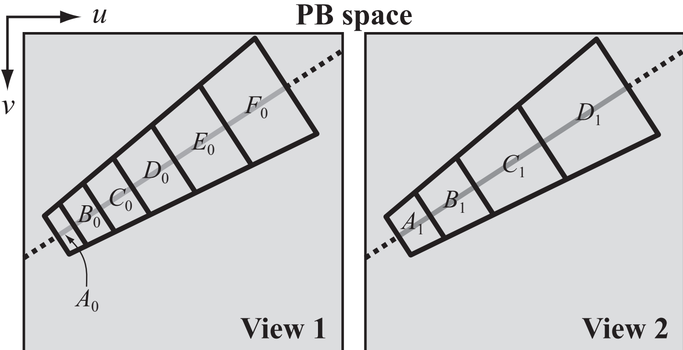


Figure 4: Tiled traversal, shown in PB space, for a row of tiles overlapping a triangle (not shown). The PB traversal direction is the gray line. Our algorithm visits the tiles in the following order: $A_0, A_1, B_0, B_1, C_0, D_0, C_1, E_0, D_1$, and finally F_0. This is the order in which the tiles appear on the PB traversal direction.

rather than pixels. A rectangular tile in screen space projects, in general, to a convex quadrilateral in PB space. In Figure 4, a triangle is assumed to have been rasterized, and for a particular row of tiles, the projection of the tiles are shown in PB space. As can be seen, the projected tiles overlap the same area in PB space for the two views, and therefore the texture cache hit ratio can be expected to be high. This is especially true if the tiles are traversed in the order in which they appear along the PB traversal direction, as we suggest in our algorithm in Section 4.1.

Practically, this amounts to two computational and algorithmic differences compared to scanline-based traversal. First, we compute the efficiency measure, E^i, and perform sorting for the each tile rather than each pixel. As reference point for the computations, we use the center of the tile, but any point inside the tile would do. The second difference is that the traversal algorithm is designed so that all tiles (overlapping a triangle), on a row of tiles, are visited before moving on to the next row of tiles.

4.3 Approximate Pixel Shader Evaluation

In this section, we present an extension for tiled traversal algorithm (Section 4.2), which adds approximated pixel shader evaluation. The general idea, inspired by the work of Cohen-Or et al. [8], is that *exact* pixel shader evaluation is done for a particular view, e, and when rasterizing to a nearby view, a, the pixel shader evaluations from view e are reused if possible. In Section 4.1, we motivated that if the pixel shader program contains no view dependencies, then it will be a deterministic function, $color = f(\mathbf{t}^i, S)$, of the PB, $\mathbf{t}^i$, of a pixel. We used this to motivate that for a given PB coordinate, the shader will always issue the same texture accesses. However, it is also true that the shader will always return the same color given the

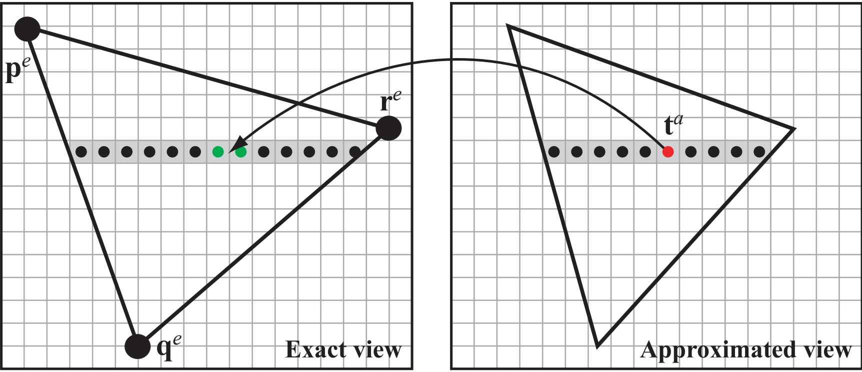


Figure 5: Illustration of how a fragment (red circle) with perspective-correct barycentric coordinates $\mathbf{t}^a$, in an approximated view, can be mapped onto a screen-space position in the exact view. The color of two nearby fragments in the exact view are interpolated to compute the color of the fragment in the approximated view.

same PB as input. This implies that we should be able to reuse the results of the pixel shader programs.

In the following, we present an algorithm that exploits this assumption to provide *approximate* pixel shader evaluations, and our results show that we can obtain high-quality renderings. Since the approximation may produce incorrect results for view-dependent shaders, we suggest that the application programmer should have fine-grained control over this feature in order to turn it off/on as desired.

We initially divide our views into sets where the view divergences of the cameras in each set are considered small enough. When a triangle is rasterized, we select an *exact view* from each set. This can be done by either setting a fixed exact view, or by choosing the view that maximizes the triangle's projected area. We refer to the remaining views in the set as *approximated views*.

Figure 5 illustrates our approximate pixel shader technique. When evaluating the pixel shader for the exact view, we execute the full pixel shader, which may depend on the camera position. Hence, the exact view will render the triangle without any approximations. Looking at a single fragment in an approximated view, we will use its perspective-corrected barycentric coordinates (PBs), $\mathbf{t}^a = (t_u^a, t_v^a)$, to compute the position of the fragment in the exact view's homogeneous screen space. Assume the homogeneous coordinates of the triangle's vertices in the exact view are denoted $\mathbf{p}^e$, $\mathbf{q}^e$, and $\mathbf{r}^e$. Now, to find out which pixel in the exact view that correspond to $\mathbf{t}^a$ in an approximated view, we can use interpolation of the homogeneous screen-space coordinates as shown below:

$$\mathbf{c} = (c_x, c_y, c_z, c_w)^T = (1 - t_u^a - t_v^a)\mathbf{p}^e + t_u^a\mathbf{q}^e + t_v^a\mathbf{r}^e. \quad (5)$$

The screen-space x-coordinate in the exact view is found by homogenization: $x =$

c_x/c_w, and the y-coordinate is implicitly known from the scanline being processed due to the findings in Section 3.2. The position, (x,y), will rarely map exactly to a sample point of a fragment in the exact view, but we can compute the color[1] of the approximated fragment by interpolating between the neighboring fragments in the exact view.

In order to approximate the pixel shader evaluation, we introduce a *shader output cache*, shown in Figure 6, which holds the pixel shader outputs (color and possibly depth) for a number of tiles rendered in the exact view. When a new triangle is being rasterized, we start by clearing the shader output cache to ensure that we only use fragment outputs of the same triangle during approximation. Each time a tile is being rasterized for the exact view, we allocate and clear the next tile-sized entry in the shader output cache. The next entry is selected in a cyclic fashion, so that the least recently traversed tile is retired from the cache. The pixel shader outputs of the fragments in the current tile are then simply written to that cache entry.

Each time we render a fragment in an approximated view, we compute the corresponding fragment position in the exact view as outlined above. The fragments to the left and right of the true fragment position are queried in the cache, and if both fragments are found, we compute the approximated pixel shader output by linear interpolation. If only one of the fragments exists, we simply set the approximated output to the value of that fragment.[2] Finally, if none of the fragments are found, we execute the full pixel shader to compute the exact output.

Our main reason for choosing two shaded fragments and weighting them to form an approximated fragment is that we can use existing hardware interpolators to do the computations. This makes the implementation very inexpensive, and in Section 6, we show that this approximation generates high-quality results for view-independent shaders. However, there is a more precise way of performing the filtering. Instead of projecting the center of the pixel, we project the endpoints of a filter kernel into the exact view. This is shown in Figure 7. When the horizontal parallax difference between the exact view and the approximated view is relatively large, the projected endpoints may span more than two pixels. To obtain a higher quality of the approximated fragments, a larger filter (e.g., tent, or Gaussian) is applied to all the pixels in the span. Note that this technique is also approximate.

The approximation method requires that the pixels in the different views are traversed in an ordered fashion, so the shader output cache is filled with the appropriate results before it is being queried. This is exactly what our algorithms (described in Section 4.1 & 4.2) do, and hence the approximation works well when used together with our traversal algorithms. However, in order to make sure that the shader output cache is filled before we start processing approximated views, we must delay the approximated views slightly. In our implementation we do this

[1] If the pixel shader also computes the depth of the fragment, our algorithm can be used to approximate the depth as well.

[2] Alternatively, one could choose to execute the full pixel shader for such fragments. This would increase the quality slightly.

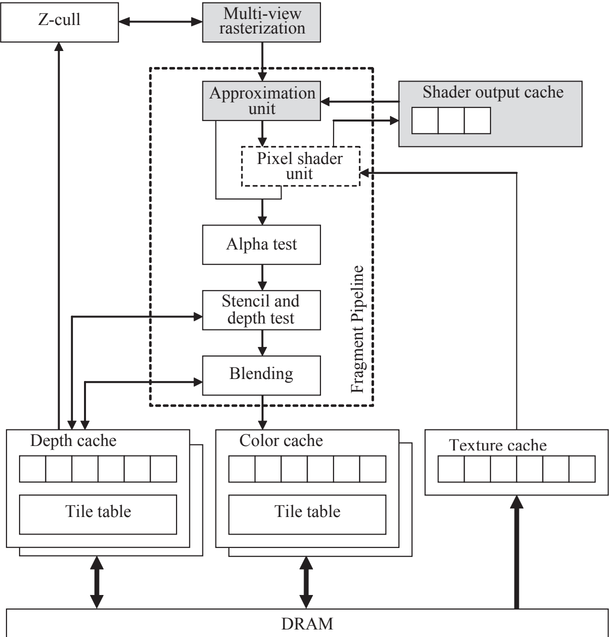


Figure 6: To support approximate pixel shader evaluation, a *shader output cache* and an *approximation unit* (AU) are introduced into the rasterization pipeline. Before the pixel shader is executed for an approximated view, the AU checks with the shader output cache whether the fragment color can be computed from existing fragment colors in the cache. If so, the pixel shader unit is bypassed. Otherwise, the pixel shader is executed as usual.

by computing the efficiency measure, E^i, for a tile located $k-1$ tiles away along the currently traversed row of tiles, where k is the number of entries in the shader output cache. For exact views, we compute the efficiency measure as usual. This will cause our sorted traversal algorithm of Section 4.1 to pre-load the cache.

Discussion One possibility we explored was to augment existing depth and color buffer caches and use them to do the job of the shader output cache. This requires a small extension of one bit per cache entry in order to be able to flag if a color or depth value was written while rasterizing the current triangle. We found that this approach works satisfactory for opaque geometry. However, we cannot perform any approximations when blending is enabled, because using the blended values for approximation may cause distortion of geometry seen through a transparent triangle. Blending is popular for particle effects, and crucial to multi-pass techniques in general. Therefore, we think that it is important to be able to support blending in our approximate algorithm as well.

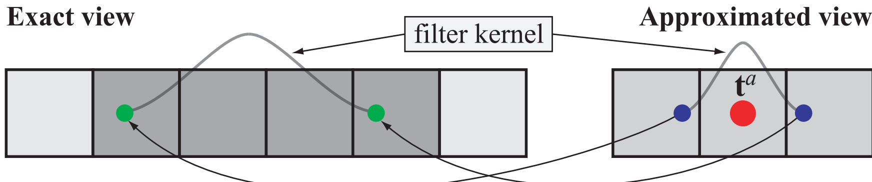


Figure 7: Higher quality filtering is obtained by projecting the blue end points of the filter kernel in the approximated view into the exact view. These projected points span a number of pixels (dark gray), which are weighted using the filter kernel to form the approximated fragment.

4.4 Accumulative Color Rendering

A very simple and worthwhile extension of our architecture is to allow all views to access a single common color buffer, while each view has its own depth and stencil buffers. This allows for acceleration of some forms of multi-sampling effects, such as, depth of field for a single view. Recall that we compute the traversal order based on a texture-cache efficiency measure in our architecture. Therefore, the rendering order will be correct in the multiple depth buffers, but we cannot make any assumptions about the rendering order in the common color buffer. However, many multi-sampling algorithms can be described on an order-independent form:

$$\mathbf{m} = \sum_{i=1}^{n} w_i \mathbf{c}_i,$$

where n is the number of samples, $\mathbf{c}_i$ is the color of a sample, w_i is the weight of the sample (typically $w_i = 1/n$), and $\mathbf{m}$ is the color of the multi-sampled fragment. This equation can be implemented by using additive blending for the summation, and the factor w_i can be included in the pixel shader.

If the programmer is aware of that a multi-view rasterizer is being used, he/she can accelerate multi-sampling in the cases mentioned above. For instance, if we have hardware capable of handling four views, an image with depth of field using 4×4 samples can be rendered as follows:

RENDERSCENEDEPTHOFFIELD()

1 **for** $y \leftarrow 1$ **to** 4
2 create all $\mathbf{M}^i$, $i \in [1,4]$
3 disable color writes
4 RASTERIZESCENETOALLVIEWS($\mathbf{M}^1, \ldots, \mathbf{M}^4$)
5 enable color writes
6 enable additive blending
7 enable pixel shader that includes the $w_i = 1/16$ scaling term
8 RASTERIZESCENETOALLVIEWS($\mathbf{M}^1, \ldots, \mathbf{M}^4$)
9 **end**

It is important to initialize the depth buffer (line 3 & 4) prior to color rendering, otherwise incorrect results will be obtained using additive blending. Note that this is also the case when using a single-view architecture, and so is not a disadvantage that stems from our architecture. An alternative approach is to render into 16 different color buffers, and combine the result in a post-process. However, our approach yields much better color buffer cache performance, since the triangles rendered simultaneously are coherent in screen space, and the post-processing stage is avoided.

In the pseudo code above, a deferred shading approach is used. Alternatively, the result could be blended directly into an accumulation buffer, but we have found that this uses more bandwidth. It is also worth pointing out that a brute-force implementation of depth of field using n samples per pixel sends the geometry n times to the graphics hardware. Our implementation sends the geometry $2n/v$, where v is the number of views (e.g. 16) the system can handle at a time.

5 Implementation

To benchmark and verify our algorithms, we have implemented a subset of OpenGL 2.0 in a functional simulator in C++. The core of the traversal algorithm and the approximation technique were implemented in less than 300 lines of code.

In our implementation, the efficiency measure E^i that is used to select the next tile to rasterize is computed in a less expensive way, compared to directly evaluating Equation 4. Since E^i is only used to sort the points, $\mathbf{t}^i$, we can evaluate this expression along the axis that corresponds to the largest of $\text{abs}(d_u)$ and $\text{abs}(d_v)$. Thus, E^i is simply the component of $\mathbf{t}^i$ that corresponds to this axis, and the computation of E^i is therefore almost for free.

Clipping a triangle against the view frustum may result in at most seven triangles. Since the projection matrices are different for each view, the number of resulting triangles may not be the same for all views. Hence, it is important to traverse unclipped triangles. We have adapted McCool et al's. [19] traversal algorithm in homogeneous space, so that we traverse tiles along the horizontal viewport direction. In order to do so, we first find a screen space axis-aligned bounding box for each triangle, using binary search and a box-triangle overlap test [5]. We then traverse the bounding box using a simple horizontal sweep. A more advanced traversal algorithm would yield higher performance, but this is outside the scope of this paper. The efficiency of the traversal algorithm will not affect bandwidth utilization in our simulator, since we detect tiles not overlapping the triangle and discard them.

In terms of culling, some special cases may occur. For instance, triangles can be backface culled or outside the view frustum in one view, while remaining visible in others. This is easily solved if our algorithms are robustly implemented. A culled triangle can simply be given a scanline width of zero pixels, which will make sure it is never rasterized. For our approximate algorithm, a culled triangle

will not generate any fragments, which means that the shader output cache will not be filled, and approximation will not be done. Thus, the visual result is not compromised.

It should be noted that it may be difficult to compute the PB traversal direction and efficiency measure in the context of a tiled rasterizer. We have favored simple code, and compute the PB traversal direction from the start and end points of a row of tiles. We also evaluate the efficiency measure in the center of each tile, regardless of whether it lies inside the triangle or not. This implementation suffers from a weakness that appear when a row of tiles cross the "horizon" of the plane that pass through the current triangle. When crossing this horizon, the PBs behave similarly to an $1/x$ function in the vicinity of the origin. This results in sign changes in the efficiency measure, and ultimately in incorrect sorting. However, it should be noted that this is a very rare occurrence. Furthermore, it will only affect the texture cache efficiency, and not the correctness of the result. In the future, we would like to investigate if there is an elegant way to extend our implementation so that correct sorting is guaranteed even in these extreme cases.

6 Results

In this section, we present the results for our exact and approximate algorithms (Section 4.2 and 4.3, respectively). The results were obtained from our functional simulator, using the test scenes summarized in the top row of Figure 8.

The Quake3 scene is a game level using multi-texturing with one texture map combined with a light map for every pixel. A *potentially visible set* is used during rendering, so the overdraw factor is similar to that of most modern games.

For our Soft Shadows test scene, we implemented Uralsky's soft shadow mapping algorithm [31] in combination with bump-mapped and gloss-mapped per-pixel Phong shading. This scene is meant to model a modern or next-generation graphics engine, targeted for real-time graphics, which makes heavy use of complex shaders containing many texture accesses.

The Ocean scene is our implementation of Pelzer's ocean shader [24]. This scene is a nightmare scenario for our algorithms, as it contains bump-mapped reflections and refractions, both view-dependent and highly diverging due to the bump mapping and high view divergence of the cameras. Thus, this scene was designed to contradict all assumptions made in our algorithms.

All scenes have an animated camera, and statistics were gathered for at least 200 frames. The Ocean scene also has an animated water surface. We chose to render at a resolution of 640×480 pixels per view, which is reasonable considering the current 3D display technology. For example, Philips has built a 3D display capable of either nine views at 533×400 pixels, or seven views at 686×400 pixels [32]. We have investigated the behavior of our algorithms with respect to rendering resolution, and conclude that they both behave robustly. Both total bandwidth and

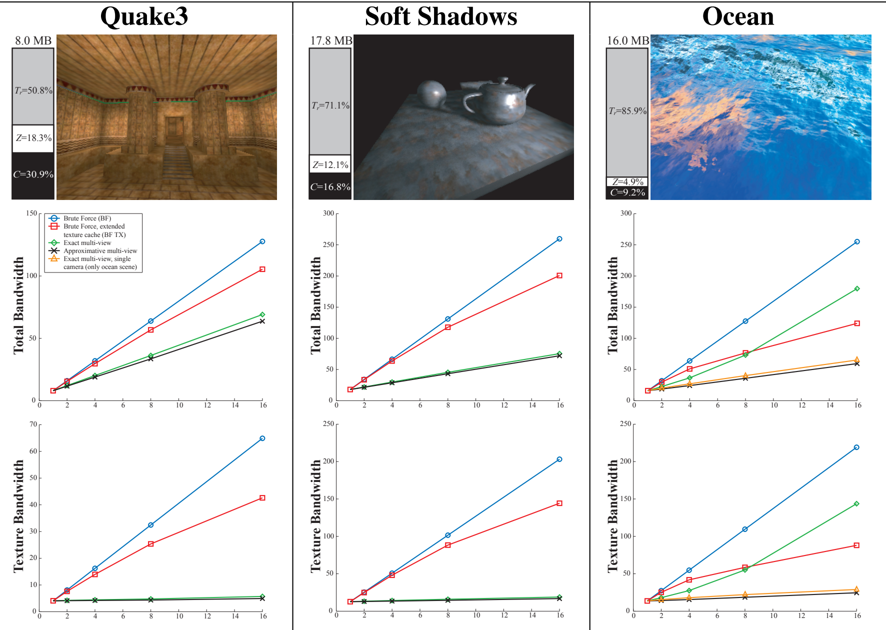


Figure 8: The first row shows a summary of the test scenes rendered at 640×480 pixels. The bandwidth (BW) figures, located on top of each bar, is the BW in megabytes (MB) per frame for a single view using a conventional rasterizer, as described in the text. Each bar is divided into texturing BW (gray), depth buffer BW (white), and color buffer BW (black). The second and third rows show total BW and texture BW per frame as a function of the number of views for each frame.

texturing bandwidth increase slightly sub-linearly with increasing resolution. This effect is due to all caches getting slightly more cache hits at higher resolutions, and the compression ratios of color and depth buffers were either constant or became slightly better. Similar behavior was observed for a brute-force architecture.

Even though there are computations that can be shared among different views in the vertex-processing units, we focus our evaluation only on the rasterizer stage, since it is very likely that it will become the bottleneck. In the following, we refer to a *conventional rasterizer* (CR) as a modern rasterizer architecture with the following bandwidth reducing algorithms: fast depth clears, depth buffer compression, depth buffer caching, Z-max culling, texture caching, color buffer compression and color buffer caching. A multi-view rasterization architecture that is implemented by rendering the scene n times using a single CR is called a *brute-force* (BF) multi-view architecture (see also Section 3.1).

For all architectures, we use a fully associative 6 kB texture cache with least-recently used (LRU) replacement policy. Since our architecture rasterizes to all

views simultaneously, it needs an increasing amount of depth and color buffer cache with an increasing number of views. For all our tests, our architecture uses $n \times 512$ bytes for the depth buffer caches, and $n \times 512$ bytes for the color buffer caches. Thus, a stereo system will use 8 kB cache memory in total.

For a fair comparison, we ensure that our architecture and the BF architecture use exactly the same amount of cache memory. The extra 1 kB of cache memory that we need per view, can be spent on either the depth and color buffer caches, or on the texture cache in a BF architecture. We call these architectures BF DC and BF TX respectively. In all our tests, we have observed that the total amount of bandwidth is reduced most if the texture cache is increased. We have therefore chosen to omit the BF DC architecture from the results.

We present statistics gathered from our test scenes in Figure 8. As can be seen in those diagrams, both our exact and our approximate rasterization algorithms perform far better than the brute-force architecture. For the Quake3 scene, the majority of bandwidth usage is spent on the color and depth buffer. However, our algorithm provides major reductions in terms of texture bandwidth. In fact, it remains almost constant over an increasing number of views. The same holds for the Soft Shadow scene, but the results are even better since the texture bandwidth is more dominating compared to the Quake3 scene.

In the case of the Ocean scene, our algorithm performs worse than BF TX when the number of views is greater than eight. This is not very surprising considering that the scene was designed as a worst case for our algorithm. The scene contains very little view coherency in the texture accesses, and the BF TX architecture will have a much bigger texture cache at 16 views. We think that the results are very good considering the circumstances, and substantial bandwidth reduction is achieved up to four views. In fact, our algorithm performs better even for 16 views, if we render four passes with a four-view architecture. It should be noted that all benchmarks once again turned completely to our favor simply by increasing the texture cache size to 12 kB. This means that for every scene, there is a texture cache size "knee" that makes our algorithm perform extremely well. The same applies to a BF TX architecture: when the texture cache size is decreased, performance will degrade gracefully. We have also included bandwidth measurements for a version of the Ocean scene that used a shared camera position for the shaders (Figure 8), and these indicate how disadvantageous the view dependencies of this scene are.

It should be noted that even though the bandwidth measurements for our approximate algorithm is only marginally better than the exact algorithm, the approximate algorithm completely avoids a very large amount of pixel shader program executions. Hence, computational resources are released and can be used for other tasks. Our tests show that about 95% of the pixels, in the approximated view in a stereo system, can be approximated. When the number of views increases, fewer pixels can be approximated due to increased view divergence, and with 16 views, our approximation ratio has dropped to approximately 80%. See Figure 9 for a visualization of the approximation in a five-view system. A major advantage of our approximate algorithm is that it always generates correct borders of the tri-

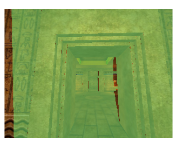
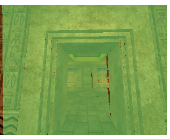
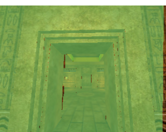
  

Figure 9: Visualization of approximation in the Quake3 scene. Green pixels have been approximated from the exact central view.

angles and correct depth—only the content "inside" a triangle can be subject to approximation.

It is also important to measure image quality of our approximate algorithm. For the Quake3 scene, the peak-signal-to-noise-ratio (PSNR) was about 40 dB for the entire animation. This is considered high even for still image compression. When the number of views increased, the PSNR remained relatively constant. This was not expected by us, since using linear interpolation for approximation gives worse results for a larger view divergence between an exact view and an approximated view. However, fewer pixels can be approximated when the view divergence is high between the approximated and exact view, and hence the quality increases.

The Soft Shadows and Ocean scene are harder cases for the approximation algorithm since they both contain view dependencies. The Soft Shadows scene contain view dependencies in the form of specular highlights, and the Ocean scene has nested view-dependent texture lookups (bump-mapped reflections and refractions). For those scenes, we must use a unified camera position for shading when the approximate algorithm is used. Otherwise visible seams may appear between approximated pixels and "exact" pixels. In our Soft Shadows scene, the unified camera position is hardly visible, and the PSNR is between 36 and 40 dB. In our Ocean scene, the differences are easily spotted when comparing to the exact solution, but the approximated version still looks good.As previously stated, we believe the application programmer should be given the appropriate control over when approximation should be used. Approximation could be turned off for surfaces with view-dependent shaders, or be controlled by a user-tweakable setting for quality or performance. In some applications, such as games, it may be more reasonable to use approximation. However, when a graphics hardware architecture is used for scientific computing, e.g. fluid dynamics on the GPU, the application programmer would probably want to turn off approximation, and only use our exact algorithm, which would still give a performance advantage.

Interestingly, for our approximate algorithm, we observed that compression of the color buffer works better than for our exact algorithm. On average, the compression ratio improved by 5–10%, most likely because of the slight low-pass effect introduced by the approximation filter.

To summarize, our results show that our multi-view rasterization architecture gives substantial reductions in terms of total bandwidth usage. The texture bandwidth re-

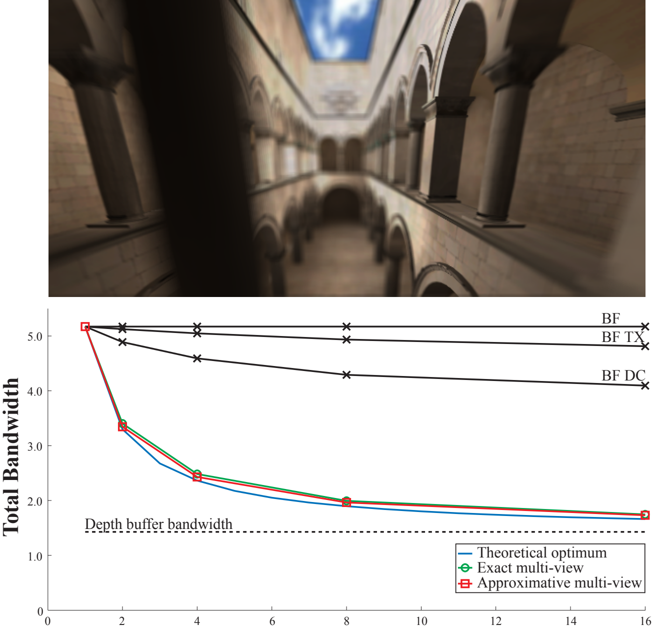


Figure 10: The Sponza atrium rendered with a depth of field (DOF) effect. The diagram shows bandwidth measurements in gigabytes per frame as a function of the number of views supported by the rasterizer. The curve named "Theoretical optimum" shows the best possible performance for our architecture at a given number of supported views. In short, we assume that the texture and color buffer bandwidth are zero for all views but one during each render pass. The BF DC architecture has been included because it performs better than the BF TX architecure in this particular benchmark. This is due to the two-pass nature of the DOF algorithm, and since the scene does not use any complicated shaders.

mains close to constant with an increasing number of views, and texture bandwidth reductions on the order of a magnitude are possible. Furthermore, our approximate technique can render high-quality images without executing the pixel shader for up to 95% of the fragments.

6.1 Accumulative Color Rendering

Figure 10 shows our final test scene, which is a *depth of field* (DOF) rendering of the Sponza atrium, using multi-texturing with decal textures and global illumination light maps. In contrast to the Ocean scene, which was designed as a nightmare scenario, this test hits the very sweet spot of our algorithm. Here, we benchmark the performance of accumulative color rendering (Section 4.4). The

tests were made using the same configurations of the rasterizer as in the previous benchmarks. However, this time we rendered a 16×16 samples DOF, where each configuration rendered the scene in as few passes as possible. For instance, a 16-view multi-view rasterization architecture would need to render 16 passes, while a 4-view system would need $4 \times 16 = 64$ passes. The BF algorithms always require all 256 passes. Our results in Figure 10 show a major reduction, not only in texture bandwidth, but also in color buffer bandwidth. This is to be expected, since all color buffer cache memory can be spent on a single color buffer, and since a projected primitive will be relatively coherent in screen space across all views when rasterizing to the same buffer. It is worth noting that the performance of our architecture is very close to its theoretical limit.

6.2 Small triangles

In this section, we will shed some light on how our architecture performs when rendering very small triangles. As we perform sorting on a per-tile level, the behavior of our algorithm will approach an architecture that renders a triangle to all views without any sorting at all (called "Tri-by-Tri" below) when rendering small triangles. In order to test sub-pixel triangle rendering, i.e., where the average number of pixels per triangle (ppt) is less than one, we rendered the Quake3 scene at low resolution using various four-view systems. In the following table, we present the texture bandwidth relative to the BF TX architecture:

	80×60, 0.8 ppt	640×480, 48 ppt
BF TX	100%	100%
Tri-by-Tri	31.3%	88.8%
Our	28.5%	27.3%

As can be seen, our algorithm handles small triangles very robustly. A view-independent texture access will be very coherent for a small triangle no matter what point in the triangle we choose to sample from, and drawing a triangle to all views will provide sufficient sorting of the texture accesses. Our opinion is that the Tri-by-Tri algorithm is much less robust since it fails horribly when the triangle area increase. Large triangles are still frequently used in games, architectural environments & particle systems, and it is therefore crucial to be able to handle them as well.

7 Discussion

Our multi-view rasterization architecture has been designed with the current technological development in mind: computing power grows at a much faster rate than memory bandwidth, and DRAM capacity is expected to double every year [22]. Hence our focus has been on reducing usage of memory bandwidth at a cost of

duplicating the depth and stencil buffers. The BF architecture would only need to duplicate the color buffer, if the depth and stencil buffers are cleared between the rendering of different views.

From a cost/performance perspective, a reasonable solution would be to implement a multi-view rasterizer with our approximate pixel shader technique and our tiled traversal for two, three, or four views. A multi-pass approach can be used when rendering to more views than supported by the architecture. For example, a system for four views can be used to render to 12 different views by rendering the scene three times.

Our traversal algorithm requires the ability to change the view which is currently being rasterized to. The same pixel shader program is used for all views, so switching views only amounts to changing the current active view. This can be done by enumerating all views, and changing the currently active index. This index points to view-dependent information, such as view parameters, and it also points to output buffers, for example. Therefore, we are confident that view switching can be efficiently implemented in hardware.

8 Conclusion and Future Work

We have presented a novel multi-view rasteriziation architecture, and shown that it is possible to exploit a substantial amount of the inherent coherency in this context. It is our hope that our work will renew interest in multi-view image generation research, a field that has received relatively little attention. Furthermore, it is our belief that our architecture may accelerate the acceptance of multi-view displays for real-time graphics.

With our current architecture, it is apparent that the bottlenecks from texturing and complex pixel shaders have moved to color and depth buffer bandwidth usage. For future work, we would therefore like to investigate whether these buffers can be compressed simultaneously for all views. This can lead to higher compression ratios. Some kind of differential encoding might be a fruitful avenue for this type of problem. Currently, we are making an attempt at augmenting our algorithms so that parallax in more directions can be obtained. This would allow us to have, for example, 2×2 viewpoints, and thus achieve both horizontal and vertical parallax. The parameter space (texture cache size, tile size, etc) involved in our architecture is large, and in future work, we want to explore various configurations in more detail.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, and LUAB. We would also like to thank Timo Aila, Petrik Clarberg, and Jacob Munkberg for proof reading and for their insightful comments.

Bibliography

[1] S. Adelson and L. Hodges. Stereoscopic Ray Tracing. *The Visual Computer*, 10(3):127–144, 1993.

[2] S. J. Adelson and C. D. Hansen. Fast Stereoscopic Images with Ray-Traced Volume Rendering. In *Symposium on Volume Visualization*, pages 3–9, 1994.

[3] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics*, 22(3):792–800, 2003.

[4] Kurt Akeley. The Elegance of Brute Force. In *Game Developers Conference*, 2003.

[5] Tomas Akenine-Möller and Timo Aila. Conservative Tiled Rasterization Using a Modififed Triangle Setup. *Journal of graphics tools*, 10(2):1–8, 2005.

[6] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics*, 22(3):801–808, 2003.

[7] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, August 1996.

[8] Daniel Cohen-Or, Yair Mann, and Shachar Fleishman. Deep Compression for Streaming Texture Intensive Animations. In *Proceedings of ACM SIGGRAPH 99*, pages 261–268, 1999.

[9] Michael Cox and Pat Hanrahan. Pixel Merging for Object-Parallel Rendering: a Distributed Snooping Algorithm. In *Symposium on Parallel Rendering*, pages 49–56, November 1993.

[10] Neil A. Dodgson. Autostereoscopic 3D Displays. *IEEE Computer*, 38(8):31–36, 2005.

[11] Ziyad S. Hakura and Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture*, pages 108–120, June 1997.

[12] Michael Halle. Multiple Viewpoint Rendering. In *Proceedings of ACM SIGGRAPH 98*, volume 32, pages 243–254, 1998.

[13] Taosong He and Arie Kaufman. Fast Stereo Volume Rendering. In *Proceedings of the 7th Conference on Visualization '96*, pages 49–56, 1996.

[14] Homan Igehy, Matthew Eldridge, and Pat Hanrahan. Parallel Texture Caching. In *Graphics hardware*, pages 95–106, 1999.

[15] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *Graphics Hardware*, pages 133–142, 1998.

[16] B. Javidi and Eds F. Okano. *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

[17] G. Knittel, A. Schilling, A. Kugler, and W. Strasser. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.

[18] Wojciech Matusik and Hanspeter Pfister. 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. *ACM Transactions on Graphics*, 23(3):814–824, 2004.

[19] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[20] Steven Molnar, Bengt-Olaf Schneider, John Montrym, James Van Dyke, and Stephen Lew. System and Method for Real-Time Compression of Pixel Colors. US Patent 6,825,847, 2004.

[21] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM Press, August 2000.

[22] John Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley Professional, 2005.

[23] Fabio Pellacini, Kiril Vidimče, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: A Hybrid Hardware-Accelerated Relighting Engine for Computer Cinematography. *ACM Transactions on Graphics*, 24(3):464–470, 2005.

[24] Kurt Pelzer. Advanced Water Effects. In *Shader X2*, pages 207–225. Wordware Publishing Inc., 2004.

[25] Dennis R. Proffitt and Mary Kaiser. Hi-Lo Stereo Fusion. In *ACM SIGGRAPH 96 Visual Proceedings*, page 146, 1996.

