IMPROVED ENCODING FOR COMPRESSED TEXTURES

Pavel Krajčevski

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the College of Arts and Sciences.

Chapel Hill
2016

Approved by:

Dinesh Manocha

Ming C. Lin

Ketan Mayer-Patel

Matt Pharr

Turner Whitted

ABSTRACT

PAVEL KRAJCEVSKI: Improved Encoding for Compressed Textures
(Under the direction of Dinesh Manocha)

For the past few decades, graphics hardware has supported mapping a two dimensional image, or texture, onto a three dimensional surface to add detail during rendering. The complexity of modern applications using interactive graphics hardware have created an explosion of the amount of data needed to represent these images. In order to alleviate the amount of memory required to store and transmit textures, graphics hardware manufacturers have introduced hardware decompression units into the texturing pipeline. Textures may now be stored as compressed in memory and decoded at run-time in order to access the pixel data. In order to encode images to be used with these hardware features, many compression algorithms are run offline as a preprocessing step, often times the most time-consuming step in the asset preparation pipeline.

This research presents several techniques to quickly serve compressed texture data. With the goal of interactive compression rates while maintaining compression quality, three algorithms are presented in the class of endpoint compression formats. The first uses intensity dilation to estimate compression parameters for low-frequency signal-modulated compressed textures and offers up to a 3X improvement in compression speed. The second, FasTC, shows that by estimating the final compression parameters, partition-based formats can choose an approximate partitioning and offer orders of magnitude faster encoding speed. The third, SegTC, shows additional improvement over selecting a partitioning by using a global segmentation to find the boundaries between image features. This segmentation offers an additional 2X improvement over FasTC while maintaining similar compressed quality.

Also presented is a case study in using texture compression to benefit two dimensional concave path rendering. Compressing pixel coverage textures used for compositing yields both an increase in rendering speed and a decrease in storage overhead. Additionally an algorithm is presented that uses a single layer of indirection to adaptively select the block size compressed for each texture, giving a 2X increase in compression ratio for textures of mixed detail. Finally, a texture storage representation that

is decoded at runtime on the GPU is presented. The decoded texture is still compressed for graphics hardware but uses 2X fewer bytes for storage and network bandwidth.

Dedicated to my mother, father, and sister.

ACKNOWLEDGEMENTS

This dissertation was made possible by all of the people who have supported me over the last five years. In particular, my mentors at UNC, Professor Russ Taylor who allowed me to pursue my work in texture compression even when it didn't align with his research interests, and Professor Dinesh Manocha who graciously allowed me to continue even though it did not fit well into any existing research project either. To my committee, that has given me invaluable feedback for how to approach some of the trickier problems. To all of the UNC faculty whose classes have exposed me to the different methods for thinking about a wide variety of issues, and helped me work through seemingly ill-defined problems on their whiteboards. Additionally the members of the GAMMA group who helped me refine the practice talks that were completely irrelevant to their research, and in particular Srihari Pratapa without whom we would not have been able to finish the final part of this thesis.

I would also like to thank all of my various internship hosts without whom I would have been restricted to the ivory tower. Dave Houlton and Doug McNabb at Intel who introduced me to the understudied topic of texture compression. Janne Kontkanen, Rob Phillips, and Matt Pharr at Google who exposed me to a variety of problems that had clear benefits from my work. Abhinav Golas and Karthik Ramani at Samsung who gave me significant freedom to pursue topics that extended the functionality of GPUs.

Much of this dissertation would not have been possible without the support prior to going to graduate school. I would like to thank John Reppy for introducing me to the wide world of research and allowing me to expand my knowledge of both programming languages and computer graphics. I would also like to thank Fletcher Dunn for encouraging me to never stop implementing my ideas. Without his wisdom, many of my projects would have been riddled with even more programming errors and poor foresight. I have learned more from this attitude of doing that I have from any other source. I would also like to thank Tom Eastman and the entire team at Trinket Studios for providing me much needed context about life outside of graduate school and for sending many of their textures for me to experiment on.

Finally, I would not have finished this thesis without the love and support of my close friends and family. I'd like to acknowledge my parents, who have always believed in me, even when I didn't. To my grandparents, to whom I promised I would finish my PhD before I knew what that entailed. To all of my friends in Shantytown that have stayed amazing and successful and set the bar for success. To all my friends in the department that have given me an ear when some experiment didn't go my way. Lastly, I would not have finished this dissertation without my partner, Kristin Sellers, who remained an idealized model of a graduate student while also providing an unprecedented level of love and support. Without her I would have thought none of this possible.

Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Texture Compression Formats:

S3TC	S3 Texture Compression (also known as DXT and BC1)
PVRTC	PowerVR Texture Compression
BPTC	Block Partitioned Texture Compression (also known as BC7 and BC6H)
ETC	Ericsson Texture Compression v1
ETC2	Ericsson Texture Compression v2
ASTC	Adaptive Scalable Texture Compression
S3TC	S3 Texture Compression (also known as S3TC)
BC*x*	Block Compression as defined in the Microsoft DirectX specification
VBR	Variable Bit Rate Compression
GST	GPU-Decodable Supercompressed Textures

Image Similarity Metrics:

PSNR	Peak Signal to Noise Ratio
SSIM	Structural Similarity Image Metric

Other Compression Formats:

JPEG	Joint Photographic Experts Group
PNG	Portable Network Graphics
DPCM	Differential Pulse Code Modulation
ANS	Asymmetric Numeral Systems

Miscellaneous:

CPU	Central Processing Unit
GPU	Graphics Processing Unit
BPP	Bits Per Pixel

CHAPTER 1: INTRODUCTION

Over the past three decades, computer graphics has matured into a robust and well-studied field. With our understanding of the physics of light and display technology, synthesized images are becoming increasingly indistinguishable from photographs. As with most aspects of modern computing, we are limited by available resources such as processor speed and memory size for each generated frame of animation. In the extreme, interactive graphics applications limit the amount of computation for a given frame to the time in between display updates, typically 16 milliseconds which allows for 59.94 frames per second on modern high definition displays. To facilitate this performance, dedicated hardware, or graphics processing units (GPUs), are used to perform large batches of computations in parallel.

One of the main uses of GPUs is to determine the visual appearance of a given set of geometric primitives on a modern display. Although many different rendering techniques have been proposed in the literature, modern hardware has been optimized for the processing and display of triangles. Many algorithms governing modern GPU architecture roughly correlate rendering performance to the number of triangles drawn in a given frame. To maintain performance, large triangles are generally used in order to better exploit the benefits of the parallelization of modern GPUs. In order to preserve the detail on these triangles, *texture mapping* is a technique that parametrically maps images (textures) onto triangle surfaces (Catmull, 1974). This technique has become so prevalent that GPUs have specialized hardware for storing and accessing textures to boost overall performance.

Over the past few decades, textures have become the predominant consumer of memory bandwidth and usually take up about 80% of the GPU memory used by a typical interactive graphics application, such as a modern AAA game title (Chen, 2016). For many of these applications, a significant portion of the time is spent loading and unloading textures, generally around 512×512 in dimension, but may be as big as 8192×16384. With these trends, texture size and storage has become a significant problem. Interactive graphics applications usually run simultaneously on traditional CPU architectures and interface with GPUs to offload the specialized rendering computations. This decoupling of computing tasks between the CPU and GPU requires a method by which data, such as textures, is transferred from a

storage device, such as a hard disk or the network, to main memory for use with the CPU, and finally to the video memory on the GPU. Even traditional personal computers are limited by the amount of data that can be transferred. Additionally, certain architectures such as the systems-on-a-chip used in mobile devices are much more sensitive to power consumption and memory bandwidth restrictions (Imagination, 2016). A significant amount of this overhead is inherent in accessing the texture data stored in memory during rendering (Nixon et al., 2014).

Recent advances in graphics hardware have pushed for compact representations of texture data. These compressed data techniques either provide additional cost savings by requiring less memory for texture storage on disk, or increase the visual quality of interactive graphics applications by allowing more detailed textures to be used. Additionally, the smaller size of textures provides faster load times from the disk or network into CPU RAM and from CPU RAM into GPU RAM. The number of memory accesses needed during rendering is also reduced due to the amount of data contained in a smaller chunk of memory. The ability to leverage these representations becomes a boon to application developers and providing this accessability is very important problem.

1.1 Texture Compression

Over the past half-century, there has been significant research in image compression algorithms (Duce, 2003; Wallace, 1992; Skodras et al., 2001). However, in order to render these images, they must be decompressed in order to get access to the raw pixel values. These pixels are usually represented using a value in the range $[0, 255]$ for each red, green, and blue channel to correspond to the underlying structure of modern displays. An optional alpha channel may be used for transparency. This corresponds to 24 or 32 bits per pixel in a decompressed image. As an example, a typical image without alpha of dimensions 768×512 is 1.18MB in memory. Using JPEG compression, this image can be stored using only 128KB, approximately a 10X savings.

Although image compression reduces the storage of an image on disk, it does not affect the decompressed image representation in main memory. In heterogeneous computing environments, such as those that use both a CPU and a GPU for rendering, this implies that on-disk image compression does not realize any benefits when transferring data across the aforementioned CPU-GPU bus. To tackle this problem, many modern commodity graphics chips support *compressed textures* through dedicated hardware decoders. These compressed representations are designed to store textures in a way that

Original ASTC 4x4 ASTC 8x8 ASTC 12x12

Figure 1.1: An example of artifacts caused by lossy compression formats used in modern GPUs. ASTC (Nystad et al., 2012) compresses $N \times M$ blocks of pixels down to a fixed number of bits across all block sizes. The larger the blocks, the smaller the resulting texture, and the more information needs to be compressed, leading to increasingly more objectionable blocky artifacts (most noticable in the corners of the image).

preserves *random access* to the pixels. In other words, for any given pixel at location (x, y) in the image, the compressed respresentation specifies a function $f : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ that maps this location to a sequence of bits shorter than the original pixel representations. These bits are read by the graphics hardware and sent through fixed-function decoding units to produce resuling pixel values.

