

# Efficient Compression and Rasterization Algorithms for Graphics Hardware

Jon Hasselgren  
Department of Computer Science  
Lund University



LUND INSTITUTE OF TECHNOLOGY  
Lund University

ISSN 1652-4691  
Licentiate Thesis 6, 2006  
LU-CS-LIC:2006-3

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

Email: [jon.hasselgren@cs.lth.se](mailto:jon.hasselgren@cs.lth.se)  
WWW: [http://www.cs.lth.se/home/Jon\\_Hasselgren](http://www.cs.lth.se/home/Jon_Hasselgren)

## **Abstract**

Despite rapid development, modern graphics hardware is still much too slow to render photo-realistic images in real time, and it will most likely remain so if we rely only on yearly growth in hardware performance. Therefore, better algorithms are constantly needed in order to advance the field.

In the first part of this thesis, we present new algorithms for hardware rasterization. We investigate different aspects of sampling, which leads to a new family of very inexpensive sampling schemes. Based on a perceptual error metric, the best performing patterns are presented. Our study of sampling also considers the use of conservative rasterization, and we present two novel algorithms designed to work on existing graphics hardware. Our final contribution for rasterization is a hardware algorithm for efficiently rasterizing multiple views of a three-dimensional scene. The current norm for multi-view rasterization is to process each view separately, but in this work, we exploit the inherent coherence by considering all views simultaneously. This is done by sorting the rasterization order among all views.

In the second part of this thesis, we consider compression algorithms for graphics hardware. Although compression does not provide anything new in terms of features, it is a good way of improving performance and lowering memory usage in a hardware system. Our contributions in this field are two new compression algorithms. The first is suited for real-time compression and decompression of depth values, and the second suited is for compression and decompression of high dynamic range textures.

---



---

## Preface

This thesis is for the Licentiate degree, and summarizes my research on rasterization and compression algorithms for graphics hardware. The following papers are included in this thesis:

- Jon Hasselgren and Tomas Akenine-Möller, “A Family of Inexpensive Sampling Schemes,” in *Computer Graphics Forum* 24(4):843–848, 2005.
- Jon Hasselgren, Tomas Akenine-Möller and Lennart Ohlsson, “Conservative Rasterization,” in *GPU Gems 2*, pages 677–690. Addison-Wesley Professional, 2005.
- Jon Hasselgren and Tomas Akenine-Möller, “An Efficient Multi-View Rasterization Architecture,” in *Eurographics Symposium on Rendering*, pages 61–72, 2006.
- Jon Hasselgren and Tomas Akenine-Möller, “Efficient Depth Buffer Compression,” in *Graphics Hardware*, pages 102–110, 2006.
- 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.

The following paper is not included in the thesis:

- Jon Hasselgren and Tomas Akenine-Möller, “Textured Shadow Volumes,” to appear in *Journal of Graphic Tools*.

## 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 during Tomas’ stay in San Diego.

I would also like to thank my colleagues in the computer graphics group, Petrik Clarberg, Calle Lejdfors, and Jacob Munkberg, and 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 funded by Vetenskapsrådet.

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

---

# Contents

1	Introduction . . . . .	1
2	Rasterization . . . . .	2
2.1	Sampling . . . . .	3
2.2	Conservative Rasterization . . . . .	4
2.3	Multi-viewpoint Rasterization . . . . .	5
3	Compression . . . . .	6
3.1	Depth buffer compression . . . . .	7
3.2	High Dynamic Range Texture compression . . . . .	8
	Bibliography . . . . .	9
<b>Paper I: A Family of Inexpensive Sampling Schemes</b>		<b>13</b>
1	Introduction . . . . .	15
2	Previous Work . . . . .	15
3	Notation . . . . .	17
4	The FLIPTRI Sampling Scheme . . . . .	18
5	Sampling patterns . . . . .	19
5.1	Initial Pattern Generation . . . . .	19
5.2	Pattern Ranking and Optimization . . . . .	19
6	Results . . . . .	19
6.1	Evaluation . . . . .	19
7	Conclusion and Future Work . . . . .	22
	Bibliography . . . . .	25
<b>Paper II: Conservative Rasterization</b>		<b>27</b>
1	Introduction . . . . .	29
2	Problem Definition . . . . .	29
3	Two Conservative Algorithms . . . . .	31

---

3.1	Clip Space . . . . .	32
3.2	The First Algorithm . . . . .	32
3.3	The Second Algorithm . . . . .	34
3.4	Underestimated Conservative Rasterization . . . . .	37
4	Robustness Issues . . . . .	37
5	Conservative Depth . . . . .	37
6	Results and Conclusions . . . . .	40
	Bibliography . . . . .	41
<b>Paper III: An Efficient Multi-View Rasterization Architecture</b>		<b>43</b>
1	Introduction . . . . .	45
2	Motivation . . . . .	46
3	Background: Multi-View Rasterization . . . . .	47
3.1	Brute-Force Multi-View Rasterization . . . . .	48
3.2	Multi-View Projection . . . . .	48
4	New Multi-View Rendering Algorithms . . . . .	49
4.1	Scanline-Based Multi-View Traversal . . . . .	51
4.2	Tiled Multi-View Traversal . . . . .	52
4.3	Approximate Pixel Shader Evaluation . . . . .	53
4.4	Accumulative Color Rendering . . . . .	57
5	Implementation . . . . .	58
6	Results . . . . .	59
6.1	Accumulative Color Rendering . . . . .	62
6.2	Small triangles . . . . .	63
7	Discussion . . . . .	63
8	Conclusion and Future Work . . . . .	64
	Bibliography . . . . .	67
<b>Paper IV: Efficient Depth Buffer Compression</b>		<b>71</b>
1	Introduction . . . . .	73
2	Architecture Overview . . . . .	74
3	Depth Buffer Compression - State of the Art . . . . .	75
3.1	Fast z-clears . . . . .	75
3.2	Differential Differential Pulse Code Modulation . . . . .	76
3.3	Anchor encoding . . . . .	77
3.4	Plane Encoding . . . . .	78
3.5	Depth Offset Compression . . . . .	80

---

4	New Compression Algorithms . . . . .	80
4.1	One plane mode . . . . .	81
4.2	Two plane mode . . . . .	83
5	Evaluation . . . . .	84
6	Conclusions . . . . .	87
	Bibliography . . . . .	89
 <b>Paper V: High Dynamic Range Texture Compression for Graphics Hardware</b>		<b>91</b>
1	Introduction . . . . .	93
2	Related Work . . . . .	94
3	Color Spaces and Error Measures . . . . .	95
3.1	Color Spaces . . . . .	96
3.2	Error Measures . . . . .	97
4	HDR S3 Texture Compression . . . . .	98
5	New HDR Texture Compression Scheme . . . . .	100
5.1	Luminance Encoding . . . . .	100
5.2	Chrominance Line . . . . .	102
5.3	Chroma Shape Transforms . . . . .	102
6	Hardware Decompressor . . . . .	105
7	Results . . . . .	107
8	Conclusions . . . . .	109
	Bibliography . . . . .	115



# 1 Introduction

Computer graphics is the process of generating, or *rendering*, images using a computer. In particular, three-dimensional computer graphics has been given much attention. Here, an image is rendered from an abstract representation of a three-dimensional scene, which is usually modeled in a hierarchical fashion. At the top level are objects, which are used to represent real geometric objects (such as, for instance, a chair or a table). As shown in Figure 1, these objects are often built from triangles that approximate the surface of the actual object. A material is also assigned to each surface. The material can be a collection of images (so called textures), tabulated physical properties of the surface, and arbitrary computations. Given such a representation, it is possible to use a computer to generate a convincing image of the scene.

Computer graphics can be divided into two main categories: *offline rendering* [12], and *real-time rendering* [3]. Offline rendering is the main interest of the movie industry and the goal is to render as realistic images as possible. Accurate physical simulations are used to ensure high quality, and as a side effect, it usually takes several minutes, or in some cases even hours to render a single image. The images are then stored as a movie sequence that can be replayed later.

The other category, real-time rendering, is the type of graphics we see in games, computer aided design (CAD) programs, and visualization. The focus is on interactivity: the user should somehow be able to interact with a virtual scene, for instance by changing the viewpoint, and thereby determine the outcome of the next rendered image. Therefore, in real-time rendering, we strive after the best possible image quality that can be rendered fast enough so that the user is not disturbed by the delay between consecutive images. This rate lies in the range of around 80 images per second for games down to 5 images per second for some visualization applications.

Real-time rendering has received tremendous attention lately, and has been the driving factor behind *graphics hardware*: circuitry dedicated entirely to the task of rendering three-dimensional graphics. During a period of ten years, dedicated graphics hardware has gone from exclusive devices to something that is included in virtually all comput-

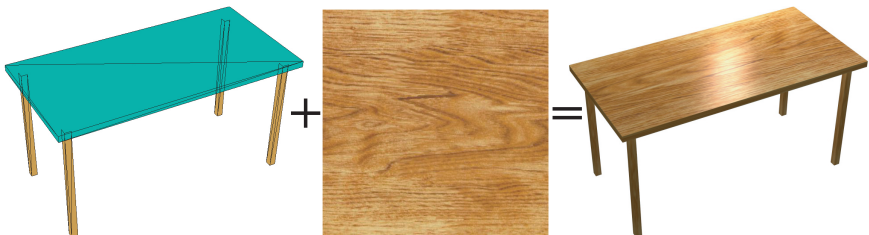


Figure 1: A three-dimensional model of a table. The figure shows, from left to right: The geometrical representation of the table, the image representing the material, and the rendered image of the table when it has been lit by a single lightsource.

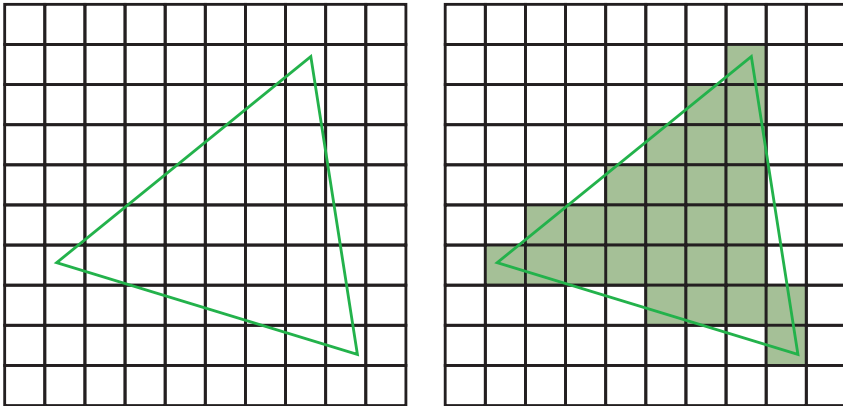


Figure 2: The purpose of the rasterization process is to identify and visit all pixels that overlap a given triangle. The left image shows a pixel grid with the geometrical representation of a triangle. In the right image, the pixels that lie within the triangle have been shaded.

ers on the market, and their speed have increased thousandfold. As a consequence, the quality gap between offline rendering and real-time rendering is steadily decreasing, but in order to close that gap we always find ourselves wanting more performance and better algorithms. That is the purpose of this thesis: to consider hardware improvements and new algorithms for rendering higher quality images with real-time performance.

This thesis summarizes five papers, where Paper 1 to 3 consider algorithms for rasterization, and Paper 4 and 5 consider performance-improving compression techniques.

## 2 Rasterization

Rasterization is the process of identifying and traversing all pixels that lie within a projected triangle, as shown in Figure 2. Popular algorithms for hardware-based rasterization [17, 20, 22] usually start from an overestimation of all pixels that overlap the triangle. Typical examples are the bounding box of the triangle, or even all pixels on the screen [17]. *Edge equations* [22] are then used to exactly determine which pixels overlap the triangle. In the simplest case, a single point is selected to represent the pixel (typically the center), and the edge equations are evaluated for this point. If the evaluated edge equations indicate that the point lie on the same side of all the edges of the triangle, then we define that the pixel lies within the triangle. Pixels within the triangle are sent to consecutive steps of the graphics pipeline for further processing. In the following, our contributions for rasterization will be summarized.



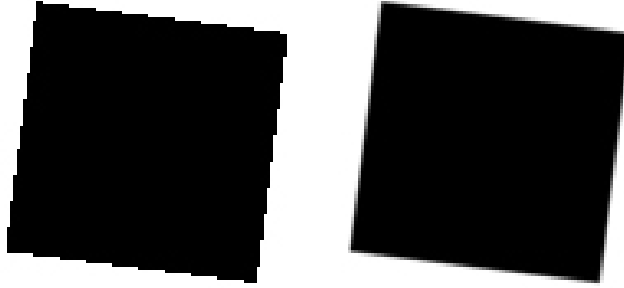


Figure 3: The left image shows a box rendered with point sampling (without supersampling) and the right image shows a box rendered using 64 samples supersampling.

## 2.1 Sampling

During the rasterization process, edge equations are evaluated in a single point in order to determine if a pixel lies within the triangle or not. This is in fact a sampling and reconstruction process, and as the Nyquist theorem suggests, we cannot accurately reconstruct a signal unless we sample with twice as high frequency as the highest frequency inherent in the signal.

In practice, this can cause annoying “staircase” effects often called “jaggies”, such as the ones shown in Figure 3. It is impossible to entirely get rid of these effects, since the frequency content of a triangle edge is infinite. However, we can greatly reduce the aliasing through supersampling [26]. That is, by using more than one sample point per pixel, and using a reconstruction filter that computes the color as a weighted average of the colors at each sample point.

Supersampling is a costly process, which increases the render time proportionally to the number of samples. Therefore, much research has been conducted on how to achieve the best possible quality for a given number of samples. This has led to a technique called sample sharing, where adjacent pixels share the same sample points. Sample sharing has the drawback of introducing a low-pass effect on the rendered image, which explains why it is used almost only by the real-time rendering community.