[26] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 283–291, 1987.

[27] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. 1992.

[28] Oles Shishkovtsov. Deferred Shading in S.T.A.L.K.E.R. In *GPU Gems 2*, pages 143–166. Addison-Wesley Professional, 2005.

[29] J. Stewart, E. P. Bennett, and L. McMillan. PixelView: A View-Independent Graphics Rendering Architecture. In *Graphics Hardware*, pages 75–84, 2004.

[30] Stanislav L. Stoev, Tobias Hüttner, and Wolfgang Strasser. Accelerated Rendering in Stereo-Based Projections. In *Third International Conference on Collaborative Virtual Environments*, pages 213–214, 2000.

[31] Yury Uralsky. Efficient Soft-Edged Shadows Using Pixel Shader Branching. In *GPU Gems 2*, pages 269–282. Addison-Wesley Professional, 2005.

[32] Cees van Berkel. Philips Multiview 3D Display Solutions. 3D Consortium, http://www.3dc.gr.jp/english/domestic_rep/040617a.php, 2004.

[33] Lance Williams. Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, pages 1–11, July 1983.

Paper IV

High Dynamic Range Texture Compression for Graphics Hardware

Jacob Munkberg Petrik Clarberg Jon Hasselgren Tomas Akenine-Möller

Lund University

{jacob|petrik|jon|tam}@cs.lth.se

ABSTRACT

In this paper, we break new ground by presenting algorithms for fixed-rate compression of high dynamic range textures at low bit rates. First, the S3TC low dynamic range texture compression scheme is extended in order to enable compression of HDR data. Second, we introduce a novel robust algorithm that offers superior image quality. Our algorithm can be efficiently implemented in hardware, and supports textures with a dynamic range of over 10^9:1. At a fixed rate of 8 bits per pixel, we obtain results virtually indistinguishable from uncompressed HDR textures at 48 bits per pixel. Our research can have a big impact on graphics hardware and real-time rendering, since HDR texturing suddenly becomes affordable.

ACM transactions on graphics 25(3):698–706, 2006.

1 Introduction

The use of high dynamic range (HDR) images in rendering [6, 7, 18, 27] has changed computer graphics forever. Prior to this, only low dynamic range (LDR) images were used, usually storing 8 bits per color component, i.e., 24 bits per pixel (bpp) for RGB. Such images can only represent a limited amount of the information present in real scenes, where luminance values spanning many orders of magnitude are common. To accurately represent the full dynamic range of an HDR image, each color component can be stored as a 16-bit floating-point number. In this case, an uncompressed HDR RGB image needs 48 bpp.

In 2001, HDR images were first used in real-time rendering [4], and over the past years, we have observed a rapidly increasing use of HDR images in this context. Game developers have embraced this relatively new technique, and several recent games use HDR images as textures. Examples include Unreal Engine 3, Far Cry, Project Gotham Racing 3, and Half-Life 2: Lost Coast.

The disadvantage of using HDR textures in real-time graphics is that the texture bandwidth usage increases dramatically, which can easily limit performance. With anisotropic filtering or complex pixel shaders, it can become even worse. A common approach to reduce the problem is *texture compression*, introduced in 1996 [1, 11, 23]. By storing textures in compressed form in external memory, and sending compressed data over the bus, the bandwidth is significantly reduced. The data is decompressed in real time using special-purpose hardware when it is accessed by the pixel shader. Several formats use as little as 4 bpp. Compared to 24 bpp RGB, such techniques can potentially reduce the texture bandwidth to only 16.7% of the original.

Texels in textures can be accessed in any order during rasterization. A fixed-rate texture compression (TC) system is desirable, as it allows random addressing without complex lookup mechanisms. Hence, JPEG and similar algorithms do not immediately qualify as reasonable alternatives for TC, since they use an adaptive bit rate over the image. The fixed bit rate also implies that all realistic TC algorithms are lossy. Other characteristics of a TC system are that the decompression should preferably be fast and relatively inexpensive to implement in hardware. However, we expect that increasingly complex decompression schemes can be accepted by the graphics hardware industry, since the available bandwidth grows at a much slower pace than the computing power [15]. A difference between LDR and HDR TC is that for HDR images, we do not know in advance what range of luminance values will be displayed. Hence, the image quality must remain high over a much larger range of luminance.

We present novel HDR TC schemes, which are inexpensive to implement in hardware. Our algorithms compresses tiles of 4×4 pixels to only 8 bpp. The compressed images are of very high quality over the entire range of input values, and are essentially indistinguishable from uncompressed 48 bpp data. An example of a compressed image is shown in Figure 1.

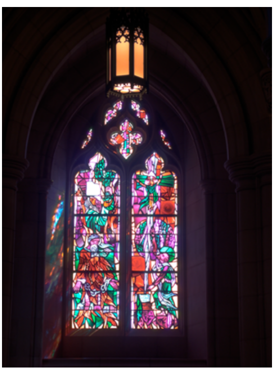
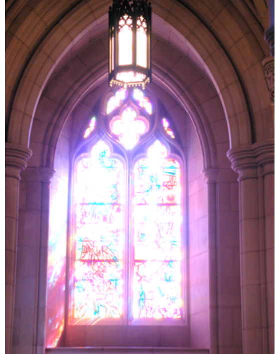



Figure 1: Example of a high dynamic range image, here shown at three different exposures, compressed with our algorithm to a fixed rate of 8 bits per pixel. Our algorithm gives excellent image quality over a large dynamic range, and is fast to decompress in hardware.

2 Related Work

Here, we first present research in LDR texture compression (TC) for graphics hardware that is relevant to our work. For a more complete overview, consult Fenney's paper [9]. Second, some attention is given to existing HDR compression systems.

LDR Texture Compression Vector quantization (VQ) techniques have been used by Beers et al. [1] for TC. They presented compression ratios as low as one or two bpp. However, VQ requires an access in a look-up table, which is not desirable in a graphics hardware pipeline. The S3TC texture compression scheme [10] has become a de facto standard for real-time rendering. Since we build upon this scheme, it is described in more detail in Section 4.

Fenney [9] presents a system where two low-resolution images are stored for each tile. During decompression, these images are bilinearly magnified and the color of a pixel is obtained as a linear blend between the magnified images. Another TC scheme assumes that the whole mipmap pyramid is to be compressed [16]. Box filtering is used, and the luminance of a 4×4 tile is decomposed using Haar wavelets. The chrominance is subsampled, and then compressed. The compression ratio is 4.6 bpp.

In a TC system called iPACKMAN [22], 4×4 tiles of pixels are used, and each tile encodes two base colors in RGB444 and a choice of modifier values. The color of a pixel is determined from one of the base colors by adding a modifier value.

HDR Image and Video Compression To store an HDR image in the RGBE format, Ward [24] uses 32 bits per pixel, where 24 bits are used for RGB, and

the remaining 8 bits for an exponent, E, shared by all three color components. A straightforward extension would be to compress the RGB channels using S3TC to 4 bits per pixel, and store the exponent uncompressed as a separate 8 bpp texture, resulting in a 12 bits per pixel format supported by current graphics hardware. However, RGBE has a dynamic range of 76 orders of magnitude and is not a compact representation of HDR data known to reside in a limited range. Furthermore, as both the RGB and the exponent channel contain luminance information, chrominance and luminance transitions are not separated, and artifacts similar to the ones in Figure 13 are likely to occur. Ward also developed the LogLuv format [28], where the RGB input is separated into luminance and chrominance information. The logarithm of the luminance is stored in 16 bits, while the chrominance is stored in another 16 bits, resulting in 32 bits per pixel. A variant using 24 bpp was also presented.

Ward and Simmons [26] use the possibility of storing an extra 64 kB in the JPEG image format. The file contains a tone mapped image, which can be decompressed using standard JPEG decompressors. In the 64 kB of data, a ratio image of the luminance in the original and the tone mapped image is stored. A loader incapable of handling the format will display a tone mapped image, while capable loaders will obtain the full dynamic range. Xu et al. [29] use the wavelet transform and adaptive arithmetic coding of JPEG 2000 to compress HDR images. They first compute the logarithm of the RGB values, and then use existing JPEG 2000 algorithms to encode the image. Impressive compression ratios and a high quality is obtained. Mantiuk et al. [14] present an algorithm for compression of HDR video. They quantize the luminance using a non-linear function in order to distribute the error according to the luminance response curve of the human visual system. Then, they use an MPEG4-based coder, which is augmented to suppress errors around sharp edges. These three algorithms use adaptive bit rates, and thus cannot provide random access easily.

There is a wide range of tone mapping operators (cf. [18]), which perform a type of compression. However, the dynamic range is irretrievably lost in the HDR to LDR conversion, and these algorithms are therefore not directly applicable for TC. Li et al. [12] developed a technique called companding, where a tone mapped image can be reconstructed back into an HDR image with high quality. This technique is not suitable for TC since it applies a global transform to the entire image, which makes random access extremely slow, if at all feasible. Still, inspiration can be obtained from these sources.

In the spirit of Torborg and Kajiya [23], we implemented a fixed-rate HDR DCT encoder, but on hardware-friendly 4×4 tiles at 8 bits per pixel. The resulting images showed severe ringing artifacts near sharp luminance edges and moderate error values. The decompressor is also substantially more complex than the algorithms we present below.

3 Color Spaces and Error Measures

In this section, we discuss different color spaces, and develop a small variation of an existing color space, which is advantageous in terms of hardware decompression and image quality. Furthermore, we discuss error metrics in Section 3.2, where we also suggest a new simple error metric.

3.1 Color Spaces

The main difficulty in compressing HDR textures is that the dynamic range of the color components can be very large. In natural images, a dynamic range of 100,000:1 is not uncommon, i.e., a factor 10^5 difference between the brightest and the darkest pixels. A 24-bit RGB image, on the other hand, has a maximum range of 255:1. Our goal is to support about the same dynamic range as the OpenEXR format [2], which is based on the hardware-supported *half* data type, i.e., 16-bit floating-point numbers. The range of representable numbers with full precision is roughly $6.1 \cdot 10^{-5}$ to $6.5 \cdot 10^4$, giving a dynamic range of 10^9:1. We aim to support this range directly in our format, as a texture may undergo complex image operations where lower precision is not sufficient. Furthermore, the dynamic range of the test images used in this paper is between $10^{2.6}$ and $10^{7.3}$. An alternative is to use a tighter range and a per-texture scaling factor. This is a trivial extension, which would increase the quality for images with lower dynamic ranges. However, this requires global per-texture data, which we have opted to avoid. We leave this for future work.

To get consistently good quality over the large range, we need a color space that provides a more compact representation of HDR data than the standard RGB space. Taking the logarithm of the RGB values gives a nearly constant relative error over the entire range of exposures [25]. Assume we want to encode a range of 10^9:1 in 1% steps. In this $\log[RGB]$ space, we would need $k = 2083$ steps, given by $1.01^k = 10^9$, or roughly 11 bits precision per color channel. Because of the high correlation between the RGB color components [19], we need to store all three with high accuracy. As we will see in Section 4, we found it difficult to reach the desired image quality and robustness when using the $\log[RGB]$ space.

In image and video compression, it is common to decorrelate the color channels by transforming the RGB values into a luminance/chrominance-based color space [17]. The motivation is that the luminance is perceptually more important than the chrominance, or *chroma* for short, and more effort can be spent on encoding the luminance accurately. Similar techniques have been proposed for HDR image compression. For example, the LogLuv encoding [28] stores a log representation of the luminance and CIE (u', v') chrominance. Xu et al. [29] apply the same transform as in JPEG, which is designed for LDR data, but on the logarithm of the RGB components. The OpenEXR format supports a simple form of compression

based on a luminance/chroma space with the luminance computed as:

$$Y = w_r R + w_g G + w_b B, \tag{1}$$

and two chroma channels, U and V, defined as:

$$U = \frac{R-Y}{Y}, \qquad V = \frac{B-Y}{Y}. \tag{2}$$

Lossy compression is obtained by subsampling the chroma components by a factor two horizontally and vertically.

Inspired by previous work, we define a simple color space denoted $\log Y\bar{u}\bar{v}$, based on log-luminance and two chroma components. Given the luminance Y computed using Equation 1, the transform from RGB is given by:

$$(\bar{Y}, \bar{u}, \bar{v}) = \left(\log_2 Y, w_b \frac{B}{Y}, w_r \frac{R}{Y}\right). \tag{3}$$

We use the Rec. 601 [17] weights (0.299, 0.587, 0.114) for w_r, w_g and w_b. With non-zero, positive input RGB values in the range $[2^{-16}, 2^{16}]$, the log-luminance $\bar{Y}$ is in the range $[-16, 16]$, and the chroma components are in the range $[0, 1]$ with $\bar{u} + \bar{v} \leq 1$.

In our color space, the HDR luminance information is concentrated to the $\bar{Y}$ component, which needs to be accurately represented, while the $(\bar{u}, \bar{v})$ components only contain chrominance information normalized for luminance. These can be represented with significantly less accuracy.

3.2 Error Measures

In order to evaluate the performance of various compression algorithms, we need an image quality metric that provides a meaningful indication of image fidelity. For LDR images, a vast amount of research in such metrics has been conducted [3]. Perceptually-based metrics have been developed, which attempt to predict the observed image quality by modeling the response of the human visual system (HVS). The prime example is the *visible differences predictor* (VDP) introduced by Daly [5].

Error measures for HDR images are not as thoroughly researched, and there is no well-established metric. The image must be tone-mapped before VDP or any other standard image quality metric, designed for LDR data, can be applied. The choice of tone mapping operator will bias the result, which is unfortunate. In our application, another difficulty is that we do not know how the HDR textures will be used or what the display conditions will be like. For example, a texture in a 3D engine can undergo a number of complex operations, such as lighting, blending and multi-texturing, which change its appearance.

Xu et al. [29] compute the *root-mean-square error* (RMSE) of the compressed image in the $\log[RGB]$ color space. Their motivation is that the logarithm is a

conservative approximation of the HVS luminance response curve. However, we argue that this error measure can be misleading in terms of visual quality. The reason is that an error in a small component tends to over-amplify the error measure even if the small component's contribution to the final pixel color is small. For example, consider a mostly red pixel, $\mathbf{r} = (1000, 1, 1)$, which is compressed to $\mathbf{r}^* = (1000, 1, 8)$. The $\log[RGB]$ RMSE is then $\log_2 8 - \log_2 1 = 3$, but the log-luminance RMSE, to which the HVS is most sensitive, is only 0.004. Still, we include the $\log[RGB]$ RMSE error because it reflects the relative, per-component error of the compressed image. It is therefore well suited to describe the expected error of the aforementioned image operations: blending, lighting etc.

To account for all normal viewing conditions, we propose a simple error metric, which we call *multi-exposure peak-signal-to-noise ratio*, or *mPSNR* for short. The HDR image is tone mapped to a number of different exposures, uniformly distributed over the dynamic range of the image. See Figure 2 for an example. For each exposure, we compute the *mean square error* (MSE) on the resulting LDR image, and then compute the peak-signal-to-noise ratio (PSNR) using the mean of all MSEs. As a tone mapping operator, we use a simple gamma-adjustment after exposure compensation. The tone mapped LDR image, $T(I)$, of the HDR image, I, is given by:

$$T(I) = \left[255\,(2^c I)^{1/\gamma}\right]_0^{255}, \tag{4}$$

where c is the exposure compensation in f-stops, γ is the display gamma, and $[\cdot]_0^{255}$ indicates clamping to the integer interval $[0, 255]$. The mean square error over all exposures and over all pixels is computed as:

$$\text{MSE} = \frac{1}{n \times w \times h} \sum_c \sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2\right), \tag{5}$$

where n is the number of exposures, c, and $w \times h$ is the image resolution. The error in the red component (similar for green and blue) at pixel (x, y) is $\Delta R_{xy} = T_R(I) - T_R(C)$, where I is the original image, and C is the compressed image. Finally, mPSNR is computed as:

$$\text{mPSNR} = 10 \log_{10}\left(\frac{3 \times 255^2}{\text{MSE}}\right). \tag{6}$$

The obtained mPSNR over all exposures gives us a prediction of the error in the compressed HDR image. The PSNR measure has traditionally been popular for evaluating the performance of TC schemes, and although no other HDR texture compression techniques exist, the use of mPSNR makes our results more easily interpreted.

Recently, Mantiuk et al. [13] have presented a number of modifications to the visual differences predictor, making it possible to predict the perceived differences over the entire dynamic range in real scenes. This novel HDR VDP takes into account a number of complex effects such as the non-linear response and local

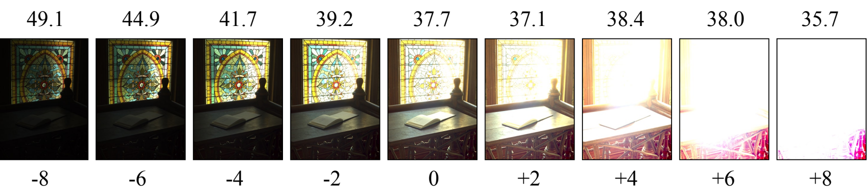


Figure 2: In our *multi-exposure PSNR* error measure, the image is tone mapped to a number of different exposures to account for all normal viewing conditions, and the PSNR is computed from the average MSE. In this case, the mPSNR is 39 dB. The top row shows the standard PSNRs, and the bottom row shows the exposure compensation, c.

adaptation of the HVS. However, their current implementation only works on the luminance, and does not take the chroma error into account.

As there is no established standard for evaluating HDR image quality, we have chosen to use a variety of error metrics. We present results for our algorithm using the mPSNR, the $\log[RGB]$ root-mean-square error, and the HDR VDP.

4 HDR S3 Texture Compression

The S3 texture compression (S3TC) method [10] is probably the most popular scheme for compressing LDR textures. It is used in DirectX and there are extensions for it in OpenGL as well. S3TC uses tiles of 4×4 pixels that are compressed to 64 bits, giving a compression rate of 4 bpp. Two base colors are stored in 16 bits (RGB565) each, and every pixel stores a two-bit index into a local color map consisting of the two base colors and two additional colors in between the base colors. This means that all colors lie on a straight line in RGB space.

A natural suggestion for an HDR TC scheme is to adapt the existing S3TC algorithm to handle HDR data. Due to the increased amount of information, we double the rate to 8 bpp. We also apply the following changes. First, we transform the linear RGB data into a more compact color space. Second, we raise the quantization resolution and the number of per-pixel index bits. In graphics hardware, the memory is accessed in bursts of 2^n bits, e.g., 256 bits. To simplify addressing, it is desirable to fetch 2^m pixels per burst, which gives 2^{n-m} bits per pixel (e.g., 4,8,16,...). Hence, keeping a tile size of 4×4 pixels is a reasonable choice, as one tile fits nicely into 128 bits on an 8 bpp budget. In addition, a small tile size limits the variance across the tile and keeps the complexity of the decompressor low.

The input data consists of three floating-point values per pixel. Performing the compression directly in linear RGB space, or in linear YUV space, produces extremely poor results. This is due to the large dynamic range. Better results are obtained in the $\log[RGB]$ and the $\log Y\bar{u}\bar{v}$ color spaces (Section 3.1). Our tests

show that 4-bit per-pixel indices are needed to accurately capture the luminance variations. We call the resulting algorithms *S3TC RGB* (using $\log[RGB]$), and *S3TC YUV* (using $\log Y\bar{u}\bar{v}$). The following bit allocations performed best in our tests:

Color space	Base colors	Per-pixel indices
$\log[RGB]$	$2 \times (11+11+10) = 64$	$16 \times 4 = 64$
$\log Y\bar{u}\bar{v}$	$2 \times (12+10+10) = 64$	$16 \times 4 = 64$

Even though these S3TC-based approaches produce usable results in some cases, they lack the robustness needed for a general HDR TC format. Some of the shortcomings of S3TC RGB and S3TC YUV are clearly illustrated in Figure 13. As can be seen in the enlarged images, both algorithms produce serious block artifacts, and blurring of some edges. This tends to happen where there is a chroma and a luminance transition in the same tile, and there is little or no correlation between these. The reason is that all colors must be located on a straight line in the respective 3D color space for the algorithms to perform well. In Figure 13, we also show the results of our new HDR texture compression scheme. As can be seen, the image quality is much higher. More importantly, our algorithm is more robust, and rarely generates tiles of poor quality.

5 New HDR Texture Compression Scheme

In the previous section, we have seen that building a per-tile color map from a straight line in some 3D color space does not produce acceptable results for S3TC-based algorithms. To deal with the artifacts, we decouple the luminance from the chrominance and encode them separately in the $\log Y\bar{u}\bar{v}$ space defined in Equation 3. By doing this, difficult tiles can be handled much better. In the following, we describe how the luminance and chrominance can be accurately represented on an 8 bpp budget, i.e., 128 bits per tile.

5.1 Luminance Encoding

In the $\log Y\bar{u}\bar{v}$ color space, the log-luminance values $\bar{Y}$ are in the range $[-16, 16]$. First, we find the minimum and maximum values, $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$, in a tile. Inspired by S3TC, we then quantize these linearly and store per-pixel indices indicating which luminance step between $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$ that is to be used for each pixel.

As we have seen, we need approximately 16 steps, i.e., 4-bit per-pixel indices, for an accurate representation of HDR luminance data. If we use 12-bit quantization of $\bar{Y}_{\min}$ and $\bar{Y}_{\max}$ as in S3TC YUV, a total of $2 \times 12 + 16 \times 4 = 88$ bits are consumed, and only 40 bits are left for the chroma encoding. This is not enough. By searching in a range around the quantized base values, it is very often possible to find a combination that gives a significantly reduced error. Thus, we manage to encode

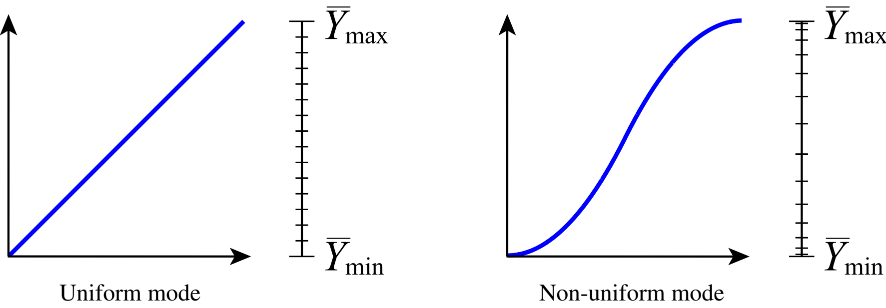


Figure 3: The two luminance quantization modes. The non-uniform mode is used for better handling tiles with sharp luminance transitions, such as edges.

the base luminances with only 8 bits each without any noticeable artifacts, even on slow gradients.

Another approach would be to use spatial subsampling of the luminance. Recent work on HDR displays by Seetzen et al. [20, 21] suggests that the human eye's spatial HDR resolution is lower than its LDR resolution. However, the techniques developed for direct display of HDR images are not directly applicable to our problem as they require high-precision per-pixel LDR data to modulate the subsampled HDR luminance. We have tried various hierarchical schemes, but the low bit budget made it difficult to obtain the required per-pixel precision. Second, our compression scheme is designed for textures, hence we cannot make any assumptions on how the images will be displayed on screen. The quality should be reasonable even for close-up zooms. Therefore, we opted for the straightforward solution of storing per-pixel HDR luminance.

The most difficult tiles contain sharp edges, e.g., the edge around the sun in an outdoor photograph. Such tiles can have a very large dynamic range, but at the same time, both the darker and the brighter areas must be represented accurately. For this, a uniform quantization between the min/max luminances is not ideal. To better handle such tiles, we add a mode using non-uniform steps between the $\bar{Y}_{\text{min}}$ and $\bar{Y}_{\text{max}}$ values. Smaller quantization steps are used near the base luminances, and larger steps in the middle. Thus, two different luminance ranges that are far apart can be accurately represented in the same tile. In our test images (Figure 10), the non-uniform mode is used for 11% of the tiles, and for these tiles, the log-luminance RMSE is decreased by 12.0% on average. The two quantization modes are illustrated in Figure 3.

To choose between the two modes, we use the mutual ordering[1] of $\bar{Y}_{\text{min}}$ and $\bar{Y}_{\text{max}}$. In decoding, if $\bar{Y}_{\text{min}} \leq \bar{Y}_{\text{max}}$, then the uniform mode is used. Otherwise, $\bar{Y}_{\text{min}}$ and $\bar{Y}_{\text{max}}$ are reversed, and we use the non-uniform mode. Hence, no additional mode bit is necessary, and the luminance encoding uses a total of $2 \times 8 + 16 \times 4 = 80$ bits,

[1] Similar ordering techniques are used in the S3TC LDR texture compression format.

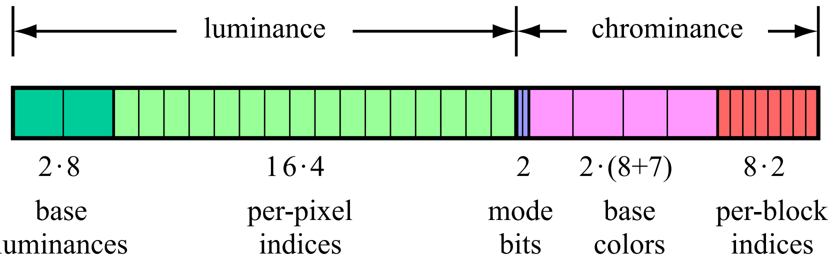


Figure 4: The bit allocation we use for encoding the luminance and chrominance of a 4×4 tile in 128 bits (8 bpp).

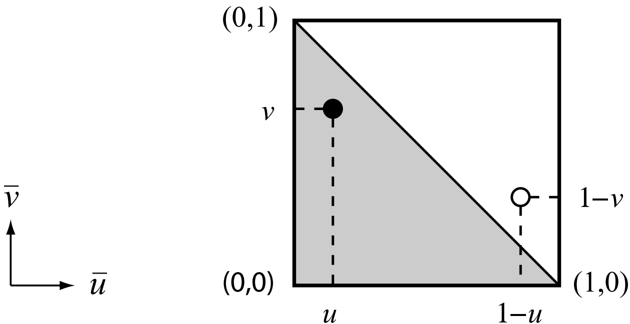


Figure 5: Illustration of the *flip trick*. By mirroring the coordinates for a base color, we exploit our triangular chrominance space in order to obtain another bit.

leaving 48 bits for the chrominance. The bit allocation is illustrated in Figure 4.

5.2 Chrominance Line

Our first approach to chrominance compression on a 48-bit budget, is to use a line in the $(\bar{u}, \bar{v})$ chroma plane. Similar to the luminance encoding, each tile stores a number of indices to points uniformly distributed along the line.

In order to fit the chroma line in only 48 bits, we sub-sample the chrominance by a factor two, either horizontally or vertically, similar to what is used in DV encoding [17]. The sub-sampling mode that minimizes the error is chosen. To simplify the following description, we define a *block* as being either 1×2 (horizontal) or 2×1 (vertical) sub-sampled pixels. The start and end points of the chroma line, each with 2×8 bits resolution, and eight 2-bit per-block indices, gives a total cost of $4 \times 8 + 8 \times 2 = 48$ bits per tile.

In our color space, the normalized chroma points $(\bar{u}_i, \bar{v}_i)$, $i \in [0, 7]$, are always

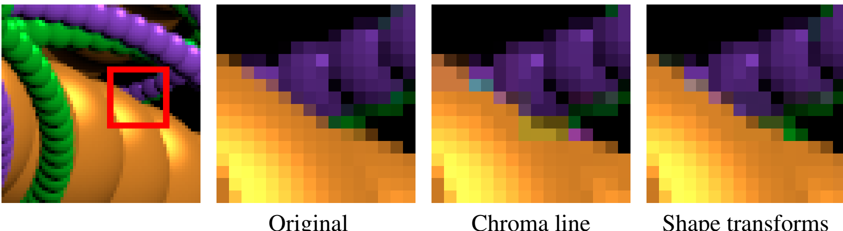


Figure 6: A region where a line in chroma space is not sufficient for capturing the complex color. With shape transforms, we get a much closer resemblance to the true color, although minor imperfections exist due to the sub-sampling.

located in the lower triangle of the chroma plane. If we restrict the encoding of the end points of the line to this area, we can get two extra bits by a *flip trick* described in Figure 5. If a chrominance value $\mathbf{c} = (\bar{u}_i, \bar{v}_i)$ is in the upper (invalid) triangle, this indicates that the extra bit is set to one, and the true chroma value is given by $\mathbf{c}' = (1 - \bar{u}_i, 1 - \bar{v}_i)$, otherwise the bit is set to zero. We can use one of these extra bits to indicate whether to use horizontal or vertical sub-sampling. The other bit is left unused.

5.3 Chroma Shape Transforms

A line in chroma space can only represent two chrominances and the gradient between them. This approximation fails for tiles with complex chroma variations or sharp color edges. Figure 6 shows an example of such a case. One solution would be to encode $\bar{u}$ and $\bar{v}$ separately, but this does not easily fit in 48 bits.

To better handle difficult tiles, we introduce *shape transforms*; a set of shapes, each with four discrete points, designed to capture chroma information. In the classic game *Tetris*, the optimal placement of a shape in a grid is found by rotating and translating it in 2D. The same idea is applied to the chrominances of a tile. We allow arbitrary rotation, translation, and uniform scaling of our shapes to make them match the eight sub-sampled chrominance values of a tile as closely as possible. The transformation of the shape can be retrieved by storing only two points.

During compression, we select the shape that most closely covers the chroma information of the tile, and store its index along with two base chrominances (start & end) and per-block indices. This allows each block in the tile to select one of the discrete positions of the shape. The shape fitting is illustrated in Figure 7 on one of the difficult tiles from the image in Figure 6. Using one of the transformed shapes, we get much closer to the actual chroma information.

The space of possible shapes is very large. In order to find a selection of shapes that perform well, we have analyzed the chrominance content in a set of images (a total of 500,000 tiles), different from our test images. First, clustering was

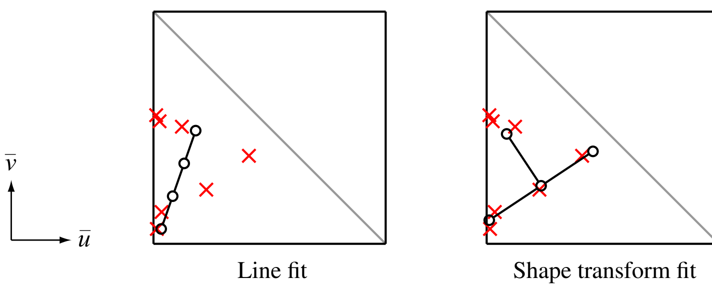
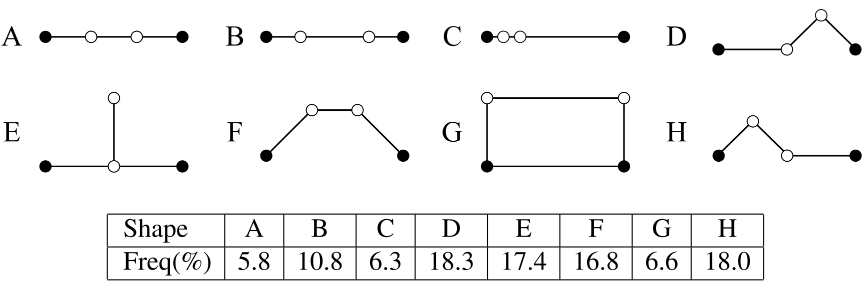


Figure 7: In tiles with difficult chrominance, such as in this example taken from Figure 6, a line in chroma space has difficulties representing the $(\bar{u}, \bar{v})$ points accurately (left). Our algorithm based on *shape transforms* generates superior chroma representations, as it is often possible to find a shape that closely matches the chrominance points (right).

done to reduce the chroma values of a tile to four chroma points. Second, we normalized the chroma points for scale and rotation, and then iteratively merged the two closest candidates until the desired number of shapes remained. Figure 8 shows our selection of shapes after a slight manual adjustment of the positions. See Appendix B for the exact coordinates. Shapes A through C handle simple color gradients, while D–H are optimized for tiles with complex chrominance. Also, by including the uniform line (shape A), we make the chrominance line algorithm (Section 5.2) a subset of the shape transforms approach. Note that the set of shapes is fixed, so no global per-texture data is needed.

Compared to the chrominance line, shape transforms need three bits per tile to indicate which of the eight shapes to use. We exploit the unused bit from the chrominance line, and the other two extra bits are taken from the quantization



Shape	A	B	C	D	E	F	G	H
Freq(%)	5.8	10.8	6.3	18.3	17.4	16.8	6.6	18.0

Figure 8: The set of shapes we use for the *shape transform* algorithm and their frequencies for our test images. The points corresponding to base colors are illustrated with solid black circles.

of the start and end points. We lower the $(\bar{u},\bar{v})$ quantization from $8+8$ bits to $8+7$ bits. Recall from Section 3.1, that in the $\log Y\bar{u}\bar{v}$ to RGB transform, the R and B components are given as $R=\bar{v}Y/w_r$ and $B=\bar{u}Y/w_b$, with $w_r=0.299$ and $w_b=0.114$. As w_b is about three times smaller than w_r, it makes the reconstructed color more sensitive for quantization errors in the $\bar{u}$ component, and therefore more bits are spent there.

With these modifications, shape transforms with eight shapes fit in precisely 48 bits. The total bit allocation for luminance and chrominance is illustrated in Figure 4. We evaluated both the chrominance line and the shape transform approach, combined with the luminance encoding of Section 5.1, on our test images. On average, the mPSNR was about 0.5 dB higher using shape transforms, and the resulting images are more visually pleasing, especially in areas with difficult chrominance.

The shape fitting step of our new algorithm is implemented by first clustering the eight sub-sampled input points to four groups. Then we apply Procrustes analysis [8] to find the best orientation of each shape to the clustered data set, and the shape with the lowest error is chosen. This is an approximate, but robust and efficient approach. We achieved somewhat lower errors by using an extensive search for the optimal shape transform, but this is computationally much more expensive.

6 Hardware Decompressor

In this section, we present a decompressor unit for hardware implementation of our algorithm. We first describe how the chrominance, $(\bar{u},\bar{v})$, is decompressed for a single pixel. In the second part of this section, we describe the color space transformation back to RGB-space. A presentation of log-luminance decompression is omitted, since it is very similar to S3TC LDR decompression. The differences are that the log-luminance is one-dimensional (instead of three-dimensional), and more bits are used for the quantized base values and per-pixel indices. In addition, we also have the non-uniform quantization, but this only amounts to using different constants in the interpolation.