Texture compression allows textures to be represented in a compact representation during transfer from the CPU to the GPU, although they are not as efficient in compressing textures for disk or network storage. The dedicated GPU decoding hardware facilitates rendering by providing fast access to pixel values directly from the compressed format. However, these random access requirements restrict the total compressibility of the textures. Certain image compression algorithms such as JPEG are able to exploit redundancies across neighboring pixels to reduce the amount of bits required to represent low-detail regions. The more redundancy there is in the pixel values, the more that can fit within a given amount of memory, such as a low-level memory cache. A hardware decoder for such a format would require multiple memory accesses that are expensive in terms of both power and latency (Nixon et al., 2014). Although these limitations may be inconsequential for a certain class of images as described in Chapter 4.3.3, compressed textures represent a fixed number of pixels using a fixed number of bits. This creates a dichotomy where solid colors are 'compressed' using the same number of bits as random noise. With only one memory lookup, to achieve any amount of compression, these formats must be lossy in order to target the common use case rather than the worst-case. As shown in Figure 1.1, the result is often a trade-off between size and data loss that dominates the design of these compression formats.

1.2 Encoding Compressed Textures

Compressed textures are typically stored by having fixed-size blocks of pixels compressed down to a fixed number of bits. The block dimensions are usually chosen to be roughly square in order to map well to access patterns during rendering. In other words, if the GPU requests a certain pixel to render from a texture, there is a high chance a neighboring pixel will be requested as well. This block structure, combined with the overall variability of image intensity values contained in a block, presents a difficult problem for the compressor. The fixed size of the compressed representation implies a significantly smaller set of blocks that can be represented exactly. Hence, the compressor must choose the compressed representation whose reconstructed block of pixels most closely matches the original block. As an example, suppose a compression format encodes 4×4 blocks of pixels as 64 bits. At 24 bits per pixel, this structure implies 2^{384} total possible blocks are restricted to 2^{64} possible compressed representations.

In order to choose the proper compressed representation for a given block of pixels, texture compressors typically are incapable of using an exhaustive method. In the previous example, there are approximately 1.84×10^{19} possible values *per-block*. For a 1024×1024 texture, this implies more possible values than molecules in a typical glass of water! Hence, most algorithms tend to approximate the best compressed representation per-block by using ad-hoc approaches with various heuristics that map well to the given representation.

Because compressed representations are designed for fast decompression, slow compression times were originally acceptable (Beers et al., 1996). However, in recent years, the amount of texture data used in interactive graphics applications has exploded. In particular, mobile UIs are increasingly being driven by GPUs requiring many re-loads of texture data (Google, 2016). For this reason, many of the billions of photographs taken each day and uploaded to social media sites such as Facebook and Twitter may need to be compressed into formats amenable to rendering on the GPU. Current game engines spend up to 80% of their asset processing time on compression of raw texture data (Chen, 2016). In order for application developers and texture artists to properly iterate over their data, compression algorithms must be relatively fast. Otherwise, optimizing for proper compression ends up causing significant bottlenecks in the overall application development pipeline.

Many compressed texture representations follow a similar structure. A given block of $N \times M$ pixels is represented using two colors and b additional bits per pixel (Delp and Mitchell, 1979). These two

colors, known as *endpoints*, are used to create a palette with 2^b colors using linear interpolation in color space. The additional b bits per pixel are then used as palette *indices* for selecting the pixel color for each block. An encoder is then tasked to determine both endpoints and indices for each input block of pixels. To vary the amount of detail in the final representation, the number of bits allotted to each endpoint and index is usually variable across compression formats.

Recent formats have expanded this basic idea to give additional detail. Some formats bilinearly interpolate the palette endpoints across blocks during decompression to give specialized per-pixel endpoints (Fenney, 2003). Others partition the block into multiple different subsets of palettes (OpenGL, 2010; Nystad et al., 2012). Due to the large number of possible partitionings, most formats limit the number of available partitions to a subset that provide the most benefit to common image features. Each of these different features creates additional complexity to the encoders that determine the final compressed representation. In particular, choosing a particular partitioning of a block to preserve the most detail is usually the most time consuming component of any encoder (Krajcevski et al., 2013).

1.3 Thesis Statement

The performance and development of interactive graphics applications can be improved through the fast computation of compressed texture representations that are either decoded or used directly with existing graphics hardware.

This statement reflects a variety of improvements developed to better facilitate the use of compressed textures from the perspective of application developers. In general, we aim to investigate the main performance bottlenecks of common encoding algorithms and eliminate them by relaxing the constraints on the search space of possible encodings. With the fast encoding algorithms, we also show that current texture compression algorithms can be improved in terms of storage space by providing additional features to target images with mixed amounts of detail. Finally, we show that the disk storage of these compressed formats can be improved by using an additional layer of compression that is amenable to fast decompression on the GPU.

1.4 Main Results

The main focus of this dissertation is to discuss how faster encoding algorithms lead to a variety of improvements for compressed textures. By developing accelerated methods for compressing textures, we can use the encoders to investigate benefits of other compressed texture methods, and use 'optimal' encodings as baselines for introducing additional error. By having this baseline, we can quickly determine the implications of making changes and additions to new compression methods and quickly determine their efficacy.

1.4.1 Accelerated Texture Compression

The first algorithm focuses on the general problem of encoding low frequency signal-modulated (LFSM) compressed textures such as PVRTC (Fenney, 2003). This method bilinearly interpolates the per-block endpoints in order to generate per-pixel endpoints defining a palette to choose from. This bilinear interpolation implies that an endpoint stored in a 4×4 block influences the final color of a 7×7 block of pixels. To search for these endpoints, we find the local minimum and maximum intensity values in this 7×7 block and average them. Using this technique, textures can be encoded into LFSM formats at 3X the speed of prior compression algorithms without any significant loss in quality, as described in Section 4.1.

