Lossy and Lossless Compression Techniques for Graphics Processors

Jim Rasmusson
Department of Computer Science
Lund University

LUND INSTITUTE OF TECHNOLOGY
Lund University

ISSN 1652-4691
Licentiate Dissertation 14, 2012

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

Email: jim@cs.lth.se
WWW: http://www.cs.lth.se/home/Jim_Rasmusson

Typeset using LaTeX2ε
Printed in Sweden by Tryckeriet i E-huset, Lund, 2012

Abstract

Computer graphics is a field that has evolved in leaps and bounds the last 10-15 years. Still, it has a seemingly endless appetite for performance. The performance is used to introduce higher levels of visual realism at higher resolutions.

Data compression is an important area in the design of a graphics processor to be able to achieve higher performance and lower power. The data flows required by graphics processors during real-time rendering are typically very large and is one of the major performance bottlenecks. For this reason, all contemporary graphics processors have compression and decompression blocks implemented in hardware, at least to some extent.

This thesis presents new algorithms for buffer- and texture compression, addressing the two parts of the graphics pipeline that are the most memory bandwidth hungry. These new algorithms offer higher visual quality and/or improved compression rates as compared to the state-of-the-art algorithms we have benchmarked against.

The suggested new buffer compression algorithms introduce, besides novel lossless techniques, new lossy techniques that enables very high compression rates while keeping the introduced errors low and the visual quality high. It does so both for standard and high dynamic range color buffers.

The new texture compression algorithm introduced herein shows that it is possible to design a more specialized texture compression algorithm that significantly outperforms the standard algorithms in terms of visual quality, when using it for new texture types, such as light maps and radiosity normal maps.

Acknowledgements

I would like to thank my main supervisor Tomas Akenine-Möller for always being highly supportive and enthusiastic in helping an old time engineer from the industry to get started with academic research in the field of computer graphics.

Thanks also to Jacob Ström, my assistant supervisor, for superb support and discussions.

Furthermore I would like to thank my former manager at Ericsson Research, Lars Tilly, for the great encouragement and for arranging funding. Thanks also to Joakim Persson at Ericsson Research for great support.

In addition, I would also like to thank my colleagues in the Computer Graphics group for great discussions and collaborations.

The work presented in this thesis was carried out within the Computer Graphics group (LUGG) at the Department of Computer Science, Lund University. It was partly funded by Ericsson AB.

Finally, I would like to thank my wife, Britt-Marie, and our kids, Max, and Ida, for all love and support.

Preface

This Licentiate thesis summarizes my research on buffer and texture compression methods for graphics processors. The following papers are included:

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

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

III. Jim Rasmusson, Jacob Ström and Tomas Akenine-Möller, "Error-bounded Lossy Compression of Floating-Point Color Buffers using Quadtree Decomposition", in *The Visual Computer*, June, 2009.

IV. Jim Rasmusson, Jacob Ström, Per Wennersten, Michael Doggett and Tomas Akenine-Möller, "Texture Compression of Light Maps using Smooth Profile Functions", in *High Performance Graphics*, pages 143–152, 2010.

Contents

1 Introduction

Computer graphics is nowadays a common technology, present in everyday display-equipped consumer electronics, such as personal computers, mobile phones, and TV sets. The display panels are built up of many small picture elements (pixels) and it is basically the task of the graphics unit (graphics processor) to calculate what colors the pixels should have.

Computer graphics has evolved with tremendous speed over the last 10-15 years. Hand in hand with the progress in silicon technology and architectural advances in both hardware and software, we have seen several orders of magnitude increase in performance and visual realism. Also in embedded battery driven consumer devices, such as mobile phones, we find highly capable graphics processors.

Today computer graphics have many applications, where the most obvious ones are computer games and movie productions. Recently also furniture and clothing companies are producing many of their catalog images using photo realistic computer graphics. The books by Akenine-Möller et al. [4], and Pharr and Humphreys [27] serve as good introductions to real-time (gaming type) and photo-realistic computer graphics respectively

Using computers to generate images is a rather old science, almost as old as computer science itself. It started already in the early fifties. At that time, it was good to have a computer to perform the plentiful and repetitive geometry calculations involved when generating a two-dimensional image out of a three-dimensional model. Even 15th century painters were dealing with the problem of getting three-dimensional objects to look correct when delineating them on a two-dimensional painting. See Figure 1.

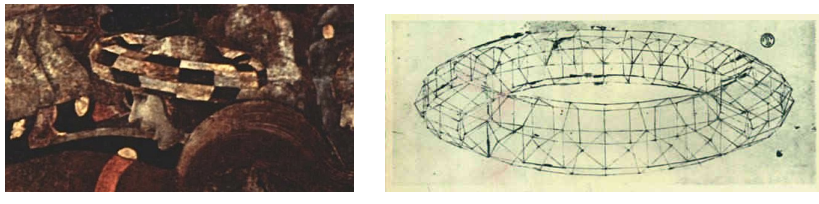


Figure 1: Paolo Uccello, 14xx-14xx, was one of the first painters who used more scientific methods to depict objects in a three-dimensional space. His methods inspired famous artists like Leonardo da Vinci and Albrecht Dürer. (From http://www.paolouccello.org/biography.html, under a Creative Commons License)

2 Three-Dimensional Graphics using a GPU

To generate a two-dimensional image from a three-dimensional model, it is common to utilize a *graphics processor* (often called GPU- Graphics Processing Unit).

Objects in a three-dimensional scene are typically built up by polygons, often triangles, that are connected together. An illustration of this is shown in Figure 2.

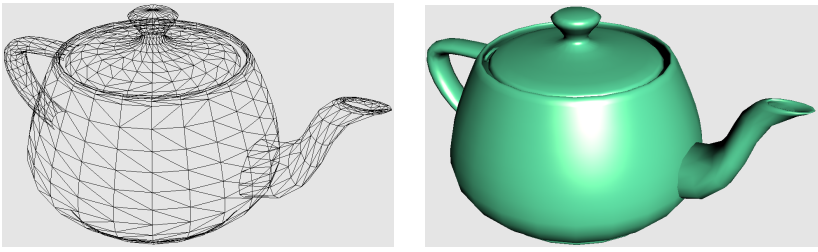


Figure 2: An example object, a teapot, is modeled by several small triangles as can be seen in the left wireframe picture. When the triangles are filled and shaded, the object may look like the figure to the right.

A triangle has 3 vertices (corners) and each vertex has an associated set of descriptive data, called *vertex attributes*. Examples of vertex attributes include the three-dimensional position (x, y, z), color and "material". To determine which part of the scene to be rendered into the two-dimensional image, a virtual camera is used. This is illustrated in Figure 3.

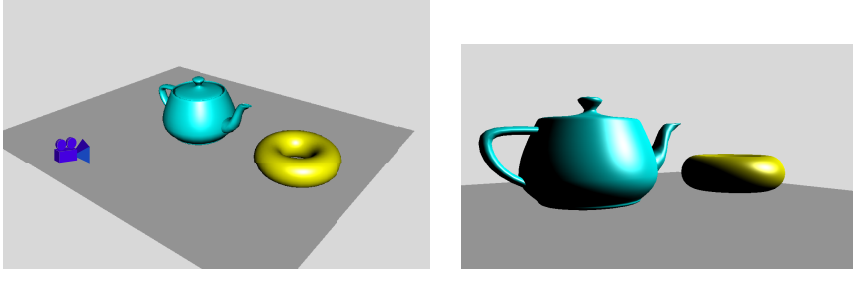


Figure 3: Left: a simple scene with a teapot and and a torus. A virtual camera (blue) is used to view the scene. Right: the scene seen form the virtual camera.

A high-level schematic rendering pipeline based on rasterization is depicted in Figure 4. An application running on a CPU handles the scene data and sends it to the graphics processing unit (GPU). The first stage of the GPU is the geometry processing stage. Here, the vertices and associated attributes of the objects are processed by software programs, called *vertex shaders*, that are executed on the GPU [20]. The objects are rotated and moved to their intended positions by the use of transformations (which is done by multiplying the vertices with transformation matrices). Lighting of the vertices and the position and viewing direction of the (virtual) camera are also calculated in the geometry stage. Finally, the vertices are projected from the three-dimensional world to the two-dimensional screen and the screen positions of the vertices are calculated. This data is then sent on to the

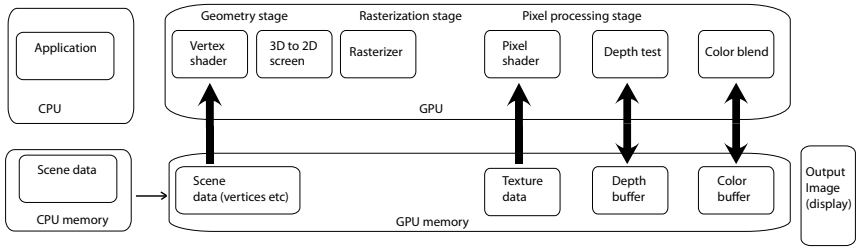


Figure 4: A high-level graphics pipeline

next stage, the *rasterization* stage. After rasterization follows the *pixel processing* stage. Rasterization and pixel processing are described in the next section. Part of the pixel processing stage is *texturing*, which is described in Section 2.2.

Note that the above description is somewhat simplified. More thorough descriptions can be found in many books, such as *Real-Time Rendering*[4].

It should also be noted here that the pixel processing stage dominates in terms of memory bandwidth and it has been an ambition in all papers of this thesis to develop methods to reduce memory bandwidth consumption.

2.1 Rasterization and Pixel Processing

The pixel grid of a display panel (Figure 5) may serve as a natural target raster for the rasterization process. A pixel consists of three color components (red, green, and blue) and, as you can see in the figure, each pixel has a red, green and blue sub-pixel. Different colors are generated by mixing different amounts of these three base colors. For example, to generate the brightest and most saturated yellow color, the red and green sub-pixels have their maximum intensity while the blue sub-pixel have its minimum intensity (zero).

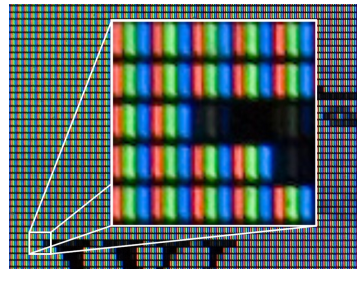


Figure 5: A close up photo of a liquid crystal display (LCD) panel. As can be seen, it is built up from a regular pattern of red, green and blue elements (sub-pixels).

In the rasterizer stage, each triangle is mapped against a raster grid and the pixels that are inside the triangle are found by using *triangle traversal* algorithms [5, 28,

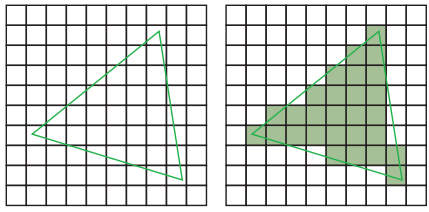


Figure 6: The rasterization process finds the pixels (or more precise: the pixel samples) that are inside the triangle

29]. The traversal algorithms are relying on *edge equations* [21, 28], to test which pixels are inside or not. An illustration of the rasterization process is shown in Figure 6. These pixels and their associated shading data are send further down the pipeline. A large part of the pixel shading data comes from interpolated vertex attributes. For example, the color attribute of a pixel inside a triangle is interpolated from the colors of each of the three triangle vertices. The actual rasterization and interpolation are typically done in hardware [24, 28].

Shading the pixels are done in a similar fashion to vertex shading. Special software programs, called *pixel shaders*, are executed for each pixel and the colors of the pixels are calculated. An important part of pixel shading is *texturing* that will be described in Section 2.2.

Once the resulting color of the pixel has been generated, it is time to apply the result to the *frame buffer*, which generally consists of a *depth buffer* (also called *Z-buffer*) and a *color buffer*. The depth buffer contains the depth values of the pixels that are (so far) closest to the camera (and thereby visible). In order to find out if the current pixel is placed 'in-front-of' the old pixel stored in the frame-buffer, a *depth test* is performed, where the depth of the current pixel is tested against the depth buffer value (at the same screen position). If the current pixel has a depth less than the depth buffer value this means that the current pixel is even closer to the camera and thereby occludes the stored pixel. The depth value of the current pixel is then written to the depth buffer and the color value in the color buffer is updated. The resulting pixel color can either simply replace the color in the frame buffer or they can be blended together in various ways, e.g., additive or multiplicative blending.

2.2 Texturing

Texture mapping (or *texturing*) was introduced by Ed Catmull [8] already in 1974 and it significantly improved the level of realism of the rendered images. It can be explained as "glueing an elastic image" on the objects. Using texturing, it is possible to add plenty of details to the objects in a simple and efficient way. For example, if we want the teapot to look like it was made out of wood, we can apply a wood texture image to it. This is shown in Figure 7.

Figure 7: The teapot textured with a wood-like texture

Texturing is done as a part of the pixel shading. The triangles' vertices have associated texture coordinates (also called *uv*-coordinates) that maps a part of the texture image to the triangle. See Figure 8 for an illustration.

This type of texturing is called *image texturing* and was the first texturing technique. Since then the texturing techniques has evolved tremendously and contemporary computer games typically make use of a number of different texturing techniques. Examples include texturing techniques to give the illusion of additional geometric detail (bump and normal mapping [9], parallax mapping [18]), and to model more realistic surface lighting and light interaction in a pre-computed fashion (light maps [8], directional light maps such as radiosity normal mapping [12, 22, 23]).

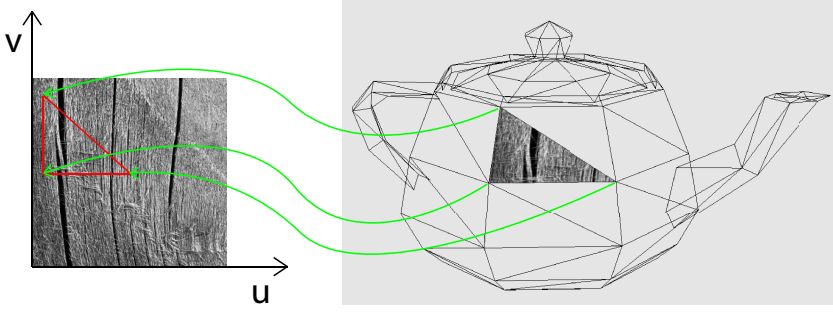


Figure 8: The teapot where only one triangle is textured. Note that each vertex has a texture coordinate, denoted *u* and *v*, that points into the texture (illustrated in the figure). The texture coordinates for the pixels inside the triangle are interpolated from the three vertices.

Texturing may require large amounts of memory from the external GPU RAM, and this explains why textures are typically *compressed*. See Section 3.2 for an introduction to texture compression.

2.3 Memory Bandwidth

The memory traffic between the GPU and the external GPU DRAM is typically dominated by the pixel processing stage (see Figure 4). This stage produces plenty of memory accesses for the color and depth buffers, as well as texturing accesses. Small on-chip cache memories are used in most architectures to mitigate the need for memory bandwidth, but even so the requirements for external DRAM bandwidth are high.

The memory transfer speed requirements of GPUs are many cases so high that even high-end graphics cards with rather 'extreme' memory systems, capable of bandwidths well above 100 GByte/s, may still be constrained by memory throughput when rendering high-end games at high resolutions.

Even more constrained are embedded GPUs in battery driven devices such as mobile phones. Mobile phones typically use a unified memory system where the CPU and the GPU share the same external DRAM. Since the application running on the CPU also may require significant amounts of DRAM accesses, memory traffic congestion is common. These embedded memory systems are also optimized for low power consumption and are thereby often clocked at lower frequencies which reduce memory transfer speeds even further.

2.4 Memory vs. Compute Performance

One would think the problem of low memory speeds would be solved by the rapid evolution in silicon technology, but it is actually getting worse over time (see Figure 9). The memory vs. compute performance gap is actually widening over time [16, 25, 26, 30]. In essence, the evolution of memory technology increase performance at a rate of around 7% per year, whereas the the processing technology increase performance at a rate of around 50% every year.

Hence, the importance of technologies that reduce memory bandwidth usage is high and increasing. Besides compression techniques, which will be discussed in Section 3, there are a number of other tricks that can be applied, such as various types of culling techniques [3, 4, 6, 13, 14, 15].

If we take a look at where the power goes, it is dominated by moving data [10, 11]. Performing calculations on the data consumes much less power and this paves the way for new algorithms with higher complexity but also higher compression performance than in the past.

The algorithms in the papers in this thesis are designed with this in mind.

3 New Compression Algorithms

Compression of rendering data is vital for the performance of graphics systems. In practice, all contemporary commercial GPUs rely on compression units to re-

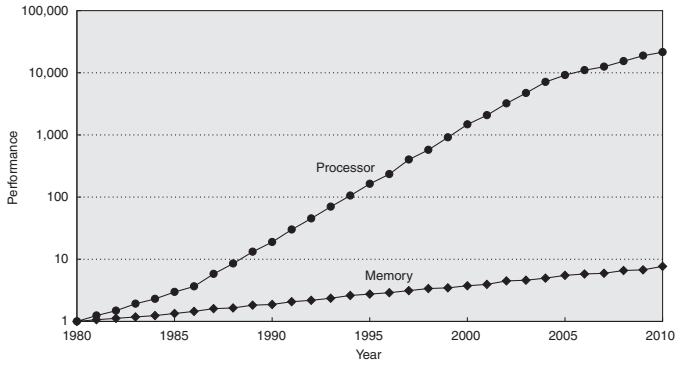


Figure 9: The CPU - Memory performance gap is widening over time. Courtesy of Morgan Kaufmann Publishers. From "Computer Architecture: A Quantitative Approach, 5th Edition," by John Hennessy and David Patterson [16]. Published by Morgan Kaufmann Publishers, an imprint of Elsevier, Inc., copyright ©2012.

duce memory bandwidths and thereby achieve intended performance. For example, most rendering buffers (e.g., color- and depth-buffers) and textures are compressed to some extent. The compression/decompression units are typically implemented as hardware blocks in the GPU. However, it may make sense to have compression/decompression implemented in software, especially in contemporary *General Purpose* capable GPU architectures (called *GPGPUs* [1]).

The papers in this thesis focus on compression algorithms for graphics rendering. These algorithms are, for the most part, designed with a complexity level and choice of algorithmic components that makes sense for hardware implementations, but they may work for software implementations as well.

3.1 Buffer compression

One of the more demanding rendering buffers in terms of memory bandwidth usage, is the color buffer. The color data for every pixel residing in the GPU DRAM is typically read and modified (e.g., blended) with the color data generated by the pixel shader and thereafter written back to the GPU DRAM. The memory bandwidth can be substantial and it is advantageous to have this data compressed as it is transferred back and forth over the memory bus.

A block diagram of a buffer compression architecture is illustrated in Figure 10. The color data is compressed before it is written out to GPU DRAM by an 'on-the-fly' compressor unit (also called encoder) and correspondingly it is decompressed by an 'on-the-fly' decompressor (also called decoder) after it is read from the DRAM and before it is used by the color blend unit. The compression schemes typically operates on 8×8 (or 4×4 or 16×16) pixel tiles (also called pixel blocks). For increased efficiency, most buffer compression implementations use a cache to

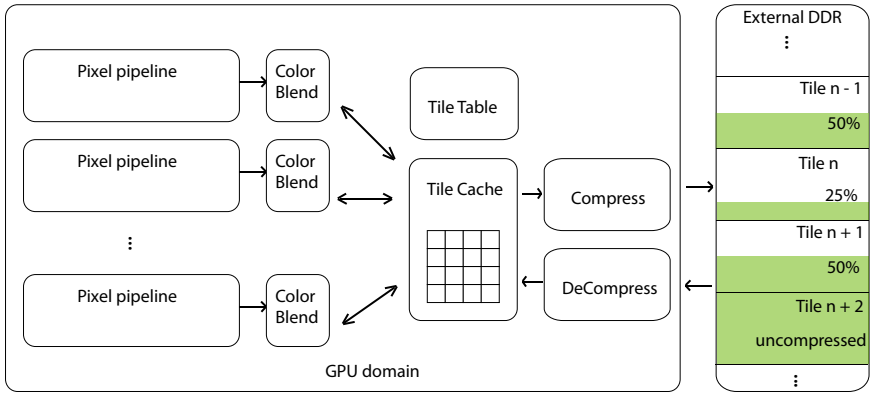


Figure 10: Buffer compression architecture. Each 8×8 pixel tile is compressed before it is written out to external DRAM and correspondingly decompressed after it is read back from the DRAM. With a variable bit rate compression scheme you get different amount of compression depending on how hard the data is to compress. You may get $4\times$ compression rate (25% of the original amount) in some cases and no compression at all for more difficult tiles (uncompressed).

further reduce memory bandwidth. The tile table is there to indicate what compression mode is used for a particular tile. For example, a tile can be compressed to 25% of the original amount of data, or, if it is a tile that is hard to compress with the compression scheme at hand, it may be transferred in uncompressed form. The tile table stores a few bits per tile to indicate this. Since each tile is compressed to a different degree (different amount of bits), the compression scheme has a bit rate that varies to some degree (called *variable bit rate* compression scheme). The compression scheme in use is often designed to reconstruct the data in a bit-exact manner. This is called *lossless* compression schemes.

With a lossless scheme you know that you will get the original data back but it is often hard to achieve high compression levels. If you instead allow a few errors in the reconstructed data (i.e., the original data is only reconstructed approximately), it is possible to achieve much higher compression levels. This is called *lossy* compression schemes.

This idea is explored in two of the papers in this thesis, Paper 1 and 3. These papers introduce the use of novel *lossy* compression schemes (in addition to novel *lossless* schemes). Lossy compression schemes are more challenging to use however. Not only is it vital to reconstruct the pixels as close to the originals pixels as possible, the color buffer data is also often read, modified and written back several times (i.e., compressed and decompressed several times), even for the same pixel. Hence, there are significant risks for error accumulation and, as a consequence, strong visible artifacts.

These lossy compression methods therefore need to be accompanied by some sort

of error-control mechanism. An example error-control mechanism is described in Paper 1. If you can control and harness the error build-up, the memory bandwidth savings can be substantial using lossy compression schemes (illustrated in Figure 11). The details can be found in Paper 1 and 3.

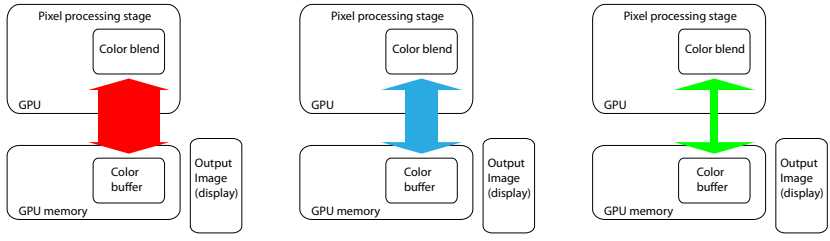


Figure 11: Left: uncompressed color buffer bandwidth is very high. Middle: lossless compression provides some bandwidth reduction (around a factor 2-3). Right: lossy compression provides more bandwidth reduction (around a factor 8-10 reduction or more with negligible artifacts). The width of the arrows indicates memory bandwidth usage

Floating point data is sometimes used when rendering graphics. It brings benefits such as high dynamic range (HDR) and enables more visual richness and some effects. It can be challenging to handle for compression algorithms, but in some cases, it is possible to cast the floating point bit pattern to an integer data type and handle it as integer data by the compression and decompression scheme, and then re-cast it back to floating point after decompression.

The problems of compressing floating point data, together with a novel lossless floating point compression scheme, is described in Paper 2. It is actually possible to compress also floating point data in a lossy fashion. This is explored in Paper 3.

The first paper (Paper 1) on color buffer compression was co-authored with Jon Hasselgren and Tomas Akenine-Möller. The author of this thesis had the initial idea of applying lossy compression to the color buffer, which had not been done before. This was very interesting because higher compression rates can be achieved. I was the main author of this paper, but Jon made significant contributions to it as well. My focus was mainly on the lossy parts of the algorithm whereas Jon focused mainly on the lossless parts.

The second paper (Paper 2) on floating point compression and the unified codec architecture was co-authored with Jacob Ström, Per Wennersten, Jon Hasselgren and Tomas Akenine-Möller. I was not the primary author of this paper. My contributions were mainly in the the unified codec part and to some extent in the color buffer compression part.

The third paper (Paper 3) was co-authored with Jacob Ström and Tomas Akenine-Möller. This is a continuation on the color buffer compression work in both paper 1 and 2. My contributions were significant in all parts of this work.

3.2 Texture Compression

Texture compression schemes are different from buffer compression schemes. The textures are typically read-only and it is only the *decompression* unit that is needed in the texturing hardware of the graphics processor. The compression is done off-line during the content creation stage and utilizes algorithms with high complexity that may take several tens of seconds, or even more time, per texture image to compress. The texture compression and decompression schemes thus typically have a highly *asymmetric* complexity. See the papers by Beers et al. [7] and Knittel et al. [19] for good introductions.

The asymmetric complexity is in contrast to the buffer compression and decompression schemes which have relatively *symmetric* complexity (both the compression and the decompression are done 'on-the-fly' in hardware).

Texture compression schemes are also lossy and introduce errors in the reconstructed texture images. These schemes are also *fixed bit rate* (in contrast to buffer compression schemes which are typically *variable bit rate*). The requirement for fixed bit rate comes from the fact that the texels needed for texturing, may be placed all over the texture in arbitrary positions. To be able to easily calculate the memory address where the data for any arbitrary texel resides in GPU DRAM, a fixed sized data structure is needed and the compression is therefore done so that it yields a fixed number of bits per texel tile.

Having a fixed number of bits per texel poses certain challenges when you are to compress areas of rapidly changing texel data. The resulting texture image quality will therefore vary quite much depending on the texture data.