The decompression of chrominance is more complex than for luminance, and in Figure 9, one possible implementation is shown. To use shape transforms, a coordinate frame must be derived from the chroma endpoints, $(\bar{u}_0,\bar{v}_0)$ and $(\bar{u}_1,\bar{v}_1)$, of the shape. In our case, the first axis is defined by $\mathbf{d}=(d_u,d_v)=(\bar{u}_1-\bar{u}_0,\bar{v}_1-\bar{v}_0)$. The other axis is $\mathbf{d}^\perp=(-d_v,d_u)$, which is orthogonal to $\mathbf{d}$. The coordinates of a point in a shape are described by two values α and β, which are both fixed-point numbers in the interval $[0,1]$, using only five bits each. The chrominance of a point, with coordinates α and β, is derived as:

$$\begin{pmatrix}\bar{u}\\ \bar{v}\end{pmatrix}=\alpha\mathbf{d}+\beta\mathbf{d}^\perp+\begin{pmatrix}\bar{u}_0\\ \bar{v}_0\end{pmatrix}=\begin{pmatrix}\alpha d_u-\beta d_v+\bar{u}_0\\ \beta d_u+\alpha d_v+\bar{v}_0\end{pmatrix}. \tag{7}$$

The diagram in Figure 9 implements the equation above. As can be seen, the hardware is relatively simple. The α and β constants units contain only the constants

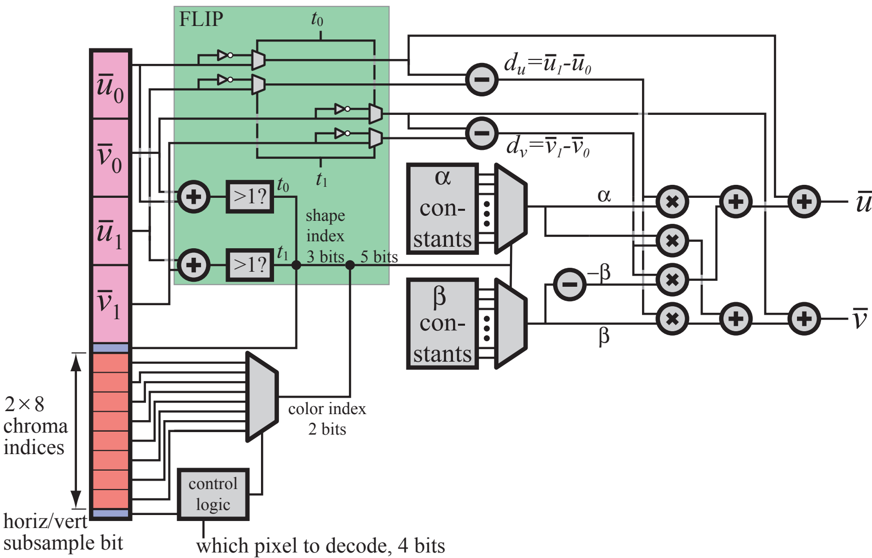


Figure 9: Decompression of chrominance, $(\bar{u}, \bar{v})$, for a single pixel. The indata is the 48 bits for chrominance (left part of the figure), and a 4-bit value indicating which pixel in a tile to decode. The outdata is $(\bar{u}, \bar{v})$ for one pixel. The green box contains the logic to implement the *flip trick*, where inverters have been used to compute the $1-x$ terms.

which define the different chrominance shapes. Only five bits per value are used, and hence the four multipliers compute the product of a 5-bit value and an 8 or 9-bit value. Note that $(\bar{u}, \bar{v})$ are represented using fixed-points numbers, so integer arithmetic can be used for the entire chroma decompressor.

At this point, we assume that $\bar{Y}$, $\bar{u}$ and $\bar{v}$ have been computed for a certain pixel in a tile. Next, we describe our transform back to linear RGB space. This is done by first computing the floating-point luminance: $Y = 2^{\bar{Y}}$. After that, the red, green, and blue components can be derived from Equation 3 as:

$$(R, G, B) = \left(\frac{1}{w_r}\bar{v}Y, \frac{1}{w_g}(1-\bar{u}-\bar{v})Y, \frac{1}{w_b}\bar{u}Y \right). \tag{8}$$

Since the weights, $(w_r, w_g, w_b) = (0.299, 0.587, 0.114)$, are constant, their reciprocals can be precomputed. We propose using the hardware-friendly constants:

$$(1/w'_r, 1/w'_g, 1/w'_b) = \frac{1}{16}(54, 27, 144). \tag{9}$$

This corresponds to

$$(w'_r, w'_g, w'_b) \approx (0.296, 0.593, 0.111). \tag{10}$$

Using our alternative weights makes the multiplications much simpler. This comes with a non-noticeable degradation in image quality.

In summary, our color space transform involves one power function, two fixed-point additions, three fixed-point multiplications, and three floating-point times fixed-point multiplications. The majority of color space transforms include at least a 3×3 matrix/vector multiplication, and in the case of HDR data, we have seen that between 1–3 power functions are also used. Ward's LogLuv involves even more operations. Our transform involves significantly fewer arithmetic operations compared to other color space transforms, and this is a major advantage of our decompressor.

The implementation shown above can be considered quite inexpensive, at least when compared to using other color spaces. Still, when compared to popular LDR TC schemes [10, 22], our decompressor is rather complex. However, we have attempted to make it simpler by designing a hardware-friendly color space, avoided using too many complex arithmetic operations, and simplified constants. In addition, we believe that in the near future, graphics hardware designers will have to look into more complex circuitry in order to reduce bandwidth, and the technological trend shows that this is the way to go [15].

7 Results

To evaluate the algorithms, we use a collection of both synthetic images and real photographs, as shown in Figure 10. Many of these are well-known and widely used in HDR research. In the following, we refer to *our algorithm* as the combination of our luminance encoding (Section 5.1) and shape transforms (Section 5.3). The results using mPSNR, $\log[RGB]$ RMSE, and HDR VDP are presented in Figure 11. After that follows visual comparisons.

In terms of the mPSNR measure, our algorithm performs substantially better over the entire set of images, with an average improvement of about 3 dB over the S3TC-based approaches. The mPSNR measure simulates the most common use of an HDR image in a real-time application, where the image is tone mapped and displayed under various exposures, either as a decal or as an environment map. The range of exposures used in mPSNR is automatically determined by computing the min and max luminance over each image, and mapping this range to exposures that give a nearly black, and a nearly white LDR image respectively. See Appendix A for the exact numbers.

The $\log[RGB]$ RMSE measures the relative error per component over the entire dynamic range of the image. In this metric, the differences between the algorithms are not as obvious. However, our algorithm gives slightly lower error on average.

The HDR VDP chart in Figure 11 shows just noticeable luminance differences. The 75%-value indicates that an artifact will be visible with a probability of 0.75, and the value presented in the chart is the percentage of pixels above this threshold.

Our test suite consists of a variety of both luminance-calibrated and relative luminance HDR images. To compare them, we multiply each image with a luminance factor so that all HDR VDP tests are performed for a global adaptation level of approximately 300 cd/m^2. Although we quantize the base luminances to only 8 bits, our algorithm shows near-optimal results, except for image **(a)**. VDP difference images for image **(i)** are shown in Figure 12. Increasing the global adaptation level increases the detection rates slightly, but the relationship between the algorithms remains.

Our algorithm is the clear winner both in terms of robustness and perceived visual quality, as can be seen in Figure 13. In general, luminance artifacts are more easily detectable, and both S3TC YUV and our algorithm handle these cases better due

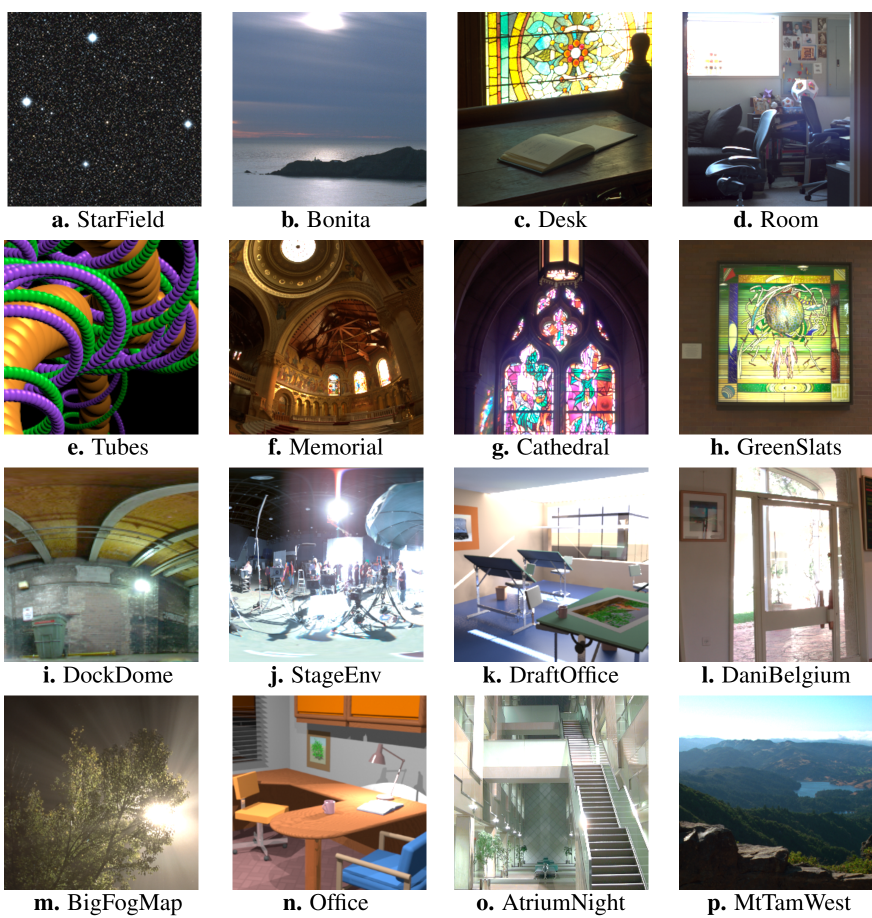


Figure 10: The HDR test images we use for evaluating our algorithms. The figure shows cropped versions of the actual test images. **(e)**, **(k)**, and **(n)** are synthetic.

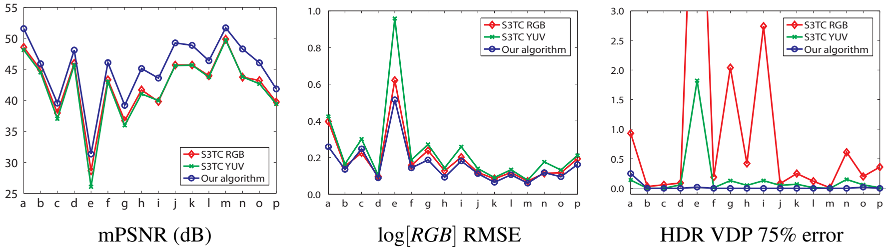


Figure 11: These diagrams show the performance of our algorithm compared to the S3TC RGB and S3TC YUV algorithms for each of the images **(a)–(p)** in Figure 10. The mPSNR measure gives consistently better values (left), while the $\log[RGB]$ RMSE (middle) is lower on nearly all of the test images. The HDR VDP luminance measure (right) indicates a perceivable error very close to 0.0% for most images with our algorithm, clearly superior to both S3TC-based algorithms.

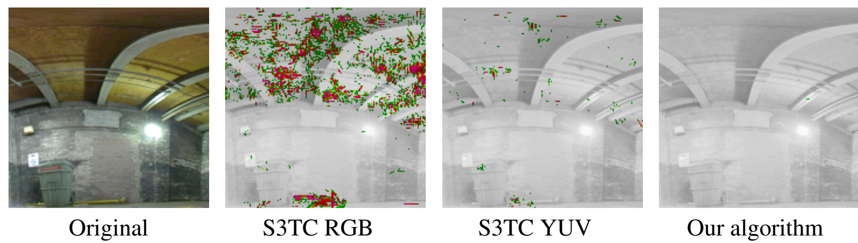


Figure 12: HDR VDP difference images. Green indicates areas with 75% chance of detecting an artifact, and red indicates areas with 95% detection probability.

to the luminance focus of the $\log Y\bar{u}\bar{v}$ color space. However, the limitation of S3TC YUV to a line in 3D makes it unstable in many cases. The only errors we have seen using our algorithm are slight chrominance leaks due to the subsampling, and artifacts in some images with high exposures, originating from the quantization of the base chrominances. Such a scenario is illustrated in Figure 14. Overall, our algorithm is much more robust since it generates significantly fewer tiles with visible errors, and this is a major strength.

It is very important that a TC format developed for real-time graphics handle mipmapping well. Figure 15 shows the average error from all our test images for the first 7 mipmap levels. The average errors grow at smaller mipmap levels due to the concentration of information in each tile, but our algorithm is still very robust and compares favorably to the S3TC-based techniques. In both error measures, our approach is consistently much better.

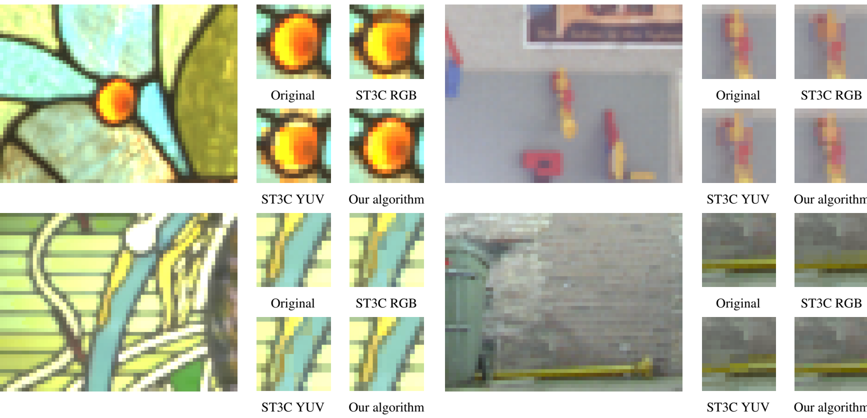


Figure 13: This figure shows magnified parts of the test images **(c)**, **(d)**, **(h)**, and **(i)**, compressed with our algorithm and the two S3TC-based methods. In difficult parts of the images, the S3TC algorithms sometimes produce quite obvious artifacts, while this rarely happens with our algorithm due to its separated encoding of luminance and chrominance.

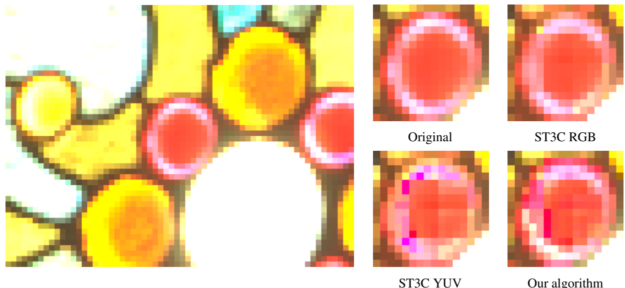


Figure 14: A difficult scenario for our algorithm is an over-exposed image **(c)** with sharp color transitions. S3TC YUV does, however, handle this case even worse. Surprisingly, S3TC RGB performs very well here.

8 Conclusions

In this work, we have presented the first low bit rate HDR texture compression system suitable for graphics hardware. In order to accurately represent the wide dynamic range in HDR images, we opted for a fixed rate of 8 bpp. Although it would have been desirable to further reduce the bandwidth, we found it hard to achieve an acceptable image quality at 4 bpp, while preserving the full dynamic range. For future work, it would be interesting to incorporate an alpha channel in the 8 bpp budget.

Our algorithm performs very well, both visually and in the chosen error metrics.

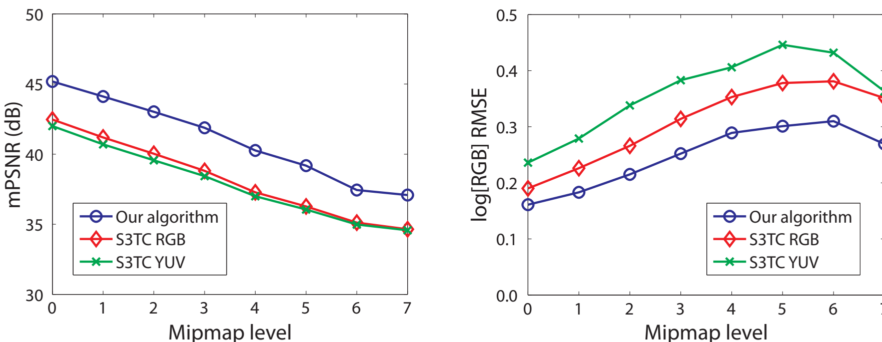


Figure 15: Here, the average mPSNR and $\log[RGB]$ RMSE values are presented for various mipmap levels of our test images, where 0 is the original image and higher numbers are sub-sampled versions.

However, more research in meaningful error measures for HDR images is needed. The HDR VDP by Mantiuk et al. [13] is a promising approach, and it would be interesting to extend it to handle chroma as well. We hope that our work will further accelerate the use of HDR images in real-time rendering, and provide a basis for future research in HDR texture compression.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks to Calle Lejdfors & Christina Dege for proof-reading, Rafal Mantiuk for letting us use his HDR VDP program, and Apple for the temporary Shake license. Image (**f**) is courtesy of Paul Debevec. (**g**) and (**l**) are courtesy of Dani Lischinski. (**h**) was borrowed from the RIT MCSL High Dynamic Range Image Database. (**i**) was created using HDRI data courtesy of HDRIMaps (`www.hdrimaps.com`) from the LightWorks HDRI Starter Collection (`www.lightworkdesign.com`). (**k**) and (**n**) are courtesy of Greg Ward. (**m**) is courtesy of Jack Tumblin, Northwestern University. (**o**) is courtesy of Karol Myszkowski. The remaining images were taken from the OpenEXR test suite.

A mPSNR Parameters

In this appendix, we summarize the parameters used when computing the mPSNR quality measure for the images in Figure 10 (**a**–**p**). For all mPSNR computations, we have computed the mean square error (MSE) only for the integers between the start and stop exposures (as shown in the table below). For example, if the start exposure is -10, and the stop exposure is $+5$, then we compute the MSE for all exposures in the set: $\{-10, -9, \ldots, +4, +5\}$.

Test image	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
Start exposure	-9	-8	-8	-9	-3	-9	-4	0
Stop exposure	+4	+2	+5	+3	+7	+3	+7	8
Test image	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**
Start exposure	-6	-12	-7	-6	-8	-6	-12	-4
Stop exposure	+3	+1	+2	+5	+2	+3	+1	+5

B Shape Transform Coordinates

Below we present coordinates, (α, β), for each of the template shapes in Figure 8.

Shape	p1	p2	p3	p4
A	(0, 0)	(11/32, 0)	(21/32, 0)	(1, 0)
B	(0, 0)	(1/4, 0)	(3/4, 0)	(1, 0)
C	(0, 0)	(1/8, 0)	(1/4, 0)	(1, 0)
D	(0, 0)	(1/2, 0)	(3/4, 1/4)	(1, 0)
E	(0, 0)	(1/2, 0)	(1/2, 1/2)	(1, 0)
F	(0, 0)	(11/32, 11/32)	(21/32, 11/32)	(1, 0)
G	(0, 0)	(0, 1/2)	(1, 1/2)	(1, 0)
H	(0, 0)	(1/4, 1/4)	(1/2, 0)	(1, 0)

Bibliography

[1] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, 1996.

[2] R. Bogart, F. Kainz, and D. Hess. OpenEXR Image File Format. In *ACM SIGGRAPH Sketches & Applications*, 2003.

[3] Alan Chalmers, Ann McNamara, Scott Daly, Karol Myszkowski, and Tom Troscianko. Image Quality Metrics. In *ACM SIGGRAPH Course Notes*, 2000.

[4] Jonathan Cohen, Chris Tchou, Tim Hawkins, and Paul Debevec. Real-Time High Dynamic Range Texture Mapping. In *Eurographics Workshop on Rendering*, pages 313–320, 2001.

[5] Scott Daly. The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity. In *Digital Images and Human Vision*, pages 179–206. MIT Press, 1993.

[6] Paul E. Debevec. Rendering Synthetic Objects into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography. In *Proceedings of ACM SIGGRAPH 98*, pages 189–198, 1998.

[7] Paul E. Debevec and Jitendra Malik. Recovering High Dynamic Range Radiance Maps from Photographs. In *Proceedings of ACM SIGGRAPH 97*, pages 369–378, 1997.

[8] I.L. Dryden and K.V. Mardia. *Statistical Shape Analysis*. Wiley, 1998.

[9] Simon Fenney. Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware*, pages 84–91, 2003.

[10] Konstantine Iourcha, Krishna Nayak, and Zhou Hong. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999.

[11] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.

[12] Yuanzhen Li, Lavanya Sharan, and Edward H. Adelson. Compressing and Companding High Dynamic Range Images with Subband Architectures. *ACM Transactions on Graphics*, 24(3):836–844, 2005.

[13] Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting Visible Differences in High Dynamic Range Images – Model and its Calibration. In *Human Vision and Electronic Imaging X*, pages 204–214, 2005.

[14] Rafal Mantiuk, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. Perception-Motivated High Dynamic Range Video Encoding. *ACM Transactions on Graphics*, 23(3):733–741, 2004.

[15] John D. Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley, 2005.

[16] Anton Pereberin. Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99*, pages 195–199, 1999.

[17] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers, 2003.

[18] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display and Image-Based Lighting*. Morgan Kaufmann Publishers, 2005.

[19] S. J. Sangwine and R. E. N. Horne, editors. *The Colour Image Processing Handbook*. Chapman and Hill, 1998.

[20] Helge Seetzen, Wolfgang Heidrich, Wolfgang Stuerzlinger, Greg Ward, Lorne Whitehead, Matthew Trentacoste, Abhijeet Ghosh, and Andrejs Vorozcovs. High Dynamic Range Display Systems. *ACM Transactions on Graphics*, 23(3):760–768, 2004.

[21] Helge Seetzen, Lorne A. Whitehead, and Greg Ward. A High Dynamic Range Display Using Low and High Resolution Modulators. *Society for Information Display Internatiational Symposium Digest of Technical Papers*, pages 1450–1453, 2003.

[22] Jacob Ström and Tomas Akenine-Möller. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware*, pages 63–70, 2005.

[23] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–364, 1996.

[24] Greg Ward. Real Pixels. In *Graphics Gems II*, pages 80–83. Academic Press, 1991.

[25] Greg Ward. High Dynamic Range Image Encodings, `http://www.anyhere.com/`, 2005.

[26] Greg Ward and Maryann Simmons. Subband Encoding of High Dynamic Range Imagery. In *Proceedings of APGV '04*, pages 83–90, 2004.

[27] Gregory J. Ward. The RADIANCE Lighting Simulation and Rendering System. In *Proceedings of ACM SIGGRAPH 94*, pages 459–472, 1994.

[28] Gregory Larson Ward. LogLuv Encoding for Full Gamut High Dynamic Range Images. *Journal of Graphics Tools*, 3(1):15–31, 1998.

[29] Ruifeng Xu, Sumanta N. Pattanaik, and Charles E. Hughes. High-Dynamic-Range Still-Image Encoding in JPEG 2000. *IEEE Computer Graphics and Applications*, 25(6):57–64, 2005.

Paper V

Efficient Depth Buffer Compression

Jon Hasselgren   Tomas Akenine-Möller

Lund University

{jon|tam}@cs.lth.se

ABSTRACT

Depth buffer performance is crucial to modern graphics hardware. This has led to a large number of algorithms for reducing the depth buffer bandwidth. Unfortunately, these have mostly remained documented only in the form of patents. Therefore, we present a survey on the design space of efficient depth buffer implementations. In addition, we describe our novel depth buffer compression algorithm, which gives very high compression ratios.

Proceedings of Graphics Hardware, pages 102–110, 2006.

1 Introduction

The depth buffer was originally invented by Ed Catmull, but first mentioned by Sutherland et al. [9] in 1974. At that time it was considered a naive brute force solution, but now it is the de-facto standard in essentially all commercial graphics hardware, primarily due to rapid increase in memory capacity and low memory cost.

A naive implementation requires huge amounts of memory bandwidth. Furthermore, it is not efficient to read depth values one by one, since a wide memory bus or burst accesses can greatly increase the available memory bandwidth. Because of this, several improvements to the depth buffer algorithm have been made. These include: the tiled depth buffer, depth caching, tile tables [7], fast z-clears [4], z-min culling [1], z-max culling [3, 4], and depth buffer compression [7]. A schematic illustration of a modern architecture implementing all these features is shown in Figure 1.

Many of the depth buffer algorithms mentioned above have never been thoroughly described, and only exist in textual form as patents. In this paper, we attempt to remedy this by presenting a survey of the modern depth buffer architecture, and the current depth compression algorithms. This is done in Section 2 & 3, which can be considered previous work. In Section 4 & 5, we present our novel depth compression algorithm, and thoroughly evaluate it by comparing it to our own implementations of the algorithms from Section 3.

2 Architecture Overview

A schematic overview implementing several different algorithms for reducing depth buffer bandwidth usage is shown in Figure 1. Next, we describe how the depth buffer collaborates with the other parts of a graphics hardware architecture.

The purpose of the *rasterizer* is to identify which pixels lie within the triangle currently being rendered. In order to maximize memory coherency for the rest of the architecture, it is often beneficial to first identify which *tiles* (a collection of $n \times m$ pixels) that overlap the triangle. When the rasterizer finds a tile that partially overlaps the triangle, it distributes the pixels in that tile over a number of *pixel pipelines*. The purpose of each pixel pipeline is to compute the depth and color of a pixel. Each pixel pipeline contains a *depth test* unit responsible for discarding pixels that are occluded by previously drawn geometry.

Tiled depth buffering in its most simple form works by letting the rasterizer read a complete tile of depth values from the depth buffer and temporarily store it in on-chip memory. The depth test in the pixel pipelines can then simply compare the depth value of the currently generated pixel with the value in the locally stored tile. In order to increase overall performance, it is often motivated to cache more than one tile of depth buffer values in on-chip memory. A costly memory access can be skipped altogether if a tile already exists in the cache. The tiled architecture

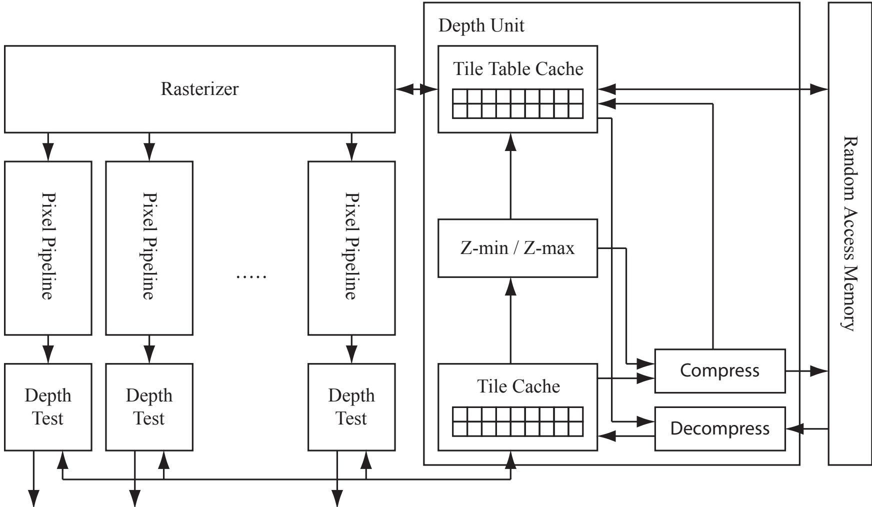


Figure 1: A modern depth buffer architecture. Only the tile cache is needed to implement tiled depth buffering. The rest of the architecture is dedicated to bandwidth and performance optimizations. For a detailed description see Section 2.

decrease the number of memory accesses, while increasing the size of each access. This is desirable since bursting makes it more efficient to write big chunks of localized data.

There are several techniques to improve the performance of a tiled depth buffer. A common factor for most of them is that they require some form of "header" information for each tile. Therefore, it is customary to use a *tile table* where the header information is kept separately from the depth buffer data. Ideally, the entire tile table is kept in on-chip memory, but it is more likely that it is stored in external memory and accessed through a cache. The cache is then typically organized in *super-tiles* (a tile consisting of tiles) in order to increase the size of each memory access to the tile table. Each tile table entry typically contains a number of "flag" bits, and potentially the minimum and maximum depth values of the corresponding tile.

The maximum and minimum depth values stored in the tile table can be used as a base for different culling algorithms. Culling mainly comes in two forms: z-max [3, 4] and z-min [1]. Z-max culling uses a conservative test to detect when all pixels in a tile are guaranteed to fail the depth test. In such a case, we can discard the tile already in the rasterizer stage of the pipeline, yielding higher performance. We can also avoid reading the depth buffer, since we already know that all depth tests will fail. Similarly, Z-min culling performs a conservative test to determine if all pixels in a tile are guaranteed to pass the depth tests. If this holds true, and the tile is entirely covered by the triangle currently being rendered, then we know that all depth values will be overwritten. Therefore we can simply clear an entry in the depth cache, and need not read the depth buffer.

The flag bits in the tile table are used primarily to flag different modes of depth buffer compression. A modern depth buffer architecture usually implements one or several compression algorithms, or compressors. A compressor will, in general, try to compress the tile to a fixed *bit rate*, and fails if it cannot represent the tile in the given number of bits without information loss. When writing a depth tile to memory, we select the compressor with the lowest bit rate, that succeeds in compressing the tile. The flags in the tile table are updated with an identifier unique to that compressor, and the compressed data is written to memory. We must write the tile in its uncompressed form if all available compressors fail, and it is therefore still necessary to allocate enough external memory to hold an uncompressed depth buffer. When a tile is read from memory, we simply read the compressor identifier from the tile table, and decompress the data using the corresponding decompression algorithm.

The main reason that depth compression algorithms can fail is that the depth compression must be lossless. The compression occurs each time a depth tile is written to memory, which happens on a highly unpredictable basis. Lossy compression amplifies the error each time a tile is compressed, and this could easily make the resulting image unrecognizable. Hence, lossy compression must be avoided.

3 Depth Buffer Compression - State of the Art

In this section, we describe existing compression algorithms. It should be emphasized that we have extracted the information below from patents, and that there may be variations of the algorithms that perform better, but such knowledge usually stays in the companies. However, we still believe that the general discussion of the algorithms is valuable.

A reasonable assumption is that each depth value is stored in 24 bits.[1] In general, the depth is assumed to hold a floating-point value in the range $[0.0, 1.0]$ after the projection matrix has been applied. For hardware implementation, 0.0 is mapped to the 24-bit integer 0, and 1.0 is mapped to $2^{24} - 1$. Hence, integer arithmetic can be used.

We define the term *compression probability* as the fraction of tiles that can be compressed by a given algorithm. It should be noted that the compression probability depends on the geometry being rendered, and can therefore only be determined experimentally.

3.1 Fast z-clears

Fast z-clears [5] is a method that can be viewed as a simple form of compression algorithm. A flag combination in the tile table entry is reserved specifically for cleared tiles. When the hardware is instructed to clear the entire depth buffer, it will instead fill the tile table with entries that are flagged as cleared tiles. This

[1] Generalizing to other bit rates is straightforward.

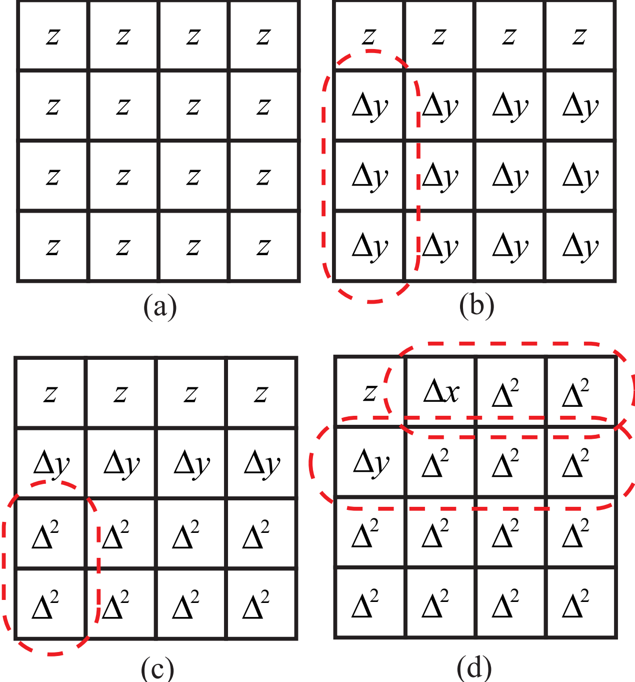


Figure 2: Computing the second order differentials. a) Original tile, b) First order column differentials, c) Second order column differentials, d) Second order row differentials.

means that the actual clearing process is greatly sped up, but it also has a positive effect when rendering geometry, since we need not read a depth tile that is flagged as cleared.

Fast z-clears is a popular compression algorithm since it gives good compression ratios and is very easy to implement.

3.2 Differential Differential Pulse Code Modulation

Differential differential pulse code modulation (DDPCM) [2] is a compression scheme, which exploits that the z-values are linearly interpolated in screen space. This algorithm is based on computing the second order depth differentials as shown in Figure 2. First, first-order differentials are computed columnwise. The procedure is repeated once again to compute the second-order columnwise differentials. Finally, the row-order differentials are computed for the two top rows, and we get the representation shown in Figure 2d. If a tile is completely covered by a single triangle, the second-order differentials will be zero, due to the linear interpolation. In practice, however, the second-order differential is a number in the set $\{-1, 0, +1\}$ if depth values are interpolated at a higher precision than they are

stored in, which often is the case.

DeRoo et al. [2] propose a compression scheme for 8×8 pixel tiles that use 32 bits for storing a reference value, 2×33 bits for x and y differentials, and 61×2 bits for storing the second order differential of each remaining pixel in the tile. This gives a total of 220 bits per tile in the best case (when a tile is entirely covered by a single triangle). A reasonable assumption would be that we read 256 bits from the memory, which would give a 8 : 1 compression when using a 32-bit depth buffer. Most of the other compression algorithms are designed for a 24-bit depth format, so we extend this format to 24 bit depth for the sake of consistency. In this case, we could sacrifice some precision by storing the differentials as 2×23 bits, and get a total of 192 bits per tile, which gives the same compression ratio as for the 32 bit mode.

In the scheme described above, two bits per pixel are used to represent the second order differential. However, we only need to represent the values: $\{-1, 0, +1\}$. This leaves one bit-combination that can be used to flag when the second-order differential is outside the representable range. In that case, we can store a fixed number of second-order differentials in a higher resolution, and pick the next in order each time an escape code occurs. This can increase the compression probability somewhat at the cost of a higher bit rate.

DeRoo et al. also briefly describe an extension of the DDPCM algorithm that is capable of handling some cases of tiles containing two different planes separated by a single edge. They compute the second order differentials from two different reference points, the upper left and lower left pixels of the tile. From these two representations, one *break point* is determined along every column, such that pixels before and after the break point belong to different planes. The break points are then used to combine the two representations to a single representation. A 24-bit version of this mode would require $24 \times 6 + 2 \times 57 + 8 \times 4 = 290$ bits of storage.