More recent formats, such as BPTC and ASTC, have much more complicated compressed representations. These formats define a way to choose from a set of predefined block partitionings that provide separate palettes for each subset of pixels (OpenGL, 2010; Nystad et al., 2012). Prior state of the art encoders would use an exhaustive search of each of the available partitionings. This exhaustive search amplifies the already difficult problem of finding a proper palette for a given set of pixels as it requires a separate palette search for each possible subset. Approximations to the appropriate palettes for given partitionings provide similar results to doing a full search with respect to PSNR. These approximations, taken from real-time texture encoding algorithms, provide orders of magnitude speed-ups for partition-based compression formats as described in Section 4.2.2.

Using approximations to the subset palettes still requires searching the entire space of block partitionings. In order to avoid this problem, a metric is used to compare partitionings against against one another. In doing so, an 'optimal' partitioning can be determined for all blocks of pixels in an image

by performing a global segmentation of image features. The boundaries of each segmentation label define the 'optimal' partition subset boundaries in a given block. The available partitioning closest to the 'optimal' choice can then be used to represent the block. Furthermore, all available blocks can be preprocessed into a VP-tree for $O(\log n)$ query given an 'optimal' block. This method provides an additional 3X speedup over prior methods as described in Section 4.2.3.

1.4.2 Applications of Interactive Compression Algorithms

In the extreme case, real-time compression algorithms can be used as an intermediate step in common rendering tasks. In particular, textures that are generated on-the-fly may benefit from real-time texture compression if the encoding is performed fast enough that the gains from uploading a smaller texture can be realized. To do this, the approach must be application-specific and tailored to the compressed format. These benefits are realized even more on mobile devices where the CPU-GPU bandwidth limitations are even greater.

One application that we investigate is the generation of coverage information in GPU-based 2D renderers. Coverage information is the per-pixel value that describes how much that pixel is covered by rendered geometric primitives. For certain primitives, such as lines, triangles, and points, these values can be analytically derived on the GPU. However, for other primitives, such as closed-loop quadratic and cubic curves that are used as the basis for many font representations, the computation may be much more difficult. To deal with this problem, some GPU-based renderers compute the coverage information using a traditional CPU rendering approach. The subsequent *coverage mask* is stored in a texture and uploaded directly to the GPU for rendering. To accelerate this process, the coverage mask can be generated directly into a compressed format circumventing both the full CPU write and the CPU-GPU bandwidth. Our method gives up to a 2X speedup over traditional GPU-based methods on certain benchmarks and up to a 9:1 savings in GPU memory as shown in Chapter 2.6.

1.4.3 Storage of Modern Compression Formats

Although texture compression formats are fixed-rate, recent formats allow variable global block sizes giving application developers a size versus compression quality trade-off. Different block sizes can be decoded in hardware by the same functional unit, effectively reducing the amount of different formats needed on a GPU. However, this limitation still implies a global fixed compression size for a

given texture. Hence, low-detail areas and high-detail areas will not have different levels of compression. By adding a single additional memory lookup, we can dynamically choose the compressed block size for a given set of pixels. In this way, we present a method for variable bit-rate compressed textures that requires a very small addition to the addressing unit of hardware decoders. In Chapter 4.3.3, we reduce the storage size of certain textures by up to 2X while maintaining acceptable compression quality.

In addition to memory storage, the disk storage of compressed textures is critical to the overall performance of interactive graphics applications. Loading textures from disk is usually more time consuming than uploading them to the GPU. However, texture compression formats are still about four times larger than image compression formats on disk. This becomes an even bigger problem for textures that are used with GPU-driven user interfaces or for applications that request texture data over the network, as image databases are growing at unprecedented rates. In order to address this issue, the endpoints and indices of compressed textures can be stored and processed independently. In particular, endpoints are treated as low resolution images and stored with image compression techniques while indices are approximated using a common dictionary. Along with a GPU-enabled decoding scheme, compressed textures can be stored on disk and uploaded all the way to the GPU for decoding, saving both CPU-GPU bandwidth and disk storage while maintaining acceptable compressed quality, as described in Chapter 5.4.

1.5 Thesis Organization

This chapter has given an overall introduction to each of the topics discussed in this thesis. Chapter 1.5 discusses the prior work in texture compression and gives a general background of the available compression formats in use today. Chapter 2.6 presents an example of fast texture compression algorithms boosting the rendering performance of GPU-based 2D rendering. With an emphasis on mobile devices, this chapter makes the case for fast compression algorithms. Chapter 3.5 gives an overview of three algorithms that provide fast encoding for a variety of texture compression formats. Given a fast encoder, Chapter 4.3.3 shows how to reduce the compressed size of mixed-detail compressed textures by allowing an additional level of indirection to determine an adaptive block size at runtime. Finally, in Chapter 5.4, we show a way in which compressed textures can be stored on disk to increase the streaming bandwidth while adding additional CPU-GPU benefits. A discussion of the current GPU architectures and what

features could be added moving forward to provide additional benefits and flexibility are discussed in Chapter 6.5. Conclusions and final remarks are given in Chapter 7.5.