Another way of increasing the quality to performance ratio is to place the sample points as efficiently as possible. Much work has been done on this in the offline rendering field [13, 26] but unfortunately these techniques do not work well with the small number of sample points required for real-time rendering. A study in this field, that we believe is well suited for real-time rendering, has been conducted by Naiman [19], who examined how humans experience aliasing of edges with different orientation. Laine and Aila [15] further formalize Naiman’s results by presenting an error metric for the perceptual aliasing of edges using a particular filter.

In Paper 1, we explore the design space of extremely inexpensive multisampling schemes that cost between 1.25 and 2 samples per pixel, which could be very useful for mo-

tile graphics [4]. We perform an exhaustive search of the target family of sampling schemes, and generate all configurations with different properties. The resulting patterns are manually examined, and we discard the patterns that would obviously not perform well according to Naiman’s study. The remaining configurations are optimized based on Laine and Aila’s perceptual error metric [15], and we present and evaluate a suite of the best performing patterns.

## 2.2 Conservative Rasterization

Conservative rasterization is a field that is closely related to sampling. Given a triangle, we want to rasterize an overestimate or underestimate of the actual triangle. For the overestimate, all pixel regions that at least partially overlap the triangle are considered to belong to the triangle, and for the underestimate, only pixel regions completely within the triangle are considered to belong to the triangle.

These properties have been shown to be important for the correctness of some hardware accelerated-algorithms, such as collision detection [9, 18] and culling [16]. Therefore, hardware extensions for supporting conservative rasterization have been proposed [2]. However, hardware extensions made for one or a few particular functions are not always desirable, especially since the graphics hardware trend drives towards more general and programmable functionality.

In Paper 2, we propose two algorithms for conservative rasterization which can be implemented as shader programs on current graphics hardware. The first algorithm is inspired by the hardware-based algorithm of Akenine-Möller and Aila [2], but modified to fit in the current graphics pipeline. We extend each rasterized triangle so that it is guaranteed to overlap the sample points that rasterization would traverse. The extended triangle may overlap more pixels than optimal overestimated rasterization would produce (although it is still overestimating in a sense), and therefore we remove the excess pixels using a simple test in the fragment program. This algorithm seems to be best suited for current low-end and older graphics hardware as there is no strong dependence of the tessellation of the scene geometry. However, the performance is tightly coupled with the rendering resolution, and the performance therefore depends on the accuracy required by the algorithm.

For the second algorithm, we take a completely different approach, and directly create triangles that cover exactly the pixels that optimal overestimated rasterization would produce. This requires sending three times as many vertices to the graphics hardware which makes the algorithm more dependent on mesh tessellation, but less dependent on resolution. We believe that this is the algorithm that will work best in the long term as it tends to perform best on modern graphics hardware. It is also the algorithm that we think will benefit most from using geometry shaders [6], which will be introduced in future graphics hardware.



Figure 4: Stereoscopic image of Manhattan from 1909, to be viewed with crossed eyes. [The image is licensed under creative commons share alike license: <http://creativecommons.org/licenses/sa/1.0/>]

### 2.3 Multi-viewpoint Rasterization

It is a well known fact that the human brain use images from both eyes to determine the depth of different objects. Different techniques for showing such *stereo* images have been more or less successful. Examples are the red-green glasses used for three-dimensional cinema, the stereo images where you need to cross your eyes (see Figure 4), or virtual reality. In recent years, displays have been developed that can directly show stereo images [7, 11] without the use of peripheral equipment. Furthermore, some of these screens can display even more views, which means that the observer may move his head to see the scene from a slightly different angle.

Interestingly, there has been very little research conducted on how to efficiently render from multiple viewpoints, especially in the field of real-time and hardware-accelerated graphics. The attempts that has been made [10, 23, 27, 28] are all based on exotic hardware architectures which do not fit well with current graphics hardware, would cost too much to implement, could only produce approximate results, or are not thoroughly evaluated. Therefore, the norm is to use a normal graphics card and draw each view sequentially, or to use one graphics card per view. This brute force way of treating the views separately does not exploit any of the redundancy inherent in multi-viewpoint rendering.

In an attempt to solve these problems, we propose a new architecture in Paper 3, which extends current graphics hardware architectures with only small changes. First, we observe that a given point on the triangle is likely to result in the same computations and texture lookups for all views. We then use this observation to formulate an algorithm that performs a sorted rasterization which considers what viewpoint and

what position on the triangle that is most optimal to traverse next. This sorting procedure greatly improves the performance of the texture cache and consequently reduces the texture bandwidth, which is the main consumer of memory bandwidth. Memory bandwidth is the most common performance bottleneck in a graphics hardware system, and computational resources has been shown to grow substantially faster than memory bandwidth [21]. Therefore, it makes sense to optimize primarily for memory bandwidth consumption.

A great strength of our algorithm is that it is the first exact algorithm for multi-viewpoint rasterization. That is, it produces the exact same result as if we rendered the viewpoints sequentially using a single graphics card. However, we found that even more performance improvements could be made if we allow approximations. Therefore, we also present an approximate algorithm that can be implemented as an extension on top of the original algorithm. This is an important feature as it allows the approximation to be turned off or on based on the preferences of the application programmer or user. The approximative algorithm saves both bandwidth and computational resources simultaneously.

We have evaluated our algorithms using four different test scenes, and observed strongly sub-linear texture bandwidth growth for an increasing number of views. Overall, we have observed bandwidth gains of up to an order of a magnitude when compared to the naive algorithm. For the approximate algorithm, we have observed approximation rates of up to 95% for approximated views.

### 3 Compression

As previously mentioned, one of the most common performance bottlenecks in a graphics hardware system is the memory bandwidth capacity. Thus, much effort is spent on lowering the bandwidth utilization [1], and this is often done by compressing the data stored in memory. The compression algorithms can be divided into two categories.

The first category is that of *buffer compression*. Buffers are subject to a frequent read-modify-write access pattern, and it is therefore crucial to use fast compression and decompression algorithms that can be performed on the fly. Furthermore the compression must be lossless, because compression errors would be accumulated for each read-modify-write operation. This also implies that the compression must have a variable bit-rate, and that an uncompressed fallback is needed. Memory allocation must be done with the worst case in mind, which means that we can reduce only bandwidth and not actual memory footprint.

The second category is the compression of application resources, of which a typical example is *texture compression* [5, 14, 29]. This type of compression algorithms is based on the assumption that a static image is compressed once and then read frequently. As a consequence, hardware decompression must be fast, but the compression process can be precomputed on the CPU, using complex and time-consuming algorithms. Also, it is possible to use lossy compression since the resource is only compressed



Figure 5: An HDR image of Petrik’s room under two different exposures. Notice how the HDR format can accurately represent the faint light in the room as well as the bright light coming through the window.

once, and since compression can be turned off for individual resources when quality is too low. Fixed-rate compression methods are often applied, as this allows both memory savings and fast random access.

### 3.1 Depth buffer compression

Depth buffer compression is the most common form of buffer compression algorithms. The depth buffer information is well suited for compression, and the major hardware manufacturers have spent much time researching different compression algorithms. Unfortunately all this research has gone directly into patents, and there is very little or no literature available on how efficient hardware depth buffer architectures work.

Paper 4 is to some extent a survey over the state of the art of a modern depth buffer system. We have summarized the information that we could get from the patents, and present the design of a modern depth buffer architecture as well as a summary of the most popular compression algorithms. In addition, we also present a novel compression algorithm.

This algorithm is, as so many others before it, based on representing the depth values of a block of pixels as one or more separate planes. We found that by viewing the problem as plane interpolation, we can save approximately one bit per pixel by applying a simple correction of the plane derivatives. We also improve on previous approaches for coding multiple planes, and present robust one and two plane modes of our algorithm.

Our evaluations show that our algorithm performs substantially better than previous algorithms when compressing small blocks of pixels. We believe that this makes it ideally suited for use in mobile graphics devices [4].

### 3.2 High Dynamic Range Texture compression

Traditionally, images and textures used in computer graphics are encoded in formats that directly relate to how an image is displayed on the screen. In these formats, a color is represented by a *(red, green, blue)* tuple where each value dictates the intensity of the corresponding color channel. The values are restricted to the range  $[0, 1]$  where 0 indicates no intensity and 1 is the maximum intensity of the display.

As these formats are directly related to the display device they fail to capture the full range of luminance in the real world, which makes it nearly impossible to simulate some phenomena. The concept of *High Dynamic Range* (HDR) images [24] relaxes this range limitation of ordinary images and allows the full physical luminance range to be stored (see Figure 5). This is done by encoding each color channel in a non-linear fashion (for instance using IEEE floating point values [8]), which naturally requires more bits per pixel than the traditional format. As a consequence the use of HDR textures increases the bandwidth utilization substantially. Once again we find ourselves having to compress data for better performance.

Even though texture compression algorithms is an established field of science, all previous algorithms have targeted the traditional texture formats and do not easily generalize to the non-linear representations of HDR color values. On the HDR image compression side, some progress has also been made, but the bulk of the algorithms are either too complicated and not meant for real-time decompression, or too simple in the sense that they are simply different quantization methods which do not provide high enough compression rates or provide too low quality.

Our contribution to this field is Paper 5, which presents the first<sup>1</sup> algorithm targeted for HDR texture compression. The algorithm is loosely based on traditional texture compression algorithms, but we found that many changes and different prioritizations had to be made to optimize the format for HDR data. Furthermore, the higher bit-rate of HDR textures allows us to employ more complex compression algorithms than traditional formats, while maintaining the same compression ratio. We thoroughly evaluate the format using a multitude of test images and error metrics, including a new error metric which is proposed in the paper.

The author was involved in the conceptual design of the algorithm, as well as some unreleased (“dead end”) implementation. He also contributed through the extended versions of S3TC compression algorithms, and as a co-author in the writing and internal reviewing process of the paper.

---

<sup>1</sup>Nokia simultaneously developed another HDR texture compression algorithm [25], which was presented at the same conference.

# 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 Timo Aila. Conservative Tiled Rasterization Using a Modified Triangle Setup. *Journal of graphics tools*, 10(2):1–8, 2005.
- [3] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A.K. Peters, 2nd edition, June 2002.
- [4] 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.
- [5] A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, 1996.
- [6] David Blythe. The Direct3D 10 System. In *ACM Transactions on Graphics*, volume 25, pages 724–734, 2006.
- [7] Neil A. Dodgson. Autostereoscopic 3D Displays. *IEEE Computer*, 38(8):31–36, 2005.
- [8] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. In *ACM Computing Surveys*, volume 23, pages 5–48, 1991.
- [9] 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.
- [10] Michael Halle. Multiple Viewpoint Rendering. In *Proceedings of ACM SIGGRAPH 98*, volume 32, pages 243–254, 1998.
- [11] B. Javidi and Eds F. Okano. *Three-Dimensional Television, Video, and Display Technologies*. Springer-Verlag, 2002.

- [12] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.
- [13] Alexander Keller. Strictly Deterministic Sampling Methods in Computer Graphics. Technical report, Mental Images, 2001.
- [14] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics*, 20(4):475–481, 1996.
- [15] Samuli Laine and Timo Aila. A Weighted Error Metric and Optimization Method for Antialiasing Patterns. *Computer Graphics Forum*, 25(1):83–94, 2006.
- [16] Brandon Lloyd, Jeremy Wendt, Naga Govindaraju, and Dinesh Manocha. CC Shadow Volumes. In *Eurographics Symposium on Rendering*, pages 197–205, 2004.
- [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] 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.
- [19] Avi C. Naiman. Jagged edges: When is filtering needed? *ACM Transactions on Graphics*, 17(4):238–258, 1998.
- [20] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 89–96. ACM Press, 1997.
- [21] John Owens. Streaming Architectures and Technology Trends. In *GPU Gems 2*, pages 457–470. Addison-Wesley Professional, 2005.
- [22] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, pages 17–20. ACM, August 1988.
- [23] Dennis R. Proffitt and Mary Kaiser. Hi-Lo Stereo Fusion. In *ACM SIGGRAPH 96 Visual Proceedings*, page 146, 1996.
- [24] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 1st edition, 2005.
- [25] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. High Dynamic Range Texture Compression. 25(3):707–712, 2006.



- [26] Peter S. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois, 1991.
- [27] J. Stewart, E. P. Bennett, and L. McMillan. PixelView: A View-Independent Graphics Rendering Architecture. In *Graphics Hardware*, pages 75–84, 2004.
- [28] 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.
- [29] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–364, 1996.



# Paper I

---

## A Family of Inexpensive Sampling Schemes

Jon Hasselgren<sup>‡</sup>   Tomas Akenine-Möller<sup>‡</sup>   Samuli Laine<sup>\*†</sup>

<sup>‡</sup>Lund University,

<sup>\*</sup>Helsinki University of Technology/TML,

<sup>†</sup>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 1.1b and 1.1c) to perform supersampling with two samples per pixel. Placing the samples diagonally instead (Figure 1.1d) gives a slight visual improvement. The  $2 \times 2$  *box pattern* (Figure 1.1e) 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  $2 \times 2$  box pattern is obviously due to its simple implementation. However, it should be noted that the *Rotated Grid Supersampling* (RGSS) pattern (Figure 1.1f) delivers much better quality with equal cost.