The biggest drawback of the suggested two plane mode is that compression only works when the two reference points lie in different planes. This will only be true in half of the cases, if we assume that all orientation and positioning of the edge separating the two plane is equally probable.

3.3 Anchor encoding

Van Dyke and Margeson [10] suggest a compression technique quite similar to the DDPCM scheme. The approach is based on 4×4 pixel tiles (although it could be generalized) and is illustrated in Figure 3. First, a fixed anchor pixel, denoted z in the figure, is selected. The depth value of the anchor pixel is always stored at full 24-bit resolution. Two more depth values, Δx and Δy, are stored relatively to the depth value of the anchor pixel, each with 15 bits of resolution. These three values form a plane, which can be used to predict the depth values of the remaining pixels. Compression is achieved by storing the difference between the predicted, and actual depth value, for the remaining pixel. The scheme uses 5 bits of resolution for each pixel, resulting in a total of 119 bits (128 with a fast clear

d	d	d	d
d	z →	Δx	d
d	↓ Δy	d	d
d	d	d	d

Figure 3: Anchor encoding of a 4×4 tile. The depth values of the z, Δx and Δy pixels form a plane. Compression is achieved by using the plane as a predictor, and storing an offset, d, for each pixel. Only 5 bits are used to store the offsets.

flag and a constant stencil value for the whole tile).

The anchor encoding mode behaves quite similar to the one plane mode of the DDPCM algorithm. The extra bits of per-pixel resolution provide for some extra numerical stability, but unfortunately do not seem to provide a significant increase in terms of compression ratio.

3.4 Plane Encoding

The previously described algorithms use a plane to predict the depth value of a pixel, and then correct the prediction using additional information. Another approach is to skip the correction factors and only store parameterized prediction planes. This only works when the prediction planes are stored in the same resolution that is used for the interpolation.

Orenstein et al. [8] present such a compression scheme, where a single plane is stored per 4×4 pixel tile. They use a representation on the form $Z(x,y) = C_0 + xC_x + yC_y$ with 40 bits of precision for each constant. A total of 120 bits is needed, leaving 8 bits for a stencil value. Exactly how the constants are computed, is not detailed. However, it is likely that they are obtained directly from the interpolation unit of the rasterizer. Computing high resolution plane constants from a set of low resolution depth values is not trivial.

A similar scheme is suggested by Van Hook [11], but they assume that the same precision (16, 24 or 32 bits) is used for storing and interpolating the depth values. The compression scheme can be seen as an extension of Orenstein's scheme, since it is able to handle several planes. It requires communication between the rasterizer and the compression algorithm. A counter is maintained for every tile cache entry. The counter is incremented whenever rasterization of a new triangle generates pixels in the tile, and each generated pixel will be tagged with that value as an identifier, as shown in Figure 4. The counter is usually given a limited resolution (4 bits is suggested) and if the counter overflows, no compression can be made. When a cache entry is compressed and written to memory, the first pixel with a particular ID number is found. This pixel is used as a reference point for the plane

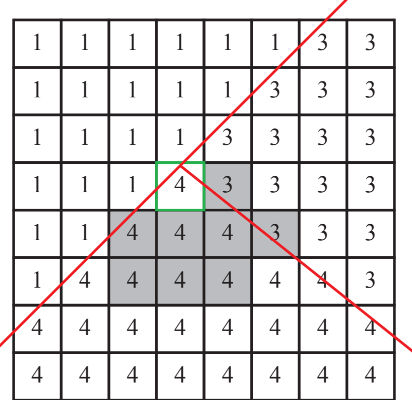


Figure 4: Van Hook's plane encoding uses ID numbers and the rasterizer to generate a mask indicating which pixels belong to a certain triangle. The compression is done by finding the first pixel with a particular ID and searching a window of nearby pixels, shown in gray, to compute a plane representation for all pixels with that ID.

equation. The x and y differentials are found by searching the pixels in a small window around the reference point. Van Hook shows empirically that a window such as the one shown in Figure 4 is sufficient to be able to compute plane equations in 96% of the cases that could be handled with an infinite size window (tests are only performed on a simple torus scene though). The suggested compression modes stores a number of planes (2,4, or 8 with 24 bits per component) and an identifier for each pixel, indicating to which plane that pixel belongs (1,2 or 3 bits depending on the number of planes), resulting in compression ratios varying from $6:1$ to $2:1$. The compression procedure will automatically collapse any pixel ID numbers that is not currently in use. ID numbers may go to waste as depth values are overwritten when the depth test succeeds. Therefore, collapsing is important in order to avoid overflow of the ID counter. When decompressing a tile, the ID counter is initialized to the number of planes that is indicated by the compression mode.

The strength of the Van Hook scheme is that it can handle a large number of triangles overlapping a single tile, which is an important feature when working with large tiles. A drawback is that we must also store the 4-bit ID numbers, and the counter, in the depth tile cache. This will increase the cache size by $4/24 = 16.6\%$, if we use a 4-bit ID number per pixel. Another weakness is that the depth interpolation must be done at the same resolution as the depth values are stored in.

3.5 Depth Offset Compression

Morein and Natale's [6] depth offset compression scheme is illustrated in Figure 5. Although the patent is written in a more general fashion, the figure illustrates its

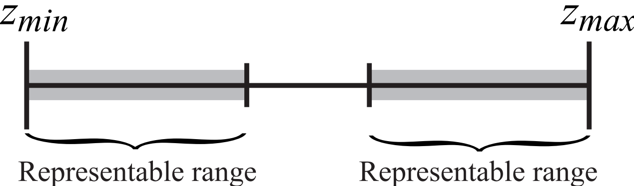


Figure 5: The depth offset scheme compresses the depth data by storing depth values in the gray regions as offsets relative to either the z-min or z-max value.

primary use. The depth offset compression scheme assumes that the depth values in a tile often lie in a narrow interval near either the z-min value or the z-max value. We can compress such data by storing an n-bit offset value for every depth value, where n is some pre-determined number (typically 8 or 12) of bits. The most significant bit indicates whether the depth value is encoded as an offset relative to the z-min or z-max value, and the remaining bits represents the offset. The compression fails if the depth offset value of any pixel in a tile cannot be represented without loss in the given number of bits.

This algorithm is particularly useful if we already store the z-min and z-max values in the tile table for culling purposes. Otherwise we must store the z-min and z-max values in the compressed data, which increases the bit rate somewhat.

Orenstein et al. [8] also present a compression algorithm that is essentially a subset of Morein and Natale's algorithm. It is intended to complement the plane encoding algorithm described in Section 3.4, but can also be implemented independently. The depth value of a reference pixel is stored along with offsets for the remaining pixels in the tile. This mode can be favorable in some cases if the z-min and z-max values are not available.

The advantage of depth offset compression is that compression is very inexpensive. It does not work very well at high compression ratios, but gives excellent compression probabilities at low compression rates. This makes it an excellent complementary algorithm to use for tiles that cannot be handled with specialized plane compression algorithms (Sections 3.2-3.4).

4 New Compression Algorithms

In this section, we present two modes of a new compression scheme. As most other schemes, we try to achieve compression by representing each tile as number of planes and predict the depth values of the pixels using these planes.

In the majority of cases, depth values are interpolated at a higher resolution than is used for storage, and this is what we assume for our algorithm. We believe that this is an important feature, especially in the case of homogeneous rasterizers where exact screen space interpolation can be difficult. Allowing higher precision interpolation allows for some extra robustness.

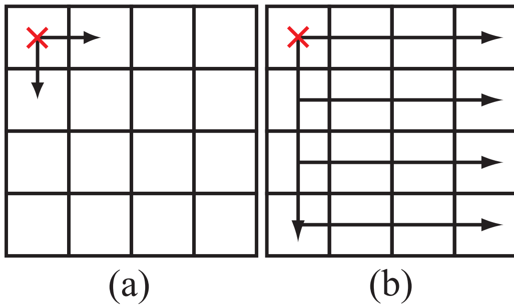


Figure 6: The leftmost image shows the points used to compute our prediction plane. The rightmost image shows in what order we traverse the pixels of a tile.

In the following we will motivate that we only need the integer differentials, and a one bit per pixel *correction term*, in order to be able to reconstruct a rasterized plane. During the rasterization process, the depth value of a pixel is given through linear interpolation. Given an origin (x_0, y_0, z_0) and the screen space differentials $(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta x})$, we can write the interpolation equations as:

$$z(x,y) = z_0 + (x - x_0)\frac{\Delta z}{\Delta x} + (y - y_0)\frac{\Delta z}{\Delta y}. \quad (1)$$

The equation can be incrementally evaluated by stepping in the x-direction (similar for y) by computing:

$$z(x+1,y) = z(x,y) + \frac{\Delta z}{\Delta x}. \quad (2)$$

We can rewrite the differential of Equation 2 as a quotient and remainder part, as shown below:

$$\frac{\Delta z}{\Delta x} = \left\lfloor \frac{\Delta z}{\Delta x} \right\rfloor + \frac{r}{\Delta x}. \quad (3)$$

Equation 2 can then be stepped through incrementally by adding the quotient, $\lfloor \frac{\Delta z}{\Delta x} \rfloor$, in each step, and by keeping track of the accumulated remainder, $\frac{r}{\Delta x}$. When the accumulated remainder exceeds one, it is propagated to the result. What this amounts to in terms of compression is that we can store the propagation of the remainder in one bit per pixel, as long as we are able find the differentials $(\lfloor \frac{\Delta z}{\Delta x} \rfloor, \lfloor \frac{\Delta z}{\Delta y} \rfloor)$. This reasoning has much in common with Bresenham's line algorithm.

4.1 One plane mode

For our one plane mode, we assume that the entire tile is covered by a single plane. We choose the upper left corner as a reference pixel and compute the differentials

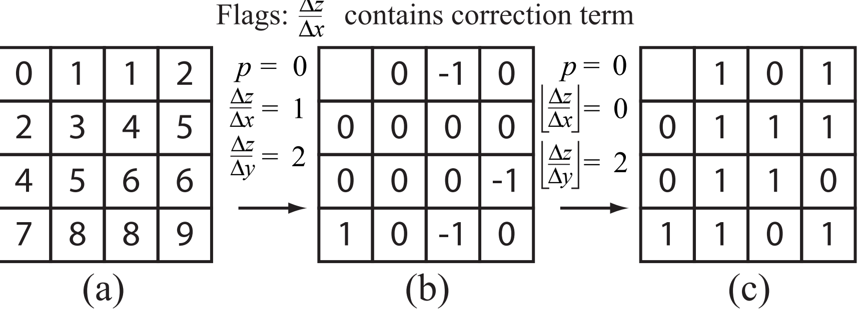


Figure 7: The different steps of the one plane compression algorithm, applied to a compressible example tile.

$(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta y})$ directly from the neighbors in the x- and y-directions, as shown in Figure 6a. The result will be the integer terms,$(\lfloor\frac{\Delta z}{\Delta x}\rfloor, \lfloor\frac{\Delta z}{\Delta y}\rfloor)$, of the differentials, each with a potential correction term of one baked into it.

We then traverse the tile in the pattern shown in Figure 6b, and compute the correction terms based on either the x or y direction differentials (y direction when traversing the leftmost column, and x direction when traversing along a row). If the first non-zero correction term of a row or column is one, we flag that the corresponding differential as correct. Accordingly, if the first non-zero element is minus one, we flag that the differential contains a correction term. The flags are sticky, and can therefore only be set once. We also perform tests to make sure that each correction value is representable with one bit. If the test fails, the tile cannot be compressed.

After the previous step, we will have a representation like the one shown in Figure 7b. Just as in the figure, we can get correction terms of -1 for the differentials that contain an embedded correction term. Thus, we want to subtract one from the differential (e.g. $\frac{\Delta z}{\Delta x}$), and to compensate for this, we add one to all the per-pixel correction terms. Adding one to the correction terms is trivial since they can only be -1 or 0. We can just invert the last bit of the correction terms and interpret them as a one bit number. We get the corrected representation of Figure 7c.

In order to optimize our format, we wish to align the size of a compressed tile to the nearest power of two. In order to do so, we sacrifice some accuracy when storing the differentials, and reference point. Since the compression must be lossless, the effect is that the compression probability is slightly decreased, since the lower accuracy means that fewer tiles can be compressed successfully. Interestingly, storing the reference point at a lower resolution works quite well if we assume that the most significant bits are set to one. This is due to the non-linear distribution of the depth values. For instance, assume we use the projection model of OpenGL and have the near and far clip planes set to 1 and 100 respectively, then 21 bits will

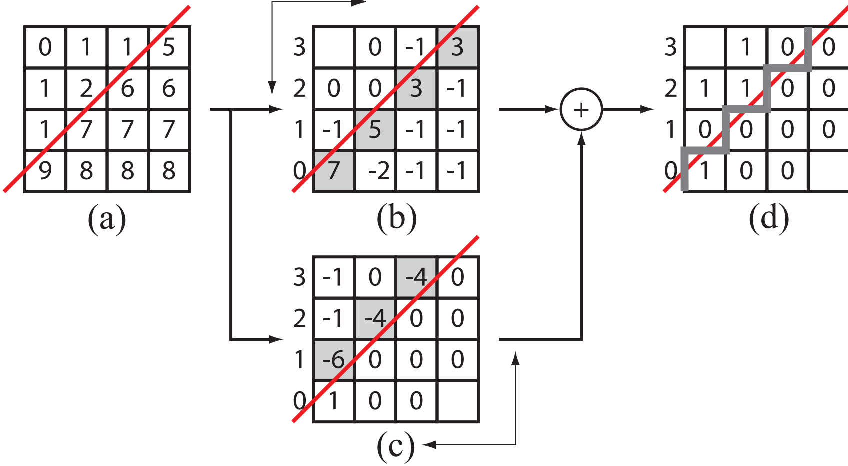


Figure 8: This figure illustrates the two plane compression algorithm. a) Shows the original tile with depth values from two different planes. The line indicates the edge separating the two planes. b & c) We execute the one plane algorithm of Section 4.1 for each corner of the tile. In this figure, we only show the two correct corners for clarity. Note that the correction terms take on unrepresentable values when we cross the separating edge. We use this to detect the breakpoints, shown in gray. d) In a final step, we stitch together the two solutions from (b) and (c), and make sure to correct the differentials so that all correction terms are either 0 or 1. The breakpoints are marked as a gray line.

be enough to cover 93% of the representable depth range. In contrast, 21 bits can only represent 12.5% of the range representable by a 24 bit number. We propose the following formats for our one plane mode

tile	point	deltas	correction	total
4×4	21	14×2	1×15	64
8×8	24	20×2	1×63	127

4.2 Two plane mode

We also aim to compress tiles that contain two planes separated by a single edge. See Figure 8a for an example. In order to do so, we must first extend our one plane algorithm slightly. When we compute the correction terms, we already perform tests to determine if the correction term can be represented with one bit. If this is not the case, then we call the pixel a break point, as defined in Section 3.2, and store its horizontal coordinate. We only store the first such break point along each row. If a break point is found while traversing along a column, rather than a

row, then all remaining rows are given a break point coordinate of zero. Figure 8b shows the break points and correction terms resulting from the tile in Figure 8a. As shown in the figure, we can use the break points to identify all pixels that belong to a specific plane.

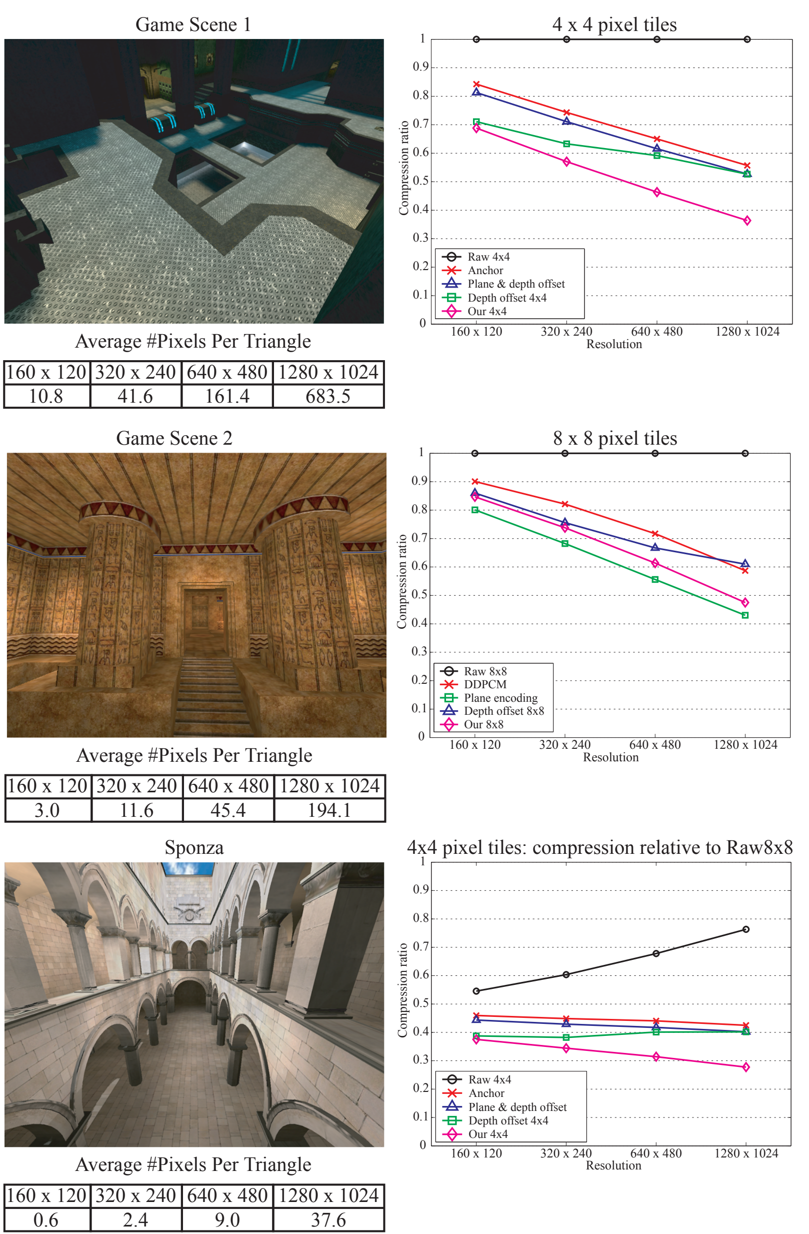
We must also extend the one plane mode so that it can operate from any of the corners as reference point. This is a simple matter of reflecting the traversal scheme, from Figure 6, horizontally and/or vertically until the reference point is where we want it to be.

We can now use the extended one plane algorithm to compress tiles containing two planes. Since we have limited the algorithm to tiles with only a single separating edge, it is possible to find two diagonally placed corners of the tile that lie on opposite sides of the edge. There are only two configurations of diagonally placed corners, which makes the problem quite simple. The basic idea is to run the extended one plane algorithm for all four corners of the tile, and then find the configuration of diagonal corners for which the break points match. We then stitch together the correction terms of both corners, by using the break point coordinates. The result is shown in Figure 8d.

It should be noted that we need to impose a further restriction on the break points. Assume that we wish to recreate the depth value of a certain pixel, p, then we must be able to recreate the depth values of the pixels that lie "before" p in our fixed traversal order. In practice, this is not a problem since we are able to chose the other configuration of diagonal corners. However, we must perform an extra test. The break points must be either in falling or rising order, depending on which configuration of diagonal corners is used. As it turns out, we can actually use this to our advantage when designing the bit allocations for a tile. Since we know that the break points are in rising or falling order, we can use fewer bits for storing them. In our 4×4 tile mode, we use this to store the break points in just 7 bits. We do not use this in the 8×8 tile mode, as the logic would become too complicated. Instead, we store the break points using $\log_2(9^8) = 26$ bits, or with 4 bits per break point when possible.

We employ the same kind of bit length optimizations as for the one plane mode. In addition, we need one bit, d, to indicate which diagonal configuration is used, and some bits for the break points, bp. Suggestions for bit allocations are shown in the following table.

tile	d	point	deltas	bp	correction	total
4×4	1	23×2	15×4	7	1×15	128
8×8	1	$22 + 21$	15×4	26	1×63	192
8×8	1	24×2	24×4	32	1×63	240



Average #Pixels Per Triangle (Game Scene 1)

160 x 120	320 x 240	640 x 480	1280 x 1024
10.8	41.6	161.4	683.5

Average #Pixels Per Triangle (Game Scene 2)

160 x 120	320 x 240	640 x 480	1280 x 1024
3.0	11.6	45.4	194.1

Average #Pixels Per Triangle (Sponza)

160 x 120	320 x 240	640 x 480	1280 x 1024
0.6	2.4	9.0	37.6

Figure 9: The left column shows a summary of the benchmark scenes. The diagrams in the right column show the average compression for all three scenes as a function of rendering resolution, for 4×4 and 8×8 pixel tiles. Finally, we show the depth buffer bandwidth of 4×4 tiles, relative to the bandwidth of a Raw 8x8 depth buffer. It should be noted that this diagram does not take tile table bandwidth into account.

5 Evaluation

In this section, we compare the performance, in terms of bandwidth, of all depth compression algorithms described in this paper. The tests were performed using our functional simulator, implementing a tiled rasterizer that traverses triangles a horizontal row of tiles at a time. We matched the tile size of the rasterizer to the tile size of each depth buffer implementation in order to maximize performance for all compression algorithms. Furthermore, we assumed a 64 bit wide memory bus, and accordingly, all our implementations of compressors have been optimized to make the size of all memory accesses aligned to 64 bits.

The depth buffer system in our functional simulator implements all features described in Section 2. We used a depth tile cache of approximately 2 kB, and full precision z-min and z-max culling. Our tests show that compression rates are only marginally affected by the cache size.[2] Similarly, the z-min and z-max culling avoids a given fraction of the depth tile fetches, independent of compression algorithm. Therefore, it should affect all algorithms equally, and not affect the trend of the results.

Most of the compression algorithms have two operational modes. Therefore, we have chosen this as our target. Furthermore, two modes fit well into a two bit tile-table assuming we also need to flag for uncompressed tiles and for fast z clears. It is our opinion that using fast clears makes for a fair comparison of the algorithms. All algorithms can easily handle cleared tiles, which means that our compressors would be favored if this mode was excluded since they have the lowest bit rate.

We evaluate the following compression configurations

- **Raw 4x4/8x8**: No compression.
- **DDPCM**: The one and two-plane mode (not using "escape codes") of the DDPCM compression scheme from Section 3.2, 8×8 pixel tiles. Bit rate: 3/5 bpp (bits per pixel)
- **Anchor**: The anchor encoding scheme (Section 3.3), 4×4 pixel tiles. Note that this is the only compression scheme in the test that only uses one compression mode. One bit-combination in the tile table was left unused. Bit rate: 8 bpp.
- **Plane encoding**: Van Hook's plane encoding mode from section 3.4, 8×8 pixel tiles. Only the two and four plane modes were used, since we only allow 2 compression modes. This algorithm was given a slight favor in form of a 16.6% bigger depth tile cache. Bit rate: 4/7 bpp.
- **Plane & depth offset**: The plane (Section 3.4) and depth offset (Section 3.5) encoding modes of Orenstein et al, 4×4 pixel tiles. Bit rate: 8/16 bpp, 8 bits for the plane mode and 16 bits for the depth offset mode.

[2]The efficiency of all algorithms increased slightly, and equally, with a bigger cache. We tested cache sizes of 0.5, 1, 2 and 4 kb

- **Depth Offset 4x4/8x8**: Morein and Natale's depth offset compression mode from Section 3.5. We used two compression modes, one using 12 bit offsets, and one with 16 bit offsets. Bit rate: 12/16 bits per pixel for both 4×4 and 8×8 tiles.
- **Our 4x4/8x8**: Our compression scheme, described in Section 4. For the 8×8 tile mode, we used the 192 bit version of the two plane mode in this evaluation. Bit rate: 4/8 bits per pixel for 4×4 tiles and 2/3 bits per pixel for 8×8 tiles.

Our benchmarks were performed on three different test scenes, depicted in Figure 9. Each test scene features an animated camera with static geometry. Furthermore, we rendered each scene at four different resolutions: 160×120, 320×240, 640×480, and 1280×1024 pixels. Varying the resolution is a simple way of simulating different levels of tessellation. As can be seen in Figure 9, we cover scenes with great diversity in the average triangle area.

In the bottom half of Figure 9, we show the compression ratio of each algorithm, grouped into algorithms for 4×4 and 8×8 pixel tiles. We also present the compression of the 4×4 tile algorithms, as compared to the bandwidth of the Raw 8x8 mode. It should be noted that this relative comparison only takes the depth buffer bandwidth into account. Thus, the bandwidth to the tile table will increase as the tile size decreases. How much of an effect this will have on the total bandwidth, will depend on the format of the tile table, and on the efficiency of the culling.

For 8×8 pixel tiles, our algorithm is the clear winner among the algorithms supporting high resolution interpolation, but it cannot quite compete with Van Hook's plane encoding algorithm. This is not very surprising considering that the plane encoding algorithm is favored by a slightly bigger depth tile cache, and avoids correction terms by imposing the restriction that depth values must be interpolated in the same resolution that is used for storage.

For 4×4 pixel tiles, the advantages of our algorithm becomes really clear. It is capable of bringing the two-plane flexibility that is only seen in the 8×8 tile algorithms down to 4×4 tiles, and still keeps a reasonably low bit rate. A two plane mode for 4×4 tiles is equal to having the flexibility of eight planes (with some restrictions) in an 8×8 pixel tile. This shows up in the evaluation, as our 4×4 tile compression modes have the best compression ratio at all resolutions.

6 Conclusions

We hope that our survey of previously existing depth buffer compression schemes will provide a valuable source for the graphics hardware community, as these algorithms have not been presented in an academic paper before. As we have shown, our new compression algorithm provides competitive compression for both 4×4 and 8×8 pixel tiles at various resolutions. We have avoided an exhaustive evaluation of whether 4×4 or 8×8 tiles provide better performance, since this is a very

difficult undertaking which depends on several other parameters. Our work here has been mostly on an algorithmic level, and therefore, we leave more detailed hardware implementations for future work. We are certain that this is important, since such implementations may reveal other advantages and disadvantages of the algorithms. Furthermore, we would like to examine how to best deal with depth buffer compression of anti-aliased depth data.

Bibliography

[1] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics*, 22(3):801–808, 2003.

[2] John DeRoo, Steven Morein, Brian Favela, and Michael Wright. Method and Apparatus for Compressing Parameter Values for Pixels in a Display Frame. In *US Patent 6,476,811*, 2002.

[3] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, pages 231–238, August 1993.

[4] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM SIGGRAPH/Eurographics, August 2000.

[5] Steven Morein. Method and Apparatus for Efficient Clearing of Memory. In *US Patent 6,421,764*, 2002.

[6] Steven Morein and Mark Natale. System, Method, and Apparatus for Compression of Video Data using Offset Values. In *US Patent 6,762,758*, 2004.

[7] Steven Morein, Michael Wright, and Kin Yee. Method and apparatus for controlling compressed z information in a video graphics system. US Patent 6,636,226, 2003.

[8] Doron Ornstein, Guy Peled, Zeev Sperber, Ehud Cohen, and Gabi Malka. Z-Compression Mechanism. In *US Patent 6,580,427*, 2005.

[9] Evan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[10] James Van Dyke and James Margeson. Method and Apparatus for Managing and Accessing Depth Data in a Computer Graphics System. In *US Patent 6,961,057*, 2005.

[11] Timothy Van Hook. Method and Apparatus for Compression and Decompression of Z Data. In *US Patent 6,630,933*, 2003.

Paper VI

Exact and Error-bounded Approximate Color Buffer Compression and Decompression

Jim Rasmusson Jon Hasselgren Tomas Akenine-Möller

Ericsson Research Lund University

{jim|jon|tam}@cs.lth.se

ABSTRACT

In this paper, we first present a survey of existing color buffer compression algorithms. After that, we introduce a new scheme based on an exactly reversible color transform, simple prediction, and Golomb-Rice encoding. In addition to this, we introduce an error control mechanism, which can be used for approximate (lossy) color buffer compression. In this way, the introduced error is kept under strict control. To the best of our knowledge, this has not been explored before in the literature. Our results indicate superior compression ratios compared to existing algorithms, and we believe that approximate compression can be important for mobile GPUs.

Proceedings of Graphics Hardware, pages 41–48, 2007.

1 Introduction

The expected yearly performance increase in terms of bandwidth and latency of DRAM is about 25% and 5%, respectively. At the same time, the expected increase in computing capability of a processor is about 71% every year [13]. Due to this, the gap between memory speeds and computational resources is steadily increasing. For desktop computer GPUs this is mitigated to some extent by wider and wider DRAM buses, a "luxury" that is basically not available for mobile devices. Hence, compression techniques aimed at saving memory bandwidth for GPUs are becoming increasingly important, especially for mobile GPUs. Examples include vertex compression, texture compression, depth buffer compression, and color buffer compression.

In this paper, we focus on *color buffer compression and decompression*. The purpose of our work is to provide the reader with a state-of-the-art report of existing algorithms, which are currently only available in the form of patents, and to introduce *new* algorithms.

In terms of new algorithms, we start by introducing a new *exact* algorithm, which first uses a reversible color transform, and then applies Golomb-Rice coding after using a simple predictor. Second, we experiment with *approximate* color buffer compression. The motivation here is that we can accept, for example, lossy video compression (e.g., MPEG), and approximate rendering using precomputed radiance transfer with spherical harmonics or wavelets. Even just after executing the pixel shader, conversion from floating point to 8-bit integers is done, and this is actually a type of lossy compression (truncation). In addition, most texture compression schemes are also lossy. Hence, one could ask whether and how this can be applied to color buffers as well.

This may sound dangerous, but we show that it is possible by developing error-bounded algorithms to keep the visual artifacts under precise control, and to avoid so called *tandem* compression artifacts, which may arise due to several passes of sequential lossy compression. We emphasize that approximate, i.e., lossy, color buffer compression is not always desired. For example, in GPGPU computations for fluid simulation, exact results is of uttermost importance, and in such cases, we suggest that the programmer can turn off this feature. However, for a GPU in a mobile phone, where it is important to reduce memory accesses over external buses [3], it can be very convenient to enable approximate compression as this can increase the use time on a battery charge at a cost of slight image degradation. The major advantage of approximate compression is that higher compression can be obtained, which reduces memory bandwidth usage compared to lossless, i.e., exact, color buffer compression.

2 State-of-the-art Color Buffer Compression

In this section we summarize the color buffer compression algorithms we have found in patent databases. A summary of existing depth buffer compression schemes

is already available [5]. The reader is referred to Section 2 of this paper for an overview of the depth buffer architecture, which is almost identical to the color buffer architecture. This paper describes the general methodology for selecting and tracking what compressor to use for a specific tile, and how to handle tiles that cannot be compressed.

2.1 Multi-Sampling Compression

In this section, we present an algorithm for compression of color buffers with multi-sampling. In the following, we assume that n samples are used per pixel.

Elder explains that due to multi-sampling, samples inside a pixel often share the exact same color, and this is an opportunity for compression [4]. If all samples inside a pixel share the same color, then it suffices to flag this mode, and store only one color instead of n colors. Another common case is when a triangle edge cuts through a pixel. In such a case, we can store two colors, and a one-bit index per sample to "point" at one of these colors. Elder also suggests that this compressed format is used inside the GPU as well. This has a number of benefits, such as using fewer operations when blending and during reconstruction of the final pixel color.

The RealityEngine [2] used a similar coverage mask approach internally in their fragment pipelines. However, the depth and color-values were decompressed prior to frame buffer operations, and consequently some of the performance benefits were lost.

2.2 Color Plane Compression

Another example of exploiting multi-sampling color redundancy is the method described by Molnar et al. [11]. In a first step, they collapse pixels with identical sample colors, similarly to Elder's work [4]. When using four samples per pixel, this by itself gives sufficient compression to reach their predetermined bit-budget. However, in the case of two samples per pixel, they need to compress the data by an additional factor of two.

To this end, they introduce a plane compression mode. A predictor plane is computed from three collapsed reference pixels, as shown in Figure 1. This plane is stored with varying accuracy, and the remaining pixels are stored as differences between the actual pixel value, and the value predicted by the plane at that pixel. Bit allocations for the plane and delta values are detailed in Figure 1. The observant reader may note that these allocations only use 127 bits. The remaining bit is used to flag that a tile is in cleared state, which saves some bandwidth when clearing the color buffer.

Flag:	1 bit (set to 0)
p:	24 bits (R8G8B8)
dx:	16 bits (R6G5B5)
dy:	18 bits (R6G6B6)
Pixel Deltas:	12 bits (R4G4B4)
Alpha:	8 bit

Flag:	1 bit (set to 1)
p:	24 bits (R8G8B8)
dx:	12 bits (R4G4B4)
dy:	14 bits (R5G5B4)
Pixel Deltas:	15 bits (R5G5B5)
Alpha:	1 bit

Figure 1: Color plane compression. For this example, two samples (gray circles) per pixel are used, and these are collapsed (black circles). Each tile of 2×4 pixels are encoded together. A prediction plane is computed from the three reference pixels (indicated by p, dx, and dy), and the remaining pixels are stored as deltas between the prediction from the reference plane and the actual color of the pixel. The tables to the right show two suggested bit-allocations.

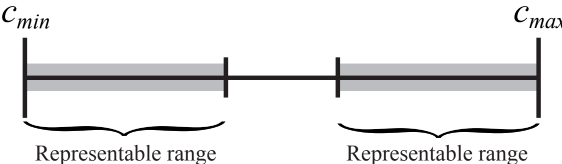


Figure 2: Color offset compression, when using the min and max colors as references. Note that the width of the representable color intervals vary with the number of bits allocated for the per pixel offsets.

2.3 Offset Compression

Some of the methods targeting depth buffer compression can also be used for color buffer compression. A good example of this is the offset compression method proposed by Morein and Natale [12].

The method compresses a tile by identifying a number of reference values. All pixels in the tile are then coded as an index to a reference value, and component-wise color offsets from that reference value. A typical implementation is to chose the minimum and maximum colors as reference values, similarly to depth offset compression. We can then represent the color range shown in Figure 2.

It should be noted that depth offset compression has one advantage over color offset compression, which is that the min and max depth values are already stored in on-chip memory for Zmax- and Zmin-culling, so we do not have to store the reference values explicitly. This makes offset compression slightly less efficient for color data than for depth data.

2.4 Entropy Coded Pixel Differences

Van Hook suggests compression schemes based on entropy coding of pixel differences [6]. First, he computes the componentwise pixel differences. Although the exact procedure is not specified, the patent indicates that different traversal orders may affect the magnitude of the pixel differences (and in the end the efficiency of the algorithm). This indicates that the pixel differences actually are the differences between the current pixel and the previously traversed pixel. The suggested implementation uses either horizontal or vertical scanline traversal of the tile, based on what gives the best compression.