CHAPTER 2: TEXTURE COMPRESSION VERSUS IMAGE COMPRESSION

Traditional image compression algorithms such as PNG and JPEG assign bits to pixels with respect to the amount of detail in an image. In other words, images that contain many common pixels or contain neighborhoods of similar pixels require fewer bits in the final encoding. Hence, traditional image compression offers *variable-rate* compression depending on the details of the image. Often, in order to remove some redundant detail, the raw RGB values usually undergo one or more transforms. For example, JPEG uses the discrete cosine transform to condense the information content of blocks of pixels (Wallace, 1992), and its successor, JPEG-2000, uses one of two wavelet transforms depending on whether or not lossless encoding is desired (Skodras et al., 2001). Similarly, colors are usually transformed into color spaces that condense information into a single channel in order to increase overall compression efficiency. One such example is the lossless conversion of 8-bit RGB to a colorspace such as YCoCg (Malvar et al., 2008).

Given an *alphabet* of *symbols*, *entropy encoding* is the practice of converting a sequence of symbols into a sequence of bits by assigning bits to symbols based on the probability of each individual symbol appearing in the given sequence. Entropy encoding techniques, such as Huffman (1952), arithmetic (Rissanen and Langdon, 1979), and ANS (Duda, 2013) encoding are the basis for many image compression formats (Buccigrossi and Simoncelli, 1999). Without additional metadata, the variable number of bits per symbol forces serial decoding algorithms for any given encoding algorithm. A variable number of bits per encoded symbol requires all previous symbols to be decoded prior to any specific symbol, although some attempts have been made to increase the scalability of these techniques to multiple processors (Klein and Wiseman, 2003; Kim and Park, 2007).

Texture compression, as discussed in Section 1.1, must support random access to the pixel data. In this context, DCT or wavelet transforms are usually superfluous without the entropy encoding stage. Classically, this requirement has led to many simple-to-decode algorithms for representing pixel data in a fixed number of bits. In this chapter we will provide an overview of the first available fixed compression

rate formats, the current state-of-the-art compression formats, and a few notes on the methods for deriving the compressed representation from a given input image.

2.1 Fixed Rate Image Compression

The origin of most fixed rate formats follow the format of a technique known as *Vector Quantization* (VQ). This technique, originally used in signal processing, represents a sequence of n values (or *codewords*) by storing a dictionary of size $k < n$ and replacing each codeword with the appropriate index into the dictionary. Lossless compression using VQ can actually require more data than storing the original set of codewords when all of the codewords are distinct. However, for data that does not need to be stored without loss, a few representative dictionary entries can be used to approximate each of the original codewords. If the data represents a quantity perceived by humans, such as audio or image data, then the loss may be acceptable without losing the perceived quality.

Fixed-rate texture compression formats are a variation of this technique. If a sequence of pixels originally stored in n bits is compressed down to a fixed $k < n$ bits, we are effectively defining a dictionary of size 2^k. The specification of this dictionary is implicit in the decoding algorithm for the texture once it has been compressed. In other words, since two sequences of k bits are decoded to the same original n bits, the 'dictionary' is defined by the logic that does this decoding. This is counter to the original formulation of VQ in which the dictionary is stored along with the k-bit entries. Hence, the size and quality of any given compressed texture representation is specified by how well its 'dictionary' is defined.

Today's fixed-rate encoding schemes mostly follow the Block Truncation Coding (BTC) technique introduced by Delp and Mitchell (1979) for compressing eight-bit grayscale images. In this format, 4×4 blocks of pixels are encoded using two eight-bit grayscale values and a per-pixel bit selecting choosing one or the other. Each block is therfore compressed into two bytes offering two *bits per pixel* (bpp). Various generalizations of this idea have been proposed by Nasrabadi et al. (1990); Fränti et al. (1994).

Campbell et al. (1986) extended the idea of BTC to include color by introducing a 256-value color palette instead of grayscale values. By specifying a single palette for the entire image, the use of eight-bit entries into this palette compresses images down to 2 bpp. However, the additional memory lookup made

it difficult to render from directly on GPUs. This method, known as *Color Cell Compression* (CCC) was used predominantly in the architecture presented by Knittel et al. (1996).

The random access properties of fixed-rate compression imply a lossy compression algorithm. However, texture mapping hardware can quickly compute an address to the underlying texture data. Similar to BTC and CCC, typical fixed-rate compression formats represent $N \times M$ blocks of pixels in some fixed number of bits. Many graphics architectures were proposed using similar compressed texture representations (Torborg and Kajiya, 1996; Knittel et al., 1996; Beers et al., 1996).

2.2 Graphics Architectures for Compressed Textures

Knittel et al. (1996) introduced a graphics architecture that supported compressed texture representations via CCC introduced by Campbell et al. (1986). In particular, Knittel et al. (1996) describe the use of a high-throughput multi-bank memory system that could handle multiple addresses corresponding to neighboring pixel values. An additional hardware decoder was used to obtain close-to-interactive decompression speeds by requiring fewer bytes in memory and allowing different page table configurations. Torborg and Kajiya (1996) take a similar approach by preserving the compressed format in memory, although the details are not disclosed.