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

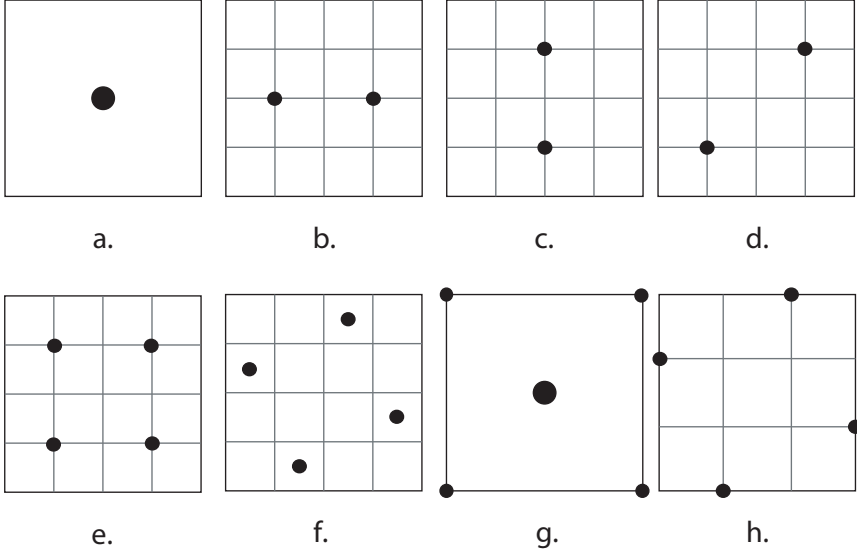


Figure 1.1: A collection of sampling schemes. From top left to bottom right: a) Centroid sampling, b) Kyrø horizontal, c) Kyrø vertical, d) Diagonal, e)  $2 \times 2$  box pattern, f) Rotated Grid Supersampling (RGSS), g) Quincunx, h) FLIPQUAD.

ing 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 1.1g). 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 1.1h, 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 1.2a. 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 importance of an-

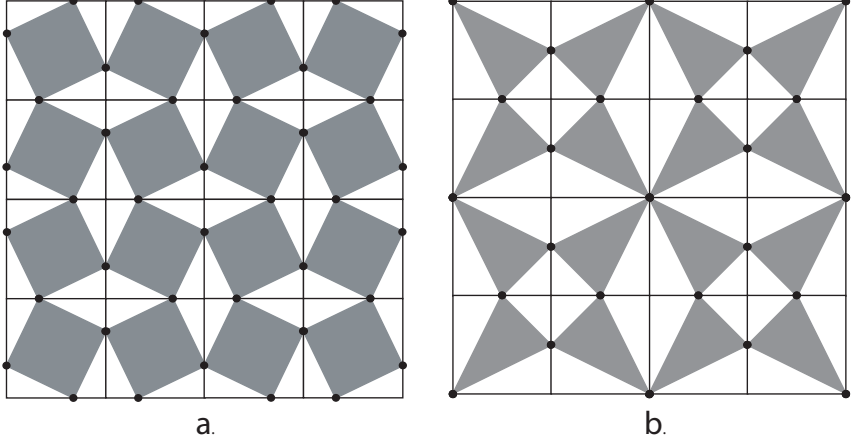


Figure 1.2: The a) FLIPQUAD and b) FLIPTRI sampling patterns repeated and reflected over  $4 \times 4$  pixels. For illustration purposes, shaded regions are added to emphasize the sample points that are used for reconstructing the pixels.

tialiasing 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  $45^\circ$ . 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  $45^\circ$  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 pixels used

Sample point position	Cost
Corner	0.25
Edge	0.5
Pixel	1

Table 1.1: Cost of differently placed sample points in the reflected  $P(4, 1^+, n)$  class

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 1.1 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 1.2b, 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 1.1, 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 ( $90^\circ$ ,  $180^\circ$  or  $270^\circ$ ) 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 1.3, 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.

### 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 1.3 and rasterized a number of test images. Figure 1.4 shows the results of drawing

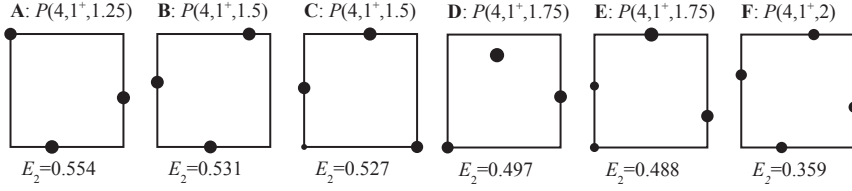


Figure 1.3: The figure shows the best schemes in terms of the error metric  $E_2$  in each class. The area of each circle is proportional to the weight of the corresponding sample point. For  $n = 1.5$  and  $n = 1.75$ , two patterns were found with similar error measures, but with different number of 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)

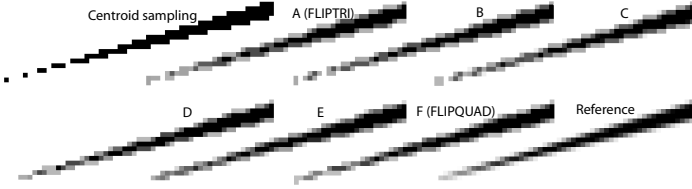


Figure 1.4: A thin polygon rendered using the sampling schemes, A through F, from Figure 1.3. The upper left polygon uses a single sample point per pixel while the reference polygon is supersampled with 1024 jittered grid sample points and uses a  $4 \times 4$  pixel Mitchell-Netravali reconstruction filter.

a thin polygon with the respective schemes as well as using centroid sampling and a high quality supersampling pattern for reference. Figure 1.5 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 1.4 and 1.5 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 the deviations to compute the Root Mean Square Error (RMSE) as

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=0}^{255} C(i) i^2} \quad (1.1)$$

Where  $C(i)$  is the number of pixels with a given deviation,  $i$ , from the reference ani-

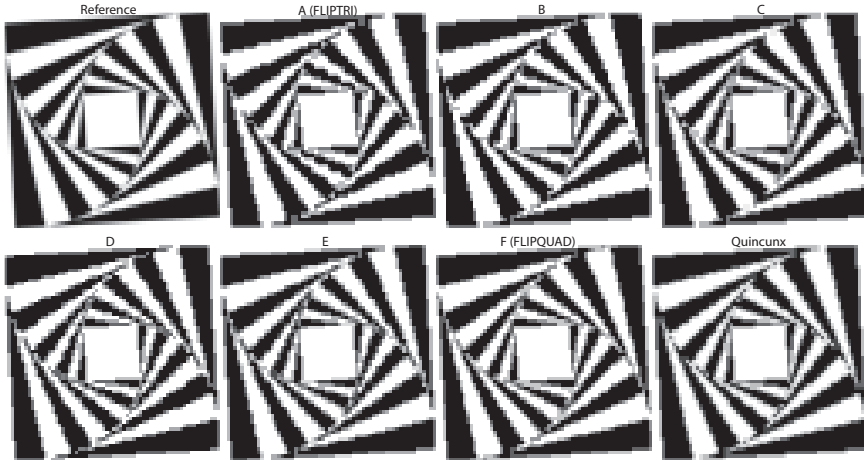


Figure 1.5: A spiral test image rendered using the sampling schemes, A through F, from Figure 1.3. The reference image is supersampled with 1024 jittered grid sample points and uses a  $4 \times 4$  pixel Mitchell-Netravali reconstruction filter. An image rendered using the Quincunx pattern is also included for comparison.

mation. The normalization constant  $n$  is the total number of pixels in the animation. In Figure 1.6 and 1.7, 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 1.6, 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 1.2b, 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 1.7, 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.

Finally, we present a summary of the errors in Table 1.2. 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

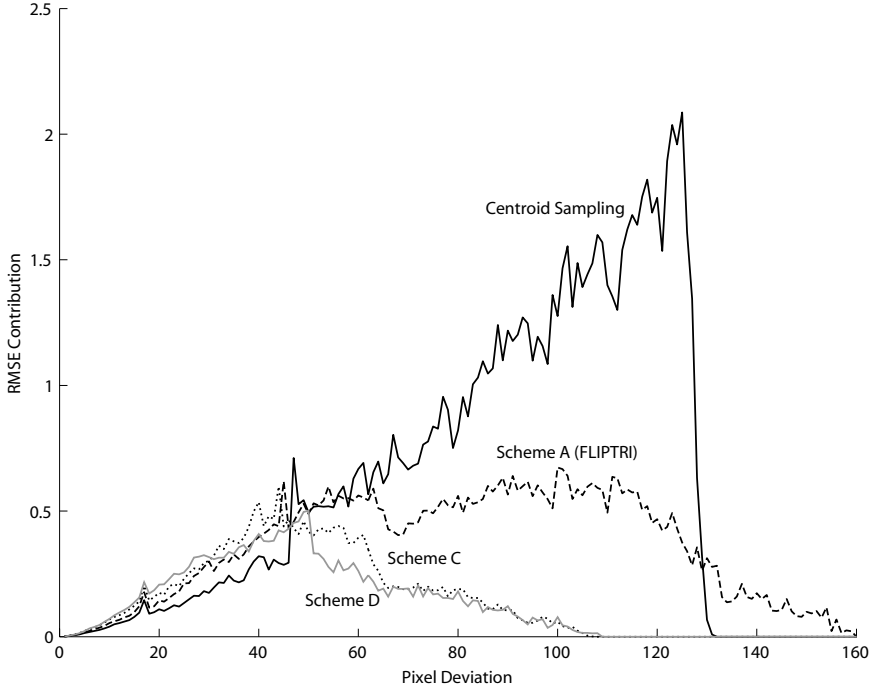


Figure 1.6: Histogram of the squared pixel error. This figure shows a comparison between the 1.25, 1.5 and 1.75 sample shemes. Sampling with a single centroid sample is also included for reference. The RMSE contribution is computed as  $C(i)i^2/n$

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 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.

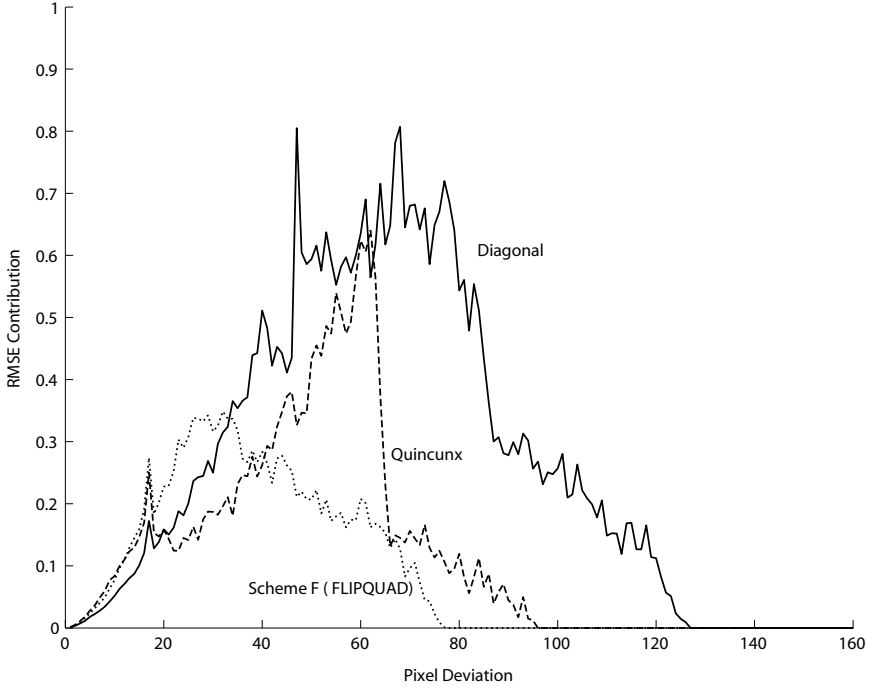


Figure 1.7: Histogram of the squared pixel error. This figure shows how scheme f (FLIPQUAD) compares to Quincunx and the Diagonal sample scheme from Figure 1.1d.

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

	$E_2$	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 1.2: The errors of our sampling schemes, using different metrics. The “Diagonal” scheme is the diagonal super sampling scheme from Figure 1.1d.

## A Sample positions and weights

The following table lists the sample point coordinates and weights for the sampling patterns in Figure 1.3. 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  $y = \pm 0.5$ .

<b>A: <math>P(4, 1^+, 1.25)</math></b>	<b>B: <math>P(4, 1^+, 1.5)</math></b>
-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
<b>C: <math>P(4, 1^+, 1.5)</math></b>	<b>D: <math>P(4, 1^+, 1.75)</math></b>
-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	
<b>E: <math>P(4, 1^+, 1.75)</math></b>	<b>F: <math>P(4, 1^+, 2)</math></b>
-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 computer-graphics. 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 2.1a, 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 result in less work for the GPU and more work for the CPU, which performs exact intersection tests. See Figure 2.1b 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.

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 §conservative rasterization.

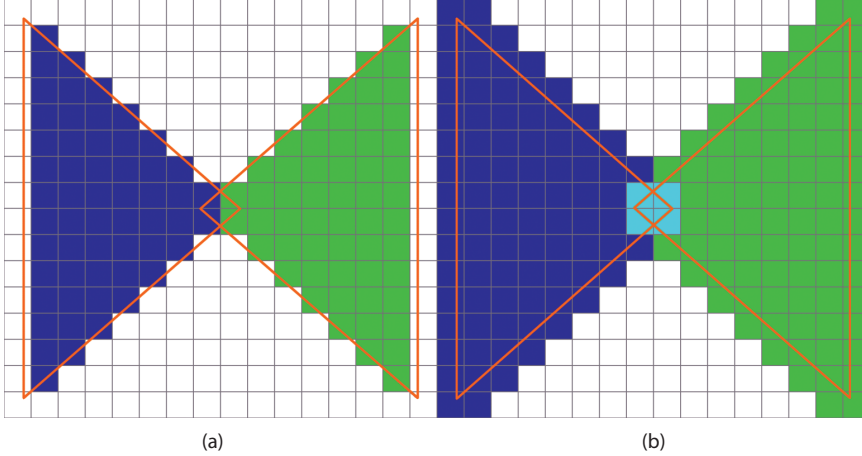


Figure 2.1: Comparing Standard and Conservative Rasterization: A blue and a green triangle rasterized at 16CE16 pixels resolution using additive blending. (a) Standard rasterization. (b) Overestimating conservative rasterization. Note that the small overlap of the triangles is missed with standard rasterization and correctly detected using conservative rasterization.