As mentioned in Section 2.2, there are several more advanced texturing techniques in use in contemporary computer games, such as bump and parallax mapping and light maps.

One technique that is used to create more realistic lighting in computer games is *radiosity normal mapping (RNM)* introduced by Valve in 2004 [12, 22, 23] and used also by Electronic Arts (EA). This technique makes use of *directional light maps* which are calculated off-line and stored in textures. Common texture compression algorithms, such as S3TC/DXTC [2, 17] have problems with compressing these type of *smooth light map textures* with decent quality.

Paper 4 explores a novel compression algorithm for these type of textures. The main idea here is to model the smooth surfaces with a small set of pre-defined *profile functions* (the concept is illustrated in Figure 12). A few modifier parameters are used to fit these functions to a 4×4 pixel tile. See paper 4 for details.

It should be noted that storage for texture data in contemporary computer games is huge. For example, the game Mirror's Edge from Electronic Arts, have around 3 GB of compressed RNM textures. Without texture compression, these textures would consume six times as much, i.e., 18 GB of data and it would have been impossible to ship the game on a DVD.

Paper 4 was co-authored with Jacob Ström, Per Wennersten, Michael Doggett, and

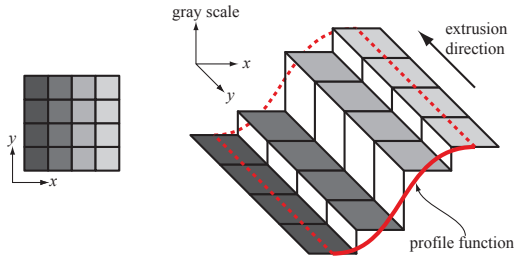


Figure 12: From Paper 4. Left: an example 4×4 pixel tile with a smooth transition. Right: illustration of how a profile function is used to describe the content of the pixel block to the left. Note how the profile function is merely a function of x, and then extruded in the y-direction.

Tomas Akenine-Möller. The original idea, to use some kind of parametric surface function to model the smooth gradients in a pixel tile of a light map was conceived by Jacob. He also made some early Matlab experiments to test the idea. I joined the discussion early on and all authors jointly developed the algorithm to the final state. With some initial help from Tomas, I did most of the multi-threaded C++ and OpenCL implementations of the algorithms and generated most of the results.

4 Conclusion and Future Work

All algorithms presented in this thesis describe compression techniques useful for graphics rendering. They target for the most part hardware implementations in graphics processors but can be useful also for software implementations.

These types of compression techniques are vital for all contemporary and future graphics processor designs, as they reduce memory bandwidth and power, enable higher rendering performance, and in some cases also reduce memory storage requirements.

Memory bandwidth is a scarce resource even in high-end graphics cards. Even worse is the fact that memory bandwidth is evolving slowly compared to other silicon based system performance metrics, such as number of calculations per second. This means that there is increasingly a need for new compression algorithms, taking advantage of the silicon evolution by exploiting higher complexity algorithms that offer higher compression rates as well as improved visual quality.

The lossy buffer compression algorithms presented in this thesis are of particular interest as they bring substantial reductions in memory bandwidth and thus enable significant performance benefits.

The texture compression algorithm presented in the last paper of this thesis may serve as an example of a new category of texture compression algorithms, that, by exploiting parametric surface models, offers superior quality and performance, at least for particular types of textures.

For the future, there are lots of things to explore. The lossy buffer compression algorithms can, although challeging, perhaps be applied also to the depth buffer.

Another topic for the future is perhaps to design a more flexible hardware accelerator for compression and decompression. Such a hardware block should be programmable or at least configurable to some extent. This will open up opportunities for new innovative algorithms with high visual quality and high compression rates.

Bibliography

[1] http://www.gpgpu.org/wiki/.

[2] http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt.

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

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

[5] 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.

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

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

[8] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[9] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 115–122, 1998.

[10] William Dally. Power efficient supercomputing. Accelerator-based Computing and M;anycore Workshop, Lawrence Berkeley National Laboratory. http://www.lbl.gov/cs/html/Manycore_Workshop09/GPU%20Multicore%20SLAC%202009/dallyppt.pdf, 2009.

[11] Michael Garland and David B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[12] Chris Green. Efficient Self-Shadowed Radiosity Normal Mapping. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)*, 2007.

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

[14] Jon Hasselgren and Tomas Akenine-Möller. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics*, 26(92), 2007.

[15] Jon Hasselgren, Jacob Munkberg, and Tomas Akenine-Möller. Automatic Pre-Tessellation Culling. *ACM Transactions on Graphics*, 28(19):19:1–19:10, 2009.

[16] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[17] 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.

[18] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, pages 205–208, 2001.

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

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

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

[22] Gary McTaggart. Half-Life 2 / Valve Source Shading. In *Game Developers Conference*, 2004.

[23] Jason Mitchell, Gary McTaggart, and Chris Green. Shading in Valve's Source Engine. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)*, 2006.

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

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

[26] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM : IRAM. *IEEE Micro*, 17:34–44, 1997.

[27] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition*. Morgan Kaufmann, 2010.

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

[29] Ramchan Woo, Sungdae Choi, Ju-Ho Sohn, Seong-Jun Song, Young-Don Bae, and Hoi-Jun Yoo. A low-power graphics lsi integrating 29mb embedded dram for mobile multimedia applications. In *Asia and South Pacific Design Automation Conference*, pages 533–534, 2004.

[30] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News*, 23(1):20–24, 1995.

Paper I

Exact and Error-bounded Approximate Color Buffer Compression and Decompression

Jim Rasmusson[‡†] Jon Hasselgren[‡] Tomas Akenine-Möller[‡]

[†]Ericsson Research [‡]Lund University

jim|jon|tam@cs.lth.se

ABSTRACT

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

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

1 Introduction

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

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

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

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

2 State-of-the-art Color Buffer Compression

In this section we summarize the color buffer compression algorithms we have found in patent databases. A summary of existing depth buffer compression schemes is already available [5]. The reader is referred to Section 2 of this paper for an overview of the depth buffer architecture, which is almost identical to the color buffer architecture. This paper describes the general methodology for selecting and tracking what compressor to use for a specific tile, and how to handle tiles that cannot be compressed.

2.1 Multi-Sampling Compression

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

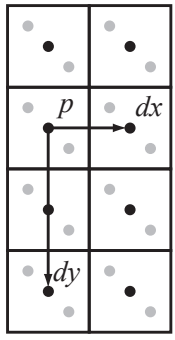
Elder explains that due to multi-sampling, samples inside a pixel often share the exact same color, and this is an opportunity for compression [4]. If all samples inside a pixel share the same color, then it suffices to flag this mode, and store only one color instead of n colors. Another common case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

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

2.2 Color Plane Compression

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

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



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

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

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

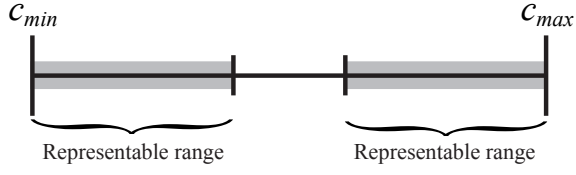


Figure 2: Color offset compression, when using the min and max colors as references. Note that the width of the representable color intervals vary with the number of bits allocated for the per pixel offsets.

2.3 Offset Compression

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

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

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

2.4 Entropy Coded Pixel Differences

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

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

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

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

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

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

3 A New Exact Color Buffer Compression Algorithm

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

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

3.1 Reversible Color Transforms

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

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

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

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

Transforming back is as simple:

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

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

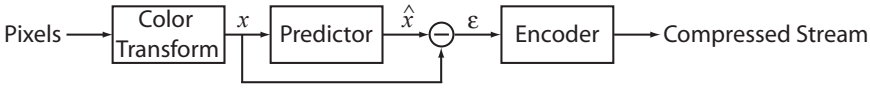


Figure 3: Overview of our compression algorithm.

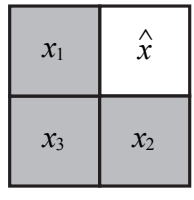
the commonly used standard Yc_rc_b transform is not, in general, reversible without loss.

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

3.2 The Algorithm

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

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



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

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

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

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

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

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

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

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

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

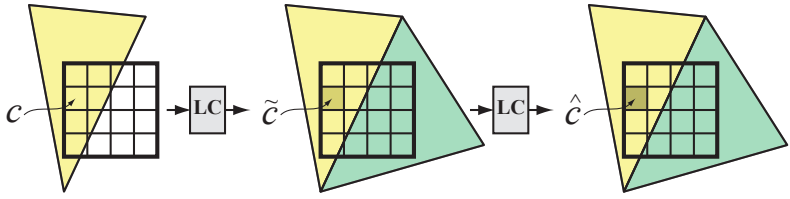


Figure 4: Illustration of tandem compression. Left to right: first a triangle is written to a tile. For one pixel, we track its original color, c. After lossy compression (LC), we obtain an approximation, $\tilde{c}$. However, when a second triangle is written to the tile, $\tilde{c}$ may be compressed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

4 Error-bounded Approximate Compression

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

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

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

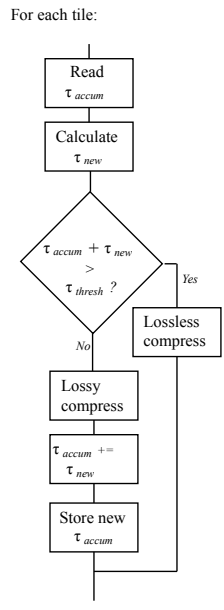
Note that buffer compression & decompression must be symmetric, i.e., execute in about the same amount of time, since these procedures run in real time inside the GPU. This means that the majority of all (lossy) texture compression schemes immediately disqualify, since compression often takes several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

4.1 The Error Control Mechanisms

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

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

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



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

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

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

4.2 A Lossy Algorithm

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

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

As an example, if $\tau^{rmse}_{thresh} = 2$ with 8×8 pixel tiles, we have $\tau^{max}_{thresh} = 8 \times 2 = 16$. In a practical implementation it may make more sense to use MSE instead of RMSE, since this avoids the expensive square root. However, that also doubles the number of bits for the accumulated error. Another useful error metric is the sum of absolute differences (SAD).

When it comes to the actual approximation, we have taken a gentle approach and use only (conditional) 2×2 subsampling of the chrominance components. Higher

compression rates can of course be obtained with more "brutal" subsampling, quantization, and other lossy methods.

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

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

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

5 Results

We have evaluated our compression algorithms using a software based simulation framework, which implements a tile-based triangle rasterizer with a modern color buffer architecture. It also simulates tile caching, using a 1 kB fully associative cache, and implements the color compression algorithms described in this paper. In addition, we used a logging OpenGL driver to record the rendering calls from actual games. This means that our results include the full, incremental, rasterization process of the games. They are not just compressed screenshots.

To benchmark the color compression algorithms, we used the four test scenes shown in Figure 5. The first scene is designed to stress high contrast colors, and the following two scenes are relatively colorful scenes taken from Quake 3 maps.[3] The final scene features complex particle effects with blending, and is taken from the game Quake 4. It should be noted that this scene use blending based on the alpha value stored in the color buffer. Therefore, we compress the full RGBA components for this scene, while we only compress the RGB components for the remaining scenes. This shows that all algorithms are suitable for compressing alpha data as well.

Note that we will refer to each compression algorithm by the names we used in Section 2. See the titles of each subsection.

[2] Chrominance errors can spread into the luminance channels due to tandem compression.

[3] The maps have been taken from the Quake 3 add-on "Urban Terror".

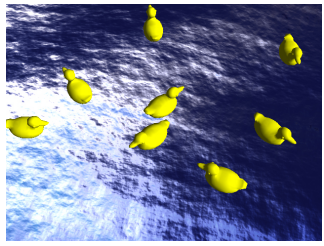
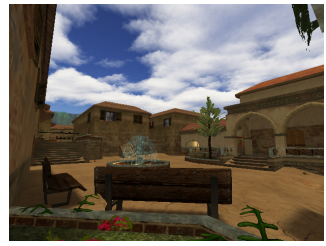
		
Plane	1.13:1	1.43:1
Color Offset	1.90:1	2.06:1
Entropy Coding	2.09:1	2.60:1
Our	2.64:1	2.87:1
		
Plane	1.40:1	1.47:1
Color Offset	2.15:1	2.30:1
Entropy Coding	2.66:1	2.93:1
Our	3.36:1	3.05:1

Figure 5: Evaluation of the compression algorithms: the table shows compression ratios for our test scenes (top left: "Ducks", top right: "Square", bottom left: "Car", and bottom right: "Quake4") using exact compression algorithms. We computed the compression ratios as the average compression ratio for rendering resolutions 320×240, 640×480 and 1280×1024 pixels. The algorithms scaled similarly with varying resolutions.

5.1 Exact Compression

The effective compression ratios of the different exact algorithms are presented in Figure 5. We used 8×8 pixel tiles and variable bit-rate encoding for offset compression, entropy coding, and our algorithm. Variable bit-rate coding comes very natural to our algorithm and entropy coding. For offset compression, we implemented variable bit-rate in the sense that we use no fixed bit allocations for the offsets. We simply use the least amount of bits that is capable of representing the largest offset in the tile.

The plane compression algorithm is special in that we used the specified 2×4 tile size and only the two modes specified in the patent. We would like to emphasize that this algorithm is disfavored in this evaluation since it is so specialized. A

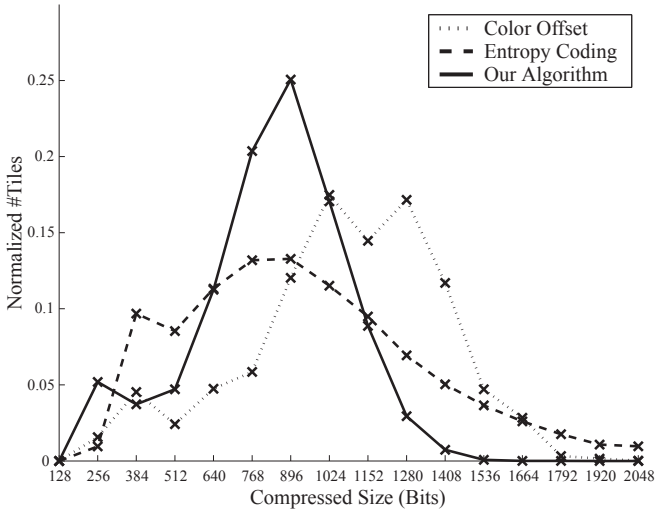


Figure 6: A normalized histogram of the number of tiles that are compressed to a given size (in multiples of 128 bits), using different algorithms. We use 8×8 pixel tiles, which means that 2048 bits indicate uncompressed tiles. The histogram is based on an average over all our test scenes. Note that our algorithm has a distinct peak, which makes it efficient when only a few compressed sizes can be used. It is of course also essential that the peak is located at good compression ratios, i.e., to the left in this diagram.

generalized version of plane compression may generate better results, but this is left for future work.

We would also like to point out that our measurements (Figure 5) do not take hardware limitations, such as the width of the memory bus, into account. Furthermore, in most hardware implementations, we can only afford a few different compressed sizes, since the compressed size of a tile is typically stored in on-chip memory so that we know beforehand how many memory words to read. In order to measure the algorithms performance with respect to these limitations, we computed the compressed size histograms shown in Figure 7. Note that we have left out plane compression since it is so specialized, and only allows for 128 or 256 bits per tile.

Next, we show how to interpret this diagram with an example. Assume that we allow two fixed sizes for tiles: 1024 bits for compressed and 2048 bits for uncompressed. In this case, the number of tiles compressed to 1024 bits will be the integral from 0 to 1024 over the histogram, while the uncompressed tile will be the integral from 1024 to 2048. Using the histogram, we can easily find the best n sizes for each algorithm using an exhaustive search. In Table 1, we present the best compressed sizes for $n = 1, 2, 3$. Note that an extra uncompressed size always needs to be available as a fallback.

Color Offset		
#Sizes	Best sizes (Bits)	Effective Compression
1	1280	1.43:1
2	1024,1408	1.58:1
3	896,1152,1408	1.61:1
∞		2.04:1
Entropy Coding		
#Sizes	Best sizes (Bits)	Effective Compression
1	1024	1.52:1
2	768,1280	1.75:1
3	640,1024,1408	1.88:1
∞		2.45 : 1
Our Algorithm		
#Sizes	Best sizes (Bits)	Effective Compression
1	1024	1.78:1
2	896,1152	2.04:1
3	640,896,1152	2.17:1
∞		2.88:1

Table 1: The tables show how the algorithms perform when given a number of allowed compressed sizes, as well as what selection of sizes that worked best for our test suite. Note that our algorithm performs very well even with very few compressed sizes.

5.2 Approximate Compression

In Figure 7, we show the results from our experiments with approximate compression. The Quake 4 scene is excluded since alpha handling is currently not implemented in the lossy part. As can be seen, the additional compression gains can be quite substantial. We can gain an additional 25–60% compression by allowing approximate compression. The visual impact is normally small as can be seen in Figure 8 and in the $SSIM_{rgb}$ plot (Figure 7c). See Section 5.3 for more information on *SSIM*. However, the "ducks" scene clearly shows artifacts (Figure 9) already for small levels of τ^{rmse}_{thresh}. This is due to that we use chrominance sub-sampling, which makes chrominance leak out to surrounding pixels. For a case like this, a more conservative error metric could be used, e.g., a maximum error threshold. This would decrease the effect of these artifacts. Our most important contribution for lossy buffer compression is the error control mechanism, and we believe our results shows that it works well, and that it can keep high image quality. However, more research is clearly needed on lossy compression algorithms.

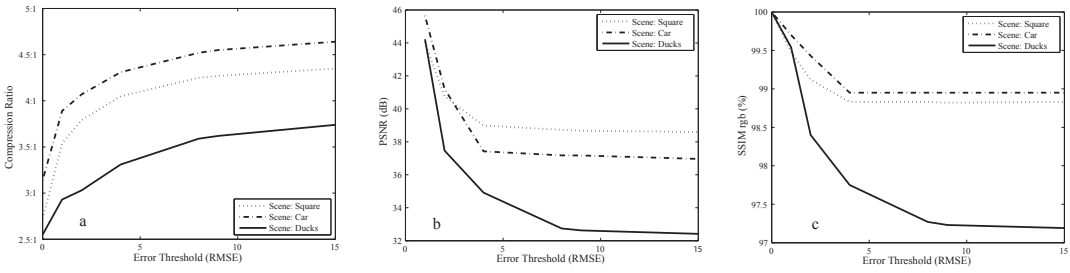
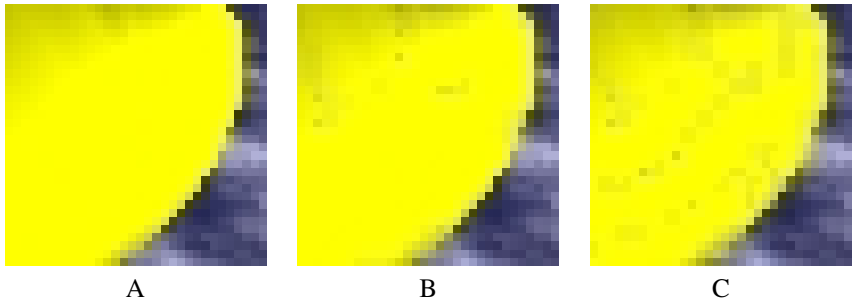


Figure 7: Approximate (lossy) compression results for three of the test scenes (average of three rendering resolutions 320×240, 640×480 and 1280×1024 pixels). From the left: a) compression ratio vs. τ^{rmse}_{thresh}, b) PSNR vs. τ^{rmse}_{thresh}, and c) $SSIM_{rgb}$ vs. τ^{rmse}_{thresh}.

A B C

Figure 8: Crops from the "Square" scene. A: original, B: $\tau^{rmse}_{thresh} = 4$, $PSNR = 39.0$ dB, $SSIM_{rgb} = 98.8\%$, compression ratio = 4.1:1, C: $\tau^{rmse}_{thresh} = 15$, $PSNR = 35.6$ dB, $SSIM_{rgb} = 98.8\%$, compression ratio = 4.3:1.



A B C

Figure 9: Crops from the "Ducks" scene. A: original, B: $\tau^{rmse}_{thresh} = 4$, $PSNR = 34.9$ dB, $SSIM_{rgb} = 97.8\%$, compression ratio = 3.3:1, C: $\tau^{rmse}_{thresh} = 15$, $PSNR = 32.4$ dB, $SSIM_{rgb} = 97.2\%$, compression ratio = 3.8:1.

5.3 Structural Similarity Index - SSIM

In addition to the common error metric, PSNR, we also use the structural similarity index, SSIM, as suggested by Wang et. al [16]. This is a visual quality metric which attempts to mimic the human visual perception. The SSIM index is a number between 0% and 100%, where 100% is perfect similarity. Note that the SSIM index is normally calculated using the luminance alone. In order to get the errors in all three color channels, R, G and B respectively, we have chosen to calculate the SSIM index for the R,G and B channels independently and combining them into a single number, $SSIM_{rgb}$, according to:

$$SSIM_{rgb} = 0.2126 * SSIM_R + 0.7152 * SSIM_G + 0.0722 * SSIM_B \quad (5)$$

where the weights comes from ITU-BT.709 [8].

6 Conclusions and Future Work

Color buffer compression is available in almost all (if not all) GPUs, but up until now, this type of algorithms have not been described in the literature. By providing an overview of existing algorithms, we now have an important stepping stone in place, which is needed to invent new algorithms.

In addition, we have presented new algorithms based on a decorrelated color transform, which is also exactly reversible. Our results show that this can improve the compression ratio compared to other algorithms. Since it is well-known in the image and video compression community that the human visual system is more sensitive to luminance than chrominance, we have also done some initial results on approximate color buffer compression with this reversible transform.

We note that it is very important to keep the accumulated error under strict control, and we presented a simple mechanism to do this. We realize that approximate compression is a feature that must be turned off for some applications, but for, e.g., gaming on mobile devices, it can be very valuable with a longer use time on the battery with a only slight degradation in image quality.

We have only experimented with simple compression algorithms for approximate color buffers, and for future work, there is much to learn and transfer from the image and video processing field. We have started to investigate more sophisticated and fine-grained sub-sampling and quantization schemes. There is definitely room for inventing new algorithms. High dynamic range (HDR) color buffer compression is also an interesting topic for further studies.

In our paper, we have not handled multi-sampling, but several of the techniques [4, 11] for this can be merged relatively quickly into our work. Finally, we note that lossy depth buffer compression might not be feasible, due to the artifacts that can arise when surfaces intersect. However, this could be worth further investigation.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, Ericsson, and NVIDIA's Fellowship program. Thanks to the anonymous reviewers for their helpful comments.

Bibliography

[1] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide. `http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_Programming_Guide_0.8.pdf`.

[2] Kurt Akeley. RealityEngine Graphics. *Readings in computer architecture*, pages 507–514, 2000.

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

[4] Gordon M. Elder. Method and Apparatus for Anti-Aliasing using Floating Point Subpixel Color Values and Compression of Same. In *US Patent Application 2006/0188161 A1*, 2006.

[5] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.

[6] Timothy J. Van Hook. Method and Apparatus for Compression and Decompression of Color Data. In *US Patent Application 7039241 B1*, 2006.

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

[8] SG06 ITU-R. ITU, Recommendation BT.709 : Parameter values for the HDTV standards for production and international programme exchange. In *ITU-R, BT.709*, 2002.

[9] JPEG2000. ISO/IEC 15444-1:2000, JPEG 2000 Image Coding System, Annex G2, Reversible Component Transformation. In *ISO/IEC 15444-1:2000*, 2000.

[10] Henrique Malvar and Gary Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. In *JVT-I014r3*, 2003.

[11] Steven E. Molnar, Bengt-Olaf Schneider, John Montrym, James M. Van Dyke, and Stephen D. Lew. System and Method for Real-Time Compression of Pixel Colors. In *US Patent Application 6825847 B1*, 2004.

[12] Steven L. Morein and Mark A. Natale. System, Method, and Apparatus for Compression of Video Data using Offset Values. In *US Patent Application 2003/0038803 A1*, 2003.

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

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

[15] Robert F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 22, Jet Propulsion Lab, 1979.

[16] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P.Simonelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing,*, 13(4):600–612, 2004.

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

Paper II

Floating-Point Buffer Compression in a Unified Codec Architecture

Jacob Ström[†] Per Wennersten[†] Jim Rasmusson[‡†] Jon Hasselgren[‡] Tomas Akenine-Möller[‡]

[†]Ericsson Research [‡]Lund University

{jacob.strom|per.wennersten}@ericsson.com
{jim|jon|tam}@cs.lth.se

ABSTRACT

This paper presents what we believe are the first (public) algorithms for floating-point (fp) color and fp depth buffer compression. The depth codec is also available in an integer version. The codecs are **harmonized**, meaning that they share basic technology, making it easier to use the same hardware unit for both types of compression. We further suggest to use these codecs in a **unified codec** architecture, meaning that compression/decompression units previously only used for color- and depth buffer compression can be used also during texture accesses. Finally, we investigate the bandwidth implication of using this in a **unified cache architecture**. The proposed fp16 color buffer codec compresses data down to 40% of the original, and the fp16 depth codec allows compression down to 4.5 bpp, compared to 5.3 for the state-of-the-art int24 depth compression method. If used in a unified codec and cache architecture, bandwidth reductions of about 50% are possible, which is significant.

Proceedings of Graphics Hardware, pages 75–84, 2008.