It is well known that differences between adjacent pixels often have small magnitudes due to the continuous nature of images. Van Hook therefore proposes a variable bit length coding of the differences, which he refers to as *exponent encoding*. The general idea is to represent a value as $s(2^x - y)$, where $y \in [0, 2^{x-1} - 1]$, and s is a sign bit. In order to compress this value, $x+1$ is stored using *unary encoding*, which simply amounts to storing $x+1$ bits set to one followed by a terminating zero-bit. For example, $x+1 = 4$ is encoded as 11110_b. Normal binary encoding is used for s and y. The reason for encoding $x+1$ instead of x is that the encoding is not capable of representing a zero value. This special case is flagged when $x+1$ is set to zero.

To illustrate the exponent coding with an example, assume we want to encode the value $\pm 5 = \pm(2^3 - 3)$. The unary encoding of $x+1$ is again 11110_b. The y-value will be in the range $[0, 2^2 - 1]$, so it can be represented using two bits with binary encoding, which gives us 11_b. Finally, we need to store the sign bit s in one bit. The final encoded value therefore becomes $11110s11_b$.

Exponent coding requires a very large amount of bits for values with large magnitudes. Van Hook therefore suggests using exponent coding only for difference values in the range $[-32, 32]$, remaining values are encoded using 16 bits, the first 8 bits must be set to 11111110_b to separate the exponent coded, and binary coded values. The full encoding is shown in the following table.

Code	Representable value
0_b	0
$10s_b$	± 1
$110s_b$	± 2
$1110sx_b$	$\pm[3,4]$
$11110sxx_b$	$\pm[5,8]$
$111110sxxx_b$	$\pm[9,16]$
$1111110sxxxx_b$	$\pm[17,32]$
$11111110xxxxxxxx_b$	8-bit absolute value

A strong feature of this scheme is that it allows for adaptive bit rate inside a tile.

3 A New Exact Color Buffer Compression Algorithm

In this section, we present a new exact, i.e., lossless, color buffer compression method. The algorithm operates on tiles, which are typically 8×8 pixels.

Note that the color buffer needs to be sent to the display in uncompressed form. Hence, there is a direct benefit from having color buffer decompression implemented in the display controller, or in any of the hardware processing blocks prior the display controller. For example, most mobile phones already have some type of display processing block which provides features like scaling, overlay, color depth transform, etc. A color buffer decompressor would fit there as well.

3.1 Reversible Color Transforms

Our new algorithms share the fact that they operate in a luminance-chrominance color space instead of the standard RGB color space. It is well-known in image and video compression that this typically enables more efficient compression due to the decorrelation of the RGB channels.

In addition, it also enables the use of slightly different compression schemes for the luminance and the chrominance components [14]. This could potentially be useful since rendered gaming scenes often provide most details and dynamics in the luminance component.

Since we need lossless compression, the color space transform needs to be exactly reversible. We have chosen the reversible color transform RGB to Yc_oc_g introduced by Malvar and Sullivan [10]. Transforming from RGB to Yc_oc_g is done as shown below:

$$\begin{aligned} c_o &= R - B \\ t &= B + (c_o >> 1) \\ c_g &= G - t \\ Y &= t + (c_g >> 1). \end{aligned} \tag{1}$$

Transforming back is as simple:

$$\begin{aligned} t &= Y - (c_g >> 1) \\ G &= c_g + t \\ B &= t - (c_o >> 1) \\ R &= B + c_o. \end{aligned} \tag{2}$$

Note that if the RGB-components are stored using n bits each, the Y-component will require n bits, and the chrominance components $n+1$ bits. So the price to pay for having a lossless reversible color transform is a small data expansion of two bits. Malvar and Sullivan also showed that this transform in certain video contexts can provide better compression ratios compared to RGB and Yc_rc_b. Note also that

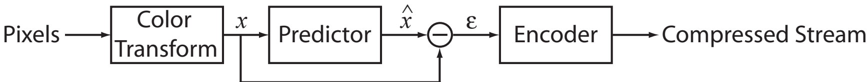


Figure 3: Overview of our compression algorithm.

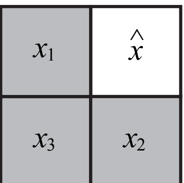
the commonly used standard Yc_rc_b transform is not, in general, reversible without loss.

An alternative color transform to Yc_oc_g would be the exactly reversible component transformation (RCT) from the JPEG2000 standardization [9]. We have empirically concluded that, for our algorithm, these color transforms are roughly equal in terms of efficiency. Our experiments also showed that the use of these color transforms improved the compression rate of our algorithm by about 10% compared to an implementation in *RGB* space. We therefore think that using a color transform is well motivated.

3.2 The Algorithm

Our lossless compression algorithm is inspired by the LOCO-I algorithm [17]. In our implementation, we work on 8×8 pixel tiles, but it should be straightforward to apply it to other tile sizes as well. The flow of our algorithm is illustrated in Figure 3. In a first step, we predict the color of each pixel based on neighbors which will be decompressed prior to the current pixel. The predicted colors are then subtracted from the actual colors to produce error residuals. Just like the differences used by Van Hook, these residuals are generally of small magnitude, and we entropy encode them using Golomb-Rice coding. Next, we describe the details of these steps.

We use the same predictor as Weinberger et. al [17]. The color, $\hat{x}$, of a pixel is predicted as specified by Equation 3 below, and based on the colors of its three neighbors shown in the figure to the right. Note that the two first cases of the equation perform a very limited form of edge detection, in which case the color is predicted based on just one of the neighbors.



$$\hat{x} = \begin{cases} min(x_1, x_2), & x_3 \geq max(x_1, x_2) \\ max(x_1, x_2), & x_3 \leq min(x_1, x_2) \\ x_1 + x_2 - x_3, & \text{otherwise.} \end{cases} \tag{3}$$

For the pixels along the lower and left edge of a tile, we only have access to one of the neighbors. In that case, we simply use the color of that neighbor as the predicted color. In addition, we use the constant zero to predict the value of the lower left pixel in the tile. The effect is that the first error residual will be given the same value as the lower left pixel.

Given these predicted values, we compute error residuals and wish to encode them using as few bits as possible. The residuals are generally of small magnitude, mixed with relatively unfrequent large values. These latter values are typically found for discontinuity edges, or where the behavior of the predictor does not match the structure of the image. We encode the residuals using a Golomb-Rice [15] coder, which is a variable bit-rate coding method similar to the exponent coding described in Section 2.4.

In Golomb-Rice encoding, we encode a residual value, $\varepsilon = x - \hat{x}$, by dividing it with a constant 2^k. The result is a quotient q and a remainder r. The quotient q is stored using unary coding, and the remainder r is stored using normal, binary coding using k bits. To illustrate with an example, let us assume that we want to encode the values 3,0,9,1 and assume we have selected the constant $k = 1$. After the division we get the following (q,r)-pairs: $(1,1),(0,0),(4,1),(0,1)$. As mentioned in Section 2.4, unary coding represents a value by as many ones as the magnitude of the value followed by a terminating zero. The encoded values therefore becomes $(10_b,1_b),(0_b,0_b),(11110_b,1_b),(0_b,1_b)$ which is 13 bits in total.

In our compression algorithm, we compute the optimal Golomb-Rice parameter k for each 2×2 pixel sub-tile using an exhaustive search. We also detect the special case, when the quotients of all values in the sub-tile is zero. This gives us the opportunity of removing the terminating zero-bit, which would otherwise be introduced by the unary coding.

We empirically examined the frequencies of different values of k, and when the special case was used. Our results indicate that k is relatively evenly distributed in the range [0,6] while the special mode was almost only used in the case $k = 0$, which is equivalent to that the whole sub-tile consists only of zero values. With this in mind, we encode each 2×2 sub-tile as a 3-bit header in which we store the value of k. If $k = 7$ the whole sub-tile is zero and we store no more data, and in the other cases the header is followed by the Golomb-Rice coded componentwise residuals.

We present the results of our lossless compression algorithm in Section 5.

Discussion Using exhaustive search to find the best Golomb parameter may seem too expensive for a real-time compression algorithm. However, we want to point out that the search is limited to 8 unique cases that can be evaluated in parallel. Furthermore, it is very inexpensive to evaluate the size of a value after it has been Golomb encoded. This requires just one shift and one addition.

One might also argue that the cost of a variable bit rate compressor is too high for practical use, but we believe it is realizable. Trying to encode a full 2048 bit vector in a single cycle is too expensive, but if we limit ourselves to compressing one sub-tile per cycle we get a more manageable 0-128 bits to write. A tile would then take a total of 16 clock cycles to compress, a delay that could most likely be hidden using pre-fetching [7]. To put this figure in perspective, the expected memory latency reported in the CUDA programming guide [1] is 200-300 cycles.

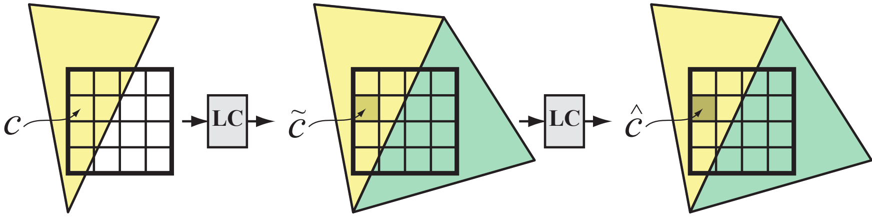


Figure 4: Illustration of tandem compression. Left to right: first a triangle is written to a tile. For one pixel, we track its original color, c. After lossy compression (LC), we obtain an approximation, $\tilde{c}$. However, when a second triangle is written to the tile, $\tilde{c}$ may be compressed again, with another loss of information, so we get yet another color, $\hat{c}$.

4 Error-bounded Approximate Compression

The obvious reason to use lossy (approximate) compression algorithms is that you are allowed to throw away information in the compressed signal, and this can make for substantially higher compression ratios. If done well, the visual impact can be marginal. Since a rather big amount of power is required to drive the capacitances of the buses to off-chip memory, battery-driven mobile devices, in particular, will benefit from lossy buffer compression. It should be noted that for both mobile devices and for desktop GPUs, we may also get higher performance due to better utilization of the memory bandwidth resources.

As argued in the introduction, lossy techniques are used in many different algorithms for graphics, video, and imaging. The prime example is probably digital TV, where we put up with pretty poor approximations in the encoded video stream. It is therefore a bit surprising that there has been no documented attempts to use approximate compression for the color buffer.

The reason for this might be that it is possible to get *unbounded*[1] errors. This can occur when lossy compression (LC) is applied several times, e.g., once per triangle written to a tile. See Figure 4, where the concept of *tandem compression* is illustrated. To counteract this, we need an error-bounded algorithm with precise control of the accumulated error. This is the topic of the next subsection.

Note that buffer compression & decompression must be symmetric, i.e., execute in about the same amount of time, since these procedures run in real time inside the GPU. This means that the majority of all (lossy) texture compression schemes immediately disqualify, since compression often takes several seconds or even minutes.

[1] Here, we used the term "unbounded" to indicate a maximum error in a value. Assuming eight bits, this happens when an original value of 255 is compressed into 0, for example.

4.1 The Error Control Mechanisms

To guarantee that the introduced error stays within bounds, we need to gauge and track the accumulated error in the image being rendered.

Our approach is to calculate and update an accumulated error measure, τ_{accum}, per tile, as illustrated to the right. As an example, we could use the accumulated mean square error. This measure is stored together with the compressed tile parameters. For even more precise control, more than one error measure can be tracked and stored. For example, it may make sense to track and store a maximum error level, which is normally a more conservative error metric than the mean square error metric. When the accumulated tile error measure has reached a configurable upper limit (threshold), τ_{thresh}, the compression stage reverts to lossless compression only in the following compression steps.

For each tile:
Read τ_{accum}
Calculate τ_{new}
$\tau_{accum} + \tau_{new} > \tau_{thresh}$?
Yes
Lossless compress
No
Lossy compress
τ_{accum} += τ_{new}
Store new τ_{accum}

This can be done by having a conditional lossy compression stage, meaning that each time an error is about to be introduced, e.g., due to when sub-sampling or quantization, we test if the updated accumulated error measure exceeds a configurable threshold τ_{thresh}. If it is still less than the threshold, we use the approximation. Otherwise, we revert to lossless compression (and do not update τ_{accum}).

Note also that if we have reverted to lossless compression, we can go back to lossy compression if all pixels are written to inside a tile.

Our approach is conservative, in that the error (in the error metric used) never grows larger than the thresholds. Hence, the introduced errors are bounded, which effectively reduces the visual quality impact (given the configured error thresholds are low enough).

4.2 A Lossy Algorithm

We have chosen to track and store the accumulated root mean square error (RMSE) error per tile, τ^{rmse}_{accum}. This measure is quantized to 16 levels and stored together with the compressed parameters as 4 bits per tile. Note that the choice of this error metric also bounds the maximum error in a tile. Assume the threshold is τ^{rmse}_{thresh}, and that we have n pixels in a tile. Some simple calculations gives:

$$\tau^{max}_{thresh} = \sqrt{n} \times \tau^{rmse}_{thresh}. \quad (4)$$

As an example, if $\tau^{rmse}_{thresh} = 2$ with 8×8 pixel tiles, we have $\tau^{max}_{thresh} = 8 \times 2 = 16$. In a practical implementation it may make more sense to use MSE instead of RMSE, since this avoids the expensive square root. However, that also doubles the number

of bits for the accumulated error. Another useful error metric is the sum of absolute differences (SAD).

When it comes to the actual approximation, we have taken a gentle approach and use only (conditional) 2×2 subsampling of the chrominance components. Higher compression rates can of course be obtained with more "brutal" subsampling, quantization, and other lossy methods.

Since the human visual system (HVS) is more susceptible to errors introduced in the luminance than the chrominance components, we use lossless compression for the luminance and lossy compression for the chrominance components respectively. This results in a good compromise between high visual quality and high compression ratios.

It should be noted here the benefits of utilizing an exactly reversible color transform. This enables the possibility to mix lossless and lossy compression freely in the same compression block. For example, it enables us to have lossless compression of the luminance components and simultaneously have lossy compression of the chrominance components[2]. When the error threshold is reached, we can revert to lossless-only chrominance compression to effectively stop further error build-up. Furthermore, if a non-exact color transform is used, that would introduce further errors, and the error accumulation mechanism would have to deal with that as well. With our approach, that can be avoided altogether.

Decompression is done in the opposite direction. The sub-sampled chrominance components are up-sampled by simply copying the sub-sampled component to the corresponding components in the 2×2 quad.

5 Results

We have evaluated our compression algorithms using a software based simulation framework, which implements a tile-based triangle rasterizer with a modern color buffer architecture. It also simulates tile caching, using a 1 kB fully associative cache, and implements the color compression algorithms described in this paper. In addition, we used a logging OpenGL driver to record the rendering calls from actual games. This means that our results include the full, incremental, rasterization process of the games. They are not just compressed screenshots.

To benchmark the color compression algorithms, we used the four test scenes shown in Figure 5. The first scene is designed to stress high contrast colors, and the following two scenes are relatively colorful scenes taken from Quake 3 maps.[3] The final scene features complex particle effects with blending, and is taken from the game Quake 4. It should be noted that this scene use blending based on the alpha value stored in the color buffer. Therefore, we compress the full RGBA components for this scene, while we only compress the RGB components for the

[2]Chrominance errors can spread into the luminance channels due to tandem compression.

[3]The maps have been taken from the Quake 3 add-on "Urban Terror".

	Ducks	Square
Plane	1.13:1	1.43:1
Color Offset	1.90:1	2.06:1
Entropy Coding	2.09:1	2.60:1
Our	2.64:1	2.87:1

	Car	Quake4
Plane	1.40:1	1.47:1
Color Offset	2.15:1	2.30:1
Entropy Coding	2.66:1	2.93:1
Our	3.36:1	3.05:1

Figure 5: Evaluation of the compression algorithms: the table shows compression ratios for our test scenes ("Ducks", "Square", "Car", and "Quake4") using exact compression algorithms. We computed the compression ratios as the average compression ratio for rendering resolutions 320×240, 640×480 and 1280×1024 pixels. The algorithms scaled similarly with varying resolutions.

remaining scenes. This shows that all algorithms are suitable for compressing alpha data as well.

Note that we will refer to each compression algorithm by the names we used in Section 2. See the titles of each subsection.

5.1 Exact Compression

The effective compression ratios of the different exact algorithms are presented in Figure 5. We used 8×8 pixel tiles and variable bit-rate encoding for offset compression, entropy coding, and our algorithm. Variable bit-rate coding comes very natural to our algorithm and entropy coding. For offset compression, we

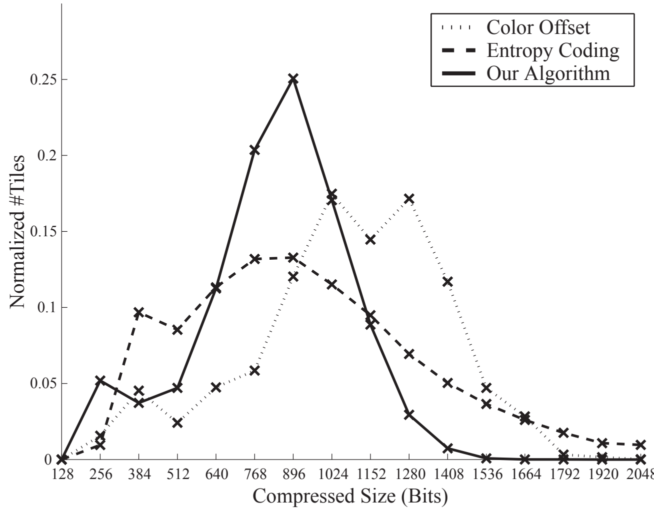


Figure 6: A normalized histogram of the number of tiles that are compressed to a given size (in multiples of 128 bits), using different algorithms. We use 8×8 pixel tiles, which means that 2048 bits indicate uncompressed tiles. The histogram is based on an average over all our test scenes. Note that our algorithm has a distinct peak, which makes it efficient when only a few compressed sizes can be used. It is of course also essential that the peak is located at good compression ratios, i.e., to the left in this diagram.

implemented variable bit-rate in the sense that we use no fixed bit allocations for the offsets. We simply use the least amount of bits that is capable of representing the largest offset in the tile.

The plane compression algorithm is special in that we used the specified 2×4 tile size and only the two modes specified in the patent. We would like to emphasize that this algorithm is disfavored in this evaluation since it is so specialized. A generalized version of plane compression may generate better results, but this is left for future work.

We would also like to point out that our measurements (Figure 5) do not take hardware limitations, such as the width of the memory bus, into account. Furthermore, in most hardware implementations, we can only afford a few different compressed sizes, since the compressed size of a tile is typically stored in on-chip memory so that we know beforehand how many memory words to read. In order to measure the algorithms performance with respect to these limitations, we computed the compressed size histograms shown in Figure 6. Note that we have left out plane compression since it is so specialized, and only allows for 128 or 256 bits per tile.

Next, we show how to interpret this diagram with an example. Assume that we allow two fixed sizes for tiles: 1024 bits for compressed and 2048 bits for uncompressed. In this case, the number of tiles compressed to 1024 bits will be the integral from 0 to 1024 over the histogram, while the uncompressed tile will be

Color Offset		
#Sizes	Best sizes (Bits)	Effective Compression
1	1280	1.43:1
2	1024,1408	1.58:1
3	896,1152,1408	1.61:1
∞		2.04:1
Entropy Coding		
#Sizes	Best sizes (Bits)	Effective Compression
1	1024	1.52:1
2	768,1280	1.75:1
3	640,1024,1408	1.88:1
∞		2.45 : 1
Our Algorithm		
#Sizes	Best sizes (Bits)	Effective Compression
1	1024	1.78:1
2	896,1152	2.04:1
3	640,896,1152	2.17:1
∞		2.88:1

Table 1: The tables show how the algorithms perform when given a number of allowed compressed sizes, as well as what selection of sizes that worked best for our test suite. Note that our algorithm performs very well even with very few compressed sizes.

the integral from 1024 to 2048. Using the histogram, we can easily find the best n sizes for each algorithm using an exhaustive search. In Table 1, we present the best compressed sizes for $n = 1, 2, 3$. Note that an extra uncompressed size always needs to be available as a fallback.

5.2 Approximate Compression

In Figure 7, we show the results from our experiments with approximate compression. The Quake 4 scene is excluded since alpha handling is currently not implemented in the lossy part. As can be seen, the additional compression gains can be quite substantial. We can gain an additional 25–60% compression by allowing approximate compression. The visual impact is normally small as can be seen in Figure 8 and in the $SSIM_{rgb}$ plot (Figure 7c). See Section 5.3 for more information on $SSIM$. However, the "ducks" scene clearly shows artifacts (Figure 9) already for small levels of τ^{rmse}_{thresh}. This is due to that we use chrominance sub-sampling, which makes chrominance leak out to surrounding pixels. For a case like this, a more conservative error metric could be used, e.g., a maximum error threshold. This would decrease the effect of these artifacts. Our most important contribution for lossy buffer compression is the error control mechanism, and we believe our results shows that it works well, and that it can keep high image quality. However, more research is clearly needed on lossy compression algorithms.

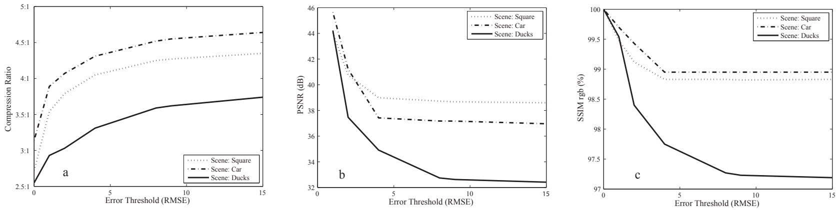


Figure 7: Approximate (lossy) compression results for three of the test scenes (average of three rendering resolutions 320×240, 640×480 and 1280×1024 pixels). From the left: a) compression ratio vs. τ_{thresh}^{rmse}, b) PSNR vs. τ_{thresh}^{rmse}, and c) $SSIM_{rgb}$ vs. τ_{thresh}^{rmse}.

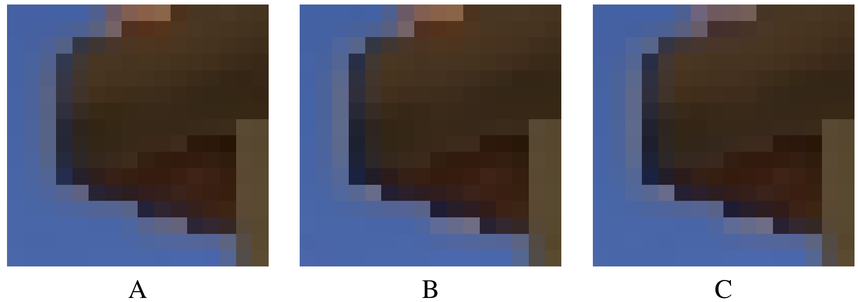


Figure 8: Crops from the "Square" scene. A: original, B: $\tau_{thresh}^{rmse} = 4$, $PSNR = 39.0$ dB, $SSIM_{rgb} = 98.8\%$, compression ratio = 4.1:1, C: $\tau_{thresh}^{rmse} = 15$, $PSNR = 35.6$ dB, $SSIM_{rgb} = 98.8\%$, compression ratio = 4.3:1.

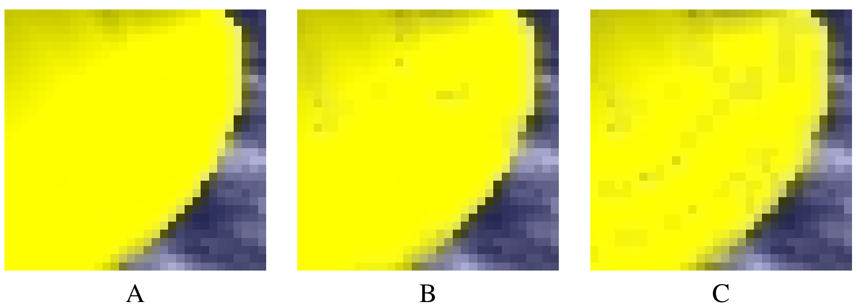


Figure 9: Crops from the "Ducks" scene. A: original, B: $\tau_{thresh}^{rmse} = 4$, $PSNR = 34.9$ dB, $SSIM_{rgb} = 97.8\%$, compression ratio = 3.3:1, C: $\tau_{thresh}^{rmse} = 15$, $PSNR = 32.4$ dB, $SSIM_{rgb} = 97.2\%$, compression ratio = 3.8:1.

5.3 Structural Similarity Index - SSIM

In addition to the common error metric, PSNR, we also use the structural similarity index, SSIM, as suggested by Wang et. al [16]. This is a visual quality metric which attempts to mimic the human visual perception. The SSIM index is a number between 0% and 100%, where 100% is perfect similarity. Note that the SSIM index is normally calculated using the luminance alone. In order to get the errors in all three color channels, R, G and B respectively, we have chosen to calculate the SSIM index for the R,G and B channels independently and combining them into a single number, $SSIM_{rgb}$, according to:

$$SSIM_{rgb} = 0.2126 * SSIM_R + 0.7152 * SSIM_G + 0.0722 * SSIM_B, \tag{5}$$

where the weights comes from ITU-BT.709 [8].

6 Conclusions and Future Work

Color buffer compression is available in almost all (if not all) GPUs, but up until now, this type of algorithms have not been described in the literature. By providing an overview of existing algorithms, we now have an important stepping stone in place, which is needed to invent new algorithms.

In addition, we have presented new algorithms based on a decorrelated color transform, which is also exactly reversible. Our results show that this can improve the compression ratio compared to other algorithms. Since it is well-known in the image and video compression community that the human visual system is more sensitive to luminance than chrominance, we have also done some initial results on approximate color buffer compression with this reversible transform.

We note that it is very important to keep the accumulated error under strict control, and we presented a simple mechanism to do this. We realize that approximate compression is a feature that must be turned off for some applications, but for, e.g., gaming on mobile devices, it can be very valuable with a longer use time on the battery with a only slight degradation in image quality.

We have only experimented with simple compression algorithms for approximate color buffers, and for future work, there is much to learn and transfer from the image and video processing field. We have started to investigate more sophisticated and fine-grained sub-sampling and quantization schemes. There is definitely room for inventing new algorithms. High dynamic range (HDR) color buffer compression is also an interesting topic for further studies.

In our paper, we have not handled multi-sampling, but several of the techniques [4, 11] for this can be merged relatively quickly into our work. Finally, we note that lossy depth buffer compression might not be feasible, due to the artifacts that can arise when surfaces intersect. However, this could be worth further investigation.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, Ericsson, and NVIDIA's Fellowship program. Thanks to the anonymous reviewers for their helpful comments.

Bibliography

[1] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide. http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_Programming_Guide_0.8.pdf.

[2] Kurt Akeley. RealityEngine Graphics. *Readings in computer architecture*, pages 507–514, 2000.

[3] Tomas Akenine-Möller and Jacob Ström. Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics,*, 22(3):801–808, 2003.

[4] Gordon M. Elder. Method and Apparatus for Anti-Aliasing using Floating Point Subpixel Color Values and Compression of Same. In *US Patent Application 2006/0188161 A1*, 2006.

[5] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.

[6] Timothy J. Van Hook. Method and Apparatus for Compression and Decompression of Color Data. In *US Patent Application 7039241 B1*, 2006.

[7] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a Texture Cache Architecture. In *Graphics Hardware*, pages 133–142, 1998.

[8] SG06 ITU-R. ITU, Recommendation BT.709 : Parameter values for the HDTV standards for production and international programme exchange. In *ITU-R, BT.709*, 2002.

[9] JPEG2000. ISO/IEC 15444-1:2000, JPEG 2000 Image Coding System, Annex G2, Reversible Component Transformation. In *ISO/IEC 15444-1:2000*, 2000.

[10] Henrique Malvar and Gary Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. In *JVT-I014r3*, 2003.

[11] Steven E. Molnar, Bengt-Olaf Schneider, John Montrym, James M. Van Dyke, and Stephen D. Lew. System and Method for Real-Time Compression of Pixel Colors. In *US Patent Application 6825847 B1*, 2004.

[12] Steven L. Morein and Mark A. Natale. System, Method, and Apparatus for Compression of Video Data using Offset Values. In *US Patent Application 2003/0038803 A1*, 2003.

[13] John D. Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley, 2005.

[14] Anton Pereberin. Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99*, pages 195–199, 1999.

[15] Robert F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 22, Jet Propulsion Lab, 1979.

[16] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P.Simonelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing,*, 13(4):600–612, 2004.

[17] Marcelo J. Weinberger, Gadiel Seroussi, and Guiellermo Sapiro. LOCO-I: A low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conference*, pages 140–149, 1996.

Paper VII

PCU: The Programmable Culling Unit

Jon Hasselgren Tomas Akenine-Möller

Lund University

{jon|tam}@cs.lth.se

Abstract

Culling techniques have always been a central part of computer graphics, but graphics hardware still lack efficient and flexible support for culling. To improve the situation, we introduce the programmable culling unit, which is as flexible as the fragment program unit and capable of quickly culling entire blocks of fragments. Furthermore, it is very easy for the developer to use the PCU as culling programs can be automatically derived from fragment programs containing a discard instruction. Our PCU can be integrated into an existing fragment program unit with a modest hardware overhead of only about 10%. Using the PCU, we have observed shader speedups between 1.4 and 2.1 for relevant scenes.

ACM Transactions on Graphics, 26(3):92, 2007.

1 Introduction

Faster rendering is a major field in computer graphics research due to the ever-increasing demands of applications. A prime example of such applications are games with high image quality, complex shading, and detailed geometry. In real-time graphics, programmable vertex, geometry, and fragment shaders provide the programmers with more flexibility for the rendering tasks, but higher image quality and using more complex shading incur a cost in performance.

The fragment pipeline is often considered to be the major bottleneck in a GPU [1], and hence it is a clear candidate for acceleration algorithms. At a high level, the task of the fragment pipeline is to *access memory* and *execute instructions* in order to produce the output (e.g., color, depth, stencil) of a fragment. The focus of our research is on the latter.

To make shader programs run faster, one can insert `KIL` (fragment discard) instructions at appropriate places, in order to provide an *early-out*. In practice, these

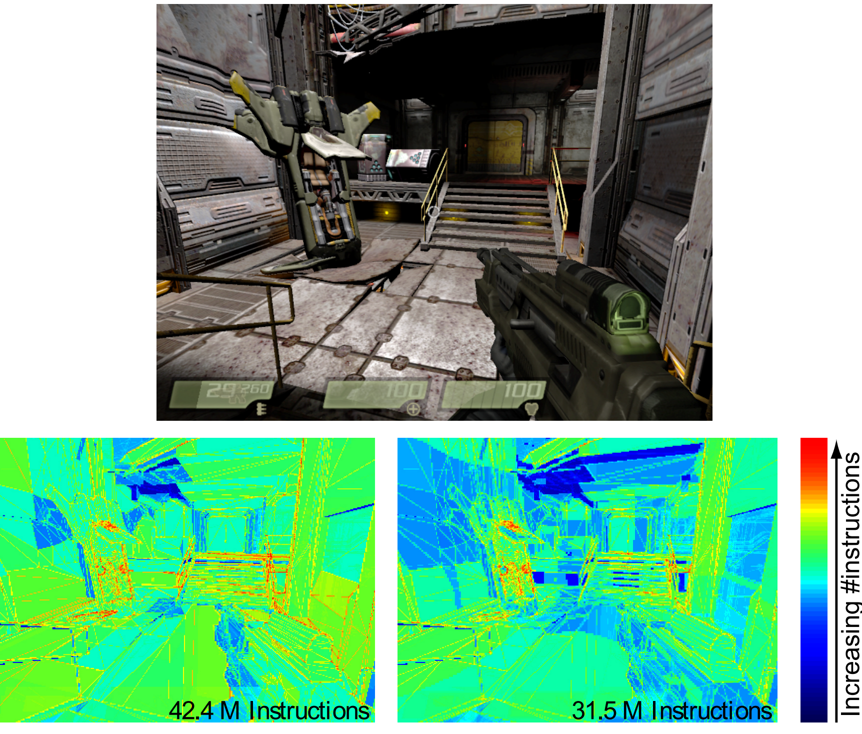


Figure 1: Our programmable culling unit applied to Quake 4 (courtesy of id Software). Top: Rendered image. Bottom left: False color image of the number of instructions executed per pixel using a standard GPU. Bottom right: Similar visualization using our programmable culling unit.

instructions seldom make the execution faster on contemporary GPUs, due to the underlying hardware design. In addition, a GPU designer can employ pipelining and parallelization techniques for faster shader instruction execution. Note, however, that the fastest instruction is the one that never is executed to begin with. Hence, the target of our paper is to avoid executing, i.e., *cull*, a substantial amount of instructions which do not contribute to the final image.

To that end, we present the *programmable culling unit* (PCU). The idea is to execute a *cull program* on tiles of pixels before the per-pixel shader processing starts. In many cases, a conservative decision can be made based on the outcome of the cull program, and per-pixel executions completely avoided. For example, if all pixels in a tile are in shadow, it may be possible to determine this on a tile-level, and shading instructions can be avoided for all pixels in the tile. Our PCU is flexible in that the programs can be automatically generated from the fragment shader programs (i.e., transparent to the programmer), or written explicitly. The core idea of the PCU is to use *interval arithmetic* [19] to bound the value of a floating-point variable, and let the fragment pipeline operate on such bounded number representations.

The PCU makes existing fragment programs run faster, and opens up new possibilities for culling algorithms specifically written for the PCU. In addition, a major advantage is that the rendering programmer can work in a more carefree way. For example, many current games are doing back-face culling from the light source on the CPU, before sending the geometry to the GPU for shading. Such culling can be done automatically on a per-tile basis using the PCU. For an example of how our PCU can improve performance, even in a highly optimized computer game, see Figure 1.

A prototype hardware implementation of a fragment pipeline has been developed, and the increase in size is about 10%. We provide extensive simulation results using a variety of benchmarks, such as commercial games and relevant shader effect demos. For these, we observed that 29–52% of the instructions can be avoided, which implies a speedup of $1.4 - 2.1\times$. The memory bandwidth utilization is also decreased by approximately 15%.

2 Previous Work

In our research, we have used *interval arithmetic* [19] (IA), which is an arithmetic defined on intervals rather than real numbers. It dates back to the early 20th century, and is now an established field of research. IA has already been used on several occasions in the field of computer graphics, and we will mention only the ones that relates to our research. For a more in-depth overview, consult Kearfott's survey [13].

The Achilles heel of interval arithmetic is that the width of an interval can grow rapidly for complex equations. Affine arithmetic [5] reduces this problem by tracking all linear dependencies in an equation and evaluates them exactly. Higher order

dependencies are solved using linear approximation and by adding an extra term that bounds the approximation error.