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 conservative rasterization, we call them the *semidiagonals*. This type of vertex movement is shown by the red lines in Figure 2.2a, which also shows the *bounding polygon* for a triangle.

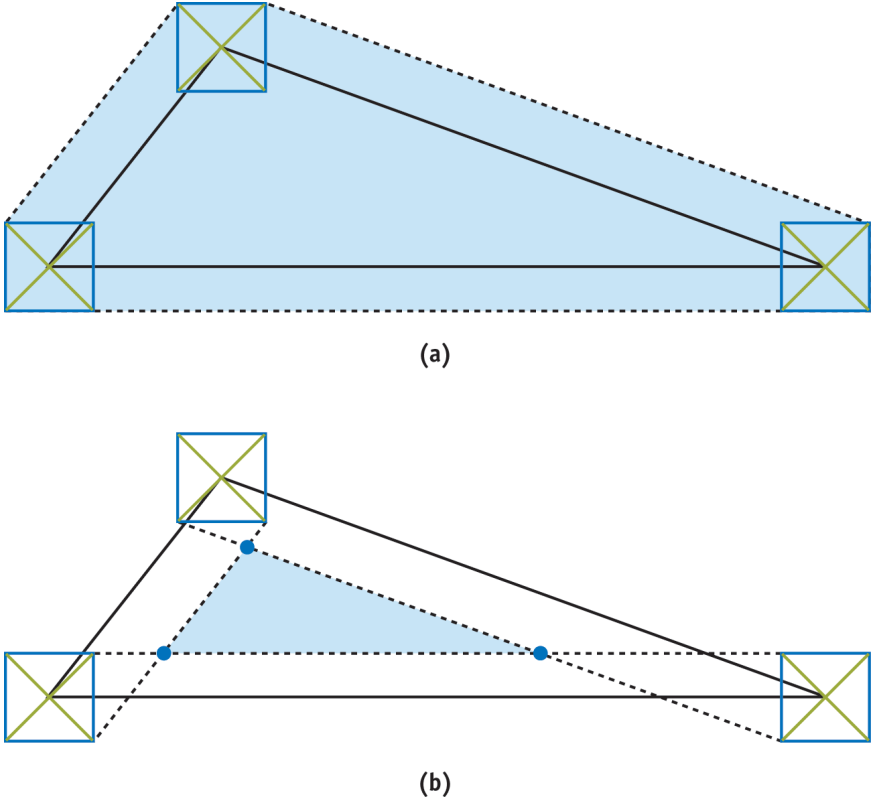


Figure 2.2: Overestimated and Underestimated Rasterization: (a) Overestimated rasterization. (b) Underestimated rasterization. Solid lines show the input triangle. The dashed lines, dots, and gray area indicate the bounding polygon's edges, vertices, and area.

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 2.2b.

### 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 2.2a. 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 2.3. Given two edges  $e_1$  and  $e_2$  connected in a vertex  $v$ , the three cases are the following:

- 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 2.3a).
- 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 2.3b).

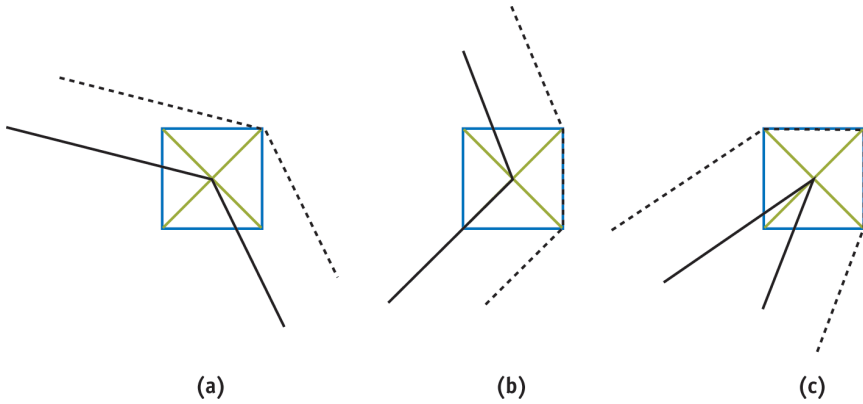


Figure 2.3: Computing an Optimal Bounding Polygon

- 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 2.3c).

### 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 2.3c. 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 2.3, 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 2.1.

---

```
// 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 2.1: 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 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 2.4. 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 2.5 illustrates this process.

In our second algorithm, we compute the bounding triangle in a vertex program and



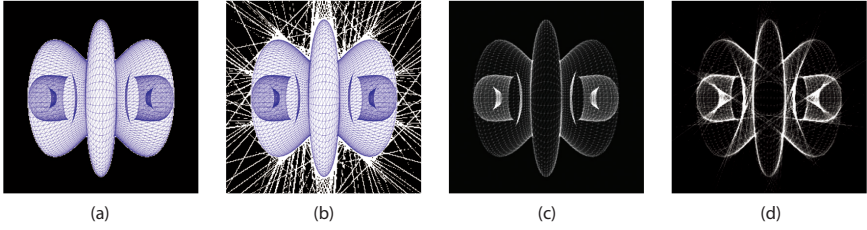


Figure 2.4: Objects Rasterized in Different Ways: (a) An object rasterized with the first algorithm. (b) The object rasterized using bounding triangles only. (c) Overdraw for the first algorithm. (d) Overdraw for the second algorithm.

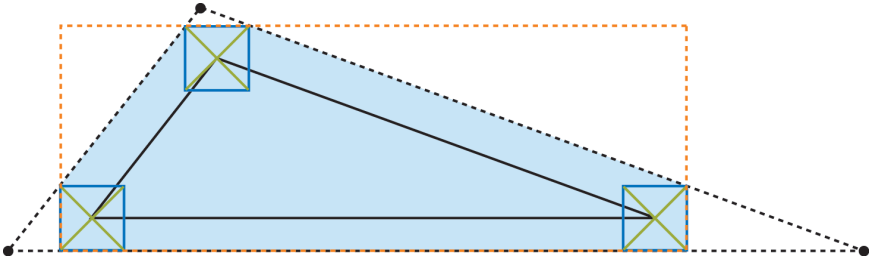


Figure 2.5: An Input Triangle and Its Optimal Bounding Polygon: The bounding triangle is shown by the dashed line. Its axis-aligned bounding box is shown by the orange dashed line. The blue area is the optimal bounding polygon.

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:

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

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 2.2 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 2.2: 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,  $\epsilon$ , small enough that we consider all errors caused by  $\epsilon$  to fall in the same category as other floating-point precision errors. If the signed distance from the plane of the triangle to the viewpoint is less than  $\epsilon$ , 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

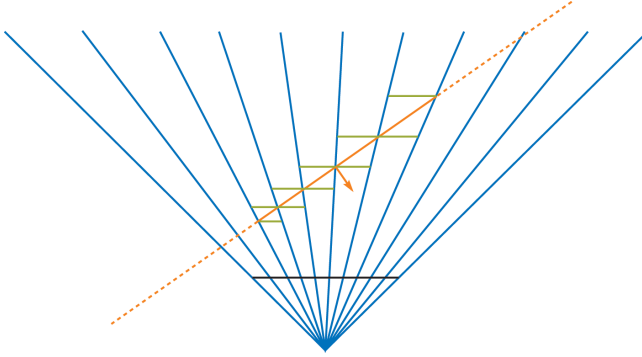


Figure 2.6: Finding the Farthest Depth Value: A view frustum, with pixel cells (blue lines) and a triangle (red), as seen from above. The dashed line is the plane of the triangle, and the orange arrow indicates its normal. The range of possible depth values is also shown for the rasterized pixels. The 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.

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 2.6.

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.

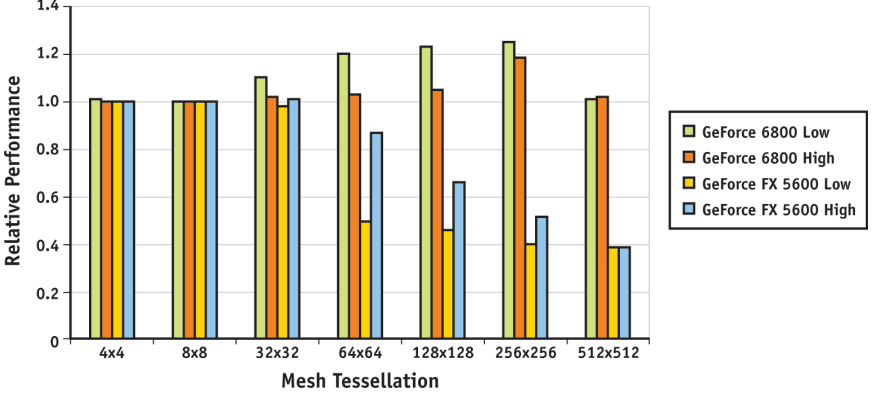


Figure 2.7: Benchmarked Results: Benchmarks on a GeForce 6800 Series GPU and a GeForce FX 5600. Tests were performed at low (128CE128 pixels) and high (1024CE1024 pixels) resolution. The graph shows the relative performance of the first algorithm compared to the second.

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:

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

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 2.7 show that the first algorithm seems to be more efficient on high-end hardware (such as GeForce 6800 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 Modified 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 Toshiyasu 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.

Eurographics Symposium on Rendering, 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 viewpoints, and thus provide motion parallax. Furthermore, the viewers can see stereoscopic 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 ( $\leq 16$ ). 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 ex-

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]:

$$o(d) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{d}. \quad (3.1)$$

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

$$b = d \times Z_r + o(d) \times (Z_w + C_w + T_r), \quad (3.2)$$

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 3.2 becomes:

$$\begin{aligned} b(n) &= n \times [d \times Z_r + o(d) \times (Z_w + C_w + m \times T_r)] \\ &= n \times \underbrace{[d \times Z_r + o(d) \times Z_w]}_{\text{depth buffer, } B_d} + \underbrace{o(d) \times C_w}_{\text{color buffer, } B_c} + \underbrace{o(d) \times m \times T_r}_{\text{texture read, } B_t} \\ &= n \times [B_d + B_c + B_t] \end{aligned} \quad (3.3)$$

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.  $\square$

**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.  $\square$

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.

### 3 Background: Multi-View Rasterization

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

```

BRUTEFORCE-MULTIVIEWRASTERIZATION()
1  for  $i \leftarrow 1$  to  $n$                 // loop over all views
2      create  $\mathbf{M}^i$                 // create projection matrix
3      for  $j \leftarrow 1$  to  $t$             // loop over all triangles
4          RASTERIZETRIANGLE( $\mathbf{M}^i, \Delta_j$ )
5      end
6  end
    
```

```

NEW-MULTIVIEWRASTERIZATION()
1  create all  $\mathbf{M}^i, i \in [1, \dots, n]$     // create projection matrices
2  for  $j \leftarrow 1$  to  $t$                 // loop over all triangles
3      RASTERIZETRIANGLETOALLVIEWS( $\mathbf{M}^1, \dots, \mathbf{M}^n, \Delta_j$ )
4  end
    