1 Introduction

To increase performance for graphics processing units (GPUs), bandwidth reduction techniques continue to be important [1]. This is especially so, since the yearly performance growth rate for computing capability is much larger than that for bandwidth and latency for DRAM [18]. The effect of this can already be seen in current GPUs, such as the GeForce 8800 architecture from NVIDIA, where there are about 14 scalar operations per texel fetch [16]. Algorithms can often be transformed into using more computations instead of memory fetches (e.g., instead of evaluating, say, $\sin(x^2+y^2)$ as a lookup in a precomputed texture, this expression could simply be evaluated by executing the relevant instructions). However, at some point, the computation needs are likely to be satisfied, and then the GPU will be idle waiting for memory access requests. Also, the brute force approach of duplicating the number of memory banks and accessing these by using more pins on the chip may not be possible in the long run. Hence, we argue that memory bandwidth reduction algorithms is a field of research worthy of more attention.

There are many types of bandwidth reduction techniques, but here we will focus mostly on algorithms related to some form of compression on the pixel or fragment level. Other techniques are mentioned in Section 5. The two major compression techniques for GPUs are *buffer compression* [6, 19] and *texture compression*, introduced in 1996 [3, 10, 22]. Since buffers, such as color, depth, and stencil, are created during rendering using both reads and writes, these algorithms must be rather symmetric in that compression and decompression are performed in similar amounts of clock cycles. Furthermore, the majority of these algorithms are exact, i.e., lossless, even if there are exceptions for error-bounded color buffer compression [19]. For lossless buffer compression, there must always be a fallback to sending uncompressed data so that all blocks can be processed in the system. Since random access is required, these algorithms do not reduce storage, only bandwidth usage is reduced when transmitting buffer data. One or a few bits are stored on-chip or in a cache to indicate whether the block is compressed or not, and this memory is called the tile table. In recent papers, there are new buffer compression algorithms and surveys for depth buffer compression [6] and for color buffer compression [19]. As a special mode in buffer compression techniques, it is also possible to implement a fast clear operation [14], which is a type of compression. When a buffer clear is requested, each block of pixels can flag (in the tile table) that the block has been cleared. When such a block needs to be accessed, the flag is first checked, and it is discovered that the block is cleared, and hence there is no need to read the block from external memory. Instead, all the pixels in the block are simply set to a "clear"-value. This makes buffer clears inexpensive.

The two most widely adopted texture compression schemes are S3TC, also known as DXTC [13], which has seen widespread adoption both for OpenGL and DirectX, and ETC [21], which has been adopted in the OpenGL ES API, targeting mobile devices. In contrast to buffer compression techniques, texture compression algorithms can be asymmetric so that the compression phase can take much longer

than the decompression. The reason for this is that textures usually are read-only data, and hence, compression can be done in a preprocess. It is only when the GPU accesses the texels that decompression needs to be done, and hardware for this must be fast and of relatively low complexity. To simplify random access, most texture compression schemes compress to a fixed rate, e.g., 4 bits per pixel. Note that texture compression actually reduces the storage needs as well as the bandwidth usage. An exception to the fixed rate rule is presented by Inada and McCool [8] who use B-tree indices to implement a lossless variable bit rate texture compression system. Rendering to such textures would be possible, which should mean that the algorithm could be used for buffer compression as well.

Overview In this paper, we present two new codecs for fp color buffer compression and fp depth buffer compression. We have not seen any previous work in either of these two fields before. Whereas these two new algorithms give rise to bandwidth savings in their own right, the paper also investigates how different types of hardware architectures influence these savings.

OpenEXR [17] is a format for fp image compression, and while it cannot be used for buffer compression as is, we have created a modified version of one of its codecs to benchmark our fp color buffer compression algorithm.

In OpenGL, the framebuffer object extension `EXT_framebuffer_object` allows render-to-texture to be performed without data being copied. For instance, it is possible to create a shadow map by rendering to a depth buffer, and later render from that depth buffer without having to first copy the data to a texture. If depth buffer compression is used, and the texturing unit cannot decode the compressed shadow map, it is necessary to first decompress the data before texturing takes place, which may be as costly as copying the data. Alternatively, compression of the shadow map can be turned off, which is approximately equally bad. Therefore, the natural next step is to allow rendering from textures irrespectively of what compression format the data is stored in. This paper refers to this as a *unified codec* architecture.

If any buffer/texture can use any compression format, it helps if the different formats are based on the same basic techniques, so that a hardware unit can decompress several formats with little extra logic. Such compression formats are denoted *harmonized codecs* in this paper.

In CPU architectures, there are systems which have separate data and instruction caches, while other systems use unified caches. Sometimes, the L1 caches are separate and the L2 cache unified. Separate caches have the advantage that instruction caches, being read-only, can be made simpler than data caches, which are read-modify-write. Unified caches, on the other hand, have the advantage that a large part of the cache can be used for instructions if instruction traffic is high and data traffic low, and vice versa. In this paper, we investigate what happens to memory bandwidth usage if the texture caches (read-only) and the buffer caches (read-modify-write) in a graphics architecture are unified. This paper denotes this a *unified cache* graphics architecture.

Next, we will describe our new floating point codecs, followed by a description on

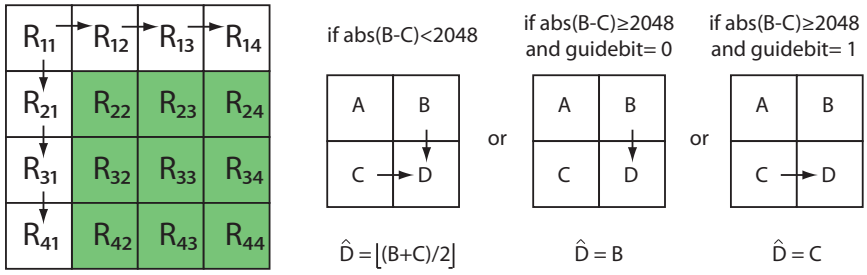


Figure 1: Left: Arrows indicate prediction. For instance, R_{12} is used to predict R_{13}. Pixels marked in green can be predicted from one or two pixels, as shown to the right.

what type of architecture they are tested in.

2 Compressing RGBA half Data

To the best of our knowledge, no (published) attempts have been made at color buffer compression for floating-point RGBA data, even though there are works in nearby fields such as texture compression of HDR textures [15, 20], floating point data compression (not for buffers) [12], integer RGBA color buffer compression [19] and lossless fp image compression [17].

Our method builds on the work of Rasmusson et al. [19]. We first divide the image into 8×8 blocks, and each block is given two bits in the tile table to indicate whether it is fast-color-cleared, uncompressed, compressed to 50% (2048 bits) of original size or compressed to 25% (1024 bits) of original size. Often, the destination alpha is not used, and therefore we assume it is 1.0 during compression. We also assume that the color components are positive, storing only the last 15 bits of the half. If a block violates either of these two assumptions, the uncompressed mode is used instead. The 8×8 block is further subdivided into four 4×4 sub-blocks. The idea is to first encode the red component separately, and then encode the difference between the green and red, and finally the difference between blue and green. This is a simple but efficient way of exploiting the correlation between the color channels, and hence an expensive YUV transformation is avoided, which lowers complexity.

The first pixel of the red component R_{11} (see left part of Figure 3) is stored directly using 15 bits. The rest of the pixels in the first row and column are predicted from the previous pixel as indicated by the arrows. For instance, R_{12} is predicted using $\hat{R}_{12} = R_{11}$, and instead of storing R_{12} directly, the prediction error $\tilde{R}_{12} = R_{12} - \hat{R}_{12} = R_{12} - R_{11}$ is calculated and stored. Hopefully, $\tilde{R}_{12}$ should be smaller and simpler to code than R_{12}. However, the floating-point nature of halfs leads to problems. Since the density of floating-point numbers is not uniform (see

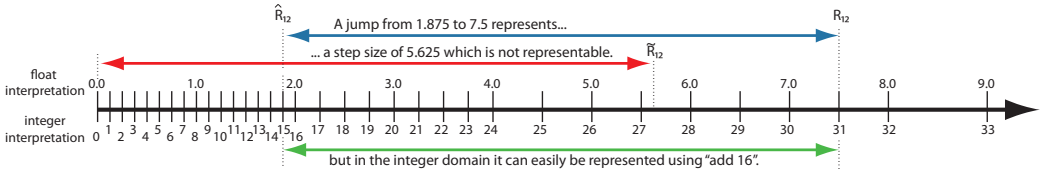


Figure 2: Lossless prediction in the floating-point domain can be tricky, since differences between two representable floating-point numbers may not be representable. In the example, both the prediction $\hat{R}_{12} = 1.875$ and the target $R_{12} = 7.5$ are representable, but the difference $\tilde{R}_{12} = 5.625$ is not. Doing the prediction in the integer domain instead solves this problem.

Figure 2), the difference, $\tilde{R}_{12}$, between two floats may not be representable (red line). This problem is solved in the seemingly counter-intuitive but well documented way [12] of treating the halfs as integers. Due to the clever way floating-point numbers are defined, neighboring halfs will be interpreted as neigbhoring integers, as can be seen below the axis in the figure. Note that this trick works because we only compress positive halfs as described above. Following the green arrow, adding the difference 16 to the prediction 15 gives the correct result 31, or 7.5 if interpreted as a float. Doing the arithmetics in this "integer domain" also avoids costly floating-point operations, and since the compression is lossless, we will also get a correct handling of NaNs, Infs and denorms.

For the values marked with green in Figure 3, we can predict using the values that we already have encoded above and to the left of the pixel. As seen to the right in Figure 3, we can predict the value D using A, B and C. Color buffer compression differs from other image compression in that there may be an unnaturally sharp color discontinuity between a rendered triangle and pixels belonging to the background or to a previously rendered triangle of a very different color. In order to avoid doing prediction across such discontinuity edges, we propose the use of *guide bits*. If the difference between B and C is larger than a threshold, one bit is used to indicate whether we will predict from B or C. If the difference is smaller than the threshold, the prediction will be $\lfloor (B+C)/2 \rfloor$ (where $\lfloor \cdot \rfloor$ denotes rounding to the nearest lower integer), and no guide bits will be used. We found that this simple predictor works better than the standard Weinberger predictor [23], even when accounting for the cost of the guide bits. Our predictor uses $\text{abs}(B-C) < t$, where we found that $t = 2048$ works well for half data. The predictor is also illustrated in Figure 3,

The leftmost block in Figure 3 shows an example of a block with such a discontinuity edge. Traversing the pixels in scan-line order, the boxed pixel will be the first different value, and we call this the *restart value*. Its position in the block (4 bits) and its value (15 bits) will be stored explicitly. We have used exhaustive search among all 15 positions to find the best restart value, but faster heuristic approaches can also be used. We describe one such heuristic for *depth* values in Figure 4 .

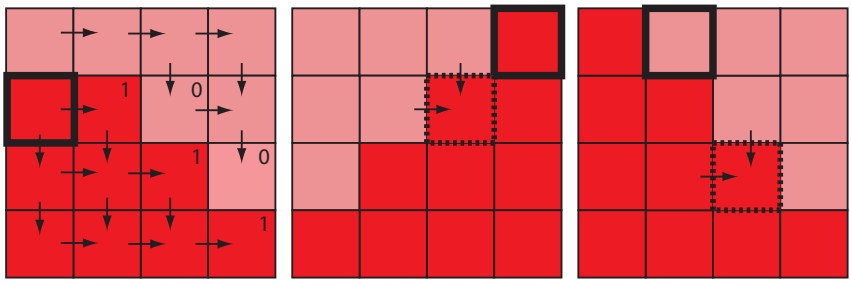


Figure 3: Left: The restart value is marked with a box. Note how the guide bits make sure no prediction is made across the discontinuity. Middle: Sometimes a pixel (marked with dots) can only choose from two erroneous predictors. This is alleviated by rotating the block 90 degrees counter-clockwise, as shown to the right.

Note how the five guide bits in Figure 3 make sure that prediction is never done across the discontinuity. In some cases, as shown in the dotted pixel in the middle illustration, neither the top nor the left pixel will give a good prediction. Therefore, we have introduced a bit to indicate whether the block is rotated or not. After rotation, the same pixel has a better chance to choose a good predictor, although there are still degenerate cases when this is not possible.

Next, the prediction errors $\tilde{R}_{xy}$ are stored. Since these are differences between 15-bit numbers, they are 16-bits signed integers. The first step is to make them positive so that a Golomb-Rice coder can be used. By applying the function $n(x) = -2x$ to negative numbers and $p(x) = 2x - 1$ to positive ones, the new arrangement will be $\{0, 1, -1, 2, -2, 3, -3 \ldots\}$, which means that numbers of small magnitudes will have a small value. Each value will then be Golomb-Rice encoded: First, it is divided by 2^k, creating a quotient and a remainder. The quotient is encoded using unary encoding according to the table to the right. Values larger than 31 are encoded using `0xffff` followed by the 16 bits of the value. The k bits of the remainder are then stored. Four pixels in a 2×2 group share the same k-value, which is stored before the quotient and remainder data. An exhaustive search between 0 and 15 is used to find the best k for the group. In practice, though, the search can be made smaller by looking at the bit position p of the most significant bit of the largest value. The best k is almost always found in the interval $[p-4, p]$.

Code	Value
0_b	0
10_b	1
110_b	2
1110_b	3
11110_b	4
...	

Whereas the red component is encoded independently of the other data, the green component is encoded relative to the red one. First, the difference between the green and the red component is calculated for each pixel, again treating the fp data as 15-bit integers. Then this difference is encoded in the same way as the red component was above, with two changes. First, the restart value and the top left

value are not treated separately, but are fed into the Golomb-Rice encoding just as the other values are, but without prediction. Second, the prediction pattern from the red component is used again, meaning that no extra guide bits need to be sent. After this, the difference between the blue and the green component is encoded in the same way as the difference between green and red.

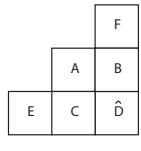
Many blocks have uniform data, and for them it may be unnecessary to encode a restart value. Therefore we reserve one bit to indicate whether we use the restart value or not. The total data structure is as follows: restart bit (1 bit), restart pos (4/0 bits), restart value (15/0 bits), rotate bit (1 bit), start value (15 bits), k-values (16 bits), guide bits (variable), and Golomb-Rice bits (variable).

2.1 Compressing Depth Data

The fp depth buffer compression algorithm, which is the first method of its kind to the best of our knowledge, is built using the same techniques as the color buffer compression system. In this way, a single hardware unit can handle both depth and color with little extra hardware. Hence, the data is also predicted, and the prediction error is encoded using Golomb-Rice. However, the way the prediction is carried out is different.

We use the complementary-Z representation of Lapidous and Jiao [11] with no sign bit, three bits of exponent and 13 bits of mantissa, which is sometimes an alternative to 24 bit integer depth buffers. Since this move from 24 bits to 16 bits is in itself a data reduction by 2/3, it gives us a head start against 24 bit integer based depth buffer compression techniques. Note that most integer depth compression algorithm rely on the fact that the depths of a plane (e.g., from a triangle) are coplanar (up to rounding accuracy) in the depth buffer. However, for floating-point depths, this may not be true, especially when the exponents differ. However, for values of the same exponent, they are still coplanar, so a planar prediction still produces good results, and the remaining deviations can be handled by the Golomb-Rice encoding. However, planar prediction requires more pixels to predict from compared to the RGBA case.

The algorithm is intended to be able to handle two different planes per 4×4 subblock. The top left pixel belongs per definition to plane 0, and its value Z_{11} will be stored. For cleared blocks, Z_{11} will be equal to the Z_{far}-value, hence one bit is used to signal whether this is the case or if it should be stored explicitly using 16 bits. The value of the first encountered pixel of plane 1, which we call the restart value or Z_R for short, will also be stored explicitly. A bit mask telling which plane each pixel belongs to is also stored. The figure to the right shows all the pixels that can be used to predict the value D. However, it can only predict from values that belong to the same plane as D. If A, B and C belongs to the same plane as D, it will use the prediction $\hat{D} = B + C - A$. Else, if B and F belong to the same plane as D, it will use $\hat{D} = 2B - F$. Else, it will try $\hat{D} = 2C - E$. If this also fails, but B and



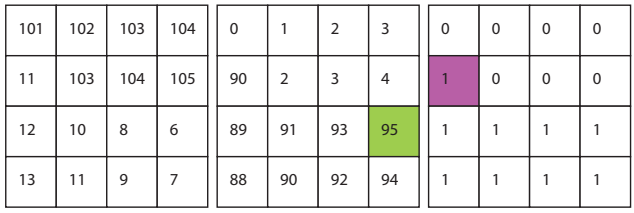


Figure 4: The position of the restart value Z_R is calculated as follows: First the absolute difference to element Z_{11} is calculated (middle). The element most different from Z_{11}, called Z_{diff}, is found (marked with green). Then each pixel that is closer to Z_{11} than to Z_{diff} is allotted a restart pixel of value of 0, else 1 (right). The restart position is the first pixel in traversal order of value 1 (purple).

C both belong to the same plane as D, one *extra guide bit* will be sent to choose between the predictions $\hat{D} = B$ and $\hat{D} = C$. If only one of them share planes with D, that one will be used and no extra guide bit will be spent. Finally, if none of these pixels are of the same plane as D, it will use the value of the first encountered pixel of that plane. For plane 0, this means $\hat{D} = Z_{11}$, and for plane 1, this means $\hat{D} = Z_R$. If any of the pixels A, B, C, E and F are outside the block, they are treated as not belonging to the same plane as D.

The position for the restart value, Z_R, is determined as shown in Figure 4. First, the pixel is found that differs the most from Z_{11}. This is done by calculating the absolute value $|Z_{xy} - Z_{11}|$ for each pixel position (x, y), as shown in the middle diagram. This pixel, which is green in the figure, is called Z_{diff}. Then a *restart bit* is given to each pixel: if $|Z_{xy} - Z_{11}| < |Z_{xy} - Z_{\text{diff}}|$ it will be 0, otherwise 1. The restart bits are then traversed in the prediction order, and the position of the first '1' will define the position of the restart value, marked with purple. The restart bits are only used for determining the restart value during compression, and are not stored. One could use the restart bits for the guide bits, but it turns out to be better to select the guide bits during compression; both '0' and '1' are tried for each guide bit, and the choice giving the best prediction will be selected. If this method of encoding two separate planes is not beneficial, the entire block can be compressed as a single plane instead. One bit is used to indicate which method was used.

The differences between the predicted and the actual values are encoded using Golomb-Rice encoding just as in the color buffer compression scheme, with one important difference. Only values predicted using two or more values, such as $\hat{D} = B + C - A$, $\hat{D} = 2B - F$ and $\hat{D} = 2C - E$, use the normal k value indicated for the 2×2 block. Values predicted from just one other pixel, i.e., using the predictions $\hat{D} = B$ and $\hat{D} = C$, will use $k_2 = \lfloor k/2 \rfloor + 10$. The reason for this is that the errors are much larger in this case, and would force too big a k value to be used if not compensated for in this way. These values can be seen as the encoding of the slope of the plane, whereas the values predicted from two values are merely

the deviations from the plane, which should be small. We have tried encoding the k-values independently from each other, but this gave very little coding gain over the simple $k_2 = \lfloor k/2 \rfloor + 10$ formula. This is fortunate, since the search space for the best k becomes much smaller. In addition, due to very small errors within a plane, $k = 0$ is a very common case, so using only one bit to indicate wether $k = 0$ leads to further gains.

The depth scheme uses two modes, where the first is 192 bits and the second 768 bits. Many 8×8 blocks consist of pixels from only one plane, and for those it is unnecessary to store several starting values for the 4×4 sub-blocks. The guide bits will also be unnecessary, since the entire block is only one plane. Therefore, the 192-bit mode uses just one 8×8 block instead of four 4×4 blocks. Also, in the 8×8 case, no guide bits or restart value are stored, and k-values are selected for 4×4 blocks instead of 2×2 blocks. In summary, the bits are as follows, for each 4×4 sub-block of the the 768-bit mode: $Z_{11} = Z_{\text{far}}$ (1 bit), Z_{11} (16/0 bits), two-plane mode (1 bit), guide bits (15/0 bits), restart value Z_R (16/0 bits, k ($4 \times 6/1$ bits), extra guide bits (variable), Golomb-Rice bits (variable). For the 192-bit mode, $Z_{11} = Z_{\text{far}}$ (1 bit), Z_{11} 16/0 bits), k ($4 \times 6/1$ bits), extra guide bits (variable), Golomb-Rice bits (variable). As a reference, we have also created a 24-bit integer version of the algorithm. It is exactly the same, except for the fact that the starting and restart values are stored with 24 bits instead of 16 bits.

Codecs can be put together in many ways, especially when it comes to details. However, we hope to have showed that some overall strategies are useful, such as not predicting across discontinuities.

3 Implementation

In order to evaluate our compression algorithms, and their impact on different architectures, we have built a software-based simulation framework, which can be configured to emulate a host of different architectures. All configurations use Zmax-culling [4, 14] and Zmin-culling [2] to avoid unnecessary fragment accesses. Pixels are rendered tile-by-tile in order to maximize data locality, and the tile size is 8×8 pixels. Fast Z-clears [14] are also used. In all our tests, we use HDR texture compression (8 bits per texel) [15], and normal map compression (8 bits per normal) when possible. The caches for all configurations store decompressed data, are fully associative and use a least recently used replacement policy.

The first configuration, A, is a conventional architecture [9] (shown on the top of Figure 5), with separate caches for textures and buffers (a non-unified cache), and where it is not possible to render from textures compressed with buffer compression schemes (a non-unified codec structure). Furthermore, no fp color buffer compression is used, however 24-bit integer depth buffer compression is employed. We have used a texture cache [5, 7] of 13 kB, a color buffer cache of 1 kB and a depth buffer cache of 512 bits. The texture cache is rather large, and the reason for this is

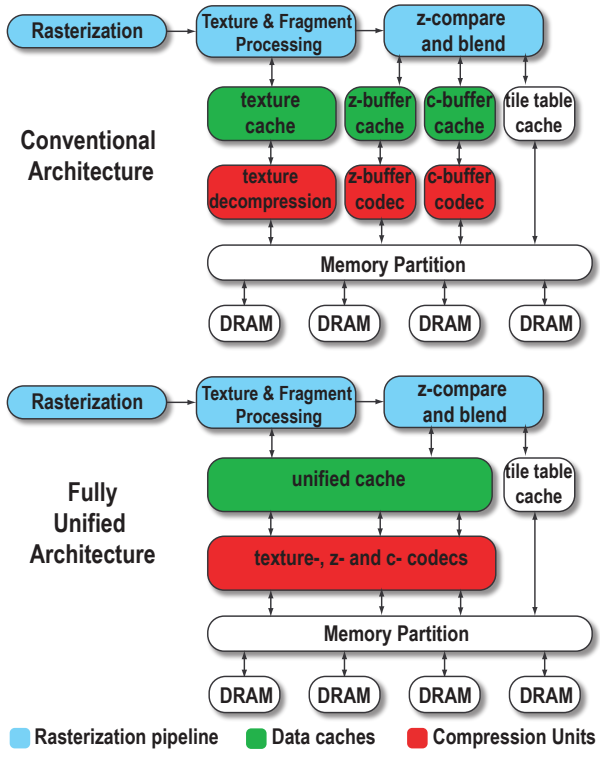


Figure 5: Conventional vs unified cache architecture. In the bottom part, both the cache and the codecs are unified.

that we use floating-point data throughout, which increases storage needs. We also use 256 bits for a tile table cache to store the tile table bits for the depth buffer.

Configuration B is equal to A except that it includes fp color buffer compression. It thus shows the benefit of adding the proposed fp color buffer compression scheme to a traditional architecture.

Configuration C equals B except that it also implements unified codecs, meaning that compressed color and depth buffers can be recast as textures without any need for decompression. However, since different caches are used, the cache must be flushed before the recast takes place.

Finally, configuration D utilizes both fp color buffer compression, unified coders and unified caches, as shown in the bottom part of Figure 5. The unified cache is of 14 kB for texture, color buffer and depth buffer, and a common tile table cache of 768 bits. Thus the configuration D is given equal amounts of cache memory as compared to A, B and C.

Example To highlight the differences between configuration A and D, consider the case of shadow mapping. The shadow map is created by rendering a depth buffer,

which is used as a texture (the "shadow map"). When rendering the final image, the shadow map is used as lookup data to determine if a fragment is in shadow or not. In configuration A, depth buffer compression can be used in the first shadow map generation pass. However, when it is time to cast the depth buffer to a texture it has to be decompressed and when it is subsequently used as a texture in the later passes it can only be used in uncompressed form. In configuration D, due to the unified cache architecture, the depth buffer can use the entire cache during the creation of the shadow map, increasing hit rate and lowering bandwidth usage. Furthermore, the cache does not need to be flushed when recasting the shadow map from a buffer to a texture. Due to the unified codec architecture, no bandwidth-consuming decompression is needed during recasting. In the final pass, the shadow map is accessed in compressed form, which further reduces bandwidth.

We use three test scenes; *Water*, *Shadows*, and *Reflections*, where all render to fp16 color buffers. HDR textures in cube maps and object textures make sure that the dynamic range of the floating-point color buffer is used. Note that the final fp color buffers are tone mapped before display, so even though a screenshot may seem simple to compress (e.g., large bright area), this is very seldom true since tone mapping is a non-linear operator with clamping at the end, and this hides details in the fp colors. *Water* is a rather regular scene, without any render-to-texture, which means that the unified codec architectures cannot exploit the important feature of reusing codecs for render-to-textures, when accessing these as textures. Still, we included this simple example in order to show that bandwidth can still be saved under those conditions. The second scene is called *Shadows*, and since it contains a shadow map, it is designed to highlight the benefits of the proposed system. *Reflections* renders to a dynamic floating-point cube map every frame, and a sphere reflects the surrounding objects using the cube map.

4 Results

In this section, we present the results from our simulations. Performance numbers for the fp16/int24 depth buffer compression and the fp16 color buffer compression are presented first, followed by overall performance results.