Culling To speed up execution in both software and hardware, it is common to use culling algorithms. Hardware culling is typically limited to fixed function mechanisms, such as back-face culling or hierarchical depth culling [11, 20].

Hierarchical depth culling (HDC) is an optimization of depth buffering. When a tile of fragments is about to be rasterized, a conservative test is performed to prove whether that tile is covered by the contents in the depth buffer. If it is, the entire tile can be skipped which results in performance gains. Aila et al. [1] improve the culling rate of HDC by introducing a delay stream, which basically performs depth buffering first, and then puts the triangles on hold until the "occluding power" of the depth buffer has been built up. Another variant of HDC, called Z-min culling [2], avoids depth reads when a tile of fragments being generated are in front of the contents of the depth buffer.

In an attempt to implement more flexible hardware culling, Purcell et al. [22] introduced *computation masks*. The basic idea is to let a fragment program set up the depth buffer so that all fragments that can be culled fail the depth test. The hierarchical depth culling hardware will then perform the actual culling. The weak points of computation masks are all related to overlapping geometry. First, the computation mask will overwrite the content of the depth buffer, which makes it hard to use depth testing in conjunction with computation masks. Second, the computation mask is given in screen space with just one entry per pixel. This makes it difficult to make different culling decisions for overlapping geometry.

Occlusion queries provide functionality to query the graphics hardware for the number of fragments that pass the depth test. The prime example of using occlusion queries is to draw the bounding box of a mesh. If no fragments pass the depth test, the entire mesh is occluded and does not need to be rendered. These algorithms require intervention by the programmer, and it can be difficult to achieve good performance due to the latency and overhead of a query, or require sophisticated CPU algorithms [3]. In general, occlusion queries are orthogonal to our work.

Programmable Shaders The core feature of modern GPUs is the availability of programmable shaders. These are commonly divided into geometry shaders [4], vertex shaders [14], and pixel/fragment shaders. However, as the programming languages are very similar, the implementation trend is to use unified shader units [8], which can execute the program for any type of shader. This enables more efficient load balancing between the different tasks. For instance, if a scene contains many small triangles, more processing units can be allocated for vertex processing. On the other hand, if we draw a typical GPGPU full-screen quad, all units can be allocated for fragment processing.

There has been some work made on software evaluation of programmable shaders

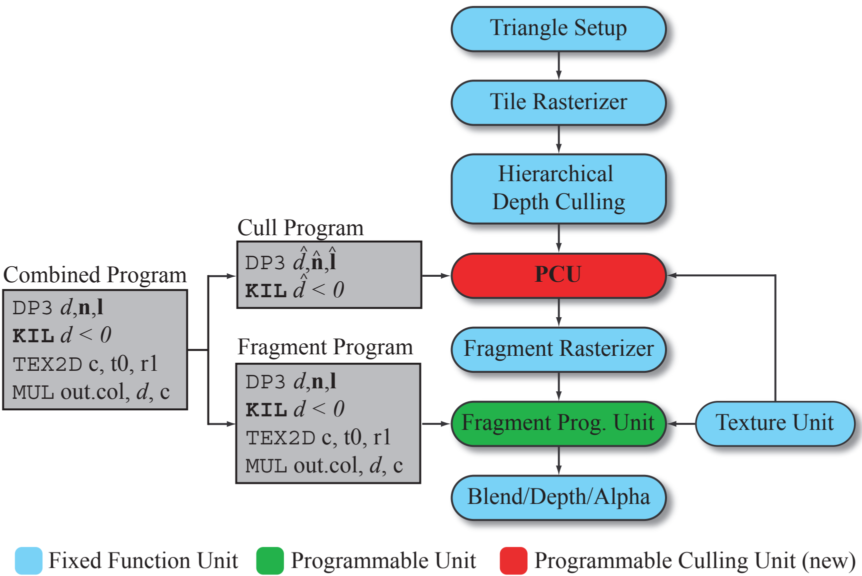


Figure 2: Behavioral model of our proposed rasterization architecture. We add a new programmable unit between the depth culling unit and the fragment-producing unit. Our unit executes a cull program, and decides whether a tile can be culled or not based on the results of the program.

using interval or affine arithmetic. Greene and Kass [10] evaluated shade trees [6] using interval arithmetic. This was used to compute bounds for the final color over a surface area element. The width of the bounds was used to guide adaptive refinement antialiasing. Heidrich et al. [12] used affine arithmetic to evaluate procedural shaders written in RenderMan. Doing so enabled them to sample a procedural shader over an area element. The authors argue that this can be used to, for instance, adaptively sample procedural textures, or taking material into account in bounded lighting equations [23]. Moule and McCool [21] use interval arithmetic to bound the approximation errors when performing adaptive tesselation of displacement maps. As a hierarchical representation of the displacement map function, they use a mipmap structure of intervals.

3 Programmable Culling Unit Overview

In this section, we give a high-level overview of our programmable culling unit, and motivate some early design choices.

Figure 2 shows a behavioral overview of our novel rasterization architecture, which includes the PCU as a new unit. The PCU is the last unit which processes entire tiles of fragments. It conservatively decides whether to terminate the tile or send

it down the pipeline for further per-fragment processing. The fundamental difference, compared to hierarchical depth culling (covered in Section 2), is that the PCU bases its decision on the output from a shader program execution, rather than fixed function computations.

One of our main ambitions was to make it as simple as possible for the programmer to take advantage of the PCU. With our approach, the programmer does not have to care about new programming languages, writing conservative algorithms, or taking tile sizes into account. The whole PCU can simply be seen as a very fast `KIL` (fragment discard) instruction that operates on a per-tile basis.

As an example, consider the "combined program" in Figure 2. This program performs diffuse lighting by computing the dot product between the normal and light vectors, and multiplies the result by a diffuse material coefficient stored in a texture. In this case, the programmer added a `KIL` instruction to terminate fragments where the normal does not face the light. We see this `KIL` instruction as an opportunity for culling a whole tile of fragments. In order to do so, we need a way to conservatively prove that the condition for the `KIL` instruction is fulfilled for every fragment within the tile. From this follows that we must also be able to conservatively evaluate the `DP3` instruction, since the `KIL` instruction depends on its result. We must also be able to find conservative bounds of the input (the normal and light vectors in this case) for a whole tile, since the `DP3` instruction in turn depends on these values.

In order to implement this chain of conservative evaluations, we base the PCU on the same instruction set as the fragment program unit. However, instead of floating-point variables as source and destination registers to an instruction, we use *intervals* and implement the instruction using the principles of interval arithmetic [19].[1] As a simple example, consider a standard `ADD` instruction:

$$\texttt{ADD}\ c,a,b \quad \Longleftrightarrow \quad c = a + b \tag{1}$$

For the corresponding PCU interval instruction, we replace the operands by intervals, $\hat{a}, \hat{b}, \hat{c}$, where an interval, e.g., $\hat{a}$ is defined as:

$$\hat{a} = [\underline{a}, \overline{a}] = \{x \,|\, \underline{a} \le x \le \overline{a}\}. \tag{2}$$

The PCU interval `ADD` instruction is then:

$$\texttt{ADD}\ \hat{c},\hat{a},\hat{b} \quad \Longleftrightarrow \quad \hat{c} = \hat{a} + \hat{b}, \tag{3}$$

where the interval addition operation is implemented as:

$$\hat{a} + \hat{b} = [\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\underline{a} + \underline{b}, \overline{a} + \overline{b}]. \tag{4}$$

As can be seen, the result of the interval addition contains all possible results of "normal" additions, or more formally, it holds that $a + b \in \hat{a} + \hat{b}$ given that $a \in \hat{a}$

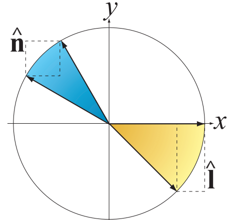
[1] We decided to use interval arithmetic because it is simple to implement in hardware. An alternative approach is to use affine arithmetic [5]. See Section 7 for further discussion.

and $b \in \hat{b}$. It is therefore conservatively correct. In similar fashion, we redefine the behavior of every instruction in the fragment program instruction set. Full details of our interval instructions, including conservative texture lookups, can be found in Appendix A.

In addition to using interval instructions, the input must also be defined as intervals. Therefore, we must be able to compute conservative bounds for quantities interpolated over an entire tile of fragments. This is treated in more detail in Section 4.2.

Given this architecture, we can execute the shader program for an entire tile at a time. When a `KIL` instruction is executed, we determine if the conditional expression is unambiguously fulfilled by checking the interval of the input. In such a case, we can quickly discard the entire tile and its fragments, which saves valuable processing time. Otherwise, per-fragment shader execution follows.

2D Interval Dot Product Example Let us once again revisit the example in Figure 2, while also looking at the figure to the right. For an entire tile of fragments, assume that we have determined that the input interval of the normals is $\hat{\mathbf{n}} = ([-\sqrt{3}/2, -1/2], [1/2, \sqrt{3}/2])$, and the interval for the light vector is $\hat{\mathbf{l}} = ([1/\sqrt{2}, 1], [-1/\sqrt{2}, 0])$, as illustrated in the figure. The dot product between these interval representations results in $\hat{d} = \hat{\mathbf{n}} \cdot \hat{\mathbf{l}} = [-(\sqrt{6}+\sqrt{3})/\sqrt{8}, -1/\sqrt{8}]$ (see the `DP3` instruction in Table A.1). We then reach the `KIL` instruction and note that the operand, $\hat{d}$, can be at most $\overline{d} = -1/\sqrt{8}$. Since this value is strictly less than zero, we can cull this whole tile without executing the fragment program for every fragment. This is the source of the performance gain in our algorithm. □



4 PCU Interaction

In this section, we will treat the PCU as a black box which takes intervals as input and determines whether a tile can be culled. It is important to note that the PCU is a part of a much larger system including both hardware and the driver. In this section, we focus on the interaction between that system and the PCU.

4.1 Driver

Here, we will introduce a new `CUL` instruction, present compilation issues, and discuss optional instruction support. All these features are implemented in the driver.

`CUL` Instruction

The system in Section 3 allows for rapid culling of fragment programs containing `KIL` instructions, but we also introduce a `CUL` instruction that can improve performance in some cases. As an example of when it is useful, consider a fragment program designed for multi-pass shading, such as the program from Figure 2. These programs rarely or never use a `KIL` instruction to discard fragments whose normals face away from the light. The light contribution will be zero anyway, so the `KIL` instruction is not needed for correctness and will in most cases only reduce performance.

In this case, we want the `KIL` instruction to exist only in the cull program, which is exactly how we define the `CUL` instruction. The `CUL` instruction, and all instructions depending on it, will only be propagated to the cull program. Thus, where a `KIL` instruction is always guaranteed to discard a fragment if the condition is fulfilled, a `CUL` instruction may or may not discard the fragment depending on if it belongs to a tile that can be culled. The `CUL` instructions can be removed for hardware not supporting cull programs.

Program Compilation and Separation

A key point of our programmable culling unit is the ease of use. The programmer writes a combined program, and it is up to the driver or compiler to separate the cull and fragment program. This process can be done easily and efficiently using dead code elimination [7]. For the fragment program, we mark the color outputs, depth outputs, and all `KIL` statements as live code. We then perform dead code elimination, efficiently removing all code not contributing to the final result. For the cull program, we mark all `CUL` and `KIL` statements as live, and again perform dead code elimination to remove all code not contributing to the result.

It should be noted that dead code elimination is a lightweight optimization technique that does not introduce any significant performance impact on the driver, even if it is carried out at runtime.

Optional Instruction Support

A convenient feature of our PCU, and culling algorithms in general, is that lack of culling does not compromise the correctness of the result. An implementation may therefore freely omit PCU support for any instructions or features present in the fragment program unit. The driver can then examine all `KIL` and `CUL` instructions and determine if they depend on some unsupported feature. If this is the case, the `KIL` or `CUL` instruction is simply marked as dead code when extracting the cull program.

We use this strategy to handle *dynamic* loops (loops with a variable number of iterations). For simple *static* loops, with a constant number of iterations, we can simply use loop unrolling. However, dynamic loops present more of a challenge,

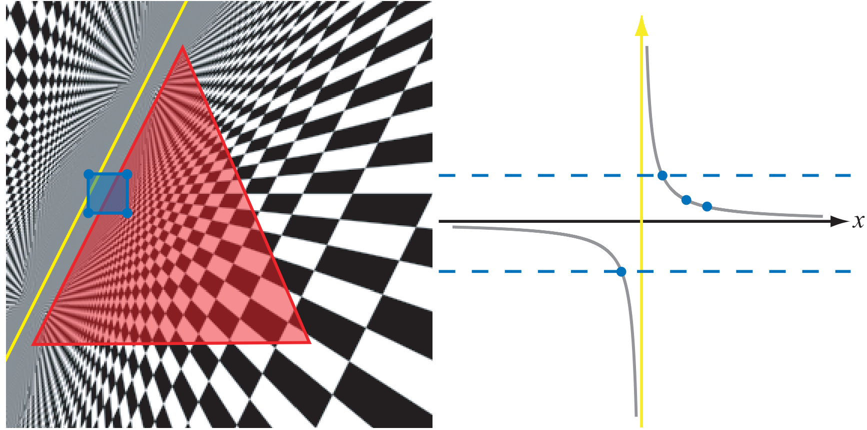


Figure 3: The perspective-correct interpolation problem. The left figure shows the perspective-correct interpolation space of a triangle, when extended outside the triangle boundaries. Note the yellow line where the checkerboard is mirrored. This is where the horizon of the triangle would be projected if it was infinitely large. The right figure shows the perspective-correct interpolation function as gray curve, and the projection line (at $x = 0$) marked in yellow. Note that computing bounds using the corners of the tile (in blue) gives an incorrect result.

and we do not currently support them in the cull program. There are ways of reformulating a dynamic loop on interval form [12], but the problem is that we cannot guarantee that a loop converging on a per-fragment level will also converge when using our interval instructions. There is a possibility of undesired infinite loops.

To support dynamic loops, we believe the best approach would be to extend the instruction set of the PCU with instructions for this purpose, and to force the programmer to write a custom cull program. Thereby, the programmer gets the responsibility to formulate a loops that converge. This is left for future work.

4.2 Hardware

When considering the PCU as a black box, there are still some issues that can be discussed when it comes to hardware. In this subsection, we will first discuss how to compute the input intervals to a tile, and then present a solution to how small triangles can be handled efficiently.

Interpolation

Given an implementation of the instruction set in Appendix A, we may execute a fragment program for a whole tile of fragments. However, in order to do so, we also need to compute bounding intervals for the varying (or interpolated) inputs.

We use a method inspired by the depth bounds computations of hierarchical depth culling. Initially, we compute the value of the varying attribute in all four corners of the tile using interpolation. We then compute the bounding interval of these four values, and call it $\hat{a}_c = [\underline{a_c}, \overline{a_c}]$. We also compute the bounding interval of the varying attribute at the triangle vertices, and call it $\hat{a}_{tri} = [\underline{a_{tri}}, \overline{a_{tri}}]$. The final bounding interval of the varying attribute over the tile can be computed as $\hat{a}_{tile} = [\max(\underline{a_{tri}}, \underline{a_c}), \min(\overline{a_{tri}}, \overline{a_c})]$.

Finally, we must take care of a special case, which is illustrated in Figure 3. Here, perspective-correct interpolation over a triangle is illustrated in form of a checkerboard texture. As can be seen, the texture is mirrored about the yellow *projection line*, which is where the horizon of the triangle would project if it was infinitely large. This mirroring effect is a form of back-projection caused by the division used in perspective-correct interpolation.

Now, assume we wish to compute the bounding interval of some varying attribute over the blue tile, which overlaps the projection line. The right part of Figure 3 shows the perspective-correct interpolation function, as well as the values we get when we interpolate the four corners of the tile. Note that the bounding interval of these corners (dashed blue lines) is obviously incorrect since the function approaches infinity at the projection line.

We handle this special case by setting $\hat{a}_{tile} := \hat{a}_{tri}$ as the bounding interval for tiles overlapping the projection line. One might argue that this interval is overly conservative, but these problematic tiles are so rare that it is hard to motivate more complex computations. In our implementation, we only traverse tiles actually overlapping the triangle, and use perspective-correct barycentric coordinates [17] to do the interpolation. We can easily detect the problematic tiles when computing perspective-correct barycentric coordinates for the corners of a tile. The perspective-correct barycentric coordinates are expressed as a rational function, and if the denominator is less than zero for any of the tile corners, then the tile crosses the projection line.

Higher Level Culling

A potential weakness of our PCU is that the culling is done on a per-tile and per-triangle basis. In the case of micro-polygons, triangles can be smaller than a pixel, which means that executing the cull program will cost at least as much as executing the fragment program. Here, we describe a solution to this problem.

In order to handle micro-triangles efficiently, we use a delay stream like unit [1]. This unit receives triangles in the same order as they are sent to the graphics card. It also keeps an internal state with bounding intervals of the varying attributes. The unit groups triangles as long as the accumulated bounding interval for the *position* attribute is smaller than the size of a tile. This means that grouping is done as long as the bounding box of the triangles is smaller than the size of a tile. When the accumulated triangles reach this limit, we execute the cull program using the accumulated bounding intervals. If the outcome indicates that culling can be done,

the entire set of small triangles is terminated. Otherwise, per-fragment processing commences for each triangle in the set.

Triangles overlapping more than one tile are rendered as usual. That is, we execute the cull program for every tile during rasterization.

Fragment Program Switching

In some cases, such as shader level of detail, it may be convenient to be able to use the cull program to specify which, out of many, fragment program should be executed for the fragments in a tile. A nice example of this is our "Soft Shadows" scene in Section 6. Here, we may skip a considerable amount of shadow map lookups if we can prove that a tile is entirely lit, but we must still perform the lighting on a per-fragment level so we cannot completely cull the tile.

We can support this if the underlying hardware is capable of rapidly switching between a small number of fragment programs. In this case, we simply attach a fragment program index to each cull instruction, and use the corresponding fragment program when rendering the fragments of a tile that has been culled by that instruction.

5 Implementation - Combined Shader Unit

So far, our description of the programmable culling unit has been on a functional level. However, our goal has been to reuse existing hardware when possible, so it is only natural to combine the PCU and the fragment program unit (or the unified shader unit, if such an architecture is used) into a single programmable unit.

Figure 4 shows a block diagram of a possible fragment program unit. The execution pipelines are organized as four separate parallel units, with a direct mapping to a quad of 2×2 fragments. This enables simple approximations of derivatives by finite differences, which is essential for correct mipmapping. The fragment program pipeline is broken down into several steps, and in order to avoid hazards, it is reasonable to assume that the GPU executes at least as many parallel program threads as there are pipeline steps in the program unit.

In order to implement our PCU, we augment the fragment program unit with new logic shown in red in Figure 4. These additions mainly consists of extra control logic before and after the ALU. The logic is responsible for value rerouting, detecting and handling the special cases of Table A.1, and handling the special case for interpolation. There are also additions in the texture unit to handle mipmap computations and filtering, and a final set of min & max units to assemble the result. These units may be taken from the ALU and moved to the end of the pipeline. No extra hardware is needed in this case.

We have implemented a simple fragment program unit augmented with a subset of the PCU, in VHDL. We restricted our hardware implementation to the arithmetic and logic functions, while omitting texturing and interpolation. The reason

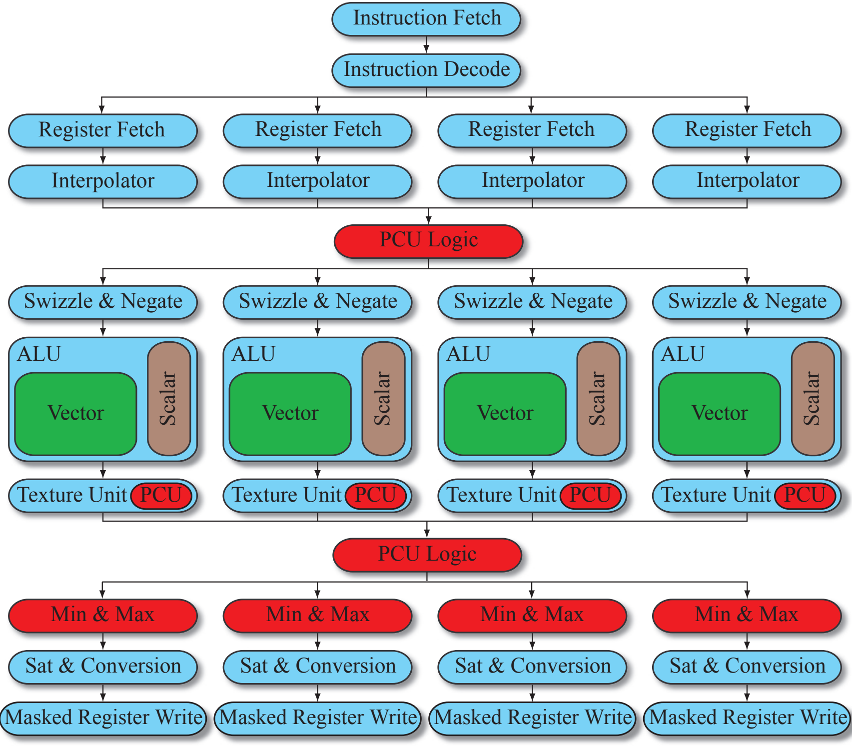


Figure 4: Conceptual diagram of a contemporary fragment program unit which we have extended to a PCU. The new units are marked in red.

for this was that we had no access to peripheral units, such as texture caching, vertex buffers, and so forth. Admittedly, our implementation is only a fraction of a GPU, but implementing the entire hardware would take considerable effort. In addition, for the same reasons, our processor model is most likely much simpler than what can be found in the GPUs on the market. However, we learned two facts which we believe will generalize to any GPU design. First, we can greatly reuse the computational resources and only need to add a small amount of logic to handle rerouting and special cases for interval arithmetic. In our particular implementation this overhead amounts to about 8%, and in this case we used duplicated min and max units. Second, using interval arithmetic increases the latency for executing an instruction. We therefore need more executing threads in order to avoid hazards, and consequently need a larger register file in on-chip memory. In our particular implementation, the execution pipeline increased from 22 to 24 steps, which gives a 9% increase in register space.

Note that we implemented the PCU by extending a fragment program unit. This indicates that it should be relatively straightforward to make the PCU a part of a unified shader architecture. No matter what architecture we choose, it should be

noted that the PCU introduces an additional delay in the pipeline, since we add an additional program that needs to be executed. However, we do not believe this will have any negative effect on the throughput.

6 Results

We have evaluated our PCU experimentally using a variety of modern shader based applications, all rendered at a resolution of 1280×1024 pixels. The evaluations are based on a functional simulator written in C++, which implements all features of the PCU specified in this paper, as well as most other features of modern graphics hardware. This includes interval-based texture lookups, texture caching, HDC, and depth & color buffer compression and caching. We implemented the PCU on top of a normal rasterizer, and also implemented an "optimal" culling unit for reference. In practice, we implement this optimal unit by executing the cull program for every fragment overlapping the triangle inside a tile, but without counting the instructions. We then make the culling decision based on the results of all fragments. This result can be seen as an upper bound for culling when using a particular program and tile size.

Quake 4 is a modern game based on the Doom 3 engine which uses advanced per-pixel lighting, bump mapping, and stencil shadows. We used a logging OpenGL driver to output frames from the actual game, which means that we can completely recreate its behavior.

The only modifications we made to the logged data was to enhance the fragment programs with culling code. Since we did not have access to the game code or any geometric data, we chose to disable shadowing in the game and concentrated our optimizations on the lighting only. With access to the full code it might be possible to use more sophisticated culling, such as clipping shadow volumes against portal frustums. We got the best balance of performance and culling rate when applying simple culling algorithms such as back-face (bump-mapped), attenuation, and spot-light frustum culling.

It should be noted that the Doom 3 game engine is optimized to perform a fill rate heavy task on a wide range of graphics cards with varying performance. It therefore makes heavy use of CPU culling, and possibly even pre-computations such as splitting geometry by static lights in order to save fill rate. Hence, as it is already optimized, this can be considered as a very difficult case for our PCU.

Soft Shadows is based on the soft shadows demo in the NVIDIA OpenGL SDK [25], which we replicated using our logging driver. This program uses a shader that draws eight jittered samples from the shadow map. If the shadow status for the samples varies, the fragment is assumed to be in the shadow penumbra and an additional 56 samples are drawn.

For this scene, we used a custom cull program. The reason is that we can use the powerful interval-based texture lookup (see Appendix A.2) to determine if a fragment is outside the penumbra. We do this using just one interval texture lookup,

where the texture coordinate interval is computed so that it covers the entire filter used for sampling. We can then determine if a tile is entirely in shadow or entirely lit. If the tile is in shadow, we immediately discard it, and if it is completely lit, we skip all shadow map sampling in the fragment program. Finally, if the tile is in the penumbra region we use the original fragment program. In addition, we have implemented back-face and attenuation culling for the light source. We do not present any results for the optimal culling unit for this example, since it is not completely defined how the interval texture lookup would work in the optimal case.

Order Independent Transparency (OIT) is based on the depth peeling algorithm [16]. Once again we used our logging driver and an example found in the NVIDIA OpenGL SDK. Depth peeling is an iterative multi-pass algorithm which creates one depth layer per pass. This is done by holding the depth values of the previous pass in a texture, and performing a two-sided depth test. This test discards all fragments with a depth component less than or equal to the value of the previous pass, or greater than the current value in the depth buffer.

Since the original shader code already contained a `KIL` instruction, we made no significant modifications to it. We simply noted that the PCU provides a two-sided hierarchical depth test, while a normal architecture only provides a one-sided hierarchical depth test (namely the depth buffer).

The results for this scene were generated under the assumption that a "sparse mipmap" of the depth buffer is available, as discussed in Appendix A.2. In this case, it simply means that the tile's Zmin and Zmax of the depth buffer are available in the PCU. This is perfectly reasonable since modern hardware supports Zmin/Zmax-culling. It should be noted that the number of instructions when using our PCU is reduced by an additional factor of two if a sparse mipmap is available for the color buffer as well. In this case, we get a massive speedup of a factor 3.7.

Spheres is a simple test scene that renders procedural spheres using billboards and a fragment program. A `KIL` instruction is used to discard all fragments in the billboard which do not overlap the sphere. The overlap test is performed in two-dimensional texture space.

An important feature for this test scene is that we also compute a custom depth value in the shader, which is simply the depth of the sphere at each pixel. This typically breaks hierarchical depth culling functionality on current hardware. However, with our PCU it is a simple extension to output a bounding interval for the depth values in a tile, and we can then feed the bounding interval directly to the hierarchical depth culling unit. This way, we can efficiently handle depth culling of shaders with custom depth computations, which was previously an unsolved problem. This is desirable in many applications, such as rendering curved surfaces [15] and parallax mapping [24]. It should be noted, however, that we must execute the cull program before the hierarchical depth test in this case. This opens a question whether the PCU should be placed before or after the hierarchical depth unit. Preferably, the order of the units should be configurable.

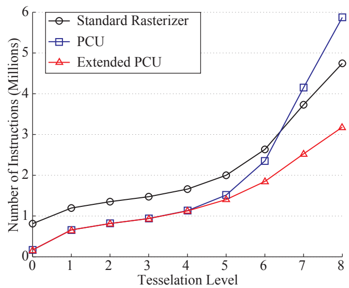
	Quake 4		Soft Shadows	OIT	Spheres
Scene	**A**	**B**			
Dynamic instructions	143 M	165 M	481 M	54 M	187 M
Instruction ratio PCU	68% (**1.5**×)	71% (**1.4**×)	48% (**2.1**×)	61% (**1.6**×)	49% (**2.0**×)
Instruction ratio optimal	60% (1.7×)	60% (1.7×)	N/A	56% (1.8×)	45% (2.2×)
Tile cull ratio PCU	40%	31%	57%	78%	58%
Tile cull ratio optimal	40%	37%	N/A	83%	82%
Total BW	**82%**	**85%**	**86%**	**87%**	**72%**

Table 1: The top section shows performance figures in terms of dynamic instructions, i.e., the total number of instructions executed when rendering a frame. We first show the number of instructions used by a normal architecture. Thereafter follows the number of instructions executed with the PCU (in percent of the original, i.e, the lower the better), and the corresponding speedup. Note that this figure includes the PCU interval instructions, where each such instruction counts as four normal fragment program instructions. Finally, we show the same statistics for an "optimal" culling unit, as described in Section 6. This unit is assumed to execute the cull program in zero time, so the culling instructions are not counted. In the middle section, we show the cull ratio in percentage of the tiles that actually use a cull program. Note that the instruction ratios include all instructions, even those when a culling program cannot be used, which explains why the tile cull ratio is better than the instruction ratio. In the lower section, we present bandwidth figures for our PCU in percent of a standard architecture. We only present the total bandwidth since the gains were quite evenly distributed over texture, depth buffer, and color buffer bandwidth.

Discussion of Results The results of our evaluations are summarized in Table 1. Note that the PCU performance figures include execution of the culling program. As can be seen, we achieve significant performance improvements for all test scenes. The improvements are larger for the simpler demos than for the game scene, as expected, since the game already makes heavy use of CPU-based culling. Still, we believe that a performance improvement of 1.4× is significant considering how easy it was to modify the game. Notice also how well our culling unit performs when compared to the "optimal" case for that scene and program. In the bottom part of Table 1, we present bandwidth figures. Bandwidth reduction was not our primary goal, but even here we noticed improvements when compared to a standard GPU.

6.1 Small Triangles

We also examined how our PCU architecture scales with decreasing triangle size, with and without the extension presented in Section 4.2. In this test, we used a very simple scene in order to focus entirely on the effects of varying tessellation. The scene consists of a unit sphere that is clipped arithmetically by a cylinder in the fragment program. We varied the tessellation of the sphere in order to study how the algorithm performs at different triangle sizes.

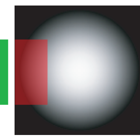


We evaluated three different rasterization algorithms, and the results are shown in the diagram to the right. The standard rasterizer emulates a modern hardware rasterization architecture. That is, it is based on pixel quads as the smallest set of pixels that can be rasterized. It should be noted that it is therefore sub-optimal for rendering extremely small triangles, but this is a problem inherent in most current hardware rasterizers. The PCU algorithm refers to our PCU architecture without the extension in Section 4.2. As we predicted, the performance decreases more rapidly with increasing tessellation than for a standard rasterizer. In this particular scene, we have found that the two curves intersect when the average projected triangle area is around three pixels. Finally, we have the extended PCU algorithm as discussed in Section 4.2. Note that the extended PCU now follows the same trend as a normal rasterizer, with a virtually constant performance advantage due to the culling.

7 Discussion

The prime use of our PCU is to speed up fragment shader execution using a very simple culling program. We believe this is the reason why we have achieved such good results using interval arithmetic, which may otherwise suffer from rapidly growing bounds when computations become too complicated. This has termi-

nated many efforts of implementing interval-arithmetic-capable CPUs. However, in culling it is more a rule than an exception to use a very simple algorithm, and this makes interval arithmetic well-suited for the task.



During the course of this project, we noted that existing fragment shader programs can be used by the PCU without any additional work by the programmer. However, with some additional effort, much better culling can be obtained in many cases. For instance, we found that Quake 4 uses a texture for spotlight angular falloff, as shown to the right. If we perform a texture lookup on the green region which is clearly outside the falloff, mipmapping and clamping may cause the result to be the interval of the texture over the red region. Much better results are achieved if the culling is done directly on the texture coordinates, rather than using the more conservative texture lookup.

The texture lookup is one of our biggest sources of interval growth, and it is a clear candidate for improvements and future work. In fact, one of our main reasons for discarding affine arithmetic (apart from interval arithmetic being better suited for hardware) is that there is no obvious way to efficiently improve texture lookups with affine arithmetic, and that the texture lookups instantly break all linear dependencies. We did some preliminary tests using affine arithmetic for the arithmetic instructions, but most results were discouraging. The cull programs were either too simple with few dependencies, or contained texture lookups.

8 Conclusion

A long awaited feature in the GPU programming community is the ability to treat the depth buffer, color buffer, and possibly even stencil buffer contents as inputs to the fragment program. This would essentially remove the need for depth/stencil/alpha tests and blending, which are the last remaining parts of the fixed function pipeline.

According to Blythe [4], the reasons why this functionality has yet to be implemented in hardware are, 1) pipeline hazards (a.k.a. read before write hazards), 2) multisampling issues, and 3) lack of culling due to the unpredictability of shader programs. Even though our PCU cannot help in the two first issues, it provides the functionality needed to solve the culling problem. For instance, it is not clear how to use hierarchical depth culling when an arbitrary depth test function is specified by the fragment program. With our PCU the culling comes automatically as long as the fragment program uses a `KIL` instruction for the depth test, and as long as the depth bounds of the triangle and depth buffer contents are given as inputs to the culling program.

The two remaining issues still are still open problems. Pipeline hazards has been, at least partially, solved by Donovan [9] and Molnar [18], who both introduce conflict detection units which ensure that no program threads are allowed to si-

multaneously access the same pixel elements. If such a conflict is detected, the conflicting threads are executed sequentially in the same order as they were issued, stalling the conflicting threads.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks to Jacob Munkberg, Petrik Clarberg, and Calle Lejdfors for inspiration and proof-reading. All Quake 4 material courtesy of id Software.

A Interval Arithmetic Instruction Set

In this section we introduce an instruction set operating on intervals. This includes arithmetic operations, conditionals, and texture lookups. We base the native instruction set of our PCU on a subset of the fragment program instructions defined by the OpenGL ARB.

A.1 Arithmetic and Conditional Instructions

Arithmetic and conditional operations have been thoroughly studied in interval arithmetic [19, 13], and we base our instruction set on those findings. The instructions and their corresponding operational behavior are summarized in Table A.1.

A.2 N-Dimensional Texture Lookups

The interval instructions for performing N-dimensional texture lookups are inspired by Moule and McCool's [21] approach, originally used for displacement map subdivision. The general idea is to provide an efficient way of computing the bounding interval of the texture data over a given area[2]. The remainder of this section will only consider two-dimensional textures, but generalization to a higher dimension is straightforward.

We initially compute two mipmap pyramids for each texture that is subject to interval-based texture lookup. As shown in Figure A.1, each element in a mipmap is computed as the component-wise minimum or maximum value of the four corresponding texels immediately below it in the pyramid. The final result can be seen as a mipmap pyramid of bounding intervals. This type of pre-computation can easily be handled by the driver, similar to how standard mipmaps are auto-generated.

When performing a texture lookup, we wish to compute the bounding interval of the texture data over an axis-aligned bounding box, which is the texture coordinate

[2] Interval-based texture example: assume we render a fence as a quad, using alpha testing and a texture to represent the geometry. It would then make sense to cull tiles where $texture_\alpha = 0$ over the whole tile. This can be done using the interval-based texture lookup.