```

Figure 3.1: 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,  $\Delta_j, j \in [1, \dots, t]$ . Note that the core of our algorithm lies in the RASTERIZETRIANGLETOALLVIEWS() function.

### 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 3.1, where it is assumed that we have a scene consisting of  $t$  triangles,  $\Delta_j, j \in [1, \dots, t]$ .

### 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 3.2, 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,  $i \in [1, \dots, n]$ , to identify a particular view. Furthermore, we have a vertex,  $\mathbf{v}$ , in object space, that should be transformed into homogeneous

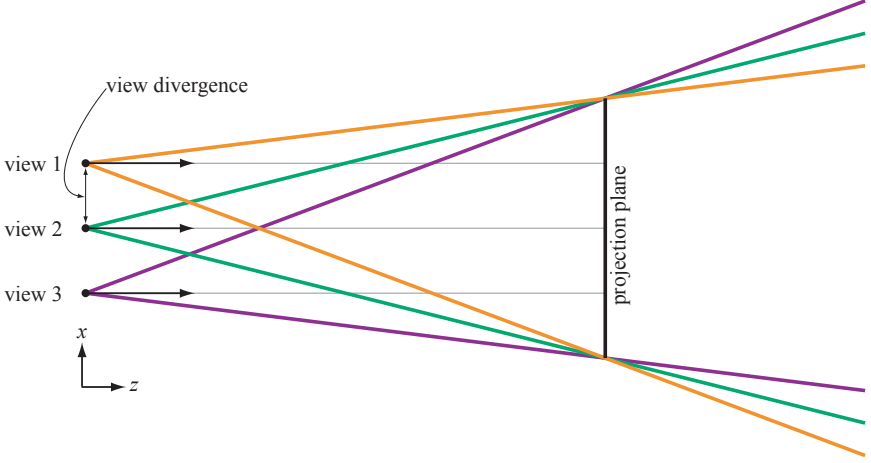


Figure 3.2: Off-axis projections for three views. As can be seen, all views share the same  $z$ -axis. We use the term “view divergence” for the distance between two views’ viewpoints.

screen space for each view. These transformed vertices are denoted  $\mathbf{p}^i$ ,  $i \in [1, \dots, n]$ . 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  $\mathbf{p}_i = (p_x^i, p_y^i, p_z^i, p_w^i)^T = \mathbf{M}^i \mathbf{v}$  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 3.3 as much as possible. Our hypothesis is that this should be possible if the 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

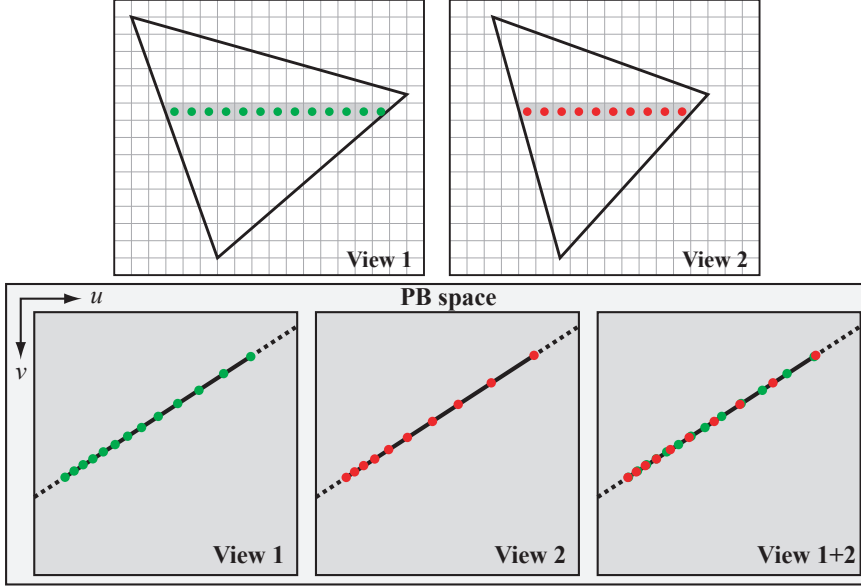


Figure 3.3: The two views in a stereo system. A single scanline is highlighted for both the left and right view. In the bottom row of the figure, we show the PB space for view 1 and view 2, where the green and red samples in PB space correspond to the samples along the selected scanlines. To the right, 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.

function `RASTERIZETRIANGLETOALLVIEWS` (Figure 3.1), 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,  $\mathbf{t}^i = (t_u^i, t_v^i)$ , 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 3.3. 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:

$$E^i = \mathbf{d} \cdot \mathbf{t}^i, \quad (3.4)$$

where  $\mathbf{d} = (d_u, d_v)$  is the PB traversal direction, which is computed as the difference 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 3.4 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,  $\lambda^i$ , 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 3.3 are tiles rather than pixels. A rectangular tile in screen space projects, in general, to a convex quadrilateral in PB space. In Figure 3.4, 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

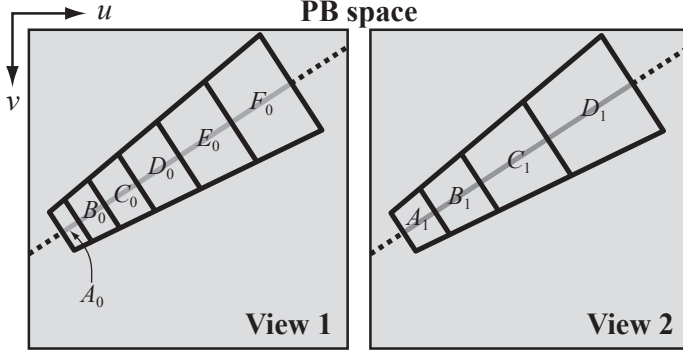


Figure 3.4: Tiled traversal, shown in PB space, for a row of tiles overlapping a triangle (not shown). The PB traversal direction is the gray line. Our algorithm visits the tiles in the following order:  $A_0$ ,  $A_1$ ,  $B_0$ ,  $B_1$ ,  $C_0$ ,  $D_0$ ,  $C_1$ ,  $E_0$ ,  $D_1$ , and finally  $F_0$ . This is the order in which the tiles appear on the PB traversal direction.

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 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*.

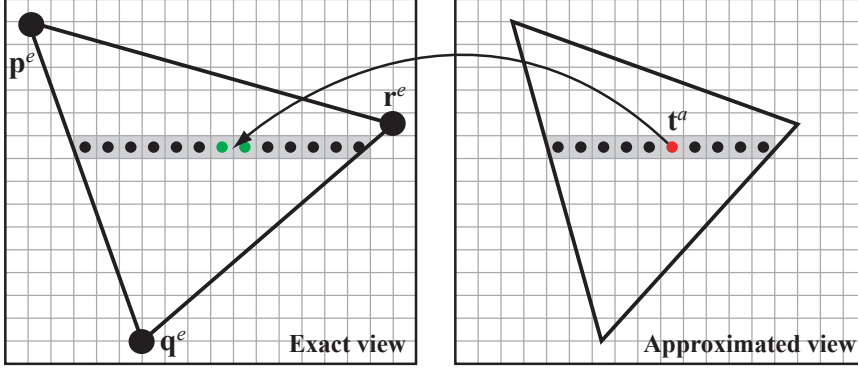


Figure 3.5: Illustration of how a fragment (red circle) with perspective-correct barycentric coordinates  $\mathbf{t}^a$ , in an approximated view, can be mapped onto a screen-space position in the exact view. The color of two nearby fragments in the exact view are interpolated to compute the color of the fragment in the approximated view.

Figure 3.5 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:

$$\mathbf{c} = (c_x, c_y, c_z, c_w)^T = (1 - t_u^a - t_v^a)\mathbf{p}^e + t_u^a\mathbf{q}^e + t_v^a\mathbf{r}^e. \quad (3.5)$$

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<sup>1</sup> 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 3.6, 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

<sup>1</sup> If the pixel shader also computes the depth of the fragment, our algorithm can be used to approximate the depth as well.

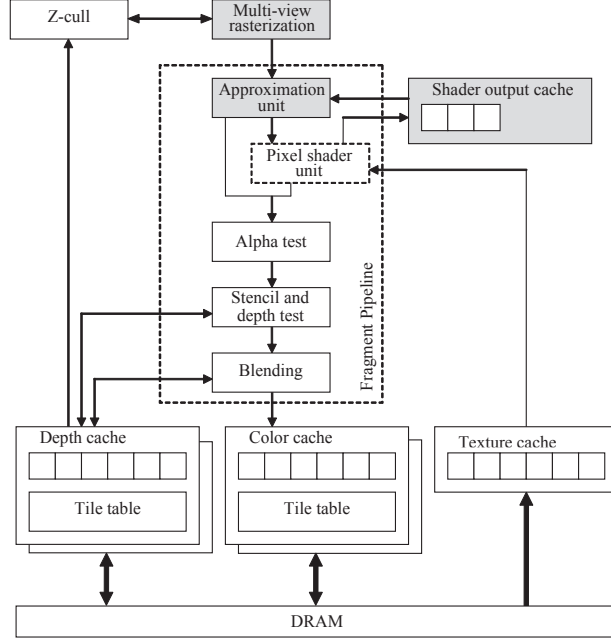


Figure 3.6: To support approximate pixel shader evaluation, a *shader output cache* and an *approximation unit* (AU) are introduced into the rasterization pipeline. Before the pixel shader is executed for an approximated view, the AU checks with the shader output cache whether the fragment color can 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.

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.<sup>2</sup> 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,

<sup>2</sup>Alternatively, one could choose to execute the full pixel shader for such fragments. This would increase the quality slightly.

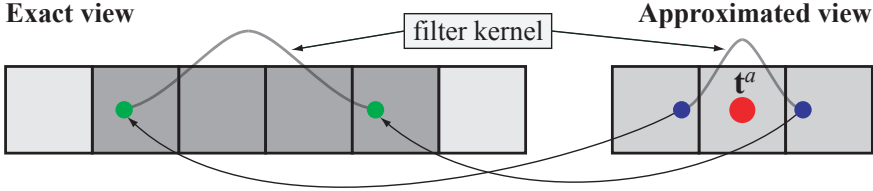


Figure 3.7: Higher quality filtering is obtained by projecting the blue end points of the filter kernel in the approximated view into the exact view. These projected points span a number of pixels (dark gray), which are weighted using the filter kernel to form the approximated fragment.

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 3.7. 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 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.

#### 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  $4 \times 4$  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    RASTERIZE_SCENE_TO_ALL_VIEWS( $\mathbf{M}^1, \dots, \mathbf{M}^4$ )
5    enable color writes
6    enable additive blending
7    enable pixel shader that includes the  $w_i = 1/16$  scaling term
8    RASTERIZE_SCENE_TO_ALL_VIEWS( $\mathbf{M}^1, \dots, \mathbf{M}^4$ )
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 3.4. 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 back-face 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.



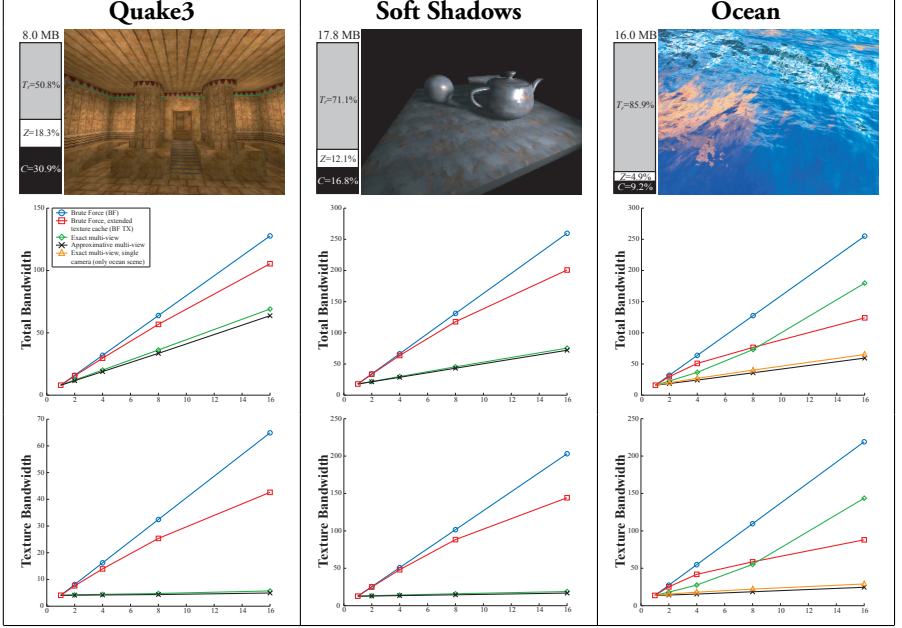


Figure 3.8: The first row shows a summary of the test scenes rendered at  $640 \times 480$  pixels. The bandwidth (BW) figures, located on top of each bar, is the BW in megabytes (MB) per frame for a single view using a conventional rasterizer, as described in the text. Each bar is divided into texturing 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.

## 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 3.8.

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  $640 \times 480$  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  $533 \times 400$  pixels, or seven views at  $686 \times 400$  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 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 3.8. 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



Figure 3.9: Visualization of approximation in the Quake3 scene. Green pixels have been approximated from the exact central view.

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 3.8), 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 3.9 (color page) 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 triangles 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 remains 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 3.10 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  $16 \times 16$  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 3.10 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 coher-

ent 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:

	$80 \times 60$ , 0.8 ppt	$640 \times 480$ , 48 ppt
<i>BF TX</i>	100%	100%