The diagram in Figure 6 shows the performance of the proposed depth buffer compression schemes (Section 3). We have rendered the *Shadows* scene in resolutions ranging from 160×120 to 1600×1200, and plotted the rate for our fp16 depth buffer compression algorithm (dashed red circles). Since there is no previous fp16 depth buffer algorithm to compare to, we plot it against the int24 algorithm of Hasselgren and Akenine-Möller [6] (blue triangles), which we believe is state-of-the-art. As can be seen, our new technique has a substantially lower rate. However, the results are not directly comparable, since the original data is fp16 and not int24. Therefore, we also show the result of our proposed int24 technique, which outperforms the state-of-the-art at lower resolutions. The reason the state-of-the-art encoder does well for large resolutions is that it can compress an 8×8 block

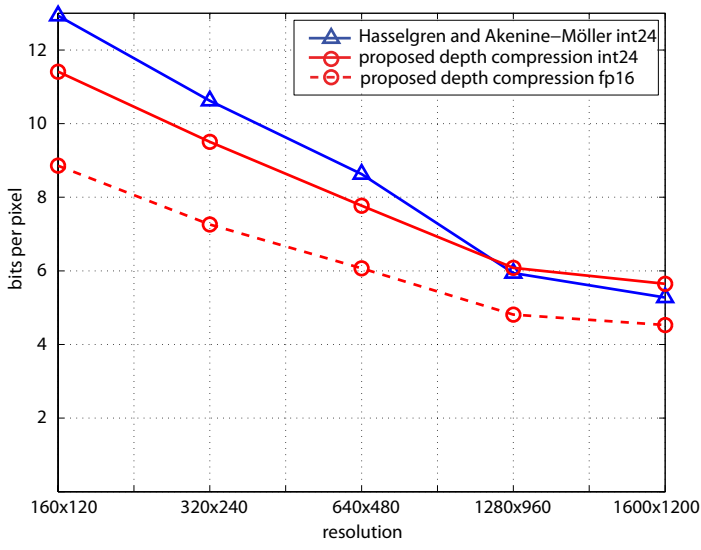


Figure 6: Depth compression results for the "Shadows" scene for different resolutions. Note how our proposed fp16 depth buffer (dashed red circles) has a significantly lower rate than the state-of-the-art int24 system (blue triangles). Our proposed int24 system (solid red circles) outperforms the state-of-the-art for high-complexity (low-resolution) scenes.

down to 128 bits if it is a single triangle covering the block, whereas the proposed algorithms will need 192 bits. For very high resolutions, this will happen more and more frequently. Note however that this means very big triangles with respect to the pixel size. Already at the cut-over resolution of 1280×960, we have an average size of 507 pixels per triangle. We argue that the lower resolutions, which are equivalent to a higher scene complexity/smaller triangles, are more important, and here both proposed methods are better than the state-of-the-art.

The proposed fp16 color buffer codec algorithm is shown in Figure 7 with green crosses. There is no previous fp color buffer compression method to compare to. As a comparison, both the HDR texture compression algorithms from Munkberg et al. [15] and Roimela et al. [20] use 8 bpp (blue squares in Figure 7), whereas our buffer compression method goes down to about 25 bpp. Note however, that the methods used for HDR texture compression are lossy, and therefore not amenable to color buffer compression. In fact, being lossy is a huge gain; as a comparison, lossy JPEG codecs typically operate at around 2 bpp whereas lossless PNG codecs manage around 12 bpp. Nor is it a good idea to use our fp color buffer compression method for texture compression; too many simultaneous textures could thrash the tile table cache, resulting in abysmal performance. The extra latency needed in order to first fetch the tile table data before knowing how many bytes to read from memory is manageable if the number of such textures is small so that the tile table data is mostly in the cache. While a scene would typically contain just a

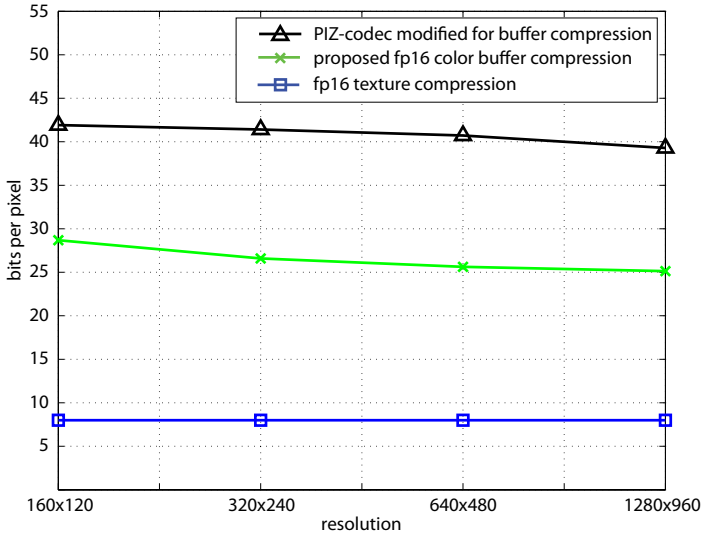


Figure 7: Fp16 color buffer compression results for the "Shadows" scene for different resolution. Black triangles: The PIZ-codec from OpenEXR, modified to be used for buffer compression, as a benchmark. Note that the PIZ-codec is designed for larger images without discontinuities, but it is included here for completeness. Green crosses: the proposed fp16 compression scheme. Blue squares: The HDR texture compression methods of Munkberg et al. and Roimela et al., which cannot be used for buffer compression.

few render-to-textures (no thrashing), it would most likely have several ordinary textures even per pixel (risk of thrashing). In its current form, we would therefore recommend not using our fp color buffer compression for all regular fp16 textures, but definitely use it for fp16 render-to-textures. Note also that the proposed method reduces bandwidth down to 40% of the original size—a significant compression. Results for the *Water* and *Reflections* scenes are similar at about 43%. Admittedly, much of this compression comes from demanding that destination alpha equals 1.0. However, even if it was assumed that the original color buffer was RGB and not RGBA, the compression would still reduce the size to 53%–60%.

We also compare our method to the fp image compression scheme in OpenEXR [17] (black triangles in Figure 7). Since it is not intended for buffer compression, we have made a number of changes to make the comparison. OpenEXR uses several codecs, of which the PIZ-codec is the one best suited for RGBA data. It performs a Haar wavelet transform and then stores the resulting symbols using run length encoded Huffman symbols. A PIZ-encoded block includes a header of 20 bytes which can most likely be made much smaller—we have therefore removed 20 bytes from the calculated byte-size of each PIZ-encoded block. Moreover, the PIZ-codec is created for RGBA textures, whereas our algorithm assumes that alpha is 1.0 everywhere. This gives the PIZ-codec an unfair disadvantage. In order

to compensate for that, we compressed an RGBA image of 16×16 pixels with the PIZ-encoder where R=2.0, G=3.0, B=4.0 and A=1.0 everywhere, which resulted in 33 bytes, of which 20 bytes was header. We have therefore removed another 13 bytes from the output of the PIZ-encoder to give it a fair comparison (in total we have removed 33 bytes). Note that the PIZ-codec is designed for large images where the cost of the Huffman table can be amortized over many pixels. It is also designed for natural images without discontinuities. However, we have included it here for completeness.

Using the PIZ-codec on 8×8 data gives no data reduction; compressed blocks are almost always bigger than 100% of the original data. Therefore, 16×16 blocks have been used for the PIZ-codec. We have also doubled the size of the color buffer cache in the PIZ-codec case so that a full block can fit. For the proposed method, we have instead halved the cache size, so that both methods can store exactly one block in the cache, to avoid that any possible cache thrashing influences the comparison.

All scenes have been rendered twice in the 640×480 resolution, first with the proposed 8×8 algorithm and then with the 16×16 algorithm based on the PIZ-codec. Bandwidth usage is reported in the table below:

	Proposed	PIZ	factor
Ocean	2.82 MB	5.56 MB	2.0×
Shadow	5.38 MB	13.66 MB	2.5×
Reflections	8.28 MB	28.18 MB	3.4×

This looks favorable for the proposed algorithm, with more than a factor of two difference on average. However, it should be noted that moving from 8×8 to 16×16 tiles in itself increases bandwidth usage even if no compression is used, since many pixels that are never used will be read/written. Therefore, we have also compared the bandwidth usage against uncompressed 8×8 and 16×16 rendering:

	Proposed as % of original 8×8	PIZ as % of original 16×16	factor
Ocean	46%	84%	1.8×
Shadow	40%	68%	1.7×
Reflect.	43%	71%	1.7×

Thus, the improvement ratio is reduced down to around $1.7\times$. Hence, even without the bandwidth gains of an 8×8 system, there is still a substantial compression advantage of the proposed algorithm compared to PIZ.

At first it may seem a bit counter-intuitive that the difference between the two algorithms is so large. One important factor here is that many blocks generated during rendering are not completely filled by a single triangle, but have pieces of the background or pieces of another triangle in the block. This yields significantly better compression for the proposed method as the following simple test will show:

When compressing blocks that are 100% full of image data (i.e., no background), the two algorithms are very similar to each other in terms of compression ratio. However, when turning several pixels to the background color (e.g., black), we

see a clear difference in the behavior of the two systems. If the majority of the pixels are black, it is easier to code the block for both systems. However, for the 25% mode to kick in for the PIZ-encoder, only about 8 out of 256 pixels can be non-black, whereas for the proposed compressor, about 20 out of 64 pixels can be non-black. This means that the 25% mode almost never kicks in for the PIZ-encoder, which we have also seen in the simulation results. At the same time, it is no surprise that the proposed method is better at these discontinuities, since it was designed to handle exactly this common case.

For this reason, the PIZ-codec works better with block sizes of 50% and 75% instead of 25% and 50%. (Block sizes of 75% and 50% are just as burst-friendly as 25% and 50%.) As an example, this lowers bandwidth for the *Shadow* scene from 68% to 63% of the original. Hence we use 50% and 75% blocks for the PIZ-codec in Figure 7. It is likely that performance would increase further if these block sizes were optimized, but we have not performed this optimization neither for the proposed algorithm nor for the PIZ-encoder. It is also clear that introducing more possible block sizes would benefit both algorithms. Therefore, to make sure that the relative superiority of the proposed algorithm is not only due to bad choices of block sizes, we have made a final test where all block sizes are allowed, which gives the rate equivalent to allowing full variable bit lengths. In this comparison we have also used 16×16 tiles for both methods (by storing four 8×8-blocks together for the proposed encoder) to remove any possible dependencies on block size. This yields the following results:

	proposed 16×16 as % of original BW (any block size)	PIZ 16×16 as % of original BW (any block size)
Ocean	33%	57%
Shadow	28%	39%
Reflec.	28%	46%

We see that our algorithm is better than the PIZ-codec by an average factor of $1.6\times$, which is substantial. Note further that 8×8 blocks are much to prefer over 16×16 blocks, not only because of the extra bandwidth usage associated with 16×16 blocks as described above, but also since a larger number of pixels with data dependencies take more clock cycles to compress/decompress. This is the reason why we use four 4×4 blocks inside the 8×8 block—these can be compressed in parallel given enough hardware resources. We have also tried an 8×8 version of our algorithm, which reduces the bandwidth by another four percentage units, but this does not give the same support for parallel hardware compression. So for fp color *buffer* compression, we have shown that our proposed algorithm performs better than OpenEXR's PIZ algorithm. In addition, we want to mention that PIZ also needs to create a unique Huffman table per tile when compressing a tile, and this is expected to be rather expensive.

Next, we evaluate our entire architecture with the new compression algorithms, unified cache, and unified codecs. Table 1 shows the bandwidth figures for the the three scenes rendered at 1024×768 for the different configurations. For the

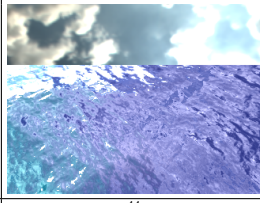
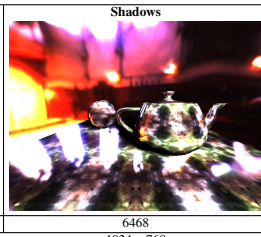
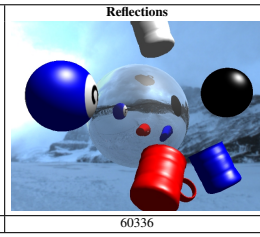
Scene	**Water**				**Shadows**				**Reflections**			
												
# triangles	44				6468				60336			
	1024×768											
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
Color BW	15.0	6.4	6.4	6.5	30.0	11.8	11.8	11.1	32.6	14.1	14.1	10.4
Depth BW	1.1	1.1	1.1	1.1	3.4	3.4	3.4	2.7	14.0	14.0	14.0	9.6
Shadow/Cube	n/a	n/a	n/a	n/a	5.9	5.9	2.0	1.5	52.3	52.3	24.4	21.9
Texture BW	17.0	17.0	17.0	16.1	12.5	12.5	12.5	11.6	7.3	7.3	7.3	7.6
Total BW	33.1	24.5	24.5	23.7	51.8	33.6	29.6	27.0	106.2	87.7	59.7	49.4
BW ratio	**100 %**	**74.1%**	**74.1%**	**71.5%**	**100 %**	**64.8%**	**57.2%**	**52.0%**	**100 %**	**82.5%**	**56.2%**	**46.5%**
	320×240											
BW ratio	**100 %**	**83.9%**	**83.9%**	**77.4%**	**100 %**	**78.9%**	**56.5%**	**48.8%**	**100 %**	**95.2%**	**59.8%**	**50.1%**
	1600×1200											
BW ratio	**100 %**	**69.6%**	**69.6%**	**68.1%**	**100 %**	**59.8%**	**54.6%**	**50.9%**	**100 %**	**75.5%**	**54.4%**	**47.2%**

Table 1: Performance figures in MB/frame for our three test scenes. Configuration *A* is a traditional architecture, with depth compression. In the *Shadows* scene, depth compression is turned off during creation of the shadow map. Configuration *B* is equal to *A*, but with fp color buffer compression turned on. Configuration *C* uses unified codecs, i.e., render-to-texture textures can be created and rendered from in compressed form. Configuration *D* shows results for a unified cache architecture. Note how color BW goes down substantially between *A* and *B*, and how overall bandwidth is reduced significantly for *C* and *D*.

Water scene, configuration *A* through *D* are as described above. Since there is no render-to-texture taking place, *B* and *C* are in fact equivalent.

For the *Shadows* scene, configuration *A* and *B* are slightly different in that we do not to compress the depth buffer during the creation of the shadow map. This is due to the fact that we want to be able to render from it later, and the texturing unit in a traditional architecture cannot read data compressed with the depth compression method. An alternative would be to create the shadow map with compression turned on, and then uncompress it when it is moved to a texture, but we found this to use slightly more bandwidth for this scene. The final depth buffer is compressed, however. In the *Reflections* scene, an HDR environment map is rendered as background, and a dynamic HDR cube map is created every frame, so that a reflection can be rendered in the sphere in the center. The numbers for depth buffer bandwidth include rendering from the camera as well as to the six faces of the cube for the environment.

Comparing columns *A* and *B* in the BW ratio row, we see that substantial gains are made over a traditional architecture just by adding the floating point color buffer compression. To ensure pixel exactness, we have used the int24 version of the depth buffer compression. Depth buffer bandwidth would be decreased by about 14% further if fp16 were used. Column C uses the proposed unified codec architecture, and we see that bandwidth is further lowered to about 60% of the

original. The reason for this is that the shadow map (*Shadows* scene) and cube map (*Reflections* scene) textures can be created and rendered from with compression turned on. Finally, on a system with a unified cache for all the buffers, bandwidth can be reduced down to below 50% for the *Reflections* scene. Detailed figures are provided for the 1024×768 resolution. For 320×240 and 1600×1200, we have presented BW ratio figures showing similar results.

5 Discussion

A unified codec architecture is certainly simpler if most codecs are harmonized. We have presented two such algorithms but it would be interesting to have more. Fp32 color and depth buffers could be compressed with the proposed methods, although we have not tried. The 8-bit color buffer by Rasmusson [19] could be made even more similar to our work. We have even tried naive adaptations of our fp16 color for vertex data and (lossy) texture compression, with premature but promising results. That said, full codec harmonization remains a hard goal. For instance, S3TC/DXTC is hard to replace for RGBA8 textures.

Although a unified cache architecture seems competitive in this evaluation, it should be noted that there may be potential implementation problems that are not discovered at this level of simulation. There is no way of finding out short of actually implementing a unified cache, which is out of the scope of this paper.

Even though some measures have been taken to lower compression/decompression latency in this paper (such as using four 4×4 tiles instead of one 8×8), a proper latency analysis would require more detailed work.

6 Conclusion

We have proposed two new algorithms for fp buffer compression, one for color and one for depth data, which are the first published algorithms on these topics. An int24 version of the depth codec has been shown to be competitive against state-of-the-art. Finally, we have demonstrated how these algorithms behave in architectures with varying degrees of unification, reaching bandwidth reductions down to 50%.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vinnova.

Bibliography

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

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

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

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

[5] 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, 1997.

[6] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.

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

[8] T. Inada and M. D. McCool. Compressed Lossless Texture Representation and Caching. In *Graphics Hardware*, pages 111–120, 2006.

[9] Emmett Kilgariff and Randima Fernando. The GeForce 6 Series GPU Architecture. In *GPU Gems 2*, pages 471–491. Addison-Wesley, 2005.

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

[11] Eugene Lapidous and Goufang Jiao. Optimal Depth Buffer for Low-Cost Graphics Hardware. In *Graphics Hardware*, pages 67–73, 1999.

[12] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics,*, 12(5):1245–1250, 2006.

[13] Dan McCabe and John Brothers. DirectX 6 Texture Map Compression. *Game Developer Magazine*, 5(8):42–46, 1998.

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

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

[16] NVIDIA. GeForce 8800 GPU Architecture Overview. Technical report, TB-02787-001_v01, 2006.

[17] OpenEXR. www.openexr.com/about.html. web site, 2008.

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

[19] Jim Rasmusson, Jon Hasselgren, and Tomas Akenine-Möller. Exact and Error-bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware*, pages 41–48, 2007.

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

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

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

[23] M. J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *In Data Compression Conference*, pages 140–149, 1996.

Paper III

Error-bounded Lossy Compression of Floating-Point Color Buffers using Quadtree Decomposition

Jim Rasmusson[‡†] Jacob Ström[†] Tomas Akenine-Möller[‡]

[†]Ericsson Research [‡]Lund University

{jim|tam}@cs.lth.se jacob.strom@ericsson.com

ABSTRACT

In this paper, we present a new color buffer compression algorithm for floating-point buffers. It can operate in either an approximate (lossy) mode or in an exact (lossless) mode. The approximate mode is error-bounded and the amount of introduced accumulated error is controlled via a few parameters. The core of the algorithm lies in an efficient representation and color space transform, followed by a hierarchical quadtree decomposition, and then hierarchical prediction & Golomb-Rice encoding. We believe this is the first lossy compression algorithm for floating-point buffers, and our results indicate significantly reduced color buffer bandwidths and negligible visible artifacts.

The Visual Computer Journal, June 4th, 2009.

1 Introduction

Rendering is essentially an *approximative* process, where an algorithm attempts to compute an image, which may resemble a photograph, for example. The algorithm may use different techniques that simulate how photons interact with an environment, and this simulation already includes a certain measure of approximation. In addition, since computers are used, there will be small approximation errors due to that floating-point numbers are used in the calculations, and at the end of the rendering pipeline, these floating-point values are often quantized to eight bits per component. Over the years, a number of *lossy* (i.e., approximative) texture compression techniques have been developed and broadly adopted in the real-time graphics industry. In addition, all precomputed radiance transfer algorithms [7] use some basis, e.g., spherical harmonics or wavelets, to approximately represent functions in. Furthermore, Monte Carlo rendering techniques use sampling to approximatively evaluate some kind of integral. These examples make it clear that approximative algorithms can be and are being used successfully in rendering.

Lossy techniques have also been used for color buffer compression and decompression [13]. However, that work targeted only low dynamic range (LDR) buffers, where each color component only has eight bits. More and more rendering is done directly to floating-point buffers, which can contain high dynamic range (HDR) content, and still, we have not seen any color buffer compression/decompression algorithms that are *lossy* (approximative) for such data.

Therefore, we present a new algorithm for lossy compression and decompression for floating-point color buffers. Our goals include a system where the introduced errors are kept under strict control, with substantially lower memory bandwidth usage, high quality of the rendered images, and reasonably low implementation complexity. This new algorithm builds on the work presented by Ström et al. [19] and by Rasmusson et al. [13] but contains several new additions each which significantly contributes to the quality and performance of the total buffer compression/decompression system:

1. The hierarchical quadtree decomposition has been extended with an adaptive flatness test during the hierarchical subdivision, which provides consistent introduced error per pixel. In the same spirit, we have developed a quantizer which adapts to the hierarchy level. These two techniques significantly reduce block artifacts.

2. The combination of using a decorrelating and reversible color transform together with an integer representation gives us the possibility to compress both lossily and losslessly in the same framework, which is important since it reduces the complexity of the system (if one want to support both lossy and lossless compression).

3. Hierarchical prediction.

4. Adaptive error thresholding: Detection of when high enough accumulated

error has been reached, and in this case, subsequent compression is made to compress less aggressively, which improves quality and performance.

5. Constant bit rate mode, which can be used for HDR texture compression.

Next, we review previous work.

2 Previous Work

In this section, we present the most relevant work and their relations to our new algorithms.

Texture compression (TC) is a technique that is heavily used in real-time graphics today. This line of research started in 1996 [1, 6, 22], and since then, two major algorithms have been adopted for implementation in graphics hardware and supported by the APIs. S3TC (called DXTC in DirectX) is a collection of formats targeting both OpenGL and DirectX, while ETC [17] is used in OpenGL ES. The majority of these schemes compress to a fixed rate per block of pixels in order to simplify random access. Both S3TC and ETC use 4×4 pixel blocks and compress down to four bits per pixel (bpp). These algorithms are therefore, by design, *lossy*. It should be noted that most algorithms for TC are highly asymmetric, which in this case means that compression often can be done offline once, and it is only the decompression process, which is supported in hardware, that needs to be fast and of low complexity. The techniques above operate on low dynamic range (LDR) data only.

To support high dynamic range (HDR) textures, where each color component may be represented using a floating-point number (e.g., using 16 bits), specialized algorithms for HDR texture compression were introduced by Munkberg et al. [11] and Roimela et al. [16]. These also operate on 4×4 pixel blocks, and compress down to eight bpp. Sun et al. [20] and Wang et al. [23] take a different approach and implement an HDR TC algorithm using existing hardware for S3TC. This is in contrast to the other HDR TC algorithms which need new hardware for efficient decompression. There have also been some recent developments to the first algorithms, and both of these increase the image quality [12, 15].

Textures are mostly read-only, and therefore it works well with asymmetric compression/decompression algorithms where compression is slower than decompression. However, the many buffers (e.g., color, depth, stencil, etc) in a real-time rendering pipeline must be treated differently for a variety of reasons. First, during the rendering of a triangle, compression and decompression may be applied several times, and hence these algorithms must be more symmetric. This also means that both compression and decompression must be supported by the hardware. See the surveys on both color buffer compression [13] and depth buffer compression [5]. In many cases, the user may need a buffer result which is lossless, i.e., exact. However, as argued in the introduction, approximate (lossy) algorithms is an interesting option. The major reason is that they offer considerably higher

compression ratios, and we note again that lossy techniques are already in use in several different places in the pipeline. Rasmusson et al. [13] propose to use a lossy buffer compression / decompression technique where the introduced error is kept under control. When the error grows too large, the method reverts to using lossless (exact) compression or no compression. That algorithm operated only on LDR data. Recently, a lossless algorithm for color and depth buffer compression of floating-point data has been proposed by Ström et al. [19]. Similarly to the other buffer compression algorithms, this technique is also block-based to provide random access, and there is a fall-back to using no compression in order to be able to guarantee an exact result at all times. However, that work does not investigate approaches for *lossy* compression/decompression of *floating-point buffers*, which is the topic of the research presented in this paper.

Note that in principle, one can use existing lossy floating-point image compression algorithms, such as the B44A codec included in OpenEXR [2], for lossy buffer compression. The B44A codec is based on delta modulation and works in a way similar to the HDR-texture compression method by Roimela et al. [16], and we have included it in our results section in order to benchmark our algorithm.

3 New Color Buffer Compression Algorithm

In this section, we present a new color buffer compression method. The algorithm operates on tiles, which are typically 8×8 pixels. It is designed to compress 16-bit floating-point (fp16) high dynamic range (HDR) color buffers, although adapting the algorithms to 32-bit floaint-point numbers is straightforward. All operations related to the compression algorithm are done in the integer domain which lowers the complexity down to a level where it is within reach also for a mobile phone implementation. Note that while the bandwidth is significantly reduced due to the compression, the amount of storage in external RAM is not affected. This is due to that the algorithm uses variable bit rate *and* have an uncompressed mode as fallback, which is used for tiles that cannot be compressed using implemented algorithms (or if the maximum allowed amount of error is reached for this tile). See the papers by Hasselgren & Akenine-Möller [5] and Rasmusson et al. [13] for detailed descriptions on how buffer compression/decompression systems work.

The same implementation can be configured to operate in 8-bit integer low dynamic range (LDR) mode or in 16-bit floating-point (fp16) high dynamic range (HDR) mode. This is possible since we reinterpret the floating-point numbers as integers and do all operations related to the compression algorithm in the integer domain. It can be configured to operate in an approximate (lossy) or exact (lossless) mode with fine-grained control to go from one mode to the other. The approximate mode has mechanisms to automatically go from lossy to lossless mode when certain error thresholds have been reached. This effectively bounds the introduced errors to configurable maximum levels.