Instruction	Operation	Condition
`MOV` d, a	$d_i \leftarrow a_i$	
`MAD` d, a, b, c	$d_i \leftarrow [\min(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}) + \underline{c_i},$ $\max(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}) + \overline{c_i}]$	
`DP4` d, a, b	$d_x \leftarrow [\sum_i \min(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i}),$ $\sum_i \max(\underline{a_i}*\underline{b_i}, \overline{a_i}*\underline{b_i}, \underline{a_i}*\overline{b_i}, \overline{a_i}*\overline{b_i})]$	
`RCP` d, a	$d_i \leftarrow [1/\overline{a_x}, 1/\underline{a_x}]$ $d \leftarrow [-\infty, \infty]$	$0 \notin a_x$ $0 \in a_x$
`RSQ` d, a	$d_i \leftarrow [1/\sqrt{\overline{a_x}}, 1/\sqrt{\underline{a_x}}]$ $d_i \leftarrow [\text{NaN}, \text{NaN}]$	$\underline{a_x} > 0$ $\underline{a_x} \leq 0$
`EX2` d, a	$d_i \leftarrow [2^{\underline{a_x}}, 2^{\overline{a_x}}]$	
`LG2` d, a	$d_i \leftarrow [\log_2 \underline{a_x}, \log_2 \overline{a_x}]$ $d_i \leftarrow [\text{NaN}, \text{NaN}]$	$\underline{a_x} > 0$ $\underline{a_x} \leq 0$
`MAX` d, a, b	$d_i \leftarrow [\max(\underline{a_i}, \underline{b_i}), \max(\overline{a_i}, \overline{b_i})]$	
`MIN` d, a, b	$d_i \leftarrow [\min(\underline{a_i}, \underline{b_i}), \min(\overline{a_i}, \overline{b_i})]$	
`SGE` d, a, b	$d_i \leftarrow 0$ $d_i \leftarrow 1$ $d_i \leftarrow [0, 1]$	$\overline{a_i} < \underline{b_i}$ $\underline{a_i} \geq \overline{b_i}$ otherwise
`SLT` d, a, b	$d_i \leftarrow 0$ $d_i \leftarrow 1$ $d_i \leftarrow [0, 1]$	$\underline{a_i} \geq \overline{b_i}$ $\overline{a_i} < \underline{b_i}$ otherwise
`FLR` d, a	$d_i \leftarrow [\lfloor \underline{a_i} \rfloor, \lfloor \overline{a_i} \rfloor]$	
`NEG` d, a `SAT` d, a	$d_i \leftarrow -[\overline{a_i}, \underline{a_i}]$ $d_i \leftarrow [\max(0, \min(1, \underline{a_i})), \max(0, \min(1, \overline{a_i}))]$	

Table A.1: The arithmetic and conditional expressions of our minimal instruction set, named in accordance to the ARB_fragment_program extension. We use d for the destination register, and a, b & c for source registers. Note that `NEG` and `SAT` are not actual instructions. They refer to the optional negation of source registers, and the optional saturation of the result. Also, the `DP3`/`DPH` are left out since they are simple modifications of the `DP4` instruction. Note that whenever i is used above, it implies componentwise computations, i.e., $i \in \{x, y, z, w\}$, and also we have omitted the hats (ˆ) to avoid cluttering.

interval. First, we compute an appropriate mipmap level as:

$$\lambda = \left\lceil \log_2 \left(\max(\overline{t_x} - \underline{t_x}, \overline{t_y} - \underline{t_y}) \right) \right\rceil \tag{A.1}$$

where $\hat{\mathbf{t}} = (\hat{t}_x, \hat{t}_y)$ is a two-dimensional interval of the unnormalized integer texture coordinates (i.e., they include the dimensions of the texture). These are conservatively rounded so that $\underline{t_i}$ is floored and $\overline{t_i}$ is ceiled for $i \in \{x, y\}$.

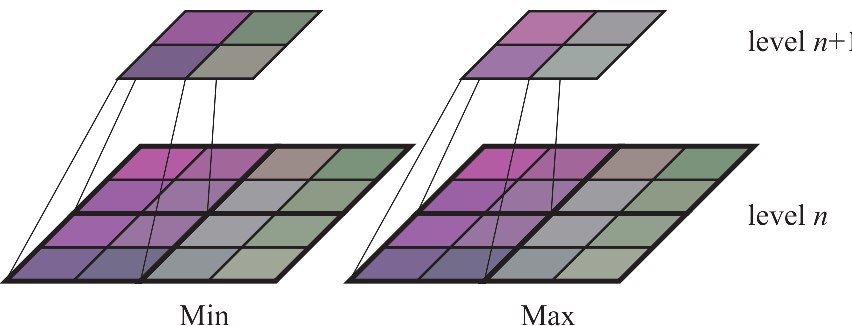


Figure A.1: The mipmap pyramids used for conservative texture lookups. We compute two pyramids, where each texel contains the minimum and maximum values respectively, of the corresponding four texels in the mipmap level directly below.

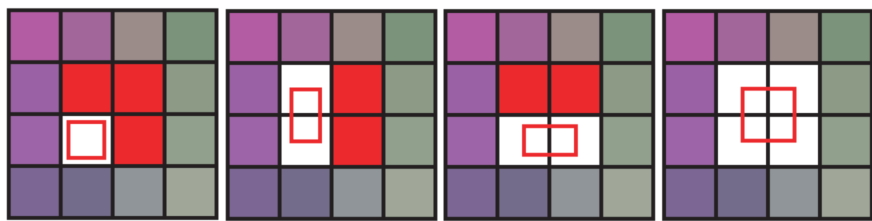


Figure A.2: The four different lookup patterns of conservative texture lookups. Mipmap selection makes sure that the texture coordinate interval (red rectangle) is never more than one texel wide. Therefore, only the four cases shown in the image are possible. Since we always read a block of 2×2 texels (in the spirit of linear filtering) we exclude the texels shaded in red from the final result.

When transformed to this mipmap level, $\hat{\mathbf{t}}$ will never be more than one texel wide in any dimension, and always least $1/2$ texels wide in the widest dimension. Thus, we get four possible cases of texture coordinate intervals as illustrated in Figure A.2. Here, we deviate slightly from Moule and McCool [21] who used these cases to determine how many texels they need to sample. Instead, we always sample a square of 2×2 texels with the lower left corner at the texel of $(\underline{t_x}, \underline{t_y})$, in the access scheme used for normal linear interpolation. The result of the texture lookup is then computed as the bounds of the colors of the texels that actually overlap the texture coordinate interval. That is, we discard the texels shaded in red in Figure A.2. Since the mipmap transformed $\hat{\mathbf{t}}$ will be rounded to the nearest integer coordinates, this overlap test can be implemented very efficiently by comparing just the final bit.

It can be shown that this texture lookup process is conservative with respect to filtered texture lookups, such as bi-linear, tri-linear and anisotropic filtering, as long as the filtered texture lookups compute derivatives using finite differences, and as long as the texture filter does not extend outside the area spanned by the derivatives. The texture lookup also natively supports all different kinds of wrapping modes, such as clamping and repeating. The appropriate wrapping mode can simply be applied to the interval coordinates, after mipmap level computation, to get the expected result. Our texture lookup process is essentially as costly as a normal tri-linearly filtered texture lookup. The biggest differences are that we need to be able to sample from the same level in two different mipmap pyramids rather than two adjacent levels, and that we compute the final result as a bounds rather than using linear interpolation.

Discussion If more texture units are available, it is possible to improve the bounds of the interval texture lookup. The normal texture lookup assumes that we can read a block of 2×2 texels at a time. If we have enough hardware resources to read a block of 4×4 texels instead, then we can move one level down in the mipmap hierarchy and get a more accurate result.

Another important observation is that we only need to create the mipmap levels that are actually used in the cull program. This optimization is particularly important for algorithms taking place in screen space such as order independent transparency (see Section 6). In this case, we know beforehand that we only need the texture at its base level and the mipmap level that corresponds to a tile on the screen. Note that such tile information is already available in modern hardware and can be read "for free". The minimum and maximum depth values can, for instance, be found in the hierarchical depth culling unit. It is also possible (but less likely) that the min and max colors are already computed for compression purposes, otherwise we need to compute them. Extensions for rendering to the base and tile mipmap level of a texture would greatly accelerate screen space algorithms.

A.3 Cube Map Lookups

Cube maps present more of a challenge than normal texture lookups. Normally, a cube map is accessed through a vector coordinate (x, y, z), and thus we get a three-dimensional coordinate interval. As can be seen in the left part of Figure A.3, we want to project that interval onto the cube map and compute the interval of the texture data under the projected area.

There are two problems with this approach. First, it is expensive to project a cube onto the shape of another cube. Second, hardware is typically designed with the restriction that each texture lookup may only access a single side of the cube. Since we strive for as modest hardware modifications as possible, we propose an alternate method which trades accuracy[3] for a simpler algorithm.

[3]In the sense of interval width. The algorithm is still fully conservative.

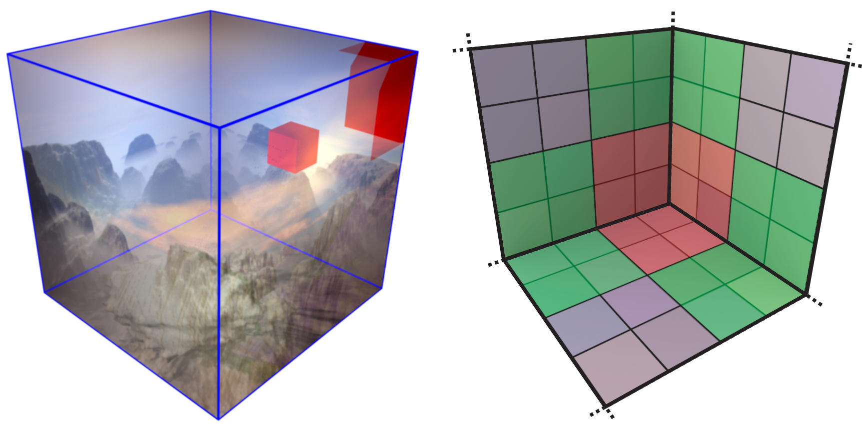
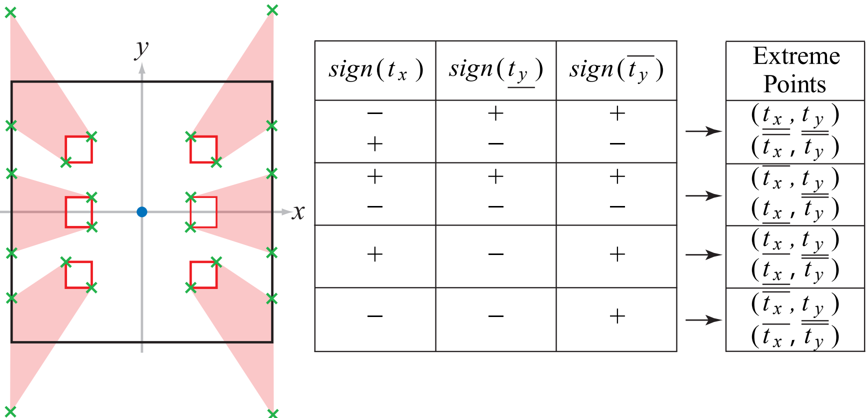


Figure A.3: *Left:* The cube mapping problem. A texture coordinate interval (the red box) may map to several sides on the cube (the projection of the red box), but current hardware only allows one side at a time to be accessed. *Right:* Min-Max mipmap generation for interval-based cube mapping. Only a small part of the cubemap is shown here. We use texels from both sides of an edge (the green regions), and texels from three sides in the corners (the red region).

We compute the min/max mipmap pyramid for the cube map, using the same approach as for two-dimensional textures. However, near edges and corners of the cube, special treatment is necessary as shown to the right in Figure A.3. For the edges, we compute the mipmap color as the min or max of four texels on both sides of the edge, and for the corners we compute the mipmap color as the min or max of four texels on all three sides emanating from that corner. Texels on opposite sides of edges will therefore share the same colors in higher mipmap levels. Similarly, the three texels in a corner will also share a common color. It should be noted that the highest level mipmap will contain the min and max value over the full cube, as expected.

We can now use this mipmap pyramid to do conservative cube map lookups with accesses to only one side of the cube. First, we compute the interval-based equivalent of the major axis. Given a texture coordinate interval $\hat{\mathbf{t}} = (\hat{t}_x, \hat{t}_y, \hat{t}_z)$, we define the major axis, i, as the axis where $\underline{t_i}$ and $\overline{t_i}$ have the same sign, and where $\min(|\underline{t_i}|, |\overline{t_i}|)$ is maximized. This is essentially the axis where the texture coordinate interval does not contain the origin, and lies nearest to a cube map face. If $\underline{t_i}$ and $\overline{t_i}$ have different signs over all axes, then we cannot find a major axis. However, this can only happen if the origin lies within the texture coordinate interval. In this case, the texture coordinate interval will project onto the entire cube map. We can easily handle this by choosing the highest mipmap level, and sample an arbitrary cube map face.

Once we have found a major axis, we conservatively project the texture coordinate



$sign(t_x)$	$sign(\underline{t_y})$	$sign(\overline{t_y})$		Extreme Points
− +	+ −	+ −	→	$(\underline{t_x}, \underline{t_y})$ $(\overline{t_x}, \overline{t_y})$
+ −	+ −	+ −	→	$(\overline{t_x}, \underline{t_y})$ $(\underline{t_x}, \overline{t_y})$
+	−	+	→	$(\underline{t_x}, \underline{t_y})$ $(\underline{t_x}, \overline{t_y})$
−	−	+	→	$(\overline{t_x}, \underline{t_y})$ $(\overline{t_x}, \overline{t_y})$

Figure A.4: Our method for projecting the bounds of a texture coordinate interval on the cube map. Here x is the major axis, and we want to find the texture coordinate interval when projected on any of the x-sides of the cube map. The figure illustrates the six possible cases of texture coordinate intervals, and the projected extreme points. The table summarizes the different cases of extreme points and show how they can be determined using signs.

interval on the corresponding side of the cube map. The projection is done by projecting the bounds of each of the two remaining axes separately. Figure A.4 shows such a projection, where x is the major axis, and y is the axis for which we want to project the bounds. The figure shows the six possible cases of texture coordinate intervals (note that no interval may cross the y-axis since the x-axis would not be a major axis in that case), and the extreme points we have to project to compute the bounds. Fortunately, it is very easy to determine which these extreme points are. It is sufficient to look at the signs of the texture coordinate interval, as shown in the table in Figure A.4.

We project the extreme points for the remaining two axes to form a two-dimensional projected coordinate interval. This interval is used to compute a mipmap level and perform a two-dimensional texture lookup, identically to the method described in Section A.2.

It is possible to show that this algorithm is conservative because of the information-bleeding during mipmap generation. Furthermore, it is computationally inexpensive. Finding the major axis, and projection can be expected to be twice as costly as a normal cube mapping implementation, which is reasonable considering we use intervals. In addition we need the tabulated function from Figure A.4, which is already very inexpensive.

Bibliography

[1] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics,*, 22(3):792–800, 2003.

[2] Tomas Akenine-Möller and Jacob Ström. Graphics for the masses: A hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics,*, 22(3):801–808, 2003.

[3] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum,*, 23(3):615–624, 2004.

[4] David Blythe. The Direct3D 10 system. *ACM Transactions on Graphics,*, 25(3):724–734, 2006.

[5] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *SIBGRAPI 1993*, pages 9–18, 1993.

[6] Robert L. Cook. Shade trees. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, pages 223–231, 1984.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Language Systems*, 13(4):451–490, 1991.

[8] Michael Doggett. Overview of the Xbox360 GPU. Keynote at *Eurographics 2005*, 2005.

[9] Walter Donovan. Pixel load instruction for a programmable graphics processor. US Patent 7,091,979, 2006.

[10] Ned Greene and Michael Kass. Error-bounded antialiased rendering of complex environments. In *Proceedings of ACM SIGGRAPH 1994*, pages 59–66, 1994.

[11] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of ACM SIGGRAPH 1993*, pages 231–238, 1993.

[12] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. In *Proceedings of ACM SIGGRAPH 1998*, pages 158–176, 1998.

[13] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[14] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, pages 149–158, 2001.

[15] Charles Loop and Jim Blinn. Real-time gpu rendering of piecewise algebraic surfaces. *ACM Transactions on Graphics,*, 25(3):664–670, 2006.

[16] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, 1989.

[17] Michael D. McCool, Chris Wales, and Kevin Moule. Incremental and hierarchical hilbert order edge equation polygon rasterization. In *Graphics Hardware*, pages 65–72, 2002.

[18] Steven Molnar and John Montrym. Position conflict detection and avoidance in a programmable graphics processor using tile coverage data. US Patent 7,053,893, 2006.

[19] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[20] Steve Morein. ATI Radeon HyperZ technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM Press, August 2000.

[21] Kevin Moule and Michael D. McCool. Efficient bounded adaptive tesselation of displacement maps. In *Graphics Interface*, pages 171–180, 2002.

[22] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Graphics Hardware*, pages 41–50, 2003.

[23] Marc Stamminger, Philipp Slusallek, and Hans-Peter Seidel. Bounded radiosity — illumination on general surfaces and clusters. *Computer Graphics Forum*, 16(3):309–317, 1997.

[24] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *Proceedings of ACM SIGGRAPH SI3D*, pages 63–69, 2006.

[25] Yury Uralsky. Efficient soft-edged shadows using pixel shader branching. In *GPU Gems 2*, pages 269–282. Addison-Wesley Professional, 2005.

Paper VIII

Automatic Pre-Tessellation Culling

Jon Hasselgren Jacob Munkberg Tomas Akenine-Möller

Lund University

{jon|jacob|tam}@cs.lth.se

ABSTRACT

Graphics processing units supporting tessellation of curved surfaces with displacement mapping exist today. Still, to our knowledge, culling only occurs *after* tessellation, i.e., after the base primitives have been tessellated into triangles. We introduce an algorithm for *automatically* computing tight positional and normal bounds on the fly for a base primitive. These bounds are derived from an arbitrary vertex shader program, which may include a curved surface evaluation and different types of displacements, for example. The obtained bounds are used for backface, view frustum, and occlusion culling *before* tessellation. For highly tessellated scenes, we show that up to 80% of the vertex shader instructions can be avoided, which implies an "instruction speedup" of 5×. Our technique can also be used for offline software rendering.

ACM Transactions on Graphics, 28(2):19, 2009.

1 Introduction

To provide rich surface representations for real-time rendering, it is expected that most graphics hardware in the near future will have support for tessellation of curved surfaces with displacement mapping. The Xbox 360 [5] and the ATI Radeon HD 2000 series [25] already have support for this. A primitive with a triangular or square domain is tessellated, and barycentric coordinates are forwarded to the vertex shader, which may compute an arbitrary position based on these coordinates, and more. To the best of our knowledge, these systems only perform culling *after* tessellation using the conventional graphics pipeline. Clearly, it would be advantageous to be able to cull *before* tessellation occurs, and Figure 1 shows an example of this.

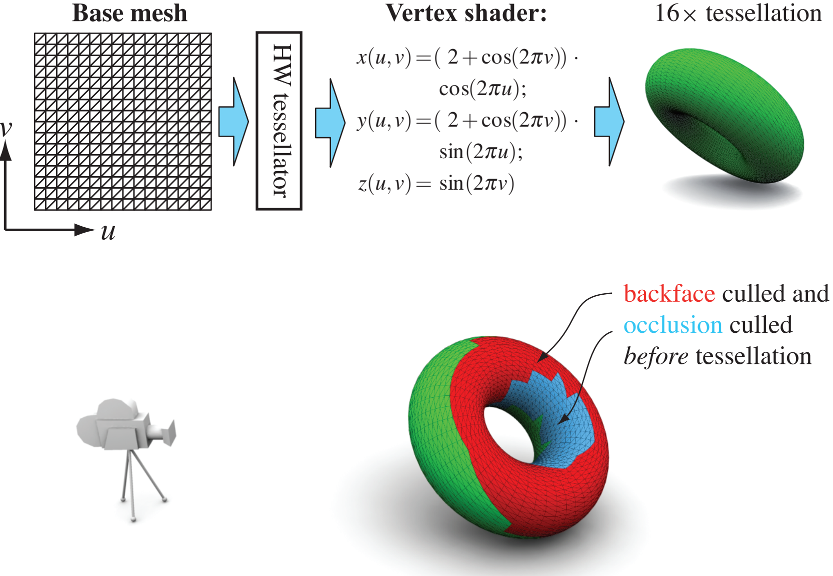


Figure 1: GPUs with tessellation hardware are given a base mesh over a parameter space, (u,v), as input. In this case, the tessellator increases the number of triangles by a factor of sixteen, and a vertex shader evaluates a point on a torus surface. In the lower part, we visualize the base triangles that our culling algorithm *automatically* can avoid to tessellate, and where vertex shader evaluations can be avoided. We are able to cull 56% of the triangles prior to tessellation.

Over the years, culling techniques have seen many uses in both real-time graphics and offline rendering. In general, RenderMan implementations [1, 4] use culling on many different levels. However, the details may vary for different implementations. View frustum and occlusion culling are performed, often prior to tessellation, and splitting of primitives may also occur. Backface culling is usually done after tessellation. Wexler et al. [27] describe a GPU-optimized implementation, where (among other things) occlusion queries are used to accelerate rendering. However, to bound a displaced surface in RenderMan, the user either has to provide the renderer with a conservative upper bound, or the displacement shader is executed on micropolygons, and exact bounds computed from these [1]. In this latter case, no culling occurs before tessellation.

Shirman and Abi-Ezzi [24] use cones to bound a set of normals on a patch, and can thus perform efficient backface culling. Kumar and Manocha [15] derive a different method for backface culling of curved surfaces, and use a conservative technique to bound the normals and then test for culling. However, neither of these techniques can handle arbitrary surface evaluations automatically on the fly. Han et al. [10] describe an alternative GPU implementation, where the part of the vertex shader that computes the position of a vertex is executed first. After that follows backface culling. If the triangle is culled then unnecessary lighting calculations are avoided. Our goal is similar, but we want to perform culling before tessellation even occurs.

There is a wealth of literature on adaptive on-the-fly tessellation, and as our work can be combined with such techniques, we only list some of them. Doggett and Hirche [6] use a summed-area table of the displacement map and a normal test to guide the tessellation. A similar approach is to use interval arithmetic and interval textures to focus the tessellation efforts [22]. To provide a continuous level of detail, Moreton [21] introduces *fractional tessellation* where tessellation factors are specified as floating-point numbers per triangle edge. This allows for adaptive tessellation across a mesh, and similar techniques are used in modern GPUs [25].

In contrast to the previous work described above, we focus on presenting a single automatic solution. Our paper contributes with a novel pre-tessellation backface, view frustum, and occlusion culling method which is:

- fully automatic, based only on arbitrary vertex shader code which can include for example deformations, curved surfaces, and displacement mapping.
- implemented with tightly bounded arithmetic on triangular domains.
- suitable for implementing in both hardware and software rendering systems.

Next, we describe our algorithm in detail.

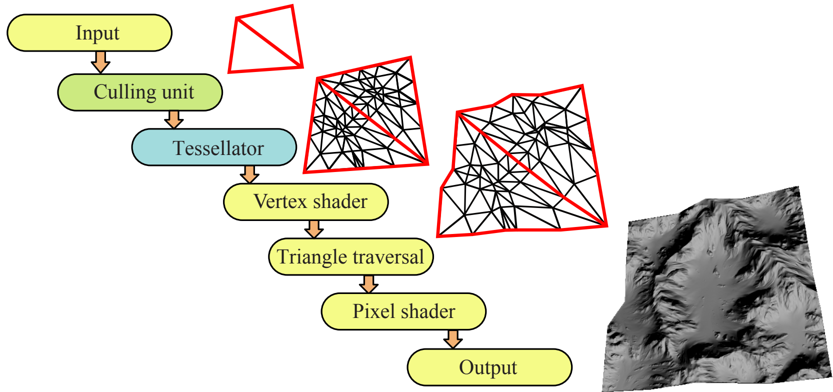


Figure 2: To support tessellation in the GPU pipeline, a tessellation unit has recently been added. We propose to add the culling unit, which automatically determines whether tessellation of a base primitive can be avoided.

2 Tessellation Culling

The goal of our work is to efficiently avoid tessellating the majority of surfaces which do not contribute to the final image. This occurs when a surface is back-facing, outside the view frustum, or occluded by previously rendered surfaces. Furthermore, we believe it is of utmost importance that fully *arbitrary* vertex displacement shaders can be handled in a completely *automatic* way. In this section, we present a novel algorithm for this. Without loss of generality, we restrict ourselves to triangular domains and tessellation.

2.1 Overview

We extend the current GPU tessellation pipeline [25] with our new culling unit as illustrated in Figure 2. Note that this type of pipeline is also rather similar to offline rasterization pipelines. Without our culling unit, *base triangles* are first injected into the pipeline, and these can be tessellated to a desired number of triangles by the hardware. For each created vertex, the tessellator forwards its barycentric coordinates, (u,v), down the pipeline. The vertex shader then computes the position, $\mathbf{p}(u,v)$, of each vertex as a function of its barycentric coordinates. This may include, e.g., the evaluation of a Bézier triangle with texture displacement, procedural noise, and transform matrices. Each term can also depend on a time parameter, in order to animate a water surface, for example.

Our culling algorithm works as outlined in Fig. 3. First, we analyze the vertex shader program and isolate all instructions that are used to compute the vertex position. We then compute geometric bounds for this position over an entire base

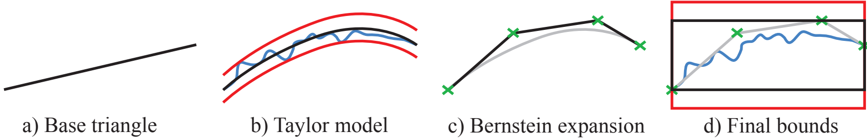


Figure 3: Algorithm overview: **a)** A base triangle (seen from the side) with pre-computed tessellation factors is sent to the tessellation unit. **b)** By expressing the vertex program in Taylor form (polynomial + interval remainder), a conservative estimate of the surface is obtained. **c)** The Taylor polynomial is expanded in Bernstein form for efficient range bounding (using the convex hull property), **d)** Finally, by adding the interval remainder term from the Taylor model to the Bernstein bounds, conservative surface bounds (red) are obtained.

triangle, and use these bounds to perform the culling.

Recently, it has been shown [12] that pixel shaders can be executed, bounded, and culled over a block of pixels using interval arithmetic [19]. In this case, the programs used for culling are often short (terminated by a `KIL` instruction). However, in our context, the shader programs are significantly more complex, and therefore we use Taylor models [2] to approximate the shader function over the triangle domain. We then use Bernstein expansion [14] to compute tight bounding boxes for the Taylor models, and use these bounding boxes for culling.

In the following, we first present some background on Taylor models in Section 2.2. Then follows an algorithm for computing tight polynomial bounds in Section 2.3, and our program analysis and generation in Section 2.4. In Section 2.5, we describe how selective execution of our culling can be done, and finally, the culling algorithms are described in Section 2.6.

2.2 Taylor Arithmetic

Taylor arithmetic has seen little use in computer graphics research, but there is a recent exception in collision detection [28]. Interval arithmetic [19], on the other hand, has been used extensively in graphics. Intervals are used in Taylor models, and the following notation is used for an interval $\hat{a}$:

$$\hat{a} = [\underline{a}, \overline{a}] = \{x \,|\, \underline{a} \leq x \leq \overline{a}\}. \tag{1}$$

Given an $n+1$ times differentiable function, $f(u)$, where $u \in [u_0, u_1]$, the Taylor model of f is composed of a Taylor polynomial, T_f, and an interval remainder term, $\hat{r}_f$ [2]. An nth order Taylor model, here denoted $\tilde{f}$, over the domain $u \in$

$[u_0, u_1]$ is then:

$$\tilde{f}(u) = \underbrace{\sum_{k=0}^{n} \frac{f^{(k)}(u_0)}{k!} \cdot (u-u_0)^k}_{T_f} + \underbrace{[\underline{r_f}, \overline{r_f}]}_{\hat{r}_f} = \sum_{k=0}^{n} c_k u^k + \hat{r}_f. \tag{2}$$

This representation is called a Taylor model, and is a conservative enclosure of the function, f over the domain $u \in [u_0, u_1]$.

Similarly to interval arithmetic, it is also possible to define arithmetic operators on Taylor models, where the result is a conservative enclosure (another Taylor model) as well [2]. Addition is defined as follows: Assume that $f+g$ shall be computed and these functions are represented as Taylor models, $\tilde{f} = T_f + \hat{r}_f$ and $\tilde{g} = T_g, \hat{r}_g$. The Taylor model of the sum is then

$$\widetilde{f+g} = (T_f + T_g) + (\hat{r}_f + \hat{r}_g). \tag{3}$$

Note here that $T_f + T_g$ is an addition of two polynomials.

Similarly, for multiplication of a Taylor model, $\tilde{f}$, by a scalar value, λ, we get that:

$$\widetilde{\lambda \cdot f} = (\lambda \cdot T_f) + (\lambda \cdot \hat{r}_f). \tag{4}$$

Multiplication between two Taylor models is more complicated. Assume again that we want to compute $f \cdot g$ where f and g are represented by Taylor models. The Taylor model of the product is then

$$\widetilde{f \cdot g} = \underbrace{\underline{T_f \cdot T_g}}_{T_{f \cdot g}} + \underbrace{B\left(\overline{T_f \cdot T_g}\right) + B\left(T_f\right) \cdot \hat{r}_g + B\left(T_g\right) \cdot \hat{r}_f + \hat{r}_f \cdot \hat{r}_g}_{\hat{r}_{f \cdot g}}. \tag{5}$$

The polynomial part of this equation, $T_{f \cdot g}$ is simply the multiplication of the polynomials T_f and T_g, but clamped (denoted $\underline{T_f \cdot T_g}$) so that all terms of higher order than the Taylor model has been removed.

The remainder has several contributing terms. First, we have the part of the polynomial multiplication that overflows and has terms only of higher order than the Taylor model ($\overline{T_f \cdot T_g} = T_f \cdot T_g - \underline{T_f \cdot T_g}$). Note that we want the remainder term on interval form, and therefore we must bound the overflow of the polynomial multiplication over the domain (this is indicated by the bounding operator, $B()$). To compute the bounds, we directly evaluate overflowing terms using interval arithmetic and accumulate them to the remainder. More complex bounding computations, such as the one presented in Section 2.3, are possible, but since multiplication is such a frequent operation, we must ensure that it is fast to compute its bounds. The other terms found in the remainder involve computing the bounds of T_f and T_g and are treated similarly to the overflow from the polynomial multiplication. It should be noted that one or more of the terms in the remainder often are zero. For instance, if $\hat{r}_f$ or $\hat{r}_g$ is zero, then the corresponding terms will be zero as well. As an optimization, we detect these cases and avoid the computations.

By using Taylor expansion, and the addition and multiplication operations presented above we can derive more complex arithmetic operators, like sine, log, exp, reciprocal, and so on. We refer to the work of Berz and Hoffstätter [2] and Makino and Berz [18] for more details.

Motivation The motivation for us to use Taylor models is that curved surfaces and subdivision schemes are often based on polynomials. Polynomial computations can be represented exactly by Taylor models (provided they are of high enough order) which leads to very tight bounds. Previous work on shader analysis [7, 13, 12] have successfully used interval and affine arithmetic, which are computationally less expensive than Taylor models. However, note that they subdivide the domain into small tiles before evaluating the bounded shader. In contrast, we must bound the shader over the entire domain (the base triangle) in a *single* evaluation, and consequently we need much tighter bounds. A side by side comparison between the tightness of interval arithmetic, affine arithmetic and Taylor models can be found in the example in Section 2.3.

Taylor models also provide a flexible framework since it is essentially a superset of interval and affine arithmetics. It allows us to tweak interval sharpness versus computational overhead by changing the order of the Taylor model. Orders zero and one correspond to interval arithmetics, and generalized interval arithmetics [11], which is similar to affine arithmetics.

2.3 Tight Polynomial Bounds

Our approach to tessellation culling is to evaluate the vertex shader using Taylor arithmetic as described above. We execute the part of the shader that affects the position attribute using Taylor arithmetic. This results in a Taylor model for each of the components in the position attribute: (x, y, z, w). To find a geometrical bounding box, one could then find local minima and maxima for each of these. However, this requires numerical, iterative methods for polynomials of degree $n > 4$, and also quickly becomes impractical due to the dependence on the two parametric coordinates, (u, v).

Instead, we use a faster, conservative approach which still produces tight bounds. The resulting Taylor polynomials are in power form, and the core idea is to convert these to Bernstein form. The convex hull property of the Bernstein basis guarantees that the actual surface or curve of the polynomial lies inside the convex hull of the control points. Thus, we compute a bounding box by finding the minimum and maximum control point value in each dimension.

In practice, we obtain bivariate polynomials from the vertex shader evaluation using Taylor arithmetic, and for a single component (e.g., x), this can be expressed in the power basis as follows (where we have omitted the remainder term, $\hat{r}_f$, for clarity):

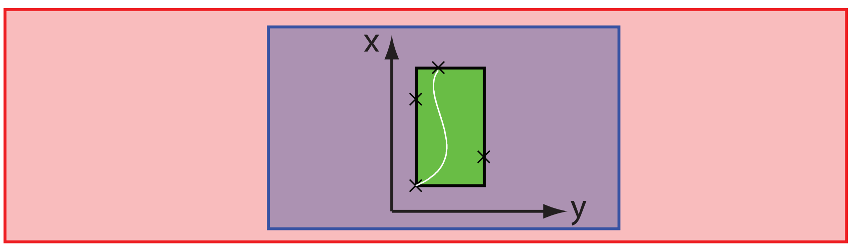


Figure 4: A comparison of the bounds for a parametric curve $(p_x(t), p_y(t))$ of degree 3 in t for interval aritmethic (red), affine arithmetic (blue) and Taylor models with Bernstein bounds (green).

$$p(u,v) = \sum_{i+j \leq n} c_{ij} u^i v^j. \tag{6}$$

We want to transform Equation 6 into the Bernstein basis:

$$p(u,v) = \sum_{i+j \leq n} p_{ij} B^n_{ij}(u,v), \tag{7}$$

where $B^n_{ij}(u,v) = \binom{n}{i}\binom{n-i}{j} u^i v^j (1-u-v)^{n-i-j}$ are the Bernstein polynomials in the bivariate case over a triangular domain. We can convert a polynomial in the power basis form, into the Bernstein form using the following formula [14]:

$$p_{ij} = \sum_{l=0}^{i} \sum_{m=0}^{j} \frac{\binom{i}{l}\binom{j}{m}}{\binom{n}{l}\binom{n-l}{m}} c_{lm}. \tag{8}$$

To compute a bounding box, we simply compute the minimum and the maximum value over all p_{ij} for each dimension, x, y, z, and w. This gives us a bounding box, $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, in clip space. Next, we will give an example of the effectiveness of this technique when compared to interval and affine arithmetic.