<i>Tri-by-Tri</i>	31.3%	88.8%
<i>Our</i>	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 rasterization 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,  $2 \times 2$  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.

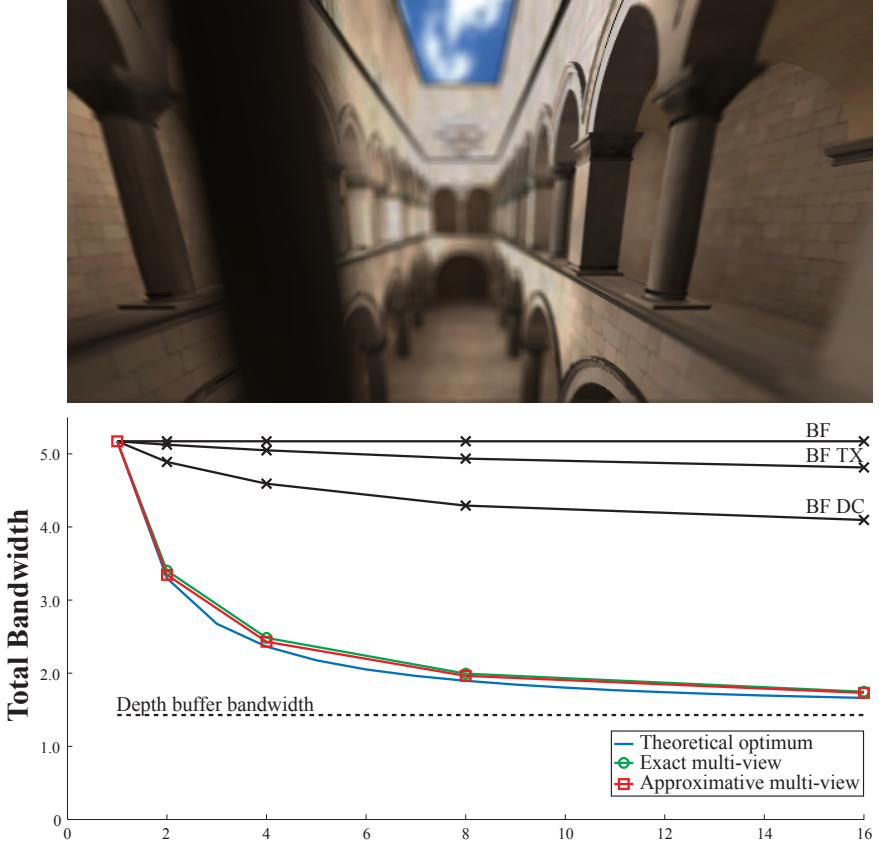


Figure 3.10: The Sponza atrium rendered with a depth of field (DOF) effect. The diagram shows bandwidth measurements in gigabytes per frame as a function of the number of views supported by the rasterizer. The curve named “Theoretical optimum” shows the best possible performance for our architecture 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 architecture 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.





# 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 Modified 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 Vidimčec, 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](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

---

# 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 2006.



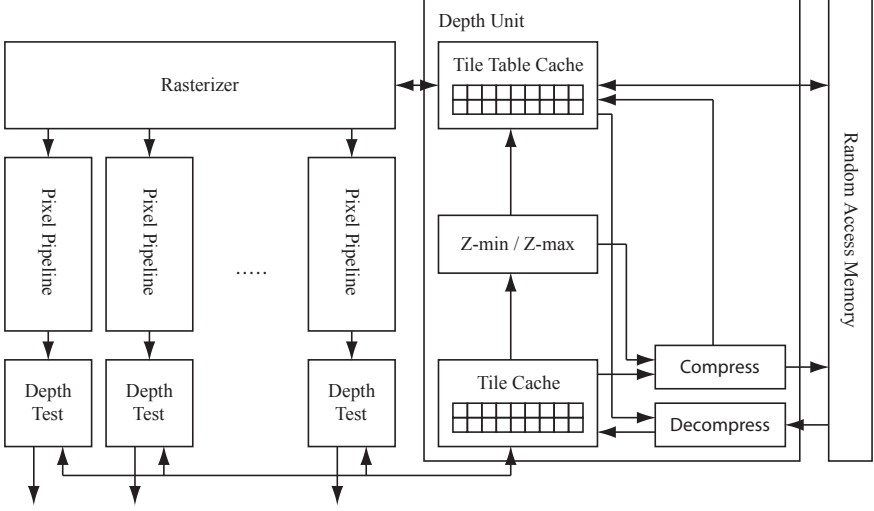


Figure 4.1: A modern depth buffer architecture. Only the tile cache is needed to implement tiled depth buffering. The rest of the architecture is dedicated to bandwidth and performance optimizations. For a detailed description see Section 2.

## 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 4.1.

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 4.1. 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 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.<sup>1</sup> 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 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.

---

<sup>1</sup>Generalizing to other bit rates is straightforward.

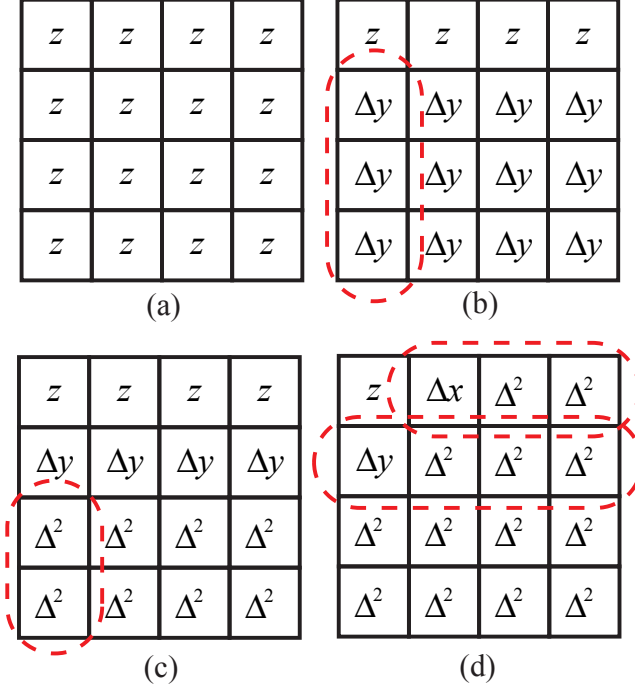


Figure 4.2: Computing the second order differentials. a) Original tile, b) First order column differentials, c) Second order column differentials, d) Second order row differentials.

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 4.2. 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 4.2d. 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  $8 \times 8$  pixel tiles that use 32 bits for storing a reference value,  $2 \times 33$  bits for  $x$  and  $y$  differentials, and  $61 \times 2$  bits for storing the second order differential of each remaining pixel in the tile. This gives a

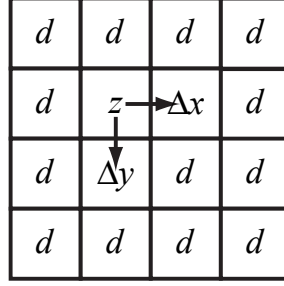


Figure 4.3: Anchor encoding of a  $4 \times 4$  tile. The depth values of the  $z$ ,  $\Delta x$  and  $\Delta y$  pixels form a plane. Compression is achieved by using the plane as a predictor, and storing an offset,  $d$ , for each pixel. Only 5 bits are used to store the offsets.

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  $2 \times 23$  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  $4 \times 4$  pixel tiles (although it could be

generalized) and is illustrated in Figure 4.3. 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,  $\Delta x$  and  $\Delta y$ , 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 flag and a constant stencil value for the whole tile).

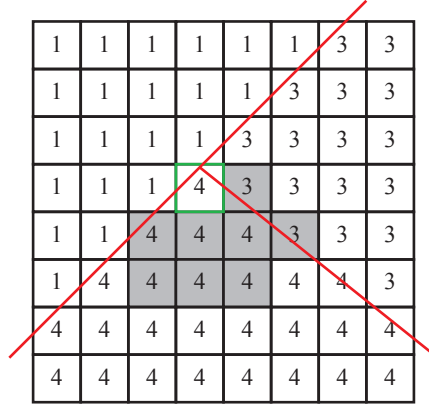
The anchor encoding mode behaves quite similar to the one plane mode of the DDPCM algorithm. The extra bits of per-pixel resolution provide for some extra numerical stability, but unfortunately do not seem to provide a significant increase in terms of compression ratio.

### 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  $4 \times 4$  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 4.4. 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 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 4.4 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



1	1	1	1	1	1	3	3
1	1	1	1	1	3	3	3
1	1	1	1	3	3	3	3
1	1	1	4	3	3	3	3
1	1	4	4	4	3	3	3
1	4	4	4	4	4	4	3
4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4

Figure 4.4: 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.

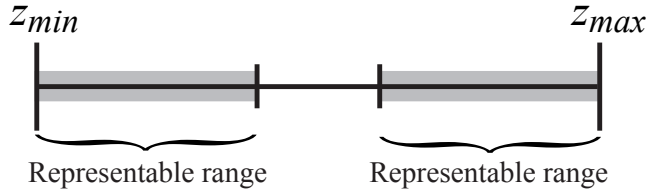


Figure 4.5: The depth offset scheme compresses the depth data by storing depth values in the gray regions as offsets relative to either the  $z_{min}$  or  $z_{max}$  value.

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 4.5. Although the patent is written in a more general fashion, the figure illustrates its 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.

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  $(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta y})$ , we can write the interpolation equations as:

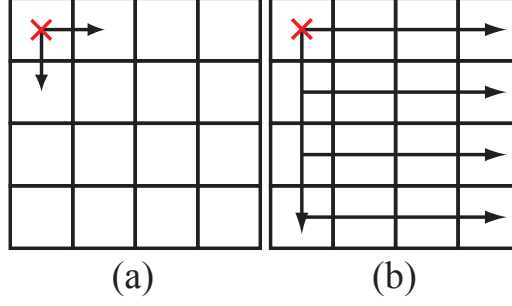


Figure 4.6: The leftmost image shows the points used to compute our prediction plane. The rightmost image shows in what order we traverse the pixels of a tile.

$$z(x, y) = z_0 + (x - x_0) \frac{\Delta z}{\Delta x} + (y - y_0) \frac{\Delta z}{\Delta y}. \quad (4.1)$$

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

$$z(x + 1, y) = z(x, y) + \frac{\Delta z}{\Delta x}. \quad (4.2)$$

We can rewrite the differential of Equation 4.2 as a quotient and remainder part, as shown below:

$$\frac{\Delta z}{\Delta x} = \left\lfloor \frac{\Delta z}{\Delta x} \right\rfloor + \frac{r}{\Delta x}. \quad (4.3)$$

Equation 4.2 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  $(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta y})$  directly from the neighbors in the  $x$ - and  $y$ -directions, as shown in Figure 4.6a. 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 4.6b, 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

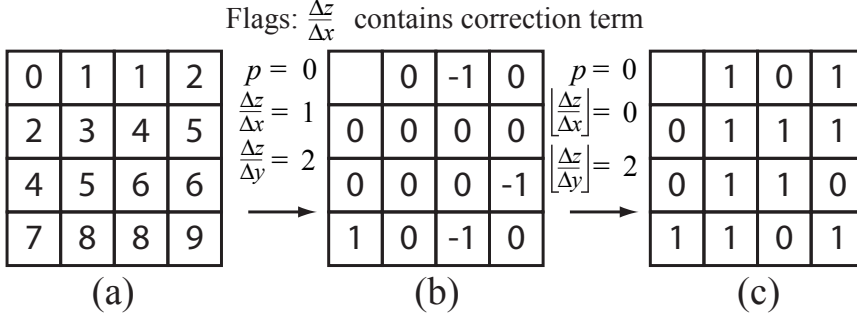


Figure 4.7: The different steps of the one plane compression algorithm, applied to a compressible example tile.

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 4.7b. 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 4.7c.

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 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



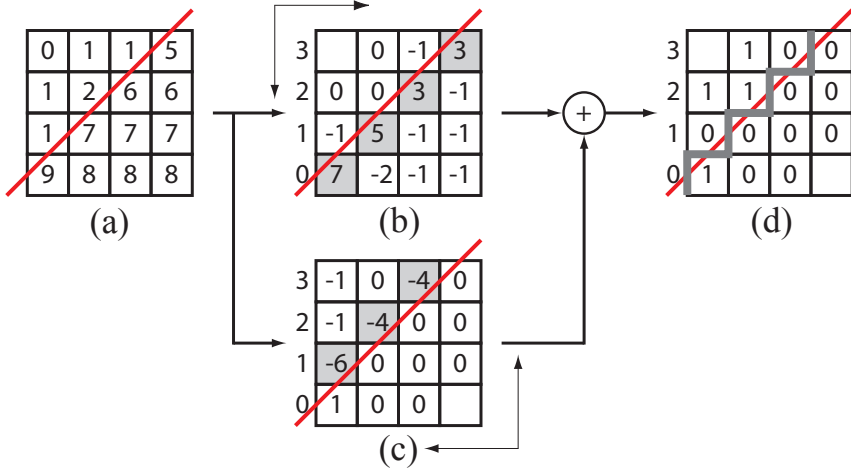


Figure 4.8: This figure illustrates the two plane compression algorithm. a) Shows the original tile with depth values from two different planes. The line indicates the edge separating the two planes. b & c) We execute the one plane algorithm of Section 4.1 for each corner of the tile. In this figure, 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.

## 4.2 Two plane mode

We also aim to compress tiles that contain two planes separated by a single edge. See Figure 4.8a 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 4.8b shows the break points and correction terms resulting from the tile in Figure 4.8a. 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 4.6, 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 4.8d.

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  $4 \times 4$  tile mode, we use this to store the break points in just 7 bits. We do not use this in the  $8 \times 8$  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
$4 \times 4$	1	$23 \times 2$	$15 \times 4$	7	$1 \times 15$	128
$8 \times 8$	1	$22 + 21$	$15 \times 4$	26	$1 \times 63$	192
$8 \times 8$	1	$24 \times 2$	$24 \times 4$	32	$1 \times 63$	240

## 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

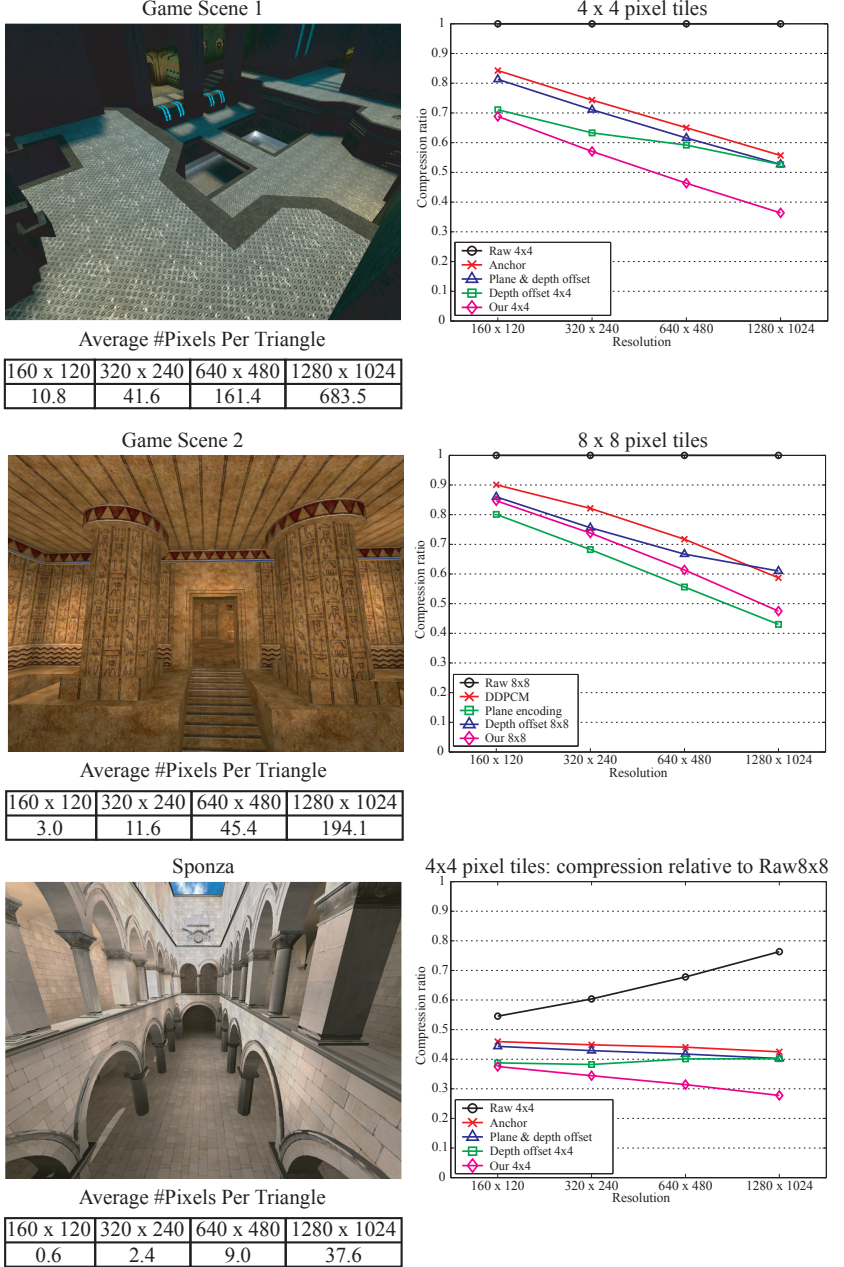