For each *tile*, a high level description of our algorithm is as follows:

1. Reinterpret the bit-pattern of the floating-point number of each color component as an integer, and transform (losslessly) from *RGB* into the decorrelated YC_oC_g color space.
2. Perform hierarchial quadtree decomposition.
3. Perform hierarchical prediction, quantization, and Golomb-Rice encoding (adaptive).

In addition, we have an error-bounded control mechanism to keep the introduced approximations under a user-defined threshold.

Our new method builds upon the color buffer compression methods by Ström et al. [19] and by Rasmusson et al. [13]. The color buffer is divided into 8×8 pixel tiles, and we also use a simplistic method to handle destination alpha, which is assumed to be 1.0 in the compression stage. If destination alpha $\neq 1.0$, or if any fp16 value is negative for any pixel in the tile, we simply revert to uncompressed mode. Alternative ways of handling alpha is to let the new compression algorithm presented in this paper also handle the alpha component, or let any existing scheme, such as DXTC or table-based methods [18], handle alpha compression for tiles with alphas between 0.0 and 1.0. At this point, we have omitted such studies since it would not advance the research field much. In addition, we leave the investigation of negative floats for future work, mostly due to the difficulty for us to find reasonable test scenes with negative values.

In the following, we describe the steps (1–3 above) in detail.

3.1 Representation and Color Space Transform

Without an excessive numbers of bits, it is impossible to represent the difference between two arbitrary floating-point numbers in a lossless manner. For positive floating-point numbers, we can circumvent this problem by taking the bit pattern of the floating-point number and interpreting them as an integer [8, 19].

After R, G, and B for each pixel in a tile have been reinterpreted as integers, we convert these into a luminance/chrominance color space. Besides decorrelation of the RGB channels, this enables the possibility to have different compression settings for the luminance and the chrominance components, respectively. For example, in a lossy mode, we typically compress the chrominance components more aggressively than the luminance components. This will introduce higher error levels in the chrominance components, but since the human visual system is less susceptible to chrominance errors than luminance errors, the resulting visual artifacts are less visible. Introducing higher error levels in the chrominance than the luminance components are common in most image and video processing methods, such as those in JPEG and MPEG.

The color space transform we use is the YC_oC_g-transform [9], which has some very attractive properties. It has low implementation complexity (highly silicon-efficient), and provides good decorrelation which in general reduces bandwidth.

Note that since we operate on pixel component values that are remapped from floating-point to integer, the behavior of this transform will not be the same as the original YC_oC_g-transform [9]. We have not done any extensive analysis of how the decorrelation properties change after the remapping operation. However, we have observed that it brings performance improvements on par with what it brings in the LDR domain (about 10% bandwidth reduction).

Another property important for us is that this transform is reversible; transforming the YC_oC_g-values back to *RGB* recreates the *RGB* values bit-exactly (that is, if the YC_oC_g-values have not been altered). This is a must for our lossless mode.

Transforming from *RGB* to YC_oC_g and back is done as follows. Note that the R, G, and B numbers are 15-bit integers (the sign bit is assumed to be zero), and all operations work on integers.

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

The reverse transforming is as simple:

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

The Y-component has 15 bits, and the C_o and C_g-components have 16 bits respectively. The color space transform has been used for LDR color buffer compression before [13], and the reinterpretation of floats as integers has been used before as well [8, 19]. However, for us, it is crucial to use a luminance and chrominance divided color space since it allows us to compress the chrominances stronger than the luminance. The decorrelation also improves compression performance (in general around 10% bandwidth reduction). Furthermore we must be able to support both lossless and lossy compression of floating-point data. The use of this combination is a small, albeit very useful and practical contribution in this context.

3.2 Hierarchical Quadtree Decomposition

The next stage is a quadtree decomposition of the transformed tile data. This creates a hierarchical tree structure of the 8×8 pixels, where "flat" (homogeneous) regions are sub-sampled and thus represented in higher levels in the tree, and using fewer samples than one per pixel. As an example, only one sample per 4×4 sub-tile may be used if that sub-tile is flat enough. For an 8×8 pixel tile, there are at most four levels, as can be seen in Figure 1.

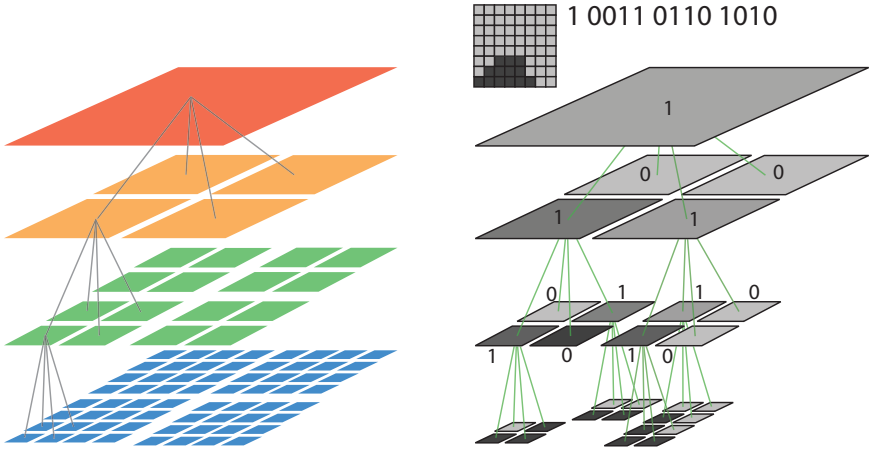


Figure 1: Left: a full hierarchical quadtree decomposition of an 8×8 pixel tile. The bottom level has full pixel resolution (8×8 pixels), and each level above is a subsampled version of the level below. Right: in this example, we show the hierarchical decomposition of the 8×8 pixels (light and dark gray) at the top. Note that if a subtile has constant color, the representation ends at that level, and that is the reason why the tree is not full. Note also that the associated 13-bit tree code is shown at the top, where 1's means that the subtile has children, and 0's means that the node is a leaf. Thus, the first bit (1) indicates that the root node (top) has children, and the first two of these children do not have children of their own (00), while the remaining two has (11), and so on.

However, it is important to avoid sub-sampling when the sub-tile region is not flat, e.g., when there are edges, as this can cause visible artifacts. We use a *flatness test*, where we first calculate the average of the 2×2 pixels in the sub-tile. If the absolute difference between each pixel value and the average is below a configurable "flatness" threshold, τ_{flat}, we assume it is reasonable to use subsampling. This is done for all pixels in the tile in a bottom-up fashion. We start with the individual pixels and attempt to sub-sample each 2×2 pixel sub-tile, and then move up in the hierarchy until all four levels have been tested. See Figure 1. If the entire 8×8 pixel tile consists of a really flat region, a single value is sufficient to represent the entire tile. This case actually occurs surprisingly often for the chrominance components, C_o and C_g. Computer generated content often decorrelate well by the YC_oC_g-transform, and this results in a Y-channel carrying most of the information, while the chrominances are more uniform (flat).

Quadtree decomposition is a well known image and graphics processing technique that yields an efficient and compact data structure. It has often been in use for image compression [24, 21]. However, to the best of our knowledge, we have not seen this being used in any buffer compression and decompression algorithms.

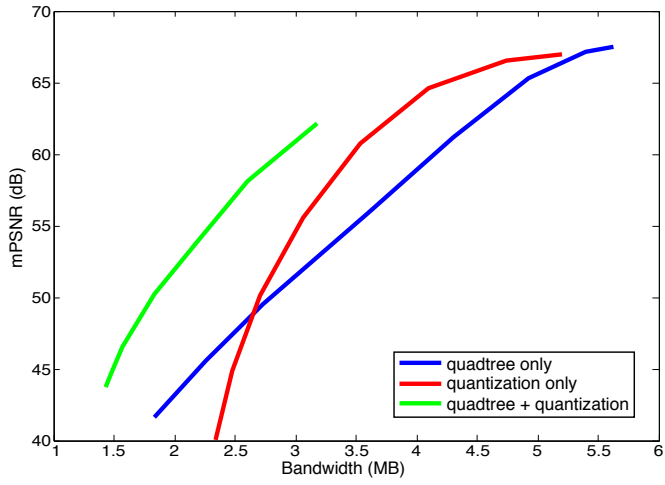


Figure 2: Rate distortion plots of the two lossy stages in the algorithm; the quadtree stage and the quantization stage. The two rightmost curves are showing the two stages working in isolation, and the leftmost curve shows the combined effect. Note that the quadtree curve's slope is flatter while the quantization curve's slope is steeper when moving towards lower bandwidths. These results were made using a near exhaustive parameter search of the Shadows scene (see Table 1) at 1024×768 resolution. Note that our algorithm uses both the quadtree and quantization.

In Figure 2, the contribution of the quadtree stage is shown in a rate-distortion plot. As can be seen, the addition of the quadtree mechanism to the algorithm brings a significant improvement in terms of reducing bandwidth while keeping the distortion reasonably low. This clearly motivates inclusion of quadtree decomposition in our system.

An associated binary tree code describes the structure of the tree in a compact way. For an 8×8 tile, the tree code varies between 1 and 21 bits. A full tree (where all nodes have four children) uses 21 bits (1+4+16), and the smallest tree uses a single bit (i.e., indicating that the 8×8 pixel tile is sub-sampled to one value), as illustrated to the right in Figure 1. The tree code is stored in the compressed data.

Note that the flatness threshold, τ_{flat}, is lower the higher up in the hierarchy the current subtile is at. That is, in order to sub-sample a region covering more pixels, "the more flat" it has to be, if the introduced errors per pixel are to be consistent. This also reduces block artifacts. Since a pixel in level l covers four pixels in level $l-1$, we divide τ_{flat} by four (by right-shifting two steps) each time we move up to a higher level. Note also that the flatness threshold for the Y-components can be lower than the corresponding thresholds for the C_o and C_g-components. Again, the reason for this is that the human visual system is more susceptible to errors in the luminance components. Also, as mentioned before, most details are often present

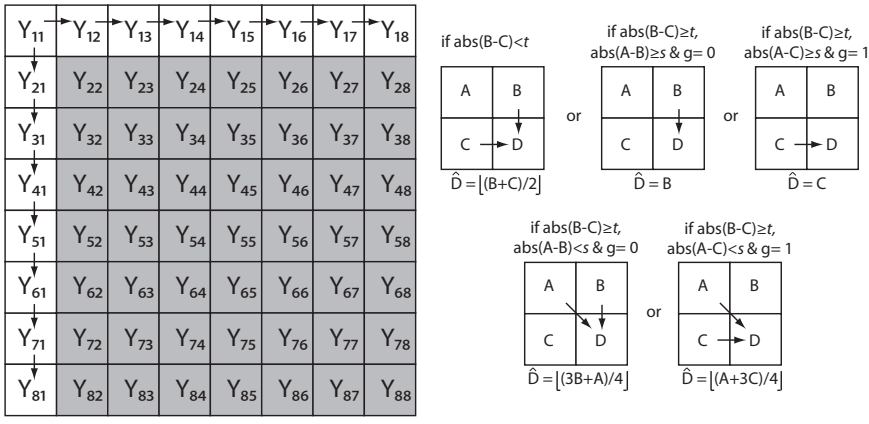


Figure 3: Arrows indicate prediction. For instance, Y_{12} is used to predict Y_{13}. Pixels marked in gray can be predicted from the pixels to the left, to the top, and/or top-left neighbors, as shown to the right. The correct D-value is predicted as $\hat{D}$ from pixels A, B, and C. See also Equation 4.

in the luminance component. For lossless compression, we simply set $\tau_{\text{flat}} = 0$, and the values in the sub-tile regions have to be exactly the same in order to be sub-sampled. This happens sufficiently often to justify the decomposition stage to be active also in our lossless mode. The flatness threshold is also dependent on the amount of accumulated error present from earlier approximations in the tile. See Section 4 for more information on the error control.

3.3 Prediction, Quantization, and Encoding

The next few steps are prediction, quantization, and finally encoding. These are presented together since they often are tightly connected in an implementation.

Prediction

For prediction, we use a modified version of the novel predictor used for color buffer compression [19]. A useful feature of this predictor is that it avoids prediction across edges. This is of particular importance during the rendering of computer generated content where sharp edged primitives (e.g. triangles) are drawn on top of other primitives or background images. Hence, the tiles that are to be compressed often contain high intensity transitions due to these edges. The predictor traverses the pixels in the tile from left to right, and in top to bottom order. See Figure 3. If the difference between pixels B and C is big enough, it is assumed that an edge has been found, and an extra guide bit, g, is used to indicate whether to predict from B or C. Otherwise, the average of B and C is used. For the gray pixels

in Figure 3, the predictor [19] is summarized below:

$$\hat{D} = \begin{cases} \lfloor \frac{1}{2}(B+C) \rfloor, & \text{if } |B-C| < t \\ B, & \text{if } |B-C| \geq t, \text{and } g = 0 \\ C, & \text{if } |B-C| \geq t, \text{and } g = 1, \end{cases} \tag{3}$$

where $\lfloor \cdot \rfloor$ denotes rounding to the nearest lower integer, and t is a threshold value. In our experiments, we have found that the following modification improves performance slightly:

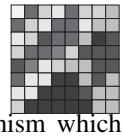
$$\hat{D} = \begin{cases} \lfloor \frac{1}{2}(B+C) \rfloor, & \text{if } |B-C| < t \\ B, & \text{if } |B-C| \geq t, |A-B| \geq s \,\&\, g = 0 \\ C, & \text{if } |B-C| \geq t, |A-C| \geq s \,\&\, g = 1 \\ \\ \lfloor \frac{3}{4}B + \frac{1}{4}A \rfloor, & \text{if } |B-C| \geq t, |A-B| < s \,\&\, g = 0 \\ \lfloor \frac{1}{4}A + \frac{3}{4}C \rfloor, & \text{if } |B-C| \geq t, |A-C| < s \,\&\, g = 1. \end{cases} \tag{4}$$

In the new predictor in Equation 4, we added an extra test to check whether the information in pixel A is useful in the predicton. If A is similar enough to B and C, respectively, we include A when calculating $\hat{D}$. We have found that $s = 512$ works well in practice. Note that the multiplications by $1/4$ and $3/4$ are done using shifts and adds in order to reduce complexity. The threshold t is set to 2048 (same as Ström et al. [19]).

We traverse the pixels left-to-right and top-to-bottom. The first upper left corner pixel is always stored separately in uncompressed form.

To indicate where a new edge is first encountered, Ström et al. [19] use a restart value, which consists of a pixel position inside the tile, and the value at that pixel. The idea is that the first time a very different value occurs, it should be possible to state that explicitly. For the rest of the pixels, it should be possible to predict either from this new value or from the previous values.

However, sometimes a block may contain more than two sets of very different values and a single restart value is not enough, as illustrated in the figure to the right.



To mitigate this situation, Ström et al. [19] have a rotation mechanism which changes the pixel traversal order, and checks if rotating the sub-tile yields a more favorable situation for the predictor. While the rotation helps, there are cases where it does not help and you end up predicting across edges, resulting in large "jumps" in the stream of residuals. This degrades the performance of the subsequent Golomb-Rice encoding.

We take a different approach to the use of restart values. While Ström et al. use a single restart value per tile and perform an exhaustive search [19] to find the best position for it in the tile, we propose to allow as many restart values as it takes to accurately represent an edge (or several edges) inside a tile. The restart values are also detected on-the-fly in the predictor and thereby we avoid the expensive ex-

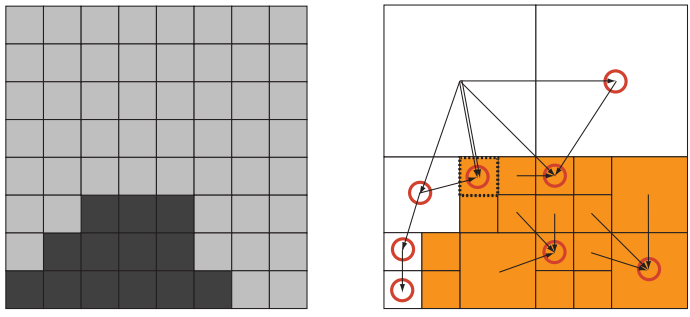


Figure 4: Illustration of a few prediction cases using the example tile from Figure 1. Note that the sub sampled values are used for prediction. The number of predictions are defined by the quad tree. Arrows indicate prediction (only a few cases, marked with red circles are illustrated). Pixels marked in orange use the three neighbor pixels (left, top, and/or top-left) and white pixels use only one (either left or top). For some pixels, two out of the three neighbors are the same (e.g. the dotted pixel). See also Equation 4.

haustive search method of Ström et al. [19]. Furthermore we also skip the rotation as it requires an extra encode session (to test if the rotated tile is better).

In order to find the restart values, we have added an extra test in the predictor that checks if the predicted value and the current value differs with more than τ_{restart} (another threshold value), i.e., $|D - \hat{D}| \geq \tau_{\text{restart}}$. In practice, we have found that $\tau_{\text{restart}} = 8192$ works well here. If this is the case, we generate a new restart value.

These restart values are basically random values, and so there is no point in using a predictor on these. Furthermore, the distribution of restart values typically does not suit the Golomb-Rice coder, and hence we code and store them separately. Pruning the restart values from the stream of residuals also has the added benefit that the Golomb-Rice coder performs well with a single divisor for all residuals, see Section 3.3.

Note that we use the subsampled pixel values when calculating the prediction values. Due to the adaptive sub-sampling scheme during the quadtree creation, a varying number of pixels needs to be predicted. See Figure 4.

The predictor also needs to handle the odd cases where the top-left pixel, due to the sub-sampling scheme, is the same as the either the left or the top pixel. The dotted pixel in Figure 4 is an example of such a case. After subtracting the predicted values from the input values (subsampled), we have a number of residual values. A last important thing to realize is that the predictor on the encoding side must behave in the same way as on the decoding side. For this reason, the pixels are at all times modified to the decoded (lossy) values while predicting. This means that, in the predictor, we quantize the residual, store the value for further entropy encoding and then do an inverse quantization, to get the decoded value back.

The quantizer and the entropy coding scheme are described next.

Quantization

In the lossy mode, the residual values are quantized. We use a uniform quantizer where the quantization step is dependent on which sub-sampled area the current pixel is in. This means that the more flat the region, the more gentle quantization. This is done to reduce the amount of introduced errors with the motivation that these sub-sampled values represent more pixels, and therefore the amount of quantization should be reduced in order to be consistent. Next, the details of our quantization are described.

We first compute the quantization step as: $q_{\text{step}} = (q_{\text{level}} \gg l) \gg q_{\text{errorweight}}$, where $\gg$ denotes right shift, and q_{level} is the basic quantization level set by the user—the higher value, the more loss in the compression. To reduce block artifacts, the subsampling level, l, reduces the amount of quantization the higher up in the hierarchy the sub-tile is located, and the right shift by l should correspond to a division by the number of pixels covered by that subtile. For example, for the top level (which covers 8×8 pixels), the quantization level should be divided by 64, i.e., $l = 6$. More generally, assume there are $2^p \times 2^p$ pixels in the current subtile. The subsampling level is then computed as $l = 2p$. The $q_{\text{errorweight}}$ parameter is described in Section 4.1. Once q_{step} has been computed, the residual values, r, are quantized according to: $\hat{r} = r \gg q_{\text{step}}$. The quantized residual values, $\hat{r}$, are then subsequently entropy coded for further compression. In the following, we argue that it is reasonable to perform the quantization in the integer domain. Recall that the float-to-integer conversion is monotonic, and neighboring positive floats are converted to neighboring positive integers. Thus, converting a float to an integer, and quantizing the integer (ie., slightly perturbing the integer), is equivalent to a slight perturbation of the original floating-point value. Hence, it is clear that we can quantize in the integer domain, with expected results.

Encoding

We use the standard Golomb-Rice technique [4, 14] for entropy encoding. When the predictor generates residual values whose distribution has higher density around zero, good compression ratios can be expected since fewer bits are spent on smaller values in Golomb-Rice. A further advantage is that the implementation is of reasonably low complexity.

For clarity, we briefly recap how the Golomb-Rice code works, and refer to recent papers [13, 19] for a detailed explanation. Each residual value is divided by a divisor, 2^k, which creates a quotient and a remainder. The quotient is encoded using unary encoding and the remainder is coded using binary encoding. A zero is used to separate the two. We have found that a single divisor per tile yields the smallest bandwidth. This is possible due to that we store the first upper left corner value and the restart values separately, which makes the stream of residuals

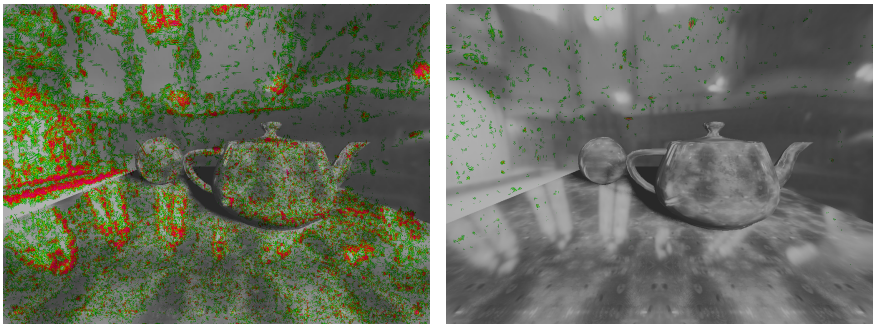


Figure 5: In these images, we visualize the error computed on the luminance channel using the HDR visual difference predictor (VDP). Green pixels are those where a human has a 75% chance of detecting the artifact, and red pixels have 95%. In the left image, *no* error control has been used, while in the right image the compression was error-bounded using our algorithm. All other parameters were the same. Clearly, there is a need for error control.

"pruned" and better suited for the Golomb-Rice coder.

To find the best k-value, we use a method [19], where the the bit position, p, of the most significant bit of the largest residual value is calculated. We then test the interval $[p-4, p]$ for the best k-value.

After this, the entire encoding is finished, and can be written over the bus to external memory.

4 Error Control

For *lossy* buffer compression, it is of utmost importance to have some sort of error control that prevents the accumulated error to grow out of reasonable proportions. The reason why this is needed is that compression can be applied many times to a tile (due to that many triangles can render to the same tile at different times), and if an error is introduced when the first triangle is rendered, then this error may grow when the second triangle is rendered, and the tile is compressed again. This is often called *tandem* compression. The difference between using and not using an error-bounded algorithm can be seen in Figure 5, where visible errors are visualized using the HDR visual difference predictor (VDP) [10].

We use an error control mechanism similar to the one proposed by Rasmusson et al. [13]. This algorithm gauges the introduced errors, and a decision is taken to use approximative compression if the newly introduced error and the previously accumulated error together are below a predetermined error threshold. If this condition is not met, then we revert to the lossless mode instead. The error metric we used is the mean square error (MSE) between the decoded RGB and the input

RGB values. See Rasmusson et al's paper for more details.

4.1 Graceful Parameter Adaptation

We now present an improvement of the previous error control mechanism. A simple observation here is that the compression algorithm should revert to lossless encoding as seldom as possible. To that end, we introduce *graceful parameter adaptation*, so that the higher the accumulated error is in the tile, the lower is the flatness threshold and also the quantization step. This allows us to use lossy compression more often and it allows us to use more aggressive compression for tiles that are compressed only a few times.

The net effect is that the newly introduced approximation errors will be smaller and smaller the closer to the error threshold we get. More importantly, the algorithm can continue to introduce small errors, without having to revert to the much more expensive (in terms of number of bytes) lossless mode. In our current implementation, we detect when the accumulated error level has reached 50% of the error threshold, and in those cases, we divide τ_{flat} by a factor of two and change $q_{\text{errorweight}}$ (used in Section 3.3) from 0 to 1 . This could be extended so that when the accumulated error has reached 75% of the error threshold, the division factor is four instead of two and set $q_{\text{errorweight}} = 2$, and so on. A further extension would be to use the full "$1/x$"-behavior.

In general, this mechanism gives an improvement of $0.5 - 2$ dB in mPSNR, while the bandwidth usage remains constant or is even reduced by up to three percent. However, for certain tiles the bandwidth may go up. Intuitively, this may happen when the τ_{flat} is lowered due to that we reach the 50% threshold, and then nothing more is rendered to that tile, which means that there was no use in lowering the threshold. It should be noted that for those tiles, the image quality is increased though.

For our lossy mode, we also quantize the restart pixel values (not the position) to 8 bits each, but when we reach the 50% error level, quantization is instead changed to 12 bits.

5 Implementation

Our algorithm evaluation (see Section 6) was done in a software-based simulation framework implementing a tiled rasterizer with a modern color buffer architecture, programmable shading, texture caching, Z-culling, and more. We have used a a color buffer cache of 1 kB, and 256 bits for a tile table cache to store the tile table bits that indicate which compression mode is used (uncompressed, tile clear, compression mode 1 or compression mode 2).

In order for us to better understand the introduced error levels, we have used mean-square (MSE) errors between the incoming (original) tile and the same tile being

encoded and decoded for the error control mechanism. This correlates well with established error metrics (e.g., mPSNR [11]) when we measure error on the resulting final image. However, for a hardware realization, methods of lower complexity could be developed. For example, error gauging can be made on-the-fly directly inside the quadtree decomposition stage and the predicton stage. Less expensive error metrics (in terms of complexity) such as sum-of-absolute-differences (SAD) may be used instead of MSE. This would need further analysis, and therefore, we leave this for future work.

While the properties of an algorithm, and its performance in terms of image quality and compression ratios can be evaluated using a software simulation, it is also important to investigate whether a hardware implementation is feasible. The different processing blocks have been designed with low complexity in mind. The YC_oC_g-transform consists solely of integer adders, subtracters and bit shifters and will incur a tiny cost in silicon. In order to reduce latency, many transform blocks can be executed in parallell. The quadtree decomposition method is also low complexity and highly parallelizable. In our predictor, a majority of the operations are bit shifters, adders and subtractors, and hence the complexity is relatively small.