Example Assume we have the following parametric curve, $\mathbf{p}(t) = (p_x(t), p_y(t))$, where $t \in [0,1]$, $p_x(t) = 1 + 3t + 3t^2 - 2t^3$, and $p_y(t) = 1 + 9t - 18t^2 + 10t^3$. We will illustrate how interval and affine arithmetic compare to our tight polynomial bounds when computing a two-dimensional axis-aligned bounding box of this curve over the domain, $t \in [0,1]$. The resulting bounds are visualized in Fig. 4. Using standard interval arithmetic, we obtain $\hat{p}_x = [1,1] + [0,3] + [0,3] + [-2,0] = [-1,7]$ and $\hat{p}_y = [1,1] + [0,9] + [-18,0] + [0,10] = [-17,20]$, and these two intervals represent a box with an area of 296. Similarly, applying affine arithmetic [3] on the same example gives us $p_x = 3 + 9/4\varepsilon_1 + 1/2\varepsilon_2 - 3/4\varepsilon_3$ and $p_y = 9/4 - 3/4\varepsilon_1 - 13/4\varepsilon_2 + 15/4\varepsilon_3$, where $\varepsilon_i \in [-1,1]$ are noise symbols. The bounding box becomes $\hat{p}_x = [-0.5, 6.5]$, $\hat{p}_y = [-5.5, 10]$, which represent a box

with an area of 108.5. To apply our tight polynomial bounds, we first observe that the polynomials for p_x and p_y are essentially in Taylor form already. Our strategy is therefore to rewrite these on Bernstein form: $p_x(t) = \mathbf{1}\cdot(1-t)^3 + \mathbf{2}\cdot 3(1-t)^2 t + \mathbf{4}\cdot 3(1-t)t^2 + \mathbf{5}\cdot t^3$, and $p_y(t) = \mathbf{1}\cdot(1-t)^3 + \mathbf{4}\cdot 3(1-t)^2 t + \mathbf{1}\cdot 3(1-t)t^2 + \mathbf{2}\cdot t^3$, where the control points have been typeset in boldface. The bounding box is then found as the minimum and maximum of the control points in x and y. This gives us $\hat{p}_x = [1,5]$ and $\hat{p}_y = [1,4]$, which has an area of 12. The tightest fit axis-aligned box has $\hat{p}_x = [1,5]$ and $\hat{p}_y = [1,2.37]$, with an area of 5.48.

2.4 Program Analysis and Generation

In a graphics pipeline with a tessellation unit, the vertex shader receives barycentric coordinates & the associated base triangle information, and then outputs a vertex position in clip space. In the simplest vertex shader, the vertex position is computed by interpolating the base triangle vertices, using the barycentric coordinates, and transforming this position into clip space by a matrix multiplication. In the general case, the vertex position is displaced using an arbitrary function (of the barycentric coordinates) before the clip space transform.

We want to bound this position over the entire barycentric domain, and must therefore evaluate the vertex shader output for every possible barycentric coordinate, since this is the only input that varies over the base triangle. To accomplish this, we reformulate the vertex shader using Taylor models.

We represent each Taylor model as a coefficient list. Each coefficient has a scalar value and an id i, indicating that it is the coefficient of the x^i term. For example, the polynomial $4+3x+0.5x^2$ over the domain $x \in [0,1]$ would be represented, as a Taylor model of order 2, by the list $[\{4,0\},\{3,1\},\{0.5,2\},\hat{r}=0]$. It could also be represented as a Taylor model of order one as $[\{4,0\},\{3,1\},\hat{r}=[0,0.5]]$.

Our only varying input, the barycentric coordinates, are expressed as two-dimensional Taylor models. Generalizing the list representation from above to two dimensions so that a polynomial term $\alpha x^i y^j$ is represented by a coefficient $\{\alpha,i,j\}$, we can write the barycentric coordinates as two-dimensional Taylor models:

$$
\begin{aligned}
u &= 0+1\cdot u+0\cdot v = [\{1,1,0\}] \\
v &= 0+0\cdot u+1\cdot v = [\{1,0,1\}] \\
w &= 1-1\cdot u-1\cdot v = [\{1,0,0\},\{-1,1,0\},\{-1,0,1\}]
\end{aligned}
$$

These are Taylor models of order 1 ($i,j \leq 1$) over the domain $u \in [0,1], v \in [0,1]$. Note that no remainder is needed.

We then proceed by evaluating all instructions using Taylor models. We will briefly exemplify the implementation of addition and multiplication of Taylor models, as more complex operations will be expressed in these in the end.

Addition: Addition is done by adding the polynomial part of each Taylor model. Our internal representation of the polynomial part is a list of non-zero coefficients.

Thus, the polynomial addition essentially becomes a sparse vector addition at runtime. Here is an example:

$$u+w=[\{1,0,0\},\{\mathbf{1}-\mathbf{1},1,0\},\{-1,0,1\}]=[\{1,0,0\},\{-1,0,1\}]=1-v. \quad (9)$$

Note that we only need to perform additions for non-zero terms existing in both u and w, as the other terms can be handled using variable renaming. The remainder term, if non-zero, is handled using normal interval arithmetic. A more realistic shader would include linear interpolation between two, at compile time unknown, positions. This requires us to work with variables rather than constants. Thus the example becomes:

$$p_1u+p_0w=[\{p_0,0,0\},\{\mathbf{p_1}-\mathbf{p_0},1,0\},\{-p_0,0,1\}]=p_0+(p_1-p_0)u-p_0v. \quad (10)$$

Multiplication: Here, we loop over the non-zero components in one Taylor model and multiply it by all non-zero components in the other. Thus, the runtime complexity is roughly $O(a\cdot b)$ multiplications, where a and b is the number of non-zero coefficients in each of the two polynomials being multiplied. We bound the remainder terms using interval arithmetics. This can be optimized by exploiting that our domain is $(u,v)\in[0,1]$, as all multiplications by zero can be omitted. For multiplication, the order of the Taylor model will increase, so we have the choice to bound the higher-order terms and add to the remainder, or increase the order of the model. A higher order Taylor model has more precision (polynomials up to the order of the model can be represented exactly), but is also more costly computationally. With the sparse list representation above, we can use a fixed order and models of lower orders will not have any computational overhead, as only non-zero terms are stored and used in the arithmetic operations.

Polynomial displacement shaders (Bézier surfaces) are simply a sequence of Taylor multiplications and additions, and elementary functions can also be bounded by Taylor models. Like standard Taylor expansions, a higher-order representation leads to tighter bounds. Once all arithmetic operations have been converted to Taylor form, we express them using regular vertex shader code. Therefore, we do not need to introduce any new specialized instruction set for our bounding shader. However, the bounding shaders will be significantly longer than the corresponding vertex shader.

Finally, our program analysis gives us a polynomial approximation of the vertex position attribute. We then compute its bounds using the algorithm in Section 2.3. Once again, we generate the necessary vertex shader code for this operation.

Discussion Program analysis is done in the exact same way as a standard implementation [2] of Taylor models, with the exception that we need to treat symbolic constants (variables) rather than values, and we need to emit code rather than executing the operations.

It should be noted that the Taylor models for the barycentric coordinates are the same for all base triangles, and thus we can treat them as constants rather than vary-

ing input. This means that the order for all Taylor instructions can be computed statically at compile time. Furthermore, we can do most standard optimizations (for example, exploiting $c \cdot 0 = 0$, and $c + 0 = c$), as well as all control flow that is internally needed in the Taylor model computations, at compile time. This greatly increases the runtime shader performance.

In conclusion, the complexity of each Taylor operation is highly dependent on the "order" of the vertex shader. For instance, for a program with only interpolation and a matrix multiplication, the Taylor models will have no non-zero coefficients over order one. In contrast, cubic Bézier triangle evaluation uses polynomials of degree 3, and consequently the Taylor models will have more higher-order coefficients. The instruction ratio between the culling program and vertex shader grows for more complex shaders (see Section 4). Note that we can determine the number of instructions during compile time. Thus we can compile the program, see how expensive it gets, and only trigger culling if there is potential for performance gain.

Texture Mapping

Shaders using texture map lookups are problematic as the texture map may contain an arbitrarily complex function. However, texture mapping is an important feature as displacement mapping is a prime use-case of a tessellation unit.

We implement texture mapping using interval-based texture lookups [22, 12], which computes a bounding interval for the texture in a given region. If, for example, we want to displace a surface in the direction of an interpolated normal, then the texture interval will be used in subsequent arithmetic computations. Therefore, we must convert the interval to Taylor form.

A naive way of doing this is to treat the texture lookup in the interval remainder term, $\hat{r}_f$, of the Taylor model. However, we found this approach to be unsatisfactory as the remainder term in Taylor models is treated using standard interval arithmetic, which cause the bounds to grow rapidly. Instead, we treat every texture lookup as a functional parameter. That is, instead of treating the shader as a two-dimensional Taylor model:

$$\tilde{f}(u,v) = \sum_{i+j \leq n} c_{ij} u^i v^j + \hat{r}_f, \tag{11}$$

we treat it as a three-dimensional Taylor model:

$$\tilde{f}(u,v,a(u,v)) = \sum_{i+j+k \leq n} c_{ijk} u^i v^j a(u,v)^k + \hat{r}_f, \tag{12}$$

where $a(u,v)$ is an unknown (texture map) function defined over the interval domain, which we computed in the interval texture lookup. By increasing the dimensionality of the Taylor models, we can track correlations for arithmetic operations which depend on texture lookups. In effect we defer the interval evaluations to the last part of the shader, which is the bounds computations. To support an arbitrary

number of texture lookups, all Taylor arithmetic, as well as the tight bounding computations of Section 2.3, can be generalized to n-dimensional domains. For details, we refer to the work by Berz and Hoffstätter [2], and Lin and Rokne [16].

Branching and Looping

We can easily support branching and looping when the conditional expression is a value (or equivalently, a zero:th order Taylor model with no remainder). In this case, it is uniquely determined which branch we should take, or how many iterations of a loop we should perform. A typical example would be looping over an, at compile time unknown, number of fractal noise octaves.

We can also handle branches with Taylor models for conditional expressions. In such cases we compute quick bounds for the Taylor model based on interval arithmetic (see multiplication in Section 2.2). If the bounds of the condition is ambiguous, we must execute both branches. Furthermore, if a variable is assigned a value in both branches, we must assign it the union of those values. A union of two Taylor models could be computed by computing the average of their polynomial parts, and growing the rest term accordingly to enclose both polynomials.

A construct that we cannot handle is loops with a Taylor model as the conditional expression. For example, some iterative computation on the barycentric coordinates, that loops until the result has converged. As previously explained by Hasselgren and Akenine-Möller [12], such computations are not guaranteed to converge when bounded arithmetics are used, and we may get an infinite loop. Fortunately, we can easily detect those cases and simply disable our culling.

2.5 Selective Execution

We have observed that the bounding shaders are roughly $3-15\times$ more expensive than the corresponding vertex shader in terms of instructions. Since this cost is rather significant, it makes sense to execute the bounding shader only in regions where we are likely to improve overall performance. A statistical analysis shows that it is beneficial to execute the bounding shader if the following holds:

$$\frac{c(cull)}{c(vertex)} \leq p(cull) \cdot n. \tag{13}$$

Where $\frac{c(cull)}{c(vertex)}$ is the cost ratio between the cull and the vertex program, $p(cull)$ is the probability that a base triangle is culled, and n is the number of vertices that will be generated during tessellation.

2.6 Culling

In this section, we will describe how the actual culling is performed. We want to emphasize that the culling algorithm per se is not a novel contribution. However,

some details are given here for the sake of completeness. Recall that the output from the bounding shader program are geometrical bounds: $\tilde{\mathbf{p}}(u,v) = (\tilde{p}_x, \tilde{p}_y, \tilde{p}_z, \tilde{p}_w)$, i.e., four Taylor models. As described above, we use the convex hull property of the Bernstein form to obtain a bounding box from these Taylor models. This box is denoted $\hat{\mathbf{b}} = (\hat{b}_x, \hat{b}_y, \hat{b}_z, \hat{b}_w)$, where each element is an interval, e.g., $\hat{b}_x = [\underline{b_x}, \overline{b_x}]$.

View Frustum Culling

For view frustum culling, we simply need to test the geometrical bounds against the planes of the frustum. Since we have the bounding box, $\hat{\mathbf{b}}$, in homogeneous clip space, we can perform the test in this space as well. We use the standard optimization for plane-box tests [9], where only a single corner of the box is used to evaluate the plane equation. Each plane test then amounts to an addition and a comparison. For example, testing if the box is outside the left plane is done with: $\overline{b_x} + \overline{b_w} < 0$. Since these tests are inexpensive, our culling always starts with the view frustum test.

Backface Culling

After the vertex shader has been executed, the vertex $\mathbf{p}$ is in homogeneous clip space (before division by w). This means that the model-view transform has been applied, so the camera position is at the origin. Now, given a point, $\mathbf{p}(u,v)$, on a surface, backface culling is in general computed as:

$$c = \mathbf{p}(u,v) \cdot \mathbf{n}(u,v), \tag{14}$$

where $\mathbf{n}(u,v)$ is the normal vector at (u,v). If $c > 0$, then $\mathbf{p}(u,v)$ is backfacing for that particular value of (u,v). For a parameterized surface, the unnormalized normal, $\mathbf{n}$, can be computed as:

$$\mathbf{n}(u,v) = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}. \tag{15}$$

After our bounding shader has been executed, we have Taylor models, $\tilde{\mathbf{p}}(u,v)$, for the position. As part of the bounding shader program, these are differentiated as well, resulting in $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ and $\partial\tilde{\mathbf{p}}(u,v)/\partial v$. Finally, the Taylor model of the normal, $\tilde{\mathbf{n}}(u,v)$, is computed using these.

There are two issues with this technique, which we need to solve. The first problem arises if $\tilde{\mathbf{p}}(u,v)$ contains a non-zero remainder term, $\hat{r}_p$, since this must be accounted for when computing the partial derivatives. We solve this by using knowledge about the tessellation frequency of the base primitive. Assume that a worst-case sawtooth tessellation pattern is generated by the remainder term, as shown in Figure 5a. The maximum slope for such a configuration is $(f(x+\Delta x) - f(x) + w)/\Delta x$, where Δx is the shortest edge generated during tessellation and w is the width of the interval remainder term. This expression is bounded by $f'(x) + w/\Delta x$

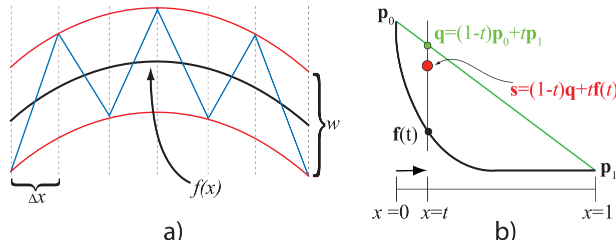


Figure 5: We must take special care of the interval remainder term when performing backface culling. Figure a) shows a worst case derivative of a Taylor model with a polynomial $f(x)$ an interval remainder term with width w. The worst case derivative that can be introduced by the remainder term is given by the blue sawtooth pattern, which has a period of $2\Delta x$ where Δx is the length of the shortest edge created during tessellation. Figure b) shows how we alter the original fractional tessellation algorithm to avoid problems that would arise in Figure a) if Δx is very small.

according to the mean value theorem. Similar reasoning holds for the minimum slope. Thus $\partial\tilde{\mathbf{p}}(u,v)/\partial u$ is bounded by $\partial T_p/\partial u \pm (\overline{r_p} - \underline{r_p})/\Delta x$.

It should be noted that fractional tessellation may introduce edges that are arbitrarily short, since new vertices may be inserted at the position of old ones. This makes it very hard to bound the derivatives of Taylor models with remainder terms, as we must assume that $\Delta x = 0$. We propose to modify the fractional tessellation algorithm so that new vertices are inserted in a bi-linearly interpolated fashion. As shown in Figure 5b, we find the point $\mathbf{q}(t)$ by linearly interpolating between the two neighbors $\mathbf{p_0}$ and $\mathbf{p_1}$. Then we interpolate again between the actual position, $\mathbf{f}(t)$, and $\mathbf{q}(t)$. Given this modification, one can show that the derivative from the previous section will behave as if the minimum edge length is half of the edge length in a corresponding uniform tessellator. This means that we can now bound the slope.

The second issue concerns treatment of texture maps. As can be seen in Equation 12, a Taylor model with texture lookups will contain terms which depend on some unknown texture function $a(u,v)$. When such a term is differentiated, we will obtain partial derivatives $\partial a(u,v)/\partial u$ and $\partial a(u,v)/\partial v$. Our solution is to evaluate these terms using textures of pre-computed differentials. These differential textures are treated just like the regular textures described in Section 2.4, and increase the dimension of the Taylor models. It should be noted that this increase in dimension is not computationally costly as we rarely get more than linear dependencies of a texture.

Occlusion Culling

Our occlusion culling technique is similar to hierarchical depth buffering [8], except that we use only a single extra level (8×8 pixel tiles) in the depth buffer. The

maximum depth value, z_{max}^{tile}, is stored in each tile. This is a standard technique in GPUs [20] used when rasterizing triangles. We project our clip-space bounding box, $\hat{\mathbf{b}}$, and visit all tiles overlapping this axis-aligned box. At each tile, we perform the classic occlusion culling test: $z_{min}^{box} \geq z_{max}^{tile}$, which indicates that the box is occluded at the current tile if the comparison is fulfilled. The minimum depth of the box, z_{min}^{box} is obtained from our clip-space bounding box, and the maximum depth of the tile, z_{max}^{tile}, from the hierarchical depth buffer (which already exists in a contemporary GPU). Note that we can terminate the testing as soon as a tile is found to be non-occluded, and that it is straightforward to add more levels to the hierarchical depth buffer. Our occlusion culling test can be seen as a very inexpensive pre-rasterizer of the bounding box of the triangle to be tessellated. Since it operates on a tile basis, it is less expensive than an occlusion query.

3 Implementation

We have implemented our automatic culling unit in a C++ software framework simulating the GPU pipeline. We execute the bounding shader program before tessellating each base primitive. We noted that both view frustum and backface culling may be realized in the bounding shader, and our implementation generates code for this. The output of our bounding shader is therefore a single boolean indicating if the base triangle should be culled or not, and a positional bounding box. The bounding box is required for the occlusion culling, which cannot be implemented in vertex shader code as it includes (coarse level) rasterization operations. Occlusion culling is implemented further down the pipeline as a quick rasterization algorithm.

We use fourth order Taylor models in our program analysis. This gives us an exact representation of the position and normal for cubic polynomial surfaces, which are frequently used. Some examples are curved PN-triangles [26] and bicubic patches, such as Loop and Shaefer's Catmull-Clark approximation [17]. Higher-order terms will be handled by the remainder term in the Taylor model.

We believe that our automatic tessellation culling could be implemented in a graphics hardware system at a moderate cost. For a full implementation, we need additional hardware that enables us to do the following:

- Execute a bounding shader once per base primitive. The instruction set and program inputs are identical to the vertex shader. With unified shader architectures, this should be fairly straightforward to add.
- Perform the occlusion culling described in Section 2.6.
- Remove a base triangle before tessellation based on a boolean culling flag.

The remaining tasks can be done either in the bounding shader code or in a preprocessing step in a driver.

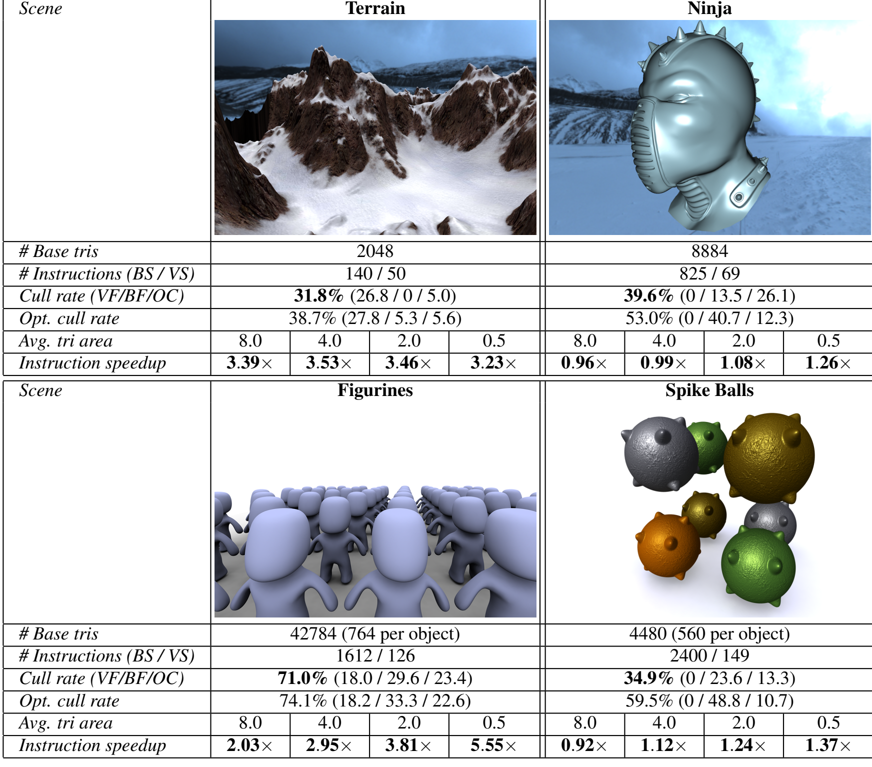
A partial implementation of our automatic culling algorithm could be realized on current hardware in two passes. First, we would execute the bounding shader program and use it to compute tessellation factors for the subsequent rendering pass. The tessellation factor can then be set to zero for all culled triangles.

4 Results

Our test setup and results will be presented in this section. We use the software GPU simulator described in the previous section, and render all images in 1920×1280 resolution. Since, to the best of our knowledge, no system exists that can *automatically* perform culling based on vertex shader analysis, cull shader generation, and on-the-fly execution, we decided to compare our system against an "optimal" culling unit. This unit can, for example, backface cull a base triangle only if all tessellated triangles are backfacing. In practice, such optimal culling uses too much resources and so will not provide much (if any) speedup. However, from a scientific point of view, it is interesting to find out how close to an optimal culling unit our algorithm performs.

To investigate the performance of our algorithm, we use four test scenes, two of which have recently been used in GPU tessellation contexts. These are *Ninja*, *Terrain*, *Figurines*, and *Spike balls*, as can be seen in Table 1. In addition, we decided to use four tessellation rates, giving approximate triangle areas of 8, 4, 2, and 0.5 pixels. We motivate these rates by the fact that GPUs were balanced for eight-pixel triangles already two years ago [23], and the introduction of tessellation units indicates an intention to further decrease the size. Our highest tessellation rate (0.5 pixels) is inspired by production rasterization pipelines [1], which is another possible application of our culling unit.

The **Terrain** scene is a common usage area for tessellation. A coarse mesh is finely tessellated and displaced. The camera moves over the landscape, and so a fair amount of view frustum culling should be possible. This scene has the most inexpensive bounding shader, which is only $2.8\times$ as expensive as the vertex shader. The **Ninja** scene uses displacement mapping along an interpolated normal. The model is always inside the view frustum, and so only backface and occlusion culling can occur. Furthermore, the base mesh is highly tesselated, which makes it a rather hard case for our algorithm. A highly tesselated base mesh will not generate as many tesselated vertices, and hence, there is not as much to be gained by the culling. The **Figurines** scene consists of a set of models using PN-triangles [26], i.e., cubic Bézier triangles. We included this scene to demonstrate that render-time mesh smoothing can be handled efficiently by our culling algorithm. The scene shows a grid of meshes seen from the front and tests all three types of culling. It has the highest number of base primitives, but also has many more separate objects and the most complex geometry. The final scene, **Spike Balls**, shows PN-triangulated



Scene	**Terrain**				**Ninja**			
# Base tris	2048				8884			
# Instructions (BS / VS)	140 / 50				825 / 69			
Cull rate (VF/BF/OC)	**31.8%** (26.8 / 0 / 5.0)				**39.6%** (0 / 13.5 / 26.1)			
Opt. cull rate	38.7% (27.8 / 5.3 / 5.6)				53.0% (0 / 40.7 / 12.3)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**3.39**×	**3.53**×	**3.46**×	**3.23**×	**0.96**×	**0.99**×	**1.08**×	**1.26**×

Scene	**Figurines**				**Spike Balls**			
# Base tris	42784 (764 per object)				4480 (560 per object)			
# Instructions (BS / VS)	1612 / 126				2400 / 149			
Cull rate (VF/BF/OC)	**71.0%** (18.0 / 29.6 / 23.4)				**34.9%** (0 / 23.6 / 13.3)			
Opt. cull rate	74.1% (18.2 / 33.3 / 22.6)				59.5% (0 / 48.8 / 10.7)			
Avg. tri area	8.0	4.0	2.0	0.5	8.0	4.0	2.0	0.5
Instruction speedup	**2.03**×	**2.95**×	**3.81**×	**5.55**×	**0.92**×	**1.12**×	**1.24**×	**1.37**×

Table 1: Performance evaluation for our four test scenes. The instructions row shows the number of scalar instructions for the vertex shader (VS), and the bounding shader (BS). The cull rate row shows how many base primitives our algorithm can automatically cull. The bold figure is the total culling rate, and the numbers in the parenthesis are for view frustum (VF), back face (BF), and occlusion (OC) culling. The Optimal cull rate row shows the best possible culling. For each scene, we then show Instruction speedup for four different tessellation rates, so that the average tessellated triangle area is 8, 4, 2, and 0.5. These figures were computed by dividing the number of instructions to compute the vertex position of every tessellated triangle by the sum of the instructions used by our bounding shader program and the instructions used for the non-culled vertices.

spheres with displacement mapping. This scene has the most expensive bounding shader program, approximately 2400 instructions long. Since everything is inside the view frustum, this scene only uses backface and occlusion culling.

We present our performance figures in Table 1. The culling rates show how our culling unit compares to the optimal culling unit described above. Note that our culling unit in some cases performs better than the optimal unit at occlusion

culling. This only occurs because the occluded triangles were removed by the backface culling test in the optimal culling unit. For the culling rate figures, we execute our bounding shader for every base triangle in order to make a fair comparison to the optimal culling unit. For the performance figures (Instruction speedup), we instead execute the bounding shader based on Equation 13, where we chose $p(cull) = 0.5$.

It should be noted that the instruction counts for bounding and vertex shaders presented in Table 1 is the number of *scalar* instructions used, and not vector instructions. The motivation for this is that modern graphics hardware architectures use scalar instructions internally, and achieve parallelism by operating on multiple vertices or pixels instead. Note also that we counted multiplications and additions separately for simplicity. It is, however, likely that the bounding shader programs can be significantly shortened using multiply-add.

It should also be noted that our performance numbers do not include the actual tessellation (i.e., generation of connected vertices) nor execution of instructions in the vertex shader not dealing with computing vertex position (e.g. vertex lighting, tangent space transforms etc.). In addition, our simple occlusion culling is not included either since it has to be implemented in custom hardware, but given its simple nature it should be very efficient. In summary, we believe that our performance would be even better if these factors were taken into account.

Discussion Given that our culling is automatically derived from a vertex shader program, we consider our culling rates very high, compared to the optimal culling rates. Note that we have intentionally avoided very simple test scenes where, for example, a detailed, tessellated character is behind a wall. In such cases, our occlusion culling would cull the entire character given that the wall was rendered first. One thing we noted in particular is that backface culling of displacement mapped surfaces is a very hard task (although our algorithm handles the Ninja and Spike Balls scenes fairly well).

We also compared our culling results with generalized interval arithmetic (first order Taylor models), and noted that the results directly dropped to 0% culling for the scenes with Bézier surfaces, namely, Figurines and Spike Balls. This clearly motivates our choice of higher-order Taylor models as a suitable arithmetic for bounding shaders. For the remaining test scenes, Terrain and Ninja, we get the exact same behavior for generalized interval arithmetic and higher order Taylor models. This is to be expected, since our Taylor model implementation never use higher order than necessary. Thus, the culling performance, and the instruction ratio between the bounding and vertex shader, are identical for these scenes.

Our PN-triangle scenes (Figurines and Spike Balls) use third order surfaces, similar to the popular Catmull-Clark subdivision schemes. We also performed initial experiments with Loop and Schaefer's [17] implementation of Catmull-Clark, for the Figurines scene. As the surfaces are bicubic, they contain more high-order terms than corresponding Bézier triangles, and consequently the bounding shader becomes more expensive (6536 instructions bounding shader, and 159 instructions

vertex shader, as compared to 1612/126 instructions with PN-triangles). However, we only need to execute the bounding shader once for every quad, in this case. The culling rate was within 2% of that of the PN-triangle version. It should be noted that shaders as long as 6536 instructions may not fit in current instruction caches, which may harm performance. However, we believe that future hardware will be able to handle longer shaders.

As can be seen in Table 1, the performance is very high for scenes with view frustum culling (the Terrain and Figurines scenes). In all scenes, we use fractional tessellation and projected edge lengths to determine the tessellation factors for each edge of the base triangles. A fundamental problem with this approach is that we cannot conservatively determine if the edge will be visible or not without tesselating it. Therefore, we chose tessellation factors based on projected edge lengths, without clipping the edges by the view frustum. This leads to highly tessellated base triangles close to the (infinite) near clipping plane, and consequently we get a substantial speedup if we can cull these. This is a general problem in tessellation, and not bound to our application. In fact, using our culling unit makes it much simpler to design a tessellation heuristic, since culling is handled automatically.

Our tessellation heuristic also includes a maximum tessellation factor to avoid generating base triangles being too highly tessellated. This limit is reached when the Terrain scene is rendered at high tessellation rates. Consequently, the vertex rate of the base triangles close to the camera (many of which we can cull) goes down, and this explains why the performance gain (instruction speedup) for this scene decreases when we increase the tessellation rate.

It should be noted that our culling technique is not limited to polynomial surfaces. Fig. 1 shows an example of a vertex shader with sines and cosines, wrapping a planar surface to a torus. Still, we can cull 56% (60% optimal) of the triangles before tessellation.

5 Conclusion and Future Work

The trend in GPU rendering is steadily continuing to close in on the quality of rasterization-based production pipelines. Using hardware to obtain highly tesselated objects is another step in this direction. We are therefore excited about the recent developments in hardware tessellation, and hopefully, our work can be used in future implementations of GPUs to accelerate rendering further. As we have shown, this would give significantly better performance, and since our technique is fully automatic, we believe the application developers would find more motivation to use hardware tessellation if the culling is done for them by the system. For future work, we would like to investigate hierarchical tessellation, so that parts of a base primitive can be culled, or even several base primitives in a single cull operation. In addition, we have realized that backface culling is the most difficult type of culling when it comes to handling arbitrary vertex shaders. Therefore, we would like to do research on novel techniques to further increase the backface cull rate at a low cost. Furthermore, our work can be used in a software rendering pipeline as

well, and it would be interesting to evaluate exactly what kind of performance can be obtained in such contexts.

Acknowledgements

Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation. In addition, we acknowledge support from the Swedish Foundation for strategic research. Thanks to Natalya Tatarchuck for giving us access to the ninja model.

Bibliography

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.

[2] Martin Berz and Georg Hoffstätter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds. *Reliable Computing,*, 4(1):83–97, 1998.

[3] J. L. D. Comba and J. Stolfi. Affine Arithmetic and its Applications to Computer Graphics. In *Proc. VII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'93)*, pages 9–18, 1993.

[4] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 96–102, 1987.

[5] Michael Doggett. Xenos: XBOX 360 GPU. Eurographics presentation, September 2005.

[6] Michael Doggett and Johannes Hirche. Adaptive View Dependent Tessellation of Displacement Maps. In *Graphics Hardware*, pages 59–66, 2000.

[7] Ned Greene and Michael Kass. Error-bounded antialiased rendering of complex environments. In *Proceedings of ACM SIGGRAPH 1994*, pages 59–66, 1994.

[8] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93*, pages 231–238, August 1993.

[9] Eric Haines and John Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proceedings of the Second Eurographics Workshop on Rendering*, pages 122–138, 1994.

[10] Chang-Young Han, Yeon-Ho Im, and Lee-Sup Kim. Geometry Engine Architecture with Early Backface Culling Hardware. *Computers & Graphics,*, 29(5):415–425, June 2005.

[11] E. R. Hansen. A Generalized Interval Arithmetic. In *Proceedings of the International Symposium on Interval Mathemantics*, pages 7–18, 1975.

[12] Jon Hasselgren and Thomas Akenine-Möller. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics,*, 26(3):92.1–92.10, 2007.

[13] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACE Transactions on Graphics,*, 17(3):158–176, 1998.

[14] Ralf Hungerbuhler and Jurgen Garloff. Bounds for the Range of a Bivariate Polynomial over a Triangle. *Reliable Computing,*, 4(1):3–13, 1998.

[15] Subodh Kumar and Dinesh Manocha. Hierarchical Visibility Culling for Spline Models. In *Graphics Interface*, pages 142–150, 1996.

[16] Qun Lin and J.G. Rokne. Interval Approximation of Higher Order to the Ranges of Functions. *Computers & Mathematics with Applications*, 31(7):101–109, 1996.

[17] Charles Loop and Scott Schaefer. Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches. Technical report, MSR-TR-2007-44, Microsoft Research, 2007.

[18] Kyoko Makino and Martin Berz. Taylor Models and Other Validated Functional Inclusion Methods. *International Journal of Pure and Applied Mathematics,*, 4(4):379–456, 2003.

[19] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[20] Steve Morein. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*. ACM Press, August 2000.

[21] Henry Moreton. Watertight Tessellation using Forward Differencing. In *Graphics Hardware*, pages 25–32, 2001.

[22] Kevin Moule and Michael D. McCool. Efficient Bounded Adaptive Tesselation of Displacement Maps. In *Graphics Interface*, pages 171–180, 2002.

[23] Matt Pharr. Interactive Rendering In The Post-GPU Era. Keynote in Graphics Hardware, 2006.

[24] Leon A. Shirman and Salim S. Abi-Ezzi. The Cone of Normals Technique for Fast Processing of Curved Patches. *Computer Graphics Forum,*, 12(3):261–272, 1993.

[25] Natalya Tatarchuk, Christopher Oat, Jason L. Mitchell, Chris Green, Johan Andersson, Martin Mittring, Shanon Drone, and Nico Galoppo. Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH course, 2007.

[26] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 159–166, 2001.

[27] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware*, pages 7–14, 2005.

[28] X. Zhang, S. Redon, M. Lee, and Y.J. Kim. Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling. *ACM Transactions on Graphics,*, 26(3):15.1–15.10, 2007.