The algorithm provided by Delp and Mitchell (1979) was able to meet the necessary requirements of texture decompression hardware as described in Section 1.1, namely the need for fixed-rate addressing and an algorithm that can be implemented in hardware. Although fixed-rate compression has been used for almost three decades (Chandler et al., 1986)(Economy et al., 1987), texture compression in graphics architectures based on Vector Quantization (VQ) was formally presented by Beers et al. (1996). In this seminal paper, Beers et al. argue for four main tenets of texture compression: fast hardware decoding speed, preservation of random-access, compression rate versus visual quality, and encoding speed. The presented argument claims that encoding speed could be sacrificed for gains in the other three, but the need for fast encoding algorithms was recognized even then, where a Generalized Lloyd's Algorithm (Gersho and Gray, 1991) was used to produce fast non-optimal encodings.

Additionally, there is extensive work on designing texture mapping hardware to provide high bandwidth to texture memory and low latency access (Hakura and Gupta, 1997)(Wei, 2004). Inada and McCool (2006) proposed texturing hardware that stores textures in a binary-tree to allow for

efficient pixel access. This architecture stored textures in a binary tree that allowed rendundancies in blocks to be eliminated by a clever differencing scheme. However, these methods require expensive addressing schemes and do not leverage existing compressed texture hardware. More recently, commercial mobile GPU manufacturers have support for surface or render target compression (nVidia, 2015, 2014; Imagination, 2016). A discussion of a compressed texture representation that proposes a simple addressing scheme but maintains some of these benefits is discussed in Chapter 4.3.3. As few hardware and algorithmic details are public about the proprietary architectures, it is difficult to compare the performance of commodity GPUs.

2.3 Modern Texture Compression

Along with the advent of commodity graphics hardware, a variety of texture compression formats emerged. These formats have largely fallen into two broad classes of compressed formats. The first class, known as *endpoint* compression formats, are extensions of the original per-block paletted formulation introduced by Delp and Mitchell (1979). The second, known as *tabled* compression formats, use a set of predefined tables that describe the compressed block. In this section we will discuss both formats in detail.

The basis of all endpoint compression formats are two low-precision RGB *endpoints* per block that generate a palette of colors by linear interpolation. Along with these two low-precision colors, a per-pixel palette *index* is stored to recreate the final pixel color, just as in the original BTC (Delp and Mitchell, 1979; Fenney, 2003; OpenGL, 2010; Nystad et al., 2012). Among the first endpoint compressed texture formats available on commodity graphics hardware was S3TC (also known as DXT and BC1), introduced by Iourcha et al. (1999). In this format, 4×4 blocks of pixels are represented using two 16-bit values and 16 two-bit values. The two 16-bit values are each interpreted as RGB endpoints with five bits for the red and blue channels and six bits for the green channel. These endpoints generate a four-color palette for the block via linear interpolation. The following 16 two-bit values index into this palette to recreate the final pixel values. Recently algorithms have been developed that provide additional quality over S3TC by either cleverly using the compression format, such as storing wavelet coefficients in the compressed texture and reconstructing pixels at run-time (Mavridis and Papaioannou, 2012), or by weighing the importance of endpoints based on the input (Krause, 2010).

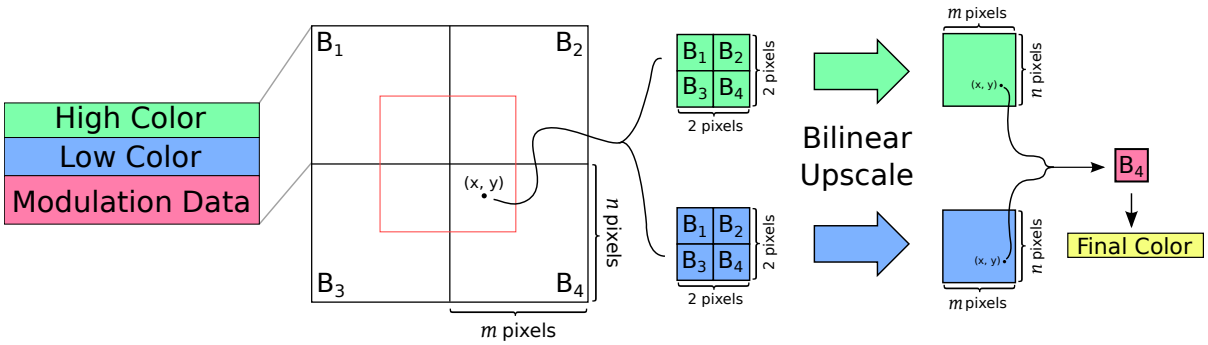


Figure 2.1: Representation of low-frequency signal modulated texture data (Fenney, 2003). The compressed data is stored in blocks that contain two colors, high and low, along with modulation data for each texel within the block. When looking up the color value for texel at location (x, y), information is used from the blocks whose centers are the four corners of a rectangle that encompass the texel. The high and low colors are separately used to generate two block sized images using bilinear interpolation, and then the modulation value of the texel's corresponding block is used in conjunction with these upscaled images in order to produce the final color.

Fenney (2003) later described decompression hardware that took advantage of the worst case of S3TC during texture filtering. In this case, the graphics hardware must generate a color by bilinearly interpolating the color between four pixels in a 2×2 square. If pixels all correspond to different blocks, four compressed block lookups need to be performed regardless. In his approach, known as Low Frequency Signal Modulated Texture Compression (LFSM), the RGB endpoints of neighboring blocks are bilinearly interpolated themselves. This gives each pixel a more accurate per-pixel endpoint palette, and as a result, smoothed gradients are better preserved using LFSM. Figure 2.1 shows an example of the LFSM pipeline.