We believe the total complexity of a combined compressor and decompressor will be low enough even for a silicon implementation in embedded battery-driven devices like mobile phones. This is supported in a Master thesis report by Caglar and Ojani [3] showing results from a silicon implementation of the lossless mode of the algorithm by Rasmusson et al. [13], including the rather complex variable bit rate parts. The resulting size of the compression and decompression blocks was well below 0.1 mm^2 in 65nm technology. That implementation operated on 8×8 pixel tiles, but only on 8-bit integer color (i.e., LDR), while we need to use 16-bit integers. Since mostly adders and shifts are used in our algorithm, we believe that an implementation in silicon will scale up in a decent way.So, while we have not implemented our algorithm in hardware exactly, the arguments above support that this will be feasible at a low cost considering that we are dealing with HDR colors.

6 Results

In Section 5, we describe the software simulation framework that we have used for algorithm evaluation. We emphasize the fact that the testing includes the full, incremental rasterization process of these scenes. This is in contrast to regular image compression, where only the final image is being compressed.

The test scenes are *Water*, *Shadows*, and *Reflections*, and all render targets are fp16 color buffers. To make certain that a large interval of the dynamic range is used, all scenes use fp16 texture maps and cube maps. In addition, the *Shadows* scene renders shadows using shadow mapping, and *Reflections* renders an fp16 cube map every frame. The sphere in the center of the scene is rendered using reflections with this dynamic cube map. Screenshots from these scenes can be seen in Table 1. Note that these screenshots have been tone mapped from fp16

RGBA down to 8 bit RGBA. This means that if a region in the screenshot appears to be simple to compress (e.g., a white or black area), this may not at all be so because the raw fp16 data can contain a lot of information even in those areas. Since we cannot know beforehand how the tone mapping operator works, we need to be able to compress even these areas with high quality.

In order to evaluate the error/quality of the final rendered images, we use HDR-VDP [10], logRGB, and mPSNR [11]. The HDR-VDP numbers presented in Table 1 are given after a manual adjustment using the multiply-lum command increasing the luminance level to 300 cd/m^2.

6.1 Contender Codecs for Benchmarking

We have compared our lossy compression algorithm against OpenEXR's B44A mode (described in Appendix 1), both in terms of compression performance and quality. B44A is a lossy compressor, and it is the one that we found most amenable for hardware implementation. We have modified the B44A algorithm to include our error control mechanism. In addition, as a benchmark for our lossless method, we have used a modified version of the method from Ström et al. [19]. The original method operates on 8×8-blocks, but divides this into four 4×4 blocks in order to enable parallel encoding. This has a negative impact on the compression efficiency, and in order to provide a fair comparison, we have implemented an 8×8 version. It works in the same way, but only one base value and one restart value per 8×8 block are now used. Note that the image error/quality measures are only relevant for the B44A lossy compressor and our lossy compressor.

6.2 Block Artifact Reduction

Since we are working with lossy compression of 8×8 pixel tiles, there is a risk for block artifacts. For this reason, we have designed block artifact reduction techniques in two places in the algorithm. The first place is in the quadtree decomposition stage where the flatness thresholds are decreased by a factor of four for each higher level up in the quadtree. In the same spirit, the quantization stage is designed to lower the quantization step and apply milder compression the more subsampled a residual value is. See Section 3 for more details. In Figure 6, the effect of the block artifact reduction methods are shown.

6.3 Target Memory System

Since we do not know the target memory system, it is hard to predict what burst sizes are better than others. The memory system of a graphics card for a PC is dramatically different compared to that of a mobile phone. For this reason, the bandwidths and associated compression ratios shown in Table 1 are given without burst size limitations. It should be noted that these are unrealistically high. In

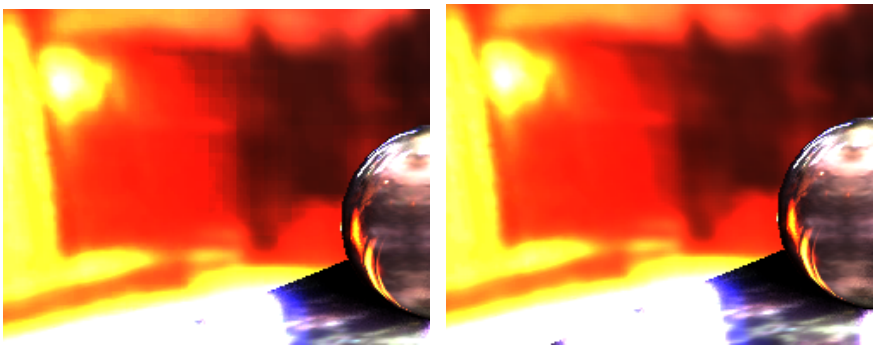


Figure 6: Crops from the shadow scene at 1024×768. Left: block artifact reduction methods disabled. Right: block artifact reduction methods enabled. In this case, mPSNR is about 4 dB higher with block artifact reduction enabled when the parameters are tuned for equal memory bandwidth usage. To clearly see the block artifacts in the left image, it may be necessary to zoom in the pdf.

order to present feasible burst sizes in a more memory system agnostic way, we use a histogram to show how often a particular "bit size bin" is used (vs bin sizes). See Figure 7.

In our tile table, we may use two bits to indicate compression mode (uncompressed, tile clear, and two different compression modes) or we may use three tilebits (uncompressed, clear and six compression modes). With detailed histogram data, it is possible to analyze what compression modes would be appropriate for a particular target memory system. If you have, for example, a mobile phone system, with a 32-bit data bus and mobile DDRAM, burst sizes of n times 256 bits (8×32 bit words) typically make sense. With three tile bits, you have 6 compression modes to choose from. Analyzing the histogram data will give the six best burst sizes. The resulting compression ratios for this example is shown in Figure 8 (shown together with the compression ratios without burst size limitations).

6.4 Performance and Image Quality Evaluation

Table 1 shows the color bandwidth figures and the image quality/error measures for the the three scenes rendered at 320×240 and 1024×768 resolutions. Note that we have tuned the thresholds of our algorithm so that the image quality/error measures are about the same for both the B44A (a lossy compressor) and our lossy compressor. As can be seen, our algorithm always outperforms the B44A algorithm. Our research in this paper has not focused on lossless compression, but our compressor can be configured as a lossless compressor by setting $\tau_{\text{flat}} = 0$, and the error threshold to zero as well. This was done for the columns **C** in Table 1, and as can be seen, our results are about the same as the lossless compressor presented by

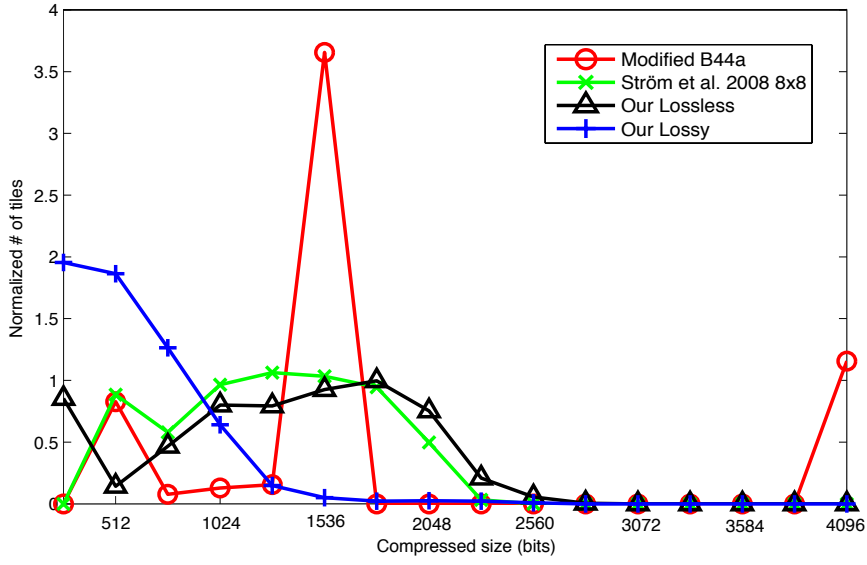


Figure 7: A normalized histogram of the number of tiles that are compressed to a given size (256 bits bins), using different algorithms. We use 8×8 pixel tiles, which means that 4096 bits indicate uncompressed tiles. The histogram is based on an average over all our test scenes.

Ström et al. [19]. It can be seen that the image error/quality measures of the rendered images have high quality overall, and hence, we believe that our algorithms make a significant contribution, since the compression factors are also rather high.

6.5 Comparison with H.264

In addition to comparing to B44A, we have also included a comparison against H.264. Although we guess that H.264 is too complex to be used for color buffer compression, we think that it is interesting to see how our codec fares against a highly optimized state-of-the-art method. For this comparison, we had to move to 16×16 rasterization (see Appendix B for details), and therefore we believe that these results are best presented in isolation. The result of the comparison can be seen in Table 2, which is an average for the three scenes rendered at 320×240. As can be seen, our 16×16 algorithm uses slightly less bandwidth and the quality is slightly better. On the face of it, it may seem as if the two 16×16 methods are more or less on par. However, the H.264 codec enjoys two advantages: First and most importantly, our 16×16 method is not allowed to predict across the 8×8 pixel boundaries within the 16×16 blocks (see Appendix B). This turned out very difficult to disable in the H.264 codec, and was thus allowed for the H.264 codec. Such prediction is highly valuable, and is generally regarded to be one of the major reasons why H.264 works so well not only as a video codec but also

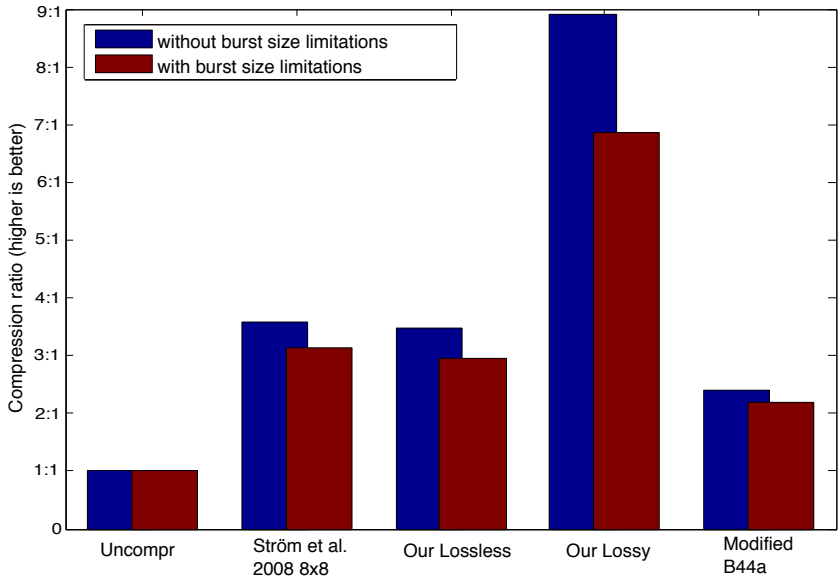


Figure 8: Compression ratios with an optimal memory system (blue bars) and a more realistic example memory system (red bars) using a minimum burst size of 256 bits, with three tile bits (uncompressed, cleared, 256, 1024, 1536, 1792, 2048, 2560 bits). The diagram is based on an average over all our test scenes.

as a still-image codec. Second, the employed H.264 encoder uses rate-distortion optimization, effectively recoding the block multiple times and choosing the best trade-off between bit rate and quality. Such coding is computationally expensive, but performs well in terms of coding efficiency.

Thus, given that our codec does not enjoy these two advantages, we find it notable to see that we are still getting slightly better results, and we think that our comparison shows that our technique is rather competitive against H.264 compression in the color buffer compression context. In the table, we have also included our 8×8 results, and as can be seen, this variant of our algorithm performs a further bit better. This is due to the fact that a system with 16×16 tiles will read and write several pixels that are never processed.

6.6 Texture Compression using Buffer Compression

Most existing HDR texture compression schemes compress down to 8 bits per pixel, i.e., at a constant bit rate (CBR) per pixel. Out of curiosity, we experimented with the idea of using our lossy buffer compression algorithm for HDR texture compression at 8 bits per pixel as a final, smaller test. The performance in terms of quality was benchmarked against three state-of-art texture compression methods [20, 12, 15]. While our CBR-decompressor is exactly the same as before, the compressor needed some minor modifications to enable a CBR mode. Our ap-

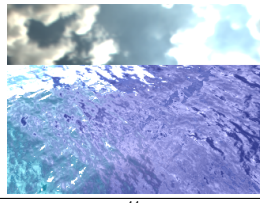
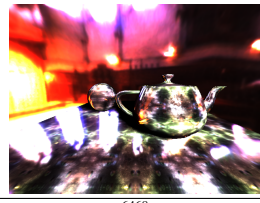
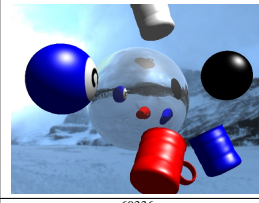
Scene	Water					Shadows					Reflections				
															
# triangles	44					6468					60336				
	A	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
Resolution	**320 × 240**														
Color BW	1.6	0.56	0.57	0.19	0.53	3.8	1.0	0.92	0.43	1.6	4.9	1.5	1.7	0.7	2.8
mPSNR (dB)	-	-	-	64.6	59.4	-	-	-	55.3	54.8	-	-	-	54.3	51.2
logERR[RGB]	-	-	-	0.026	0.029	-	-	-	0.029	0.032	-	-	-	0.028	0.038
HDR VDP	-	-	-	0.00	0.00	-	-	-	0.00	0.01	-	-	-	0.00	0.01
Compr factor	**1.0**	**2.9**	**2.9**	**8.7**	**3.1**	**1.0**	**3.8**	**4.1**	**8.8**	**2.3**	**1.0**	**3.3**	**2.9**	**6.8**	**1.7**
Resolution	**1024 × 768**														
Color BW	14.8	4.1	4.2	1.4	4.8	29.7	6.9	6.1	3.0	9.9	29.2	7.7	8.7	2.9	14.9
mPSNR (dB)	-	-	-	68.8	62.1	-	-	-	60.8	59.0	-	-	-	55.7	54.1
logERR[RGB]	-	-	-	0.021	0.023	-	-	-	0.015	0.019	-	-	-	0.024	0.027
HDR VDP	-	-	-	0.00	0.00	-	-	-	0.00	0.01	-	-	-	0.01	0.06
Compr factor	**1.0**	**3.6**	**3.5**	**10.5**	**3.1**	**1.0**	**4.3**	**4.8**	**9.8**	**3.0**	**1.0**	**3.8**	**3.4**	**10.1**	**2.0**

Table 1: Performance evaluation. The different columns are: **A** = uncompressed, **B** = Ström2008, **C** = our lossless, **D** = our lossy, **E** = OpenEXR B44A. The color buffer bandwidth (BW) is measured in MB/frame. Note that the parameters of our algorithm have been tuned so that the quality/error measures are approximately the same for column D and E. Since the quality is about the same, the compression factors can easily be compared. The HDR VDP rows show percentage of pixels where a human has a 75% chance of detecting an error. In general, one strives after a high mPSNR and low values on logRGB and HDR VDP.

	Bandwidth (MB/frame)	mPSNR (dB)
H.264 16×16	0.397	49.1
Our 16×16	0.385	49.5
Our 8×8	0.295	49.5

Table 2: A comparison with the H.264 codec.

proach is simple, and the core is a basic iterative search method to find one of 32 possible parameter sets, where a parameter set consists of seven parameters; flatness thresholds and quantization levels for each of the Y, C_o and C_g-components, and a restart quantization level. Parameter set 0 represents the mildest compression setting (highest quality, highest number of bits) with increasingly stronger compression settings as we move up towards parameter set 31, which represents the strongest compression setting (lowest quality, lowest number of bits). The chosen parameter set is signalled with five extra bits in the compressed tile data. The stopping criteria for the iterative search method is to test if the number of generated bits is within the range $512 - \tau_{\text{close}}$ and 512. 512 is the number of bits of an 8×8 pixel tile at 8 bits per pixel. τ_{close} is configurable and we used a value of 32 in

our test. As an additional stopping criteria, there is a maximum iterations counter which was set to 6 in our experiment. The search method works as follows: in the first iteration, the tile is encoded using the mid parameter set 15, which is the center of the range 0 to 31. If the first stop criteria is met, we stop and parameter set 15 is chosen. In the second iteration the next candidate parameter set is 23 (half way between 15 and 31) if the compressed tile size was above 512 bits, or 7 (halfway between 0 and 15) if it was below $512 - \tau_{\text{close}}$. Since we in the following iterations have at least two previous attempts, the remaining candidate parameter sets (3rd, 4th, 5th..) are calculated as the linear interpolation of the last two. The candidate parameter sets are clamped to be within the range $0 - 31$. If we cannot find a parameter set that reach the stop range $512 - \tau_{\text{close}}$, the maximum iteration counter halt the search. The chosen parameter set is then the best candidate (closest to, but below 512 bits) of the earlier attempts. For the test images in our experiment, the average number of iterations was around four. With this configuration, the execution time of our CBR-encoder was three orders of magnitude faster than that of Munkberg et al. [12]. Both implementations were written in non-optimzed C++. While our algorithm is almost symmetrical in terms of complexity, the algorithm proposed by Munkberg et al. represents a typical highly asymmerical algorithm.

The results can be seen in Table 3, and our algorithm performs quite well also as an HDR texture compressor. It should be noted, however, that this comparison is a bit unfair in that our algorithm operates on 8×8 pixel tiles, while the other algorithms operate on smaller 4×4 tiles. For one thing, this triggers the high performance two-dimensional predictor for more pixels in the tile (49 out of 64 pixels (76.5%) for 8×8 compared to 9 out of 16 (56.2%) for the 4×4 methods). On the other hand, our algorithm was not designed as a texture compressor, but we thought it would be interesting to find out whether a single codec could work for both texture and buffer compression, and we believe that our experiment indicates that this is so.

7 Conclusions and Future Work

In this paper, we have presented the first lossy floating-point buffer compression algorithm, with results indicating virtually lossless image quality. The compression factors were between $2 - 3$ times larger than state-of-the-art lossless color buffer compression algorithms. If a bit more loss of image quality can be accepted, the compression gain can be much larger. Due to that computation capability continues to grow at a much faster pace than memory bandwidth and latency, we believe that our work can influence the overall performance substantially for future GPUs. We therefore hope our work will spur a renewed interest in high-dynamic range color buffer and texture compression.

As computations become less and less expensive in relation to memory bandwidth usage, it would be interesting to investigate computationally (very) expensive compression/decompression algorithms with better compression ratios and image qual-

Textures	mPSNR (dB)			
	Our	**Sun 2008**	**Munkberg 2008**	**Roimela 2008**
BigFogMap	50.8	51.0	51.9	50.4
Cathedral	37.5	39.7	40.0	34.3
Memorial	44.3	46.8	46.5	41.7
Room	46.9	48.1	48.6	44.0
Desk	39.7	41.5	40.3	28.4
Tubes	32.2	35.7	35.7	27.0

Table 3: Our algorithm used as a texture compressor/decompressor. Benchmarked against three state-of-the-art texture compression algorithms [20, 12, 15] for objective quality. Note that the values in this table for the three other codecs are copied from the paper by Sun et al. [20]. We have left out logRGB and HDR-VDP results since they indicate similar relations between the contenders. The images cann be found in Munkberg et al's paper [12].

ity. This is left for future work.

8 Supplementary Material: Animated content

In order to inspect that the lossy algorithm does not introduce additional artifacts during motion, an animated scene have been put together in a video. The video shows the uncompressed sequence of frames to the left and the lossy compression to the right. These frames have been rendered using the Shadow scene using the same compression parameter settings as the results in Table 1 but at 640×480 resolution. The color buffer bandwidth is thus about a factor of 9 lower than uncompressed mode (the left video) and about a factor of 2 lower than our algorithm in lossless mode.

9 Appendix

1 OpenEXR B44A Lossy Compressor

The B44A encoder is implemented in OpenEXR, and is loosely based on the work by Roimela et al. [16]. In Section 6, we use this algorithm for our lossy compression comparisons, and hence include a more detailed description of the algorithm here. The B44A algorithm operates on the individual color channels of 4×4 blocks. The bit patterns are treated as integers, and the top left value is first stored in full 16 bits. Each pixel is then predicted from its left neighbor, except for the first pixel on each row, which is predicted from its upper neighbor. The prediction

errors between the original and the predicted values are then formed. If all of these prediction errors are in the interval $[-32,31]$, they are stored using six bits each. If not, the original values are divided by 2^k and rounded (starting with $k = 1$), and the prediction errors are formed again. If they still do not fit the $[-32,31]$ interval, the process repeats, etc. For some k, the prediction errors will be small enough so that six bits is sufficient, and the block is then successfully compressed. After compression each channel contains the first value (16 bits), the k-value (6 bits) and the prediction errors (15 6-bit values), all in all 112 bits (14 bytes). However, using six bits to store the k-value allows for a division of 2^{63}, which will never be needed on a 16-bit number. Thus, the B44A encoder reserves this value for the case when the entire block is constant. In this case, only the first value (16 bits) and the shift value (6-bits) are needed, giving 22 bits or 3 bytes. Our version of the B44A encoder always assumes that the alpha component is 1.0, and hence only compresses 3 channels. Also, it compresses 8×8 blocks by concatenating the output from four 4×4 blocks. The resulting bit rate is therefore between $4 \cdot 3 \cdot 3 = 36$ bytes and $4 \cdot 14 \cdot 3 = 168$ bytes.

2 H.264

H.264 is a state-of-the-art video codec that can also be used for compressing still images. While there exists no profile that can handle 15-bit color component depths, there is a 14-bit version. We have created a lossy coder based on H.264 by right-shifting our 15-bit data one step to 14 bits, basically destroying the least significant bit. The resulting 14-bit data is then compressed using the JM (v14.2) reference implementation of H.264 using the High 4:4:4 profile.

The smallest image size H.264 can handle is 16×16 pixels. It would have been possible to modify the H.264 codec so that it could compress 8×8 tiles (like the other algorithms in our comparison), but given the size and complexity of the H.264 code, we refrained from that option. Instead, we have chosen to change the rasterization block size from 8×8 to 16×16 for this comparison only, and hence the H.264 codec can be used unaltered. Our proposed algorithm is adapted to 16×16 pixels by calling our 8×8 compression function four times, which implies that no prediction is done between the four 8×8 blocks. The H.264 encoder, on the other hand, will make prediction across the 8×8 borders inside a 16×16 block, which gives the H.264 code an advantage.

We configured the H.264 codec to work in 4:4:4 mode, which means that no subsampling is used. The reason for this is that subsampling the chrominances gave highly disturbing artifacts along triangle edges. This is a consequence of the fact that H.264 does not have a special mode for prediction across edges.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vinnova. Thanks to Rickard and Clinton from Ericsson Research for Python script

hacking, and to Jon Hasselgren for coding help and reviewing. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg

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] Ahmet Caglar and Amin Ojani. Evaluation and Hardware Implementation of Real-Time Color Buffer Compression Algorithms. Master's thesis, Linköping University, 11 2008.

[4] Solomon W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, (July):399–401, 1966.

[5] Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.

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

[7] Jaakko Lehtinen. A Framework for Precomputed and Captured Light Transport. *ACM Transactions on Graphics,*, 26(4):13, 2007.

[8] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualisation and Computer Graphics,*, 12(5):1245–1250, 2006.

[9] Henrique Malvar and Gary Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. In *JVT-I014r3*, 2003.

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

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

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

[13] Jim Rasmusson, Jon Hasselgren, and Tomas Akenine-Möller. Exact and Error-bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware*, pages 41–48, 2007.

[14] Robert F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 22, Jet Propulsion Lab, 1979.

[15] K. Roimela, T. Aarnio, and J. Itäranta. Efficient High Dynamic Range Texture Compression. *Symposium on Interactive 3D Graphics and Games*, pages 207–214, 2008.

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

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

[18] Jacob Ström and Per Wennersten. Low-Bitrate Table-Based Alpha Compression. In *to appear in Eurographics*, 2009.

[19] Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren, Jacob Munkberg, Petrik Clarberg, and Tomas Akenine-Möller. Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware*, pages 96–101, 2008.

[20] Wen Sun, Yan Lu, Feng Wu, and Shipeng Li. DHTC: an effective DXTC-based HDR texture compression scheme. In *Graphics Hardware*, pages 85–94, 2008.

[21] J. Teuhola. Fast Image Compression by Quadtree Prediction. *Real-Time Imaging*, (4):299–308, 1998.

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

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

[24] Roland Wilson. Quad-tree predictive coding: A new class of image data compression algorithms. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 9:527–530, 1984.

Paper IV

Texture Compression of Light Maps using Smooth Profile Functions

Jim Rasmusson[‡†] Jacob Ström[†] Per Wennersten[†] Michael Doggett[‡]
Tomas Akenine-Möller[‡]

[†]Ericsson Research [‡]Lund University

{jim.rasmusson|jacob.strom|per.wennersten}@ericsson.com
{jim|jon|tam}@cs.lth.se

ABSTRACT

Light maps have long been a popular technique for visually rich real-time rendering in games. They typically contain smooth color gradients which current low bit rate texture compression techniques, such as DXT1 and ETC2, do not handle well. The application writer must therefore choose between doubling the bit rate by choosing a codec such as BC7, or accept the compression artifacts, neither of which is desirable. The situation is aggravated by the recent popularity of radiosity normal maps, where three light maps plus a normal map are used for each surface. We present a new texture compression algorithm targeting smoothly varying textures, such as the light maps used in radiosity normal mapping. On high-resolution light map data from real games, the proposed method shows quality improvements of 0.7 dB in PSNR over ETC2, and 2.8 dB over DXT1, for the same bit rate. As a side effect, our codec can also compress many standard images (not light maps) with better quality than DXT1/ETC2.