Figure 4.9: The left column shows a summary of the benchmark scenes. The diagrams in the right column show the average compression for all three scenes as a function of rendering resolution, for  $4 \times 4$  and  $8 \times 8$  pixel tiles. Finally, we show the depth buffer bandwidth of  $4 \times 4$  tiles, relative to the bandwidth of a Raw  $8 \times 8$  depth buffer. It should be noted that this diagram does not take tile table bandwidth into account.

z-min and z-max culling. Our tests show that compression rates are only marginally affected by the cache size.<sup>2</sup> 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,  $8 \times 8$  pixel tiles. Bit rate: 3/5 bpp (bits per pixel)
- **Anchor:** The anchor encoding scheme (Section 3.3),  $4 \times 4$  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,  $8 \times 8$  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,  $4 \times 4$  pixel tiles. Bit rate: 8/16 bpp, 8 bits for the plane mode and 16 bits for the depth offset mode.
- **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  $4 \times 4$  and  $8 \times 8$  tiles.
- **Our 4x4/8x8:** Our compression scheme, described in Section 4. For the  $8 \times 8$  tile mode, we used the 192 bit version of the two plane mode in this evaluation. Bit rate: 4/8 bits per pixel for  $4 \times 4$  tiles and 2/3 bits per pixel for  $8 \times 8$  tiles.

Our benchmarks were performed on three different test scenes, depicted in Figure 4.9. Each test scene features an animated camera with static geometry. Furthermore, we rendered each scene at four different resolutions:  $160 \times 120$ ,  $320 \times 240$ ,  $640 \times 480$ , and  $1280 \times 1024$  pixels. Varying the resolution is a simple way of simulating different

---

<sup>2</sup>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

levels of tessellation. As can be seen in Figure 4.9, we cover scenes with great diversity in the average triangle area.

In the bottom half of Figure 4.9, we show the compression ratio of each algorithm, grouped into algorithms for  $4 \times 4$  and  $8 \times 8$  pixel tiles. We also present the compression of the  $4 \times 4$  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  $8 \times 8$  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  $4 \times 4$  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  $8 \times 8$  tile algorithms down to  $4 \times 4$  tiles, and still keeps a reasonably low bit rate. A two plane mode for  $4 \times 4$  tiles is equal to having the flexibility of eight planes (with some restrictions) in an  $8 \times 8$  pixel tile. This shows up in the evaluation, as our  $4 \times 4$  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  $4 \times 4$  and  $8 \times 8$  pixel tiles at various resolutions. We have avoided an exhaustive evaluation of whether  $4 \times 4$  or  $8 \times 8$  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. ACM Press/ACM SIGGRAPH, New York, J. Kajiya, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 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 Comput. Surv.*, 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 V

---

# 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  $4 \times 4$  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 5.1.

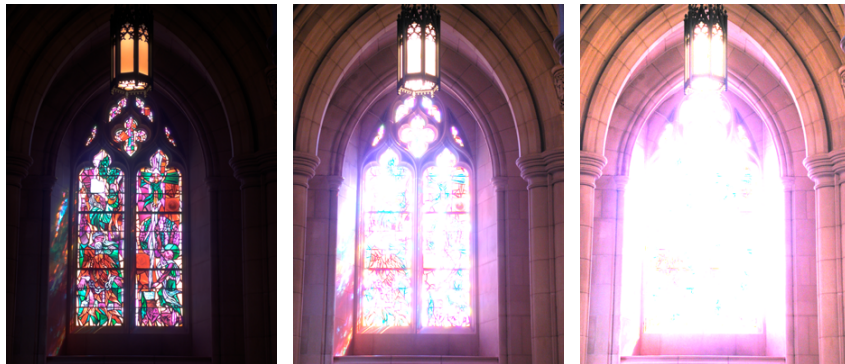


Figure 5.1: Example of a high dynamic range image, here shown at three different exposures, compressed with our algorithm to a fixed rate of 8 bits per pixel. Our algorithm gives excellent image quality over a large dynamic range, and is fast to decompress in hardware.

## 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  $4 \times 4$  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],  $4 \times 4$  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 straight-

forward 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 5.13 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  $4 \times 4$  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:

$$Y = w_r R + w_g G + w_b B, \quad (5.1)$$

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

$$U = \frac{R - Y}{Y}, \quad V = \frac{B - Y}{Y}. \quad (5.2)$$

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 5.1, the transform from RGB is given by:

$$(\bar{Y}, \bar{u}, \bar{v}) = \left( \log_2 Y, w_b \frac{B}{Y}, w_r \frac{R}{Y} \right). \quad (5.3)$$

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 5.2 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:

$$T(I) = \left[ 255 (2^c I)^{1/\gamma} \right]_0^{255}, \quad (5.4)$$

where  $c$  is the exposure compensation in f-stops,  $\gamma$  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:

$$\text{MSE} = \frac{1}{n \times w \times h} \sum_c \sum_{x,y} (\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2), \quad (5.5)$$

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:

$$\text{mPSNR} = 10 \log_{10} \left( \frac{3 \times 255^2}{\text{MSE}} \right). \quad (5.6)$$

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 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



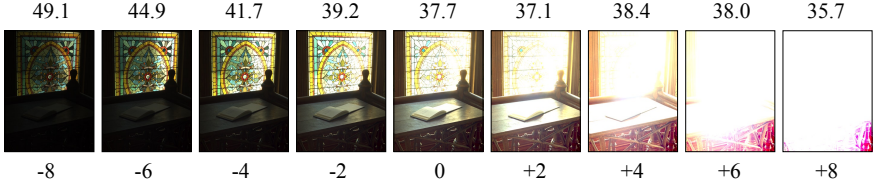


Figure 5.2: In our *multi-exposure PSNR* error measure, the image is tone mapped to a number of different exposures to account for all normal viewing conditions, and the PSNR is computed from the average MSE. In this case, the mPSNR is 39 dB. The top row shows the standard PSNRs, and the bottom row shows the exposure compensation,  $c$ .

in OpenGL as well. S3TC uses tiles of  $4 \times 4$  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  $4 \times 4$  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 5.13. 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 5.13, 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 5.3. 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 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

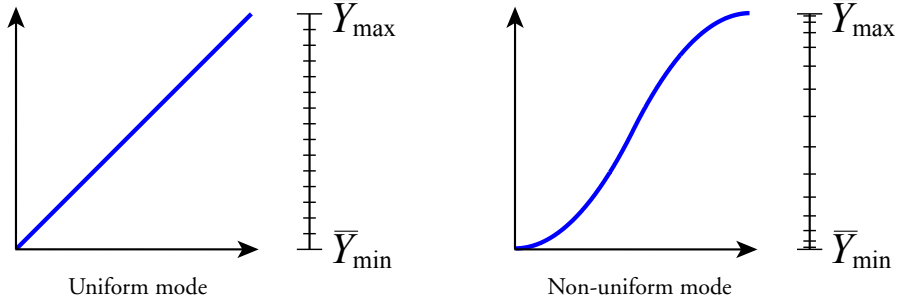


Figure 5.3: The two luminance quantization modes. The non-uniform mode is used for better handling tiles with sharp luminance transitions, such as edges.

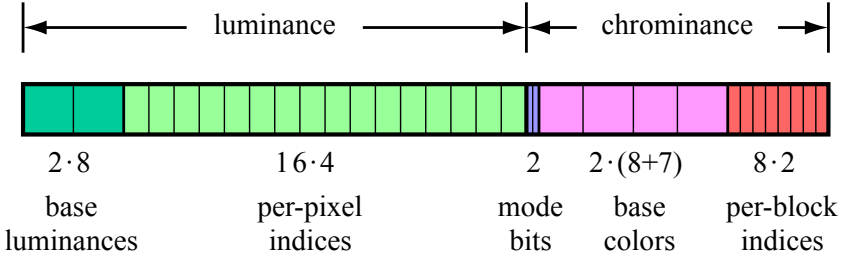


Figure 5.4: The bit allocation we use for encoding the luminance and chrominance of a  $4 \times 4$  tile in 128 bits (8 bpp).

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  $\bar{Y}_{\min}$  and  $\bar{Y}_{\max}$  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 5.10), 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 5.3.

To choose between the two modes, we use the mutual ordering<sup>1</sup> of  $\bar{Y}_{\min}$  and  $\bar{Y}_{\max}$ . In decoding, if  $\bar{Y}_{\min} \leq \bar{Y}_{\max}$ , then the uniform mode is used. Otherwise,  $\bar{Y}_{\min}$  and  $\bar{Y}_{\max}$  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, leaving 48 bits for the chrominance. The bit allocation is illustrated in Figure 5.4.

<sup>1</sup> Similar ordering techniques are used in the S3TC LDR texture compression format.

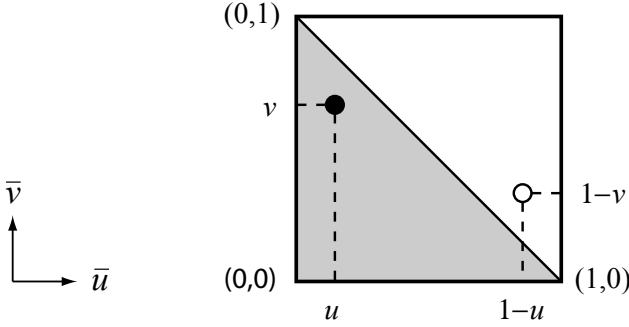


Figure 5.5: Illustration of the *flip trick*. By mirroring the coordinates for a base color, we exploit our triangular chrominance space in order to obtain another bit.