The second class of texture compression formats is known as *tabled* compression formats. These formats store a single color per block of $N \times M$ pixels and use per-pixel offsets in the luminance direction. These offsets are stored in tables defined by other encoded properties of the block. Ström and Akenine-Möller (2004, 2005) introduced the first tabled formats as PACKMAN and iPACKMAN using 2×4 and 4×2 sized blocks. Later, Ström and Pettersson (2007) improved upon iPACKMAN by exploiting unused encoded representations to provide additional detail, known as ETC2. In particular, they use undefined ETC encodings in 4×4 blocks. The base color of one of the underlying blocks 2×4 can be used as an offset from the other. Additional bits are used to select from predefined offset patterns that describe which base color to choose from when dealing with the offset tables.

ETC2 was the first format to provide partitioning in a given compression block. Compressors could effectively choose between a 2×4 or 4×2 partitioning of a 4×4 block by setting the appropriate bit (Ström and Pettersson, 2007). Recently, higher quality endpoint formats, such as BPTC (OpenGL, 2010) and ASTC (Nystad et al., 2012), have emerged that split fixed blocks into partitions that are encoded separately. Block Partition Texture Compression (BPTC, a.k.a. BC7, BC6H) was introduced as a high quality compression format that partitions a 4×4 pixel block and compresses each subset separately using the technique from S3TC (OpenGL, 2010). Additionally, BPTC supported the idea of *p-bits*: low-order bits that are shared across all values in one or more endpoints. BPTC provides eight compression modes per 4×4 block, each which varies the amount of precision given to the endpoints and indices. Additionally, of these eight modes, five support partitioning the block. Modes zero and two partition the block into three subsets, and modes one, three, and seven partition the block into two subsets where each partition is specified by a four or six bit partition index (OpenGL, 2010). Similarly, Nystad et al. (2012) introduced Adaptive Scalable Texture Compression (ASTC), a diverse new format that supports partitioning similar to BPTC. Additionally, ASTC allows each block to vary the number of bits allotted to both endpoints and pixel indices to treat a variety of image types. In ASTC, partitions are specified using a ten bit partition ID, and are determined by evaluating a function that takes this ID, the number of partitions, and the texel location as arguments (Nystad et al., 2012). ASTC also supports multiple global block sizes with the same decoding hardware providing ratios from 0.89bpp up to 8bpp.

There are other endpoint based texture formats, such as FTC, that are not currently supported in hardware (Krause, 2010). Similarly, there were formats similar to S3TC, such as FXT1, that never gained widesperead acceptance due to market forces (OpenGL, 2000). A variant that uses many more endpoints to generate a larger palette was also introduced by Levkovich-Maslyuk et al. (2000). Another approach, introduced by (Pereberin, 1999), aims to replace the 4×4 blocks of DXT with a simple wavelet decomposition of each block thereby also storing the mip-maps for each texture. For a single mip level, this approach was not effective, but it provided better compression than DXT for the three mip levels that it supported. Finally, Ivanov and Kuzmin (2000) shows that certain classes of images can be compressed effectively if we select colors from neighboring blocks when generating the DXT palette.

Other texture compression formats have been proposed that are not currently supported in modern graphics hardware. For example, formats for high dynamic range textures to complement similar schemes developed in recent years (Roimela et al., 2006; Munkberg et al., 2008; Sun et al., 2008). Some tabled

formats target specific use-cases for textures such as the grayscale variations introduced by Wennersten and Ström (2009). Additionally, there has been approaches to unconventionally using compressed textures and store transformed data to be reconstructed at runtime. As an example, Waveren and Castaño (2007) use DXT5, a variant of DXT that includes a separate BTC encoded alpha channel, to store YCoCg transformed endpoints to get better quality out of the compressed representations.

2.4 Measuring Compression Quality

For any texture compression algorithm, the main aspect for which we optimize may be different. For some algorithms, compression speed is more important while for others storage size takes priority. Because texture compression formats are inherently lossy, any algorithm must show that the level of compression quality remains roughly the same. To do this, most contributions use the *peak signal-to-noise* (PSNR) measurement of the reconstructed image as defined by

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^3 \times w \times h}{\sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2 \right)} \right)$$

(Salomon and Motta, 2010). While there is still much discussion about how to properly evaluate the differences between two images, PSNR provides a strict energy metric that determines the total difference in pixel values.

Although PSNR is the most commonly used metric, the structural similarity image metric (SSIM) has been introduced as an alternative way to measure the perceptual differences between two images (Wang et al., 2004). With this metric, Wang et al. (2004) showed that for a fixed PSNR, the image can be manipulated to an almost imperceptible level. By treating neighboring pixels as a statistical distribution around the current pixel, Wang et al. (2004) showed that we can compare two images in a way that relates much better to the human visual system. One drawback, however, is that the metric only considers single-channel images and cannot support the correlation between multiple colors.

In modern day applications, textures are not used strictly for visual appearance. For example, they may be used for varying the material properties used for lighting calculations, and hence any perceptual or application-specific metric must be taken into account by the developer. To this effect, Griffin and Olano (2014) show that the best way to actually compare compressed texture approaches is to test the final rendered image using the textures. The resulting metric is a combination of PSNR and the SSIM of

the scene after converting the final rendered images into grayscale. In this thesis, we will focus only on PSNR to evaluate texture differences.