Proceedings of High Performance Graphics, June 2010

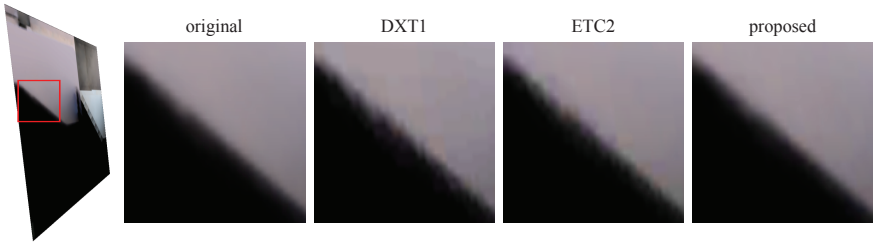


Figure 1: (Paper teaser figure) From left to right: Image rendered with original lightmap, zoom-in on the rendering using original texture, using DXT1, using ETC2 and finally using the proposed method.

1 Introduction

Texture compression [3, 10, 18] continues to be a very important technology in real-time rendering due to lower bandwidth consumption and less memory usage. At the same time, compute power increases at a much faster pace than DRAM latency and bandwidth [15], making techniques like texture compression potentially even more important in the future.

Light maps have been used to increase realism of lighting in real-time rendering for a long time. Since lighting often changes quite slowly, a low-resolution light map can often be combined with a (repeated) high resolution texture in order to create a convincing effect. However, if sharp shadows are desired, a higher resolution must be used in the light map, and this will increase the need for texture compression. The situation is aggravated by higher quality rendering techniques, such as radiosity normal mapping (RNM) [12, 13, 8], since they in effect need three or more light maps per object. Examples include Valve's Source engine (e.g., Half-life 2) [12] and Mirror's Edge & Medal of Honor by DICE/EA [11]. Sometimes about 300 MB of (compressed) RNM textures are needed for a single level in a game, which puts pressure even on high-end graphics cards. In general, the RNM techniques precompute three light maps (each for a different normal direction), and at render time, a linear combination of these light maps is computed per pixel based on the direction on the normal. The normal is typically accessed through a high-resolution, repeated normal map.

As can be seen in Figure 1, the industry standard, DXT1, fails to compress blocks with smoothly varying content with sufficient quality. Both DXT1 [9] and ETC1 [16] work by using a small color palette that the pixels can choose color from. Since they are designed to handle quite arbitrary texture content, neighboring pixels are allowed to choose palette entries completely different from each other. This flexibility is expensive in terms of bits, and therefore the number of possible colors in the block must be restricted. For instance, DXT1 can only display four different colors within a 4×4 block, and ETC1 can display four different colors within a 4×2 block. Apparently, this is not enough for slowly varying textures such as light

maps from radiosity normal maps. It should be noted that ETC2 [17] has a special mode for planar content in a block, but that too does not give enough flexibility for light maps.

This paper proposes a new texture compression algorithm to solve this problem. Instead of allowing neighboring pixels to obtain completely different colors, we exploit the spatial redundancy in the data by letting the position of the pixel in the block control the interpolation between two base colors. To allow for edges, a non-linear function is used to produce the interpolation value. This makes it possible to give unique colors to every pixel of a 4×4 block. For "non-smooth" blocks, we propose using a variant of ETC2 as a fallback.

2 Previous Work

In this section we review the most relevant previous work, which in our case leaves out all the work on alpha map compression, high dynamic range (HDR) texture compression, and lossless compression.

In 1996, the first three papers about texture compression were published [3, 10, 18]. Beers et al. [3] present a texture compression scheme based on vector quantization (VQ) which can compress down to two bits per pixel (bpp). They compress 2×2 RGB888 pixels (12 bytes) at a time using a code book of 256 entries, thus achieving a compression ratio of 12:1. The main issue with VQ based schemes is that they create memory indirection; the decompression hardware must fetch the index from memory before it knows from where in the code book to fetch data. This creates extra latency that can be hard to hide. To reach sufficient quality, one optimized code book is typically needed for each texture, which makes it harder to cache the code books.

The Talisman architecture [18] uses a discrete cosine transform (DCT) codec, but this has seen little use. We speculate that this has to do with the steps following the DCT, which typically include run-length coding and Huffman coding. Both these steps are serial in nature, and cannot be decoded in a fixed number of steps. These are features that are seldomly desired in a hardware decompressor for graphics. Also, often the hardware decompressor is located *after* the texture cache, and hence, the decompressor only needs to decompress a single pixel, which also does not fit well with Huffman & run-length decoding; if the last pixel is requested, all previous pixels must anyway be decoded. Finally, Huffman coding produces variable length data, whereas most texture compression systems use a fixed rate in order to preserve random access. You could also imagine a DCT-based scheme where we avoid Huffman & run-length decoding in order to solve this problem, but that would reduce the compression ratio, and more importantly, each DCT coefficient would still affect every pixel to be decoded, making decompression times long. Hence, it seems that DCT-based codecs are not well-suited for hardware texture compression, but in other contexts, such as video compression, they work very well.

Delp et al. describe an gray scale image compression system called block truncation coding (BTC) [5], where the image is divided into 4×4 blocks. Two gray scale values are stored per block, and each pixel within the block has a bit indicating whether it should use the first or second color. By storing colors instead of gray scale values, Campbell et al. extend this system to color under the name color cell compression (CCC) [4]. Knittel et al. [10] develop a hardware decompressor for CCC and use it for texture compression (as opposed to image compression). The S3TC texture compression system by Iourcha et al. [9] can be seen as an extension of CCC; two base colors are still stored per block (in RGB565), but two additional colors per block are created by interpolation of the first two, creating a palette of four colors. Each pixel then uses two bits to choose from the four colors, with a considerable increase in quality as a result. In total 64 bits are used for a block, giving 4 bits per pixel (bpp). S3TC is now the de facto standard on computers and game consoles under the name DXT1.

Fenney [7] also stores two colors per block, but uses the colors of neighboring blocks: The first color is used together with the first colors of the neighboring blocks to bilinearly interpolate the color over the area of the block. This way a smooth gradient can be obtained. The second color is treated similarly, creating two layers of color. Finally two bits per pixel are used to blend between these two layers, producing the final pixel color. Fenney present both a 4 bpp and a 2 bpp variant of the codec.

The ETC scheme [16] also uses 4×4 blocks, and compress down to 4 bpp. Otherwise the approach is rather different. One color is chosen per 2×4 pixel block. The color of the second block is either encoded separately or differentially, where the latter allows for higher chrominance precision. Each block also encodes a table index, where each entry has four values of the form: $\{-b, -a, +a, +b\}$. Each pixel uses two bits to point into this table entry. The value from the table is added to all three color components, and can hence be viewed as a luminance modification. ETC is standardized in OpenGL ES as an OES extension. By using invalid combinations in the ETC format, Ström and Pettersson [17] manage to squeeze in three more modes in ETC. This new format, called ETC2, is better at handling chrominance edges, and has a special mode for planar transitions.

Recently, new codecs with higher bit rates have been introduced, such as Microsoft's BC7 (called BPTC in OpenGL [1]) which operates at 8 bpp. However, faced with the prospect of doubling storage and bandwidth requirements, game developers may choose the imperfect quality of 4 bpp systems instead.

One technique in the field of image analysis and perception is to represent an image as a summation of a small number of oriented functions [14, 6]. Our technique also uses oriented functions, but represents the blocks using only one function per block, not using a summation of functions.

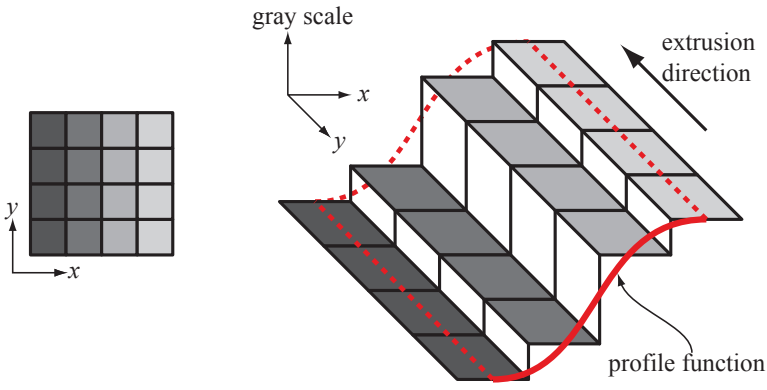


Figure 2: Left: 4×4 pixels with a very regular pixel pattern. Right: illustration of how a profile function is used to describe the content of the pixel block to the left. Note how the profile function is merely a function of x, and then extruded in the y-direction.

3 Compression using Smooth Functions

In this paper, we focus on the compression of smooth light maps, and a general observation is that they often contain rather *little* information, while at the same time, each pixel in the block can have a *unique* color, even if the differences are not always that big. Another observation is that many blocks are *directional*, i.e., they contain more information in one direction (across an edge) than in the other direction (along an edge).

We start by describing the algorithm for gray scale light maps using a very simple example, illustrated in Figure 2. We have divided the image into blocks of 4×4 pixels, and are concentrating on one block. The x-coordinate of each pixel is used to evaluate a function, here called profile function, and the result is used as the gray shade for the pixel. Thus, a single profile function is sufficient to describe the gray scale content of the entire block. However, more flexibility is needed to be able to accurately represent a large number of blocks. For example, a key parameter is to be able to *rotate* the profile function in the xy-plane.

Hence, the core idea of our research is to encode the orientation of an edge in each pixel block, and then specify a *profile* across the edge using a function with a small number of parameters.

To enable orientation and translation, we first establish two orthogonal directions in the block, and an origin. Here, we would like to arrange the directions so that the gray scale varies maximally along the first direction, and minimally along the other direction. The "origin" is placed in the lower left corner of the block. An example is shown to the left in Figure 3. Mathematically, this can be done by fitting a line, $a_1x + b_1y + c_1 = 0$, so that the "normal," (a_1, b_1), of the line coincides with

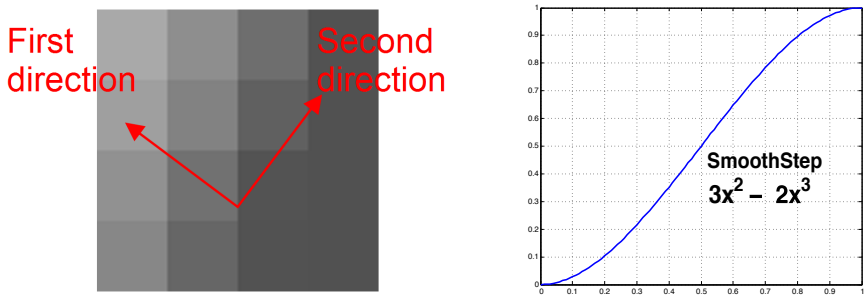


Figure 3: Left: Two orthogonal directions are placed in a block of 4×4 pixels, so that the block varies maximally along the first direction, and minimally along the second. Right: A typical function that can be used to describe the variation along the first direction.

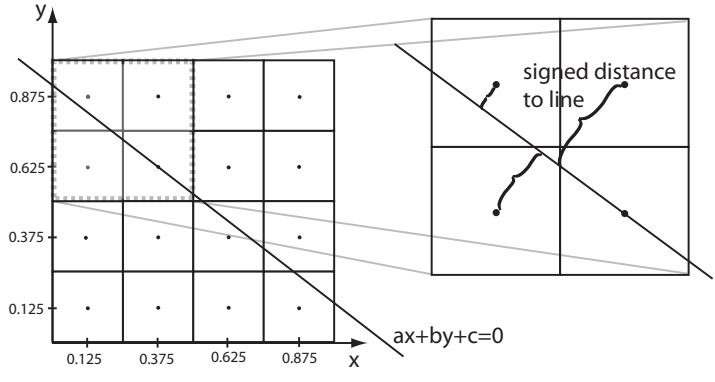


Figure 4: Illustration of the decoding of the block. The edge line is positioned in the block and each pixel computes its signed distance to the edge. This in turn is fed into a smooth profile function in order to compute the value of the pixel.

the first direction. Note that the subscript is used to distinguish it from other lines. By normalizing the equation so that $a_1^2 + b_1^2 = 1$, we can compute the distance, d_1, from any point, (p_x, p_y), to the line by simply inserting the point into the line equation: $d_1 = d_1(p_x, p_y) = a_1 p_x + b_1 p_y + c_1$. This distance is signed meaning that it is negative on one side of the line and positive on the other. See Figure 4 for an illustration of the edge line and the signed distances from the pixel centers to this line.

Next, we use a scalar-valued *profile* function, $f(d_1)$, to represent the variation inside the block. Since d_1 will be constant for points on a line parallel with the second direction, $f(d_1)$ and hence the gray scale value of the block will also be constant in that direction. Points along the other direction will give rise to a varying

d_1, and hence different gray scale values, $f(d_1)$. Using the line definition above, we obtain the gray scale value in a pixel, (p_x, p_y), as $f(d_1) = f(a_1 p_x + b_1 p_y + c_1)$. The profile function, $f(d_1)$, can be a function going from dark to bright, such as the one depicted to the right in Figure 3. This works well for blocks where there is an edge in the block. It could also be a function going from dark, bright and then dark again. Such a function would be better suited for blocks depicting a white line on a black background.

3.1 Extension to Color

A naïve way of extending our method to color would be to duplicate the above-mentioned procedure three times, i.e., once for each color component. However, that would cost too many bits, and would not exploit the fact that the color channels often are well correlated. Instead of directly calculating the grayscale value using the function $f(d_1)$, we use it to calculate an interpolation factor, $i = f(d_1)$, where $i \in [0,1]$. We then use this interpolation factor to interpolate between two different colors, $\mathbf{c}_A$ and $\mathbf{c}_B$, which are representative for the block. The interpolation is then done as:

$$\mathbf{c}(p_x, p_y) = (1-i)\mathbf{c}_A + i\mathbf{c}_B, \tag{1}$$

where $i = f(d_1) = f(a_1 p_x + b_1 p_y + c_1)$. As can be seen, the same interpolation factor, i, is used for all three color channels. So far, we only need to store the colors $\mathbf{c}_A$ and $\mathbf{c}_B$, the line equation (for instance using the constants a_1, b_1, c_1), and indicate which function, f, was used for calculating the interpolation factor in order to decompress a block.

3.2 Second Direction Tilt

Up to this point, we have assumed that all the variation is *across* the edge, i.e., in the first direction. The variation *along* the edge, i.e., in the second direction (see Figure 3), is often non-negligible in practice and needs to be represented in our model as well. Most of the bits will be allocated for the placement of the edges and for the variation across the edge, and hence, we need to limit the number of bits describing the variation in the second direction. We have found that a simple linear variation, i.e., a slope, along the edge does a decent job. This slope is described with a single parameter, γ, which is the same for all three color components, i.e., a linear luminance variation. An additional term, γd_2, is added to each component of the color:

$$\mathbf{c}(p_x, p_y) = (1-i)\mathbf{c}_A + i\mathbf{c}_B + \gamma d_2(1,1,1), \tag{2}$$

where $d_2 = a_2 p_x + b_2 p_y + c_2$ is the signed distance from the pixel, (p_x, p_y), to a line orthogonal to the first line, and $(1,1,1)$ is the maximum color. The line equation for this second line is $a_2 x + b_2 y + c_2 = 0$, where:

$$(a_2, b_2, c_2) = (-b_1, a_1, -0.5(a_1 + b_1)). \tag{3}$$

This means that the direction of the first line is rotated by 90 degrees, and that the translation, c_2, is computed so that the line goes through the origin $(0.5, 0.5)$ of the pixel block. This new term gives us the opportunity to add a luminance tilt along the edge, as illustrated in Figure 6d.

3.3 Profile Functions

We currently use four different profile functions, $f_i(d_1)$, $i \in \{1,2,3,4\}$, as shown in Figure 5, and we pick the profile function that best suits the content of each pixel block. All four functions are based on the simple *smoothstep* function, $s(d)$, shown below in pseudo-code:

```
x = d/w          //scale by width, w
x+ = 0.5         //center function around origin
x = clamp(x,0,1) //clamp between 0 to 1
return 3x^2 - 2x^3 //evaluate smoothstep
```

Note that a width parameter, w, is used to control how rapidly the smoothstep function should increase. A small value will give rise to a sharp edge, whereas a large value will create a smooth transition. The value 0.5 is added so that the function is centered around $d = 0.0$

Figure 5a shows a function we call the *symmetric single*, which is the basic smoothstep function, $f_1 = s(d)$ with a *width* parameter to control the shape. Figure 5b shows the *asymmetric single* function, f_2, where the smoothstep function also is used. However, two different width values are used here; one to the left of the y-axis, and one to the right. Figure 5c shows the *asymmetric double* function, f_3, which uses two "concatenated" smoothstep functions (hence the name 'double'), again using different widths on either side of the y-axis. In contrast to the two previous profile functions, this function uses three colors for the interpolation calculation. A third color is used at $d_1 = 0$, and to save storage this color is not a separate color but instead generated as the average of the other two and scaled with a scale factor.

The fourth function is different since it adds another smoothstep in an arbitrary direction, $d_3 = a_3 p_x + b_3 p_y + c_3$, and multiplies two smoothstep functions, $s(\cdot)$, together:

$$f_4(d_1, d_3) = s(d_1)s(d_3). \tag{4}$$

Hence, each smoothstep uses its own line equation, but they also use their own widths. The scalar value, $f_4(d_1, d_3) \in [0,1]$, is used as an interpolation factor in exactly the same way as for the other functions. See Figure 5d for an example. As can be seen, this gives us the ability to represent various smooth corners in a block. Since the fourth function is already two-dimensional, no second direction tilt is needed.

Our choice of profile functions is the result of a combination of reasoning and trial and error, therefore it may seem a bit arbitrary. We started out with the assumption that the single symmetric function should capture smooth and sharp edges well,

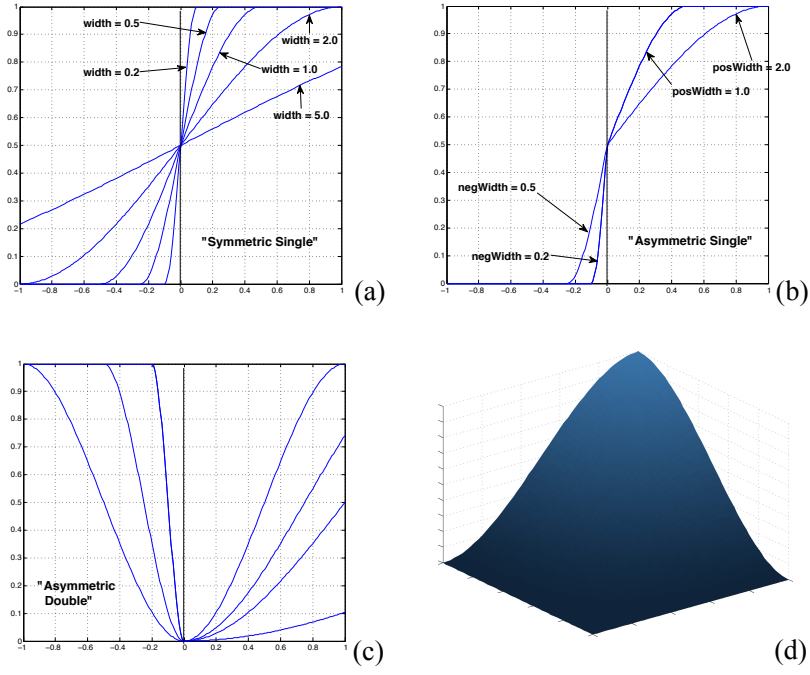


Figure 5: The four functions used to fit the color variation. a) symmetric single, b) asymmetric single, c) asymmetric double, and d) corner, which is the multiplication of two smoothstep functions along two different directions.

which turned out to be true. We then tried over 20 different functions, and picked the ones that gave the best results. In retrospect, it is clear that the asymmetric single captures blocks containing discontinuous edges, and that the asymmetric double can capture a smooth line inside a block, as mentioned earlier. The corner function captures corners, which often occurs in light maps.

3.4 Fallback method

While many blocks will compress well using the above approach, there will be others that do not have any directional structure, or any structure at all. For such blocks, we use a variant of ETC2 as a fall-back coder [17]. One bit per block will therefore be used to indicate whether profile functions should be used, or the ETC2 variant should be used. To preserve a bit rate of 64 bits per block, we need to steal one bit from the ETC2 codec. This is done by disabling the individual mode in ETC2, which frees up the diff-bit. Thus, 63 bits are left to compress the block using profile functions.

Number of bits	SSingle	ASingle	ADouble	Corner
ColorA ($\mathbf{c}_A$)	666	565	555	555
ColorB ($\mathbf{c}_B$)	666	565	555	555
Function widths ($w_1 w_2$)	5-	44	55	44
Rotations ($\theta_1 \theta_3$)	7-	8-	6-	66
Translations ($c_1 c_3$)	7-	9-	6-	56
Scale factor (m)	-	-	5	-
Second direction tilt (γ)	6	4	4	-
Total	**61**	**61**	**61**	**61**

Table 1: Bit distribution for the four modes; symmetric single (SSingle), asymmetric single (ASingle), asymmetric double (ADouble), and corner smoothstep mode.

3.5 Function Parameters, Quantization, and Encoding

In this section, we will describe, in detail, all the parameters of our codec, how they are quantized and encoded. Our codec can use either of the four profile functions in Figure 5, and hence, two bits are needed to select profile function. This leaves $63-2=61$ bits to encode the parameters for the selected profile function.

In Figure 6, a visualization of the key parameters for symmetric single are shown. There are the two end-point colors, ColorA ($\mathbf{c}_A$) & ColorB ($\mathbf{c}_B$), and line placement parameters, consisting of orientation, θ_1, and translation, c_1. The line equation then becomes: $a_1 x + b_1 y + c_1 = 0$, where $a_1 = \cos\theta_1$ and $b_1 = \sin\theta_1$. As we have seen earlier, this line is used to place the profile function in its best possible location. Furthermore, Figure 6c shows the function *width*, which determines how rapidly the function rises (see the width parameter, w, in Section 3.3). The effect of the width can be seen in Figure 5a. Finally, Figure 6d shows the second direction tilt parameter, γ, described in Section 3.2. This improves image quality in that it often reduces block artifacts.

The bit distribution varies depending on the choice of profile function. We chose a starting distribution for each function and then iteratively optimized them over a small training image. The final bit distributions for our parameters in the four smoothstep-based profiles are listed in Table 1. The symmetric single uses $3 \cdot 6$ bits per color, five bits for its width, w, seven bits for its orientation angle, θ_1, seven bits for its translation, c_1, and six bits for its tilt parameter, γ.

Compared to the symmetric single, the only new parameter for the asymmetric single is that it has two widths, w_1 and w_2, instead of only one. The asymmetric double has an additional parameter, called the scale factor, m. As mentioned briefly in Section 3.3, this factor is used to compute the color in the "middle" of the asymmetric double function. The color is computed as: $\mathbf{c}_{mid} = m(\mathbf{c}_A + \mathbf{c}_B)/2$. Finally, the corner codec uses two line equations, also described in Section 3.3, and hence encodes two orientation, θ_1 and θ_3, and two translations, c_1 and c_3.

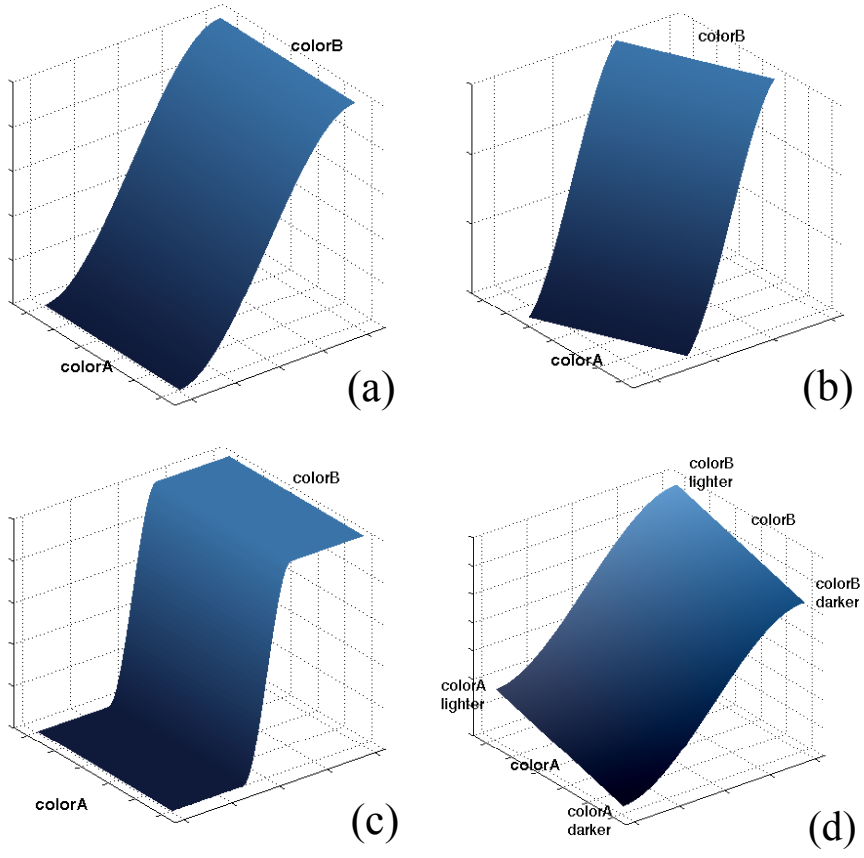


Figure 6: Illustration of the function parameters of the symmetric single smoothstep function: a) two end-point colors, b) rotation and translation, c) width, and d) slope.