## 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  $1 \times 2$  (horizontal) or  $2 \times 1$  (vertical) sub-sampled pixels. The start and end points of the chroma line, each with  $2 \times 8$  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 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 5.5. 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 5.6 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

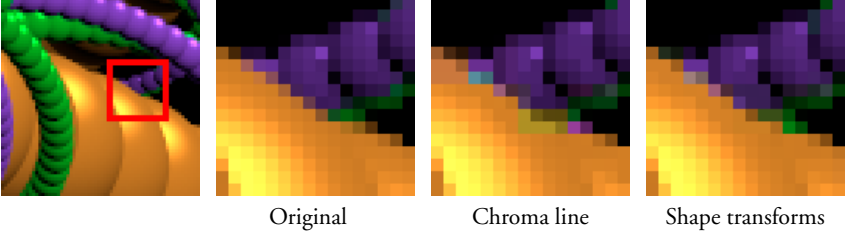


Figure 5.6: A region where a line in chroma space is not sufficient for capturing the complex color. With shape transforms, we get a much closer resemblance to the true color, although minor imperfections exist due to the sub-sampling.

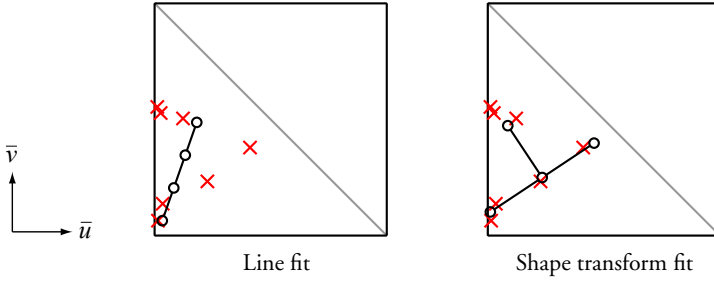


Figure 5.7: In tiles with difficult chrominance, such as in this example taken from Figure 5.6, a line in chroma space has difficulties representing the  $(\bar{u}, \bar{v})$  points accurately (left). Our algorithm based on *shape transforms* generates superior chroma representations, as it is often possible to find a shape that closely matches the chrominance points (right).

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 5.7 on one of the difficult tiles from the image in Figure 5.6. 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 done to reduce the chroma values of a tile to four chroma points. Second, we normalized the chroma

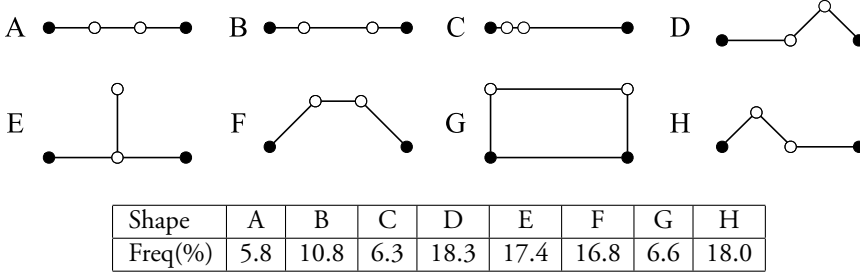


Figure 5.8: The set of shapes we use for the *shape transform* algorithm and their frequencies for our test images. The points corresponding to base colors are illustrated with solid black circles.

points for scale and rotation, and then iteratively merged the two closest candidates until the desired number of shapes remained. Figure 5.8 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 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 5.4. 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.

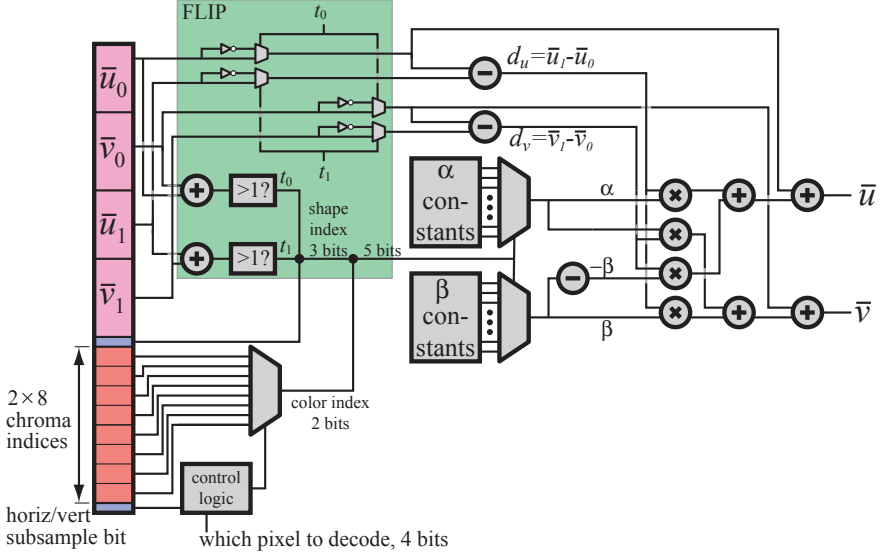


Figure 5.9: Decompression of chrominance,  $(\bar{u}, \bar{v})$ , for a single pixel. The indata is the 48 bits for chrominance (left part of the figure), and a 4-bit value indicating which pixel in a tile to decode. The outdata is  $(\bar{u}, \bar{v})$  for one pixel. The green box contains the logic to implement the *flip trick*, where inverters have been used to compute the  $1 - x$  terms.

## 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 5.9, 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  $\alpha$  and  $\beta$ , which are both fixed-point numbers in the interval  $[0, 1]$ , using only five bits each. The chrominance of a point, with coordinates

$\alpha$  and  $\beta$ , is derived as:

$$\begin{pmatrix} \bar{u} \\ \bar{v} \end{pmatrix} = \alpha \mathbf{d} + \beta \mathbf{d}^\perp + \begin{pmatrix} \bar{u}_0 \\ \bar{v}_0 \end{pmatrix} = \begin{pmatrix} \alpha d_u - \beta d_v + \bar{u}_0 \\ \beta d_u + \alpha d_v + \bar{v}_0 \end{pmatrix}. \quad (5.7)$$

The diagram in Figure 5.9 implements the equation above. As can be seen, the hardware is relatively simple. The  $\alpha$  and  $\beta$  constants units contain only the constants 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 5.3 as:

$$(R, G, B) = \left( \frac{1}{w_r} \bar{v} Y, \frac{1}{w_g} (1 - \bar{u} - \bar{v}) Y, \frac{1}{w_b} \bar{u} Y \right). \quad (5.8)$$

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:

$$(1/w'_r, 1/w'_g, 1/w'_b) = \frac{1}{16} (54, 27, 144). \quad (5.9)$$

This corresponds to

$$(w'_r, w'_g, w'_b) \approx (0.296, 0.593, 0.111). \quad (5.10)$$

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  $3 \times 3$  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 5.10. 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 5.11. After that follows visual comparisons.

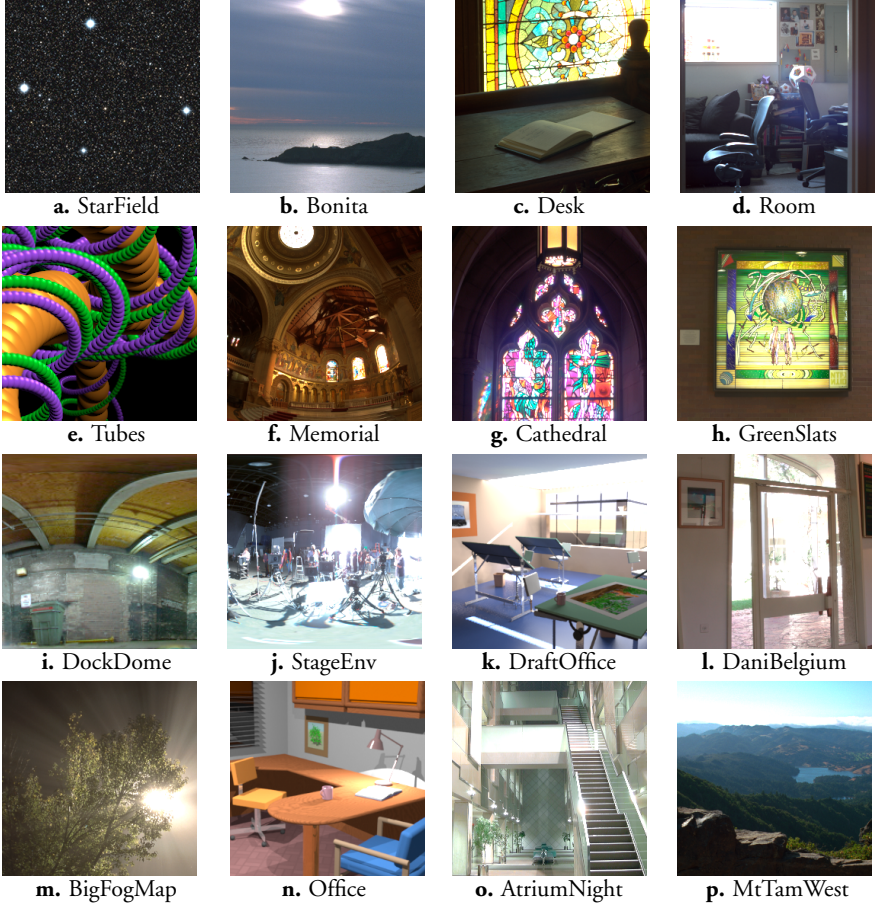


Figure 5.10: The HDR test images we use for evaluating our algorithms. The figure shows cropped versions of the actual test images. (e), (k), and (n) are synthetic.

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 im-

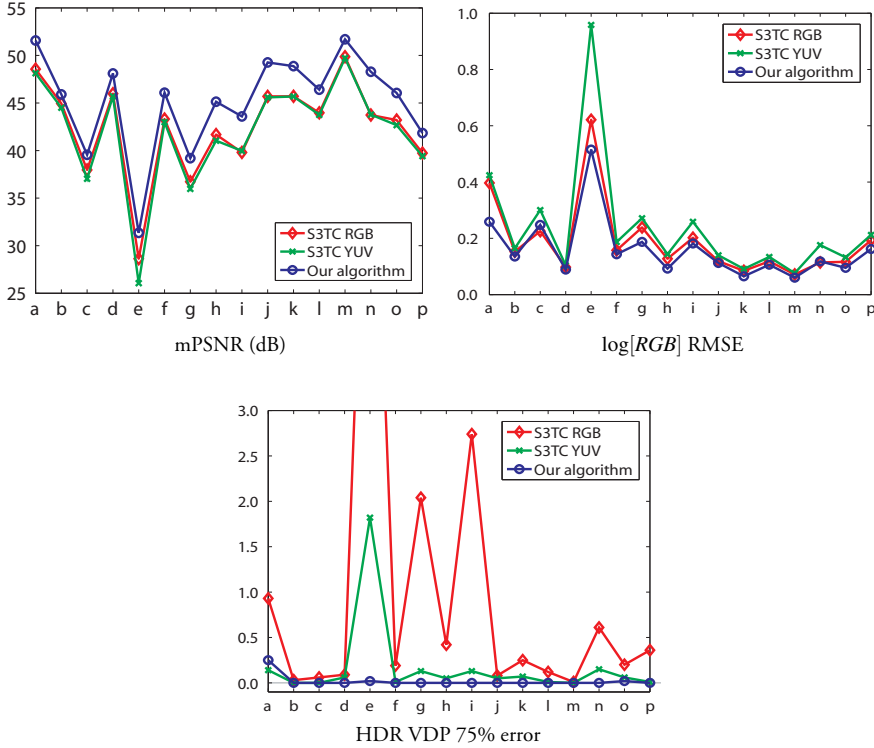


Figure 5.11: These diagrams show the performance of our algorithm compared to the S3TC RGB and S3TC YUV algorithms for each of the images (a)–(p) in Figure 5.10. The mPSNR measure gives consistently better values (left), while the  $\log[RGB]$  RMSE (middle) is lower on nearly all of the test images. 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.

age 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 5.11 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

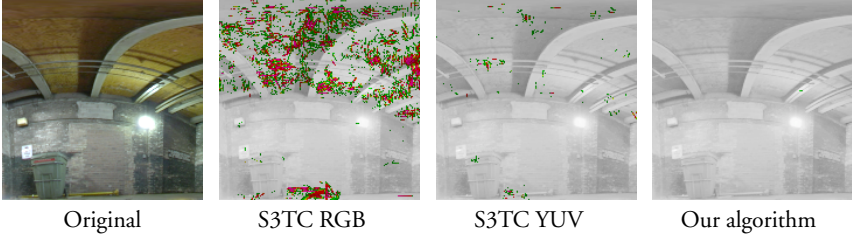


Figure 5.12: HDR VDP difference images. Green indicates areas with 75% chance of detecting an artifact, and red indicates areas with 95% detection probability.

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  $300 \text{ cd/m}^2$ . 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 5.12. 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 5.13. In general, luminance artifacts are more easily detectable, and both S3TC YUV and our algorithm handle these cases better due 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 sub-sampling, and artifacts in some images with high exposures, originating from the quantization of the base chrominances. Such a scenario is illustrated in Figure 5.14. 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 5.15 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.

## 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. 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](http://www.hdrimaps.com)) from the LightWorks HDRI Starter Collection ([www.lightworkdesign.com](http://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 5.10 (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:  $\{-10, -9, \dots, +4, +5\}$ .

Test image	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>e</b>	<b>f</b>	<b>g</b>	<b>h</b>
Start exposure	-9	-8	-8	-9	-3	-9	-4	0
Stop exposure	+4	+2	+5	+3	+7	+3	+7	8

Test image	<b>i</b>	<b>j</b>	<b>k</b>	<b>l</b>	<b>m</b>	<b>n</b>	<b>o</b>	<b>p</b>
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,  $(\alpha, \beta)$ , for each of the template shapes in Figure 5.8.

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)

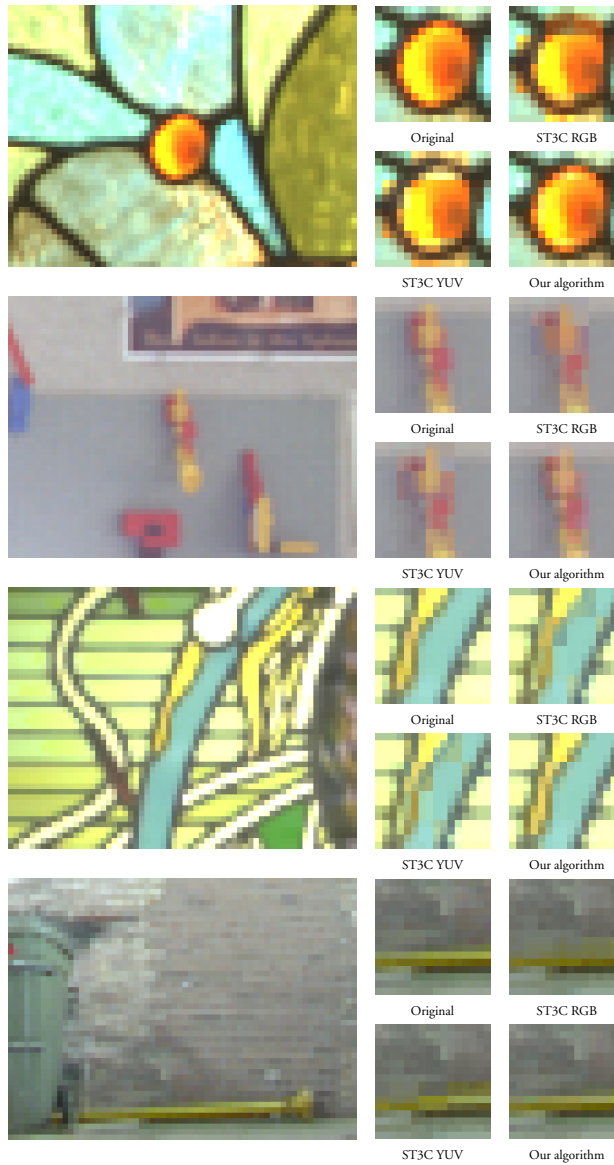


Figure 5.13: This figure shows magnified parts of the test images (c), (d), (h), and (i), compressed with our algorithm and the two S3TC-based methods. In difficult parts of the images, the S3TC algorithms sometimes produce quite obvious artifacts, while this rarely happens with our algorithm due to its separated encoding of luminance and chrominance.



Figure 5.14: 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.

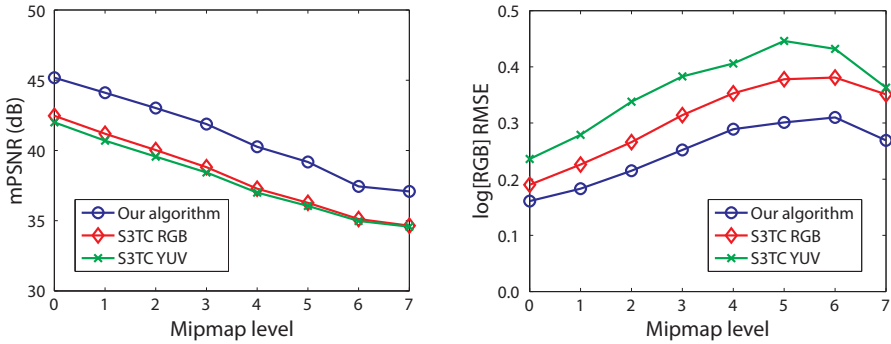


Figure 5.15: Here, the average mPSNR and  $\log[RGB]$  RMSE values are presented for various mipmap levels of our test images, where 0 is the original image and higher numbers are sub-sampled versions.





# 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 Troschianko. 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 Com-panding 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. Pre-dicting 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 *Proceed-ings 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 Dy-namic 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 Hand-book*. 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.anywhere.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.