2.5 Encoding speed

Beers et al. (1996) introduced the idea of compressing textures using vector quantization while maintaining relatively good quality and decoding speed. They also claimed that compression speed was not an issue because it could be performed offline. However, with the need for fast iteration times during content creation and multi-platform applications such as Google Maps, compression speed is becoming a major issue in the design of texture compression algorithms.

Fast texture compression has been studied comparatively less than new compression methods. J.M.P. van Waveren was the first to develop a real-time compression scheme for DXT1 (Waveren, 2006a). His technique compressed a 4×4 texel block by using the endpoints of the axis aligned bounding box diagonal in three dimensional RGB space. This technique not only proved effective, but also opened the doors for similar techniques for normal maps or YCoCg encoded textures (Waveren and Castaño, 2007) (Waveren and Castaño, 2008). Because formats such as DXT are easily parallelizable, many compression techniques also leverage GPUs to generate very good results very quickly (Castaño, 2007).

Although many popular offline encoders exist for S3TC (Bloom, 2009; Donovan, 2010; AMD, 2008; Brown, 2006), initial compressors for ASTC and BPTC could be prohibitively slow (Donovan, 2010). Recent work has focused on increasing the speed of BPTC compression algorithms and this remains an active area for research (Dufresne, 2013). Additionally, although the flexibility of the ASTC compression format allows a large quality versus compression size trade-off, developing real-time encoders for ASTC remains difficult (Oom, 2016).

2.6 Improved Compressed Texture Storage through Image Compression

Many of the motivations for image compression, such as improved storage on disk and transmission over the network, are still desired in the domain of compressed textures. Recent applications are becoming increasingly connected where the resources used during rendering are not stored locally. In these situations the storage size of textures is becoming an increasing concern, as this data must travel over the network and becomes a bottleneck for latency sensitive applications. The main benefit that

compressed textures eschew for random access is the notion of variable bit-rate compression, or using more bits where there is more detail. A few approaches have been proposed that attempt to regain that property. Additionally, other approaches attempt to apply an additional layer of compression to eliminate the remaining redundancy of the compressed textures.

The entropy encoding stage of image compression algorithms is usually an inherently serial procedure that is difficult to parallelize. However, Inada and McCool (2006) introduced a variable bit-rate format for storing sparse textures with a lot of redundancy. They store their textures in an efficient binary tree that admits hardware decoding. Additional variable bit-rate compressed texturing solutions have been proposed by Olano et al. (2011) by using a range coder and decompressing them in a shader program on the GPU. This compression scheme uses texture mipmap levels to predict the higher resolution colors and encodes the prediction error. Although this scheme generates small textures, providing savings in bandwidth over the shared memory bus, the textures are still ultimately stored at full resolution in GPU memory. Additionally, the amount of bandwidth used to fetch texel data has a direct correlation with power consumption, which is especially a concern for embedded devices, such as mobile phones.

In order to reduce the amount of processing needed at runtime, most applications store these compressed texture formats (e.g., DXT, ASTC, ETC) on disk as-is (Pohl et al., 2014). Recently, however, there has been progress into targeting both on-disk compression and in-memory compression. These *supercompressed* textures provide an additional layer of compression to be decoded on the CPU on top of the already compressed GPU formats. Ström and Wennersten (2011) proposed a scheme for further compressing ETC2 textures. In their formulation, they predict the final pixel colors in order to predict the per-pixel indices for the given block. They observe a gain of up to 3X in some cases over existing ETC2 textures. A different approach, known as *Crunch*, developed by Geldreich (2012), uses a Huffman-encoded dictionary of endpoints and index blocks to further compress a DXT-encoded texture. Blocks are stored in order using the differences between successive dictionary entries. Both of these methods decode the compressed texture on the CPU before sending them to the GPU. In Chapter 5.4, we present an approach for supercompression that preserves bandwidth using GPU decompression similar to Olano et al. (2011), but whose decompressed output maintains the benefits of DXT, such as smaller memory representation and increased bandwidth during rendering.

CHAPTER 3: CASE STUDY: COVERAGE MASK GENERATION [1]

One of the classic problems in computer graphics is the discretization of continuous functions used to display objects at a finite resolution. Improper discretization may lead to aliasing artifacts from insufficient sampling. In order to alleviate these artifacts, different techniques have emerged for computing proper discretizations (Barros and Fuchs, 1979)(Lane and M. Rarick, 1983). When rasterizing geometric objects, the main difficulty is determining what percentage of a pixel is covered by the screen-space projection of the object. This information, once calculated, can be stored in an image known as a *coverage mask*. Coverage masks are usually stored as eight-bit grayscale images and can be used in a variety of different ways in order to speed up the rendering of geometric primitives, including caching (Fiume et al., 1983) and GPU based rendering of 2D curves (Google, 2016).

Pixel coverage remains an instrumental part of computer graphics. There are many applications where coverage masks are useful, from culling (Zhang et al., 1997) to visibility determination for more efficient lighting (Kautz et al., 2004). In this chapter, we mainly focus on coverage masks used in rendering non-convex piece-wise two-dimensional cubic and quadratic curves, or *paths*, with anti