All of the parameters used are viewed as floating point values, with individual ranges as shown in Table 2. The stored bits are treated as fixed point representations, with the decimal point placed to achieve a similar range as in the floating point domain. For example, in the 6-bit second direction tilt used in the symmetric single mode, the decimal point is before the least significant bit, giving a range of -16 to +15.5. In the other modes which use 4 bits for the second direction tilt, a zero is inserted after the last of the four bits to get a 5-bit integer value, which will be in the range of -16 to +14. There are two exceptions to this rule: first, the bits representing the color components are repeated which ensures we are always able to represent both extreme points 0 and 255. Second, the width parameters use a non-linear quantization, which in the 5-bit case is:

$$w = 0.005 \times (1000^{\frac{1}{31}})^q \tag{5}$$

Parameter	*min float value*	*max float value*
Color components	0	255
Function widths (w_1&w_2)	0.005	5
Rotations (θ_1&θ_3)	0°	180°
Translations (c_1&c_3)	-2	+2
Scale factor (m)	0	2
Second direction tilt (γ)	-16	+16

Table 2: This table shows the valid range for each parameter *before* quantization. The minimum values are the same after quantization, but the maximum values are slightly smaller depending on the number of bits used.

where q is the stored value. In the four-bit case, the equation is slightly altered to get a similar range of values:

$$w = 0.005 \times (1000^{\frac{1}{31}})^{2q} \tag{6}$$

4 Compression algorithm

Compression is done in two iteration stages, the first with the parameters in the floating point domain and the second, after parameter quantization, in the fixed point domain. We use cyclic coordinate search [2] to minimize the error function, i.e., for each coordinate (i.e., parameter, such as the *width*) we approximate the gradient with respect to that parameter, and go a step in the opposite direction.

However, if the error function changes very rapidly, a small step size is necessary in order to be guaranteed a lower error in the new point. This small step may be smaller than what it is possible to step in the quantized domain. To solve that problem, we do the first round of optimization using floating point arithmetic. After a global minimum has been found, and cyclic coordinate search is no longer fruitful, we quantize the values and do a second round of optimization in the quantized domain. This is necessary since we will not be able to reach the floating point position exactly when quantizing it, so we must try a number of quantized positions around that point.

For each 4×4 pixel block the following is performed:

1. Find the direction of the maximum variation. This is represented as the rotation angle, θ.

2. Use this rotation for initial orientation of the function.

3. For initial ColorA ($\mathbf{c}_A$) and ColorB ($\mathbf{c}_B$), we use the "min" and "max"- colors along the maximum variation line.

4. Iteratively search over the entire floating-point parameter space (ColorA, ColorB, rotation, offset, width and slope) using cyclic coordinate search to minimize the image error.
5. Quantize all parameters.
6. Perform a second iterative search of the quantized parameters.
7. Choose the mode/function with the smallest error.
8. Pack into 64 bits.

To ensure a "somewhat" exhaustive search of the possible parameter space, we ran 200 iterations of the floating point parameter search and 6 iterations of the fixed point parameter search. This is of course a trade-off between compression time and performance. However, performance did not improve much above 200 float / 6 fixed iterations. Our compression algorithm compresses an image with 192×192 pixels in about 50 seconds. This is done with a multi-threaded implementation on a MacPro with dual Intel quad core CPUs at 2.26 GHz. Early on, we implemented the encoder in OpenCL on an NVIDIA GPU and managed to get a speed up factor of $42\times$ compared to a single-threaded CPU version. However, we decided to focus our programming efforts on a multi-threaded CPU implementation, since the OpenCL tools for debugging and profiling were rudimentary at best (Mac OSX 10.6.1/2).

5 Decompression algorithm

The decompressor is much simpler and faster. It is designed to have low complexity in order to ensure inexpensive hardware implementation. To decompress a single pixel in a 4×4 block, the following is done:

1. The rotation angle, θ, and offset parameter, c, are used to reconstruct the edge line, $ax + by + c = 0$, where $a = cos(\theta)$ and $b = sin(\theta)$.
2. For the pixel (x, y), calculate a signed distance, $d = ax + by + c$. See Figure 4.
3. An interpolation value, i, is calculated using the selected base function, $i = f(d), i \in [0, 1]$.
4. i is then used to interpolate between ColorA and ColorB.
5. Finally, the slope parameter is used to add a luminance ramp (along the edge line) to the output color.

For example, in the symmetric single case, Step 3 uses the smoothstep function, $s(d)$, as shown in Section 3.3. For Step 5 the distance from the orthogonal line must be first calculated using an orthogonal line equation. This orthogonal distance is multiplied by a scaling factor to vary the color along the edge line.

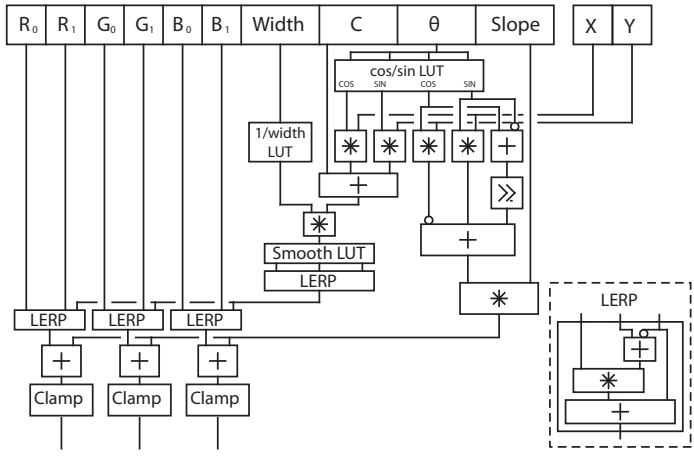


Figure 7: Hardware design for a *symmetric single* decompression unit. Linear interpolation is replaced with a LERP unit as shown in the lower right of the diagram. Three lookup tables are used for sin/cos, $\frac{1}{width}$ and smoothfunc evaluation.

5.1 Fixed function decompression algorithm

We have implemented a fixed function version of the decompressor algorithm. This decompressor uses the functions shown above with only fixed point addition and multiplication with 7 bits precision up until the final lerp, addition and clamp, and several lookup tables for the more complex functions. A possible hardware design is shown in Figure 7.

The scaling of the width of the smoothstep function requires a division. We replace this division with a lookup of $\frac{1}{width}$ and a multiplication. The width value is represented with 3 bits of integer and 7 bits of fractional precision.

The *sin* and *cos* used in computing the coefficients a and b from the rotation angle parameter θ in Step 1 are stored in a lookup table with 128 entries and 7 bits per entry for values ranging from 0 to 180 degrees.

We approximate the smoothstep equation, $3x^2 - 2x^3$, with a piecewise linear function stored in a lookup table. This table only stores the region between 0.5 and 1 since the function is symmetrical. We then split the 0.5 to 1 region in half recursively to create 8 segments and store the end point values in a table with 7 bits of precision. When decoding we find the segment by counting the number of leading zeros and then interpolate the endpoints from the lookup table.

We have performed a rough hardware complexity estimate, using estimates such as 2.2 gates per bit of a MUX and 4.4 gates per bit of an ADD, where one gate equals a 2-input NAND. The result was a gate count somewhere between 4000 and 5000 for our decoder, including the ETC2 fallback. This is roughly 4 times the size of the ETC2 decoder and 5 times the size of a DXT1 decoder estimated in a similar

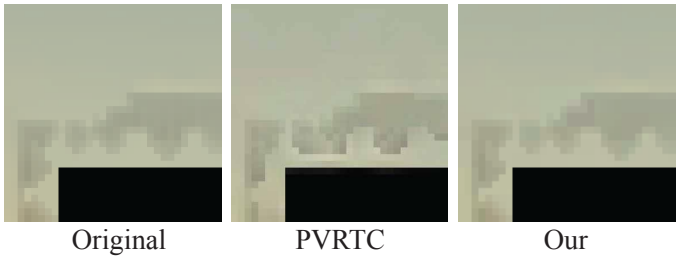


Figure 8: PVRTC handles smooth transitions well due to its bilinear interpolation. Instead, compression artifacts manifest themselves in other parts of the textures, as shown above.

Figure 9: Texture compression results using a test set of 24 radiosity normal maps from the game *Mirror's Edge*. Our method is 0.7dB better than ETC2 and 2.8 dB better than DXT1 when all 24 results are combined together.
manner.

6 Results

We have tested our algorithms on radiosity normal maps (RNMs) from two real games, namely *Mirror's Edge* and *Medal of Honor* from DICE/EA. While our main target is smooth light maps, we also wanted to test the hypothesis that our codec can also be used as a more generic texture compression method. Therefore, we also used the 64 regular textures which were used for evaluating ETC2 [17]. We have compared our results against DXT1 and ETC2 for all of the test sets. For DXT1 we used The Compressonator version 1.50.1731, and for ETC2 exhaustive compression was used. In both cases, error weights of (1,1,1) were used to maximize PSNR. We contemplated also comparing to the 4 bpp version of PVRTC [7], especially since its use of bilinear interpolation handles smooth transitions well. Unfortunately, compression artifacts creep up in other places instead as shown in Figure 8, and on the Mirror's Edge test set, it was 1.9 dB lower than DXT1 when compressed using PVRTexTool Version 3.7. For these reasons, we have excluded PVRTC from our results.

In the following subsections, we first present the results for Mirror's Edge, which

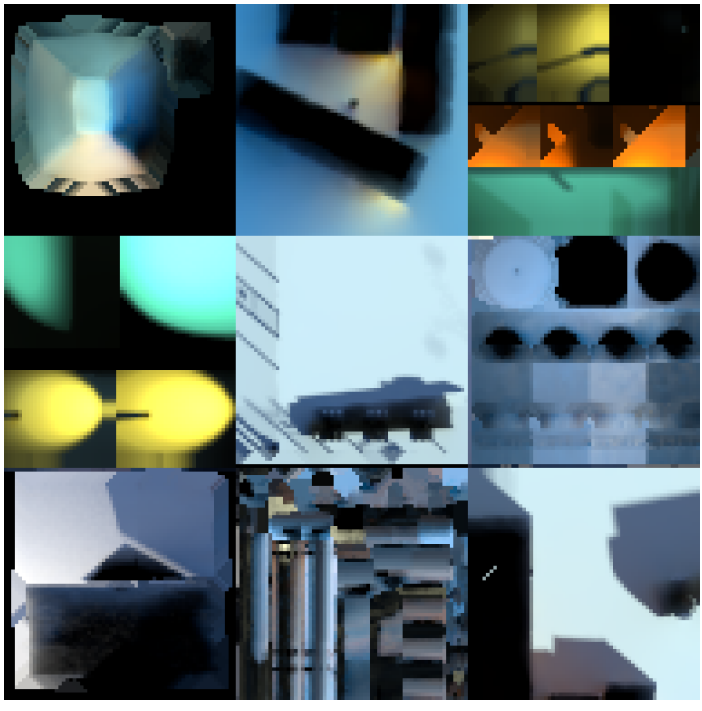


Figure 10: Original light maps from Mirror's Edge, used for training our algorithm. Images courtesy of Electronic Arts.

is followed by Medal of Honor's results. In Section 6.3, we present results for the 64 regular (non-light map) textures used to evaluate ETC2 and also a test done with a set of 24 regular photos from Kodak. Finally, we show zoomed-in crops of some textures in Figure 16.

6.1 Mirror's Edge

During the development of our algorithm (including bit distribution, profile function selection, quantization, and more), we used nine representative crops from the game Mirror's Edge as a training set. These are shown in Figure 10, and for all our results, these images were excluded in order to avoid biasing the result. The remaining RNM images from Mirror's Edge were used to form a larger test set of 24 RNMs.

The results are shown in Figure 9. The combined PSNR results for this set are 41.2 dB, 40.5 dB and 38.4 dB for our method, ETC2, and DXT1, respectively. Our method consistently performs better than ETC2 and DXT1 for this entire test set. A few resulting images (with zoomed-in crops) from this test set are shown in the top two rows of Figure 16. It is interesting to note that for the images with

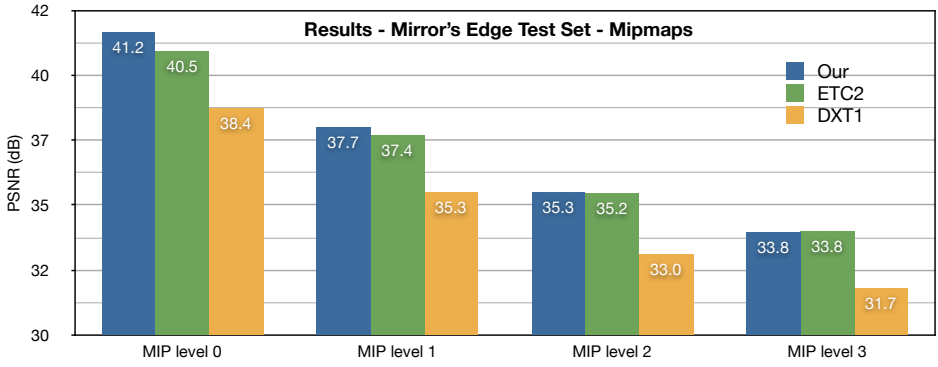


Figure 11: Results for the mipmapped versions of the 24 image test set from Mirror's Edge. Note that mipmap level 0 is the full resolution.

lowest PSNR, the improvement from our algorithm is almost zero compared to ETC2. The reason for this is that when ETC2 and DXT1 perform poorly, the image content is usually rather noisy, in which case our codec has little chance of improving quality.

We also generated mipmapped versions of these 24 RNMs to investigate how our method behaves with low-pass filtered and lower resolution versions of the RNMs. The results are shown in Figure 11. With decreasing resolution, ETC2 gradually approaches our method and in the lowest mipmap level, ETC2 is slightly ahead. This is expected since the smooth areas get smaller the lower the resolution, and the texture becomes gradually more noisy.

6.2 Medal of Honor

We compressed a set of 36 RNMs from the recent game *Medal of Honor* from EA. The combined PSNR results were 37.06 dB, 37.01 dB, and 34.15 dB for our method, ETC2, and DXT1, respectively. As can be seen, our method was only insignificantly better than ETC2 for this test set. In fact, in 8 out of 36 images, the performance of our method was lower than ETC2. Those images contain large numbers of very small light maps baked into a large texture, as shown in Figure 12. If we remove all light maps containing any sub-textures smaller than or equal to 16×16 pixels, the relative PSNR result changes: our method now has a 0.53 dB advantage over ETC2 and 3.05 dB over DXT1. Thus, the Medal of Honor test set clearly shows that our method is only improving quality for relatively large textures, i.e., bigger than 16×16 pixels. This is not surprising given the nature of our method, which depends on exploiting smooth structures.

Even for the textures where the proposed coder is worse than ETC2, it is hard to spot flaws visually. This may be because the textures are very small and quite noisy, and difficult for a human eye to understand what is the "correct" structure.

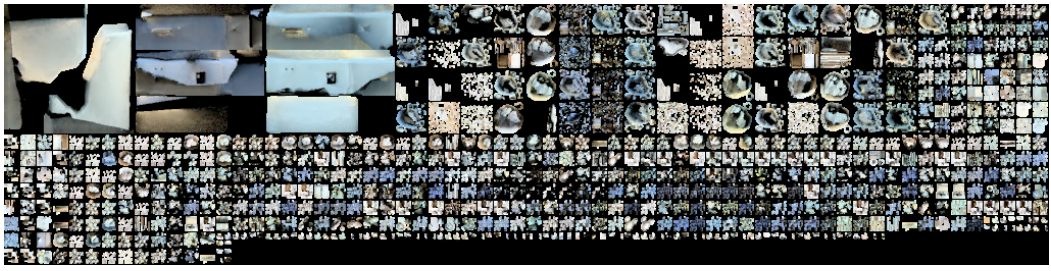


Figure 12: An example where our method performs rather poorly from the *Medal of Honor* test set. This light map was used for radiosity normal mapping in the game.

Original ETC2 Our

Figure 13: This shows the 16×16 block where our method performed the worst (PSNR wise) compared to ETC2. This is part of the texture in Figure 12 from Medal of Honor.

To give a feeling for the worst case, we automatically found the 16×16 block which gave the worst performance relative ETC2, and it is depicted in Figure 13. Note that it is only in blocks where the individual mode is chosen that ETC2 can be at an advantage over the proposed codec, since the latter includes a subset of ETC2.

6.3 Regular Textures and Photos

We also tested our method using the same set of regular texture as used in the evaluation of ETC2 [17]. This test set is a broad mixture of photos, game textures, and some computer generated images. While our algorithm was not designed with such diverse textures in mind, it was still 0.34 dB better than ETC2, and 1.25 dB better than DXT1 for the full resolution. The performance for the mipmapped versions was similar to our previous mipmap results, but at the highest level (smallest texture), ETC2 performed slightly better. The combined results for the full resolution and the mipmapped versions are illustrated in Figure 14. A few zoomed in examples from this test set are shown in Figure 16.

To compare against publicly available data, we tried our method on the Kodak data

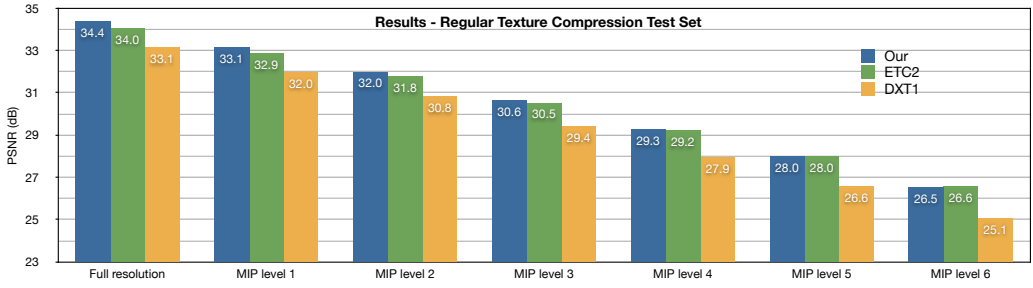


Figure 14: Results from the regular test set of 64 texture images. Our method is 0.34 dB better than ETC2 and 1.25 dB better than DXT1 for the full resolution. The results for the mipmapped versions are in here as well.

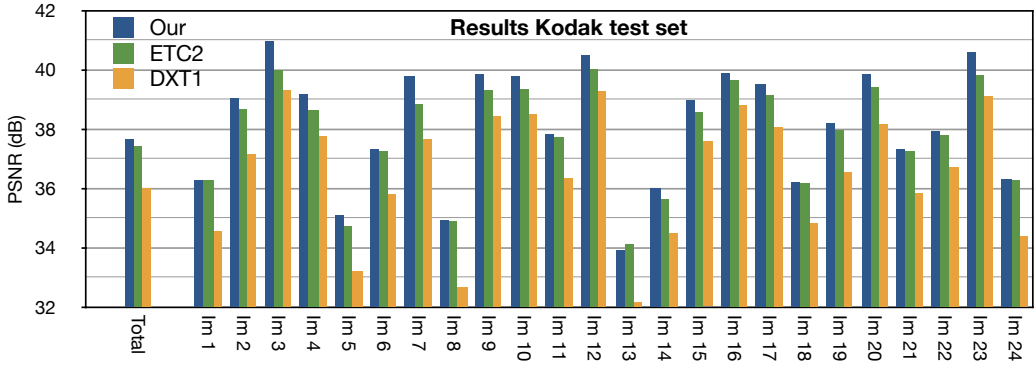


Figure 15: Results from a public test set of 24 photos from Kodak. Combining the results from all 24 photos, our method is 0.24 dB better than ETC2 and 1.65 dB better than DXT1.

set http://r0k.us/graphics/kodak/. The results are shown in Figure 15.

7 Conclusion

We have presented a new codec for texture compression. It is based on parameterized smooth profile functions, with a subset of ETC2 as a fallback for noisy blocks. Our method often generates images with much higher quality than competing algorithms for light maps, and as a positive side effect, our codec also increases the image quality on regular images.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Many thanks to Henrik Halén at EA/DICE for providing us with light map textures from recent games.

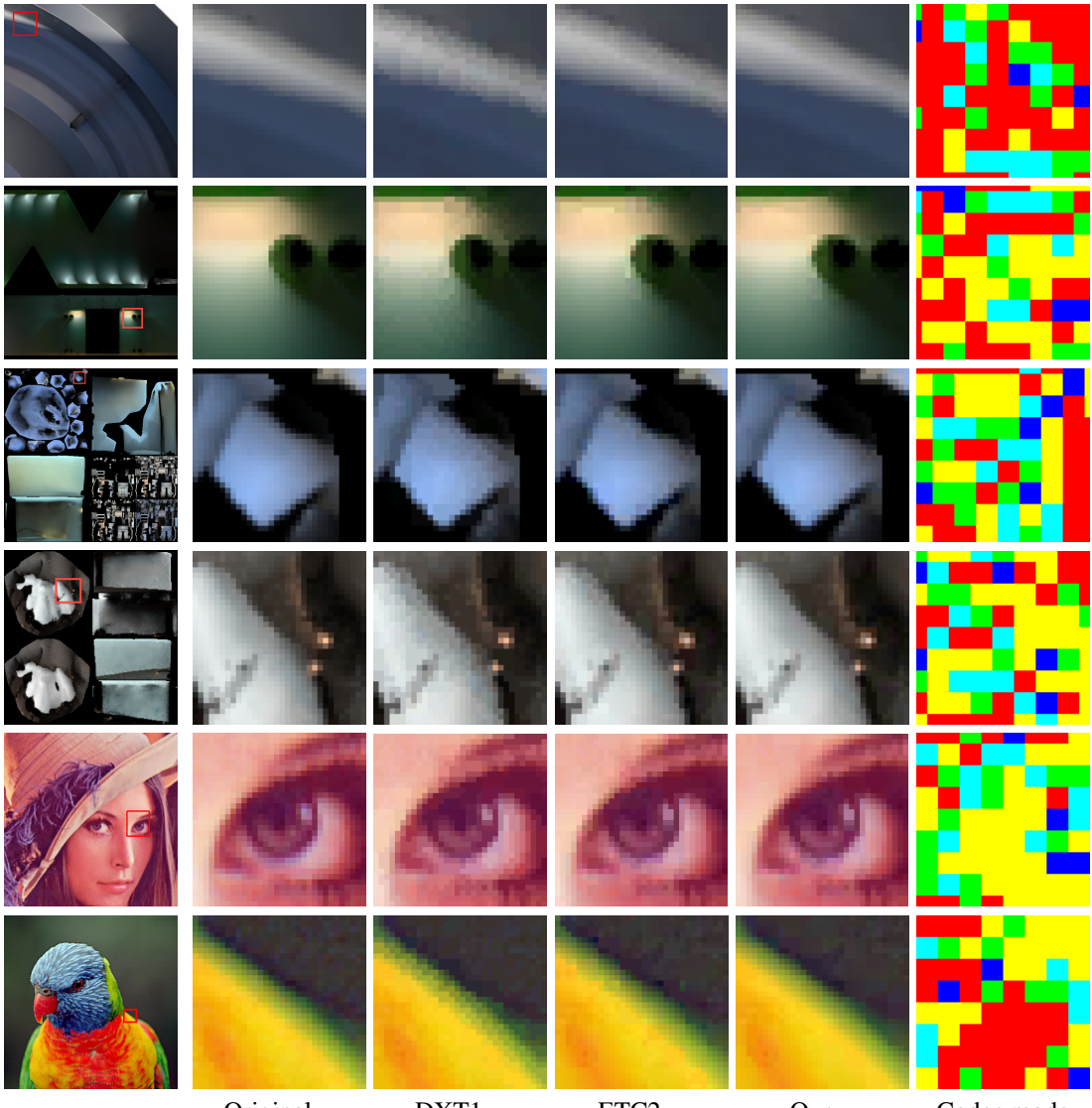


Figure 16: The top two rows show light maps from Mirror's Edge. The mid two rows show light maps from Medal of Honor, and the last two rows are examples of regular textures. The rightmost column shows which of our 5 codec modes that was selected: Red=SymmetricSingle, Green=AsymmetricSingle, Blue=Double, LightBlue=Corner, Yellow=modified ETC2 (fallback mode)

Bibliography

[1] `http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt`.

[2] M. S. Bazaraa, H. D. Sherali, and C.M. Shetty. *Nonlinear Programming — Theory and Algorithms*. Wiley, 1993.

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

[4] Graham Campbell, Thomas A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, Lawrence A. Leske, John A. Lindberg, and Daniel J. Sandin. Two Bit/Pixel Full Color Encoding. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, pages 215–223, 1986.

[5] E. J. Delp and O. R. Mitchell. Image Compression using Block Truncation Coding. *IEEE Transactions on Communications*, 2(9):1335–1342, 1979.

[6] Minh N. Do and Martin Vetterli. The Finite Ridgelet Transform for Image Representation. *IEEE Transactions on Image Processing*, 12(1):16–28, 2003.

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

[8] Chris Green. Efficient Self-Shadowed Radiosity Normal Mapping. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)*, 2007.

[9] 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.

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

[11] David Larsson and Henrik Halén. The Unique Lighting of Mirrors Edge. In *Game Developers Conference*, 2009.

[12] Gary McTaggart. Half-Life 2 / Valve Source Shading. In *Game Developers Conference*, 2004.

[13] Jason Mitchell, Gary McTaggart, and Chris Green. Shading in Valve's Source Engine. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)*, 2006.

[14] Bruno A. Olshausen and David J. Field. Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images. *Nature*, 381:607–609, 1996.

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

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

[17] Jacob Ström and Martin Pettersson. ETC2: Texture Compression using Invalid Combinations. In *Graphics Hardware*, pages 49–54, 2007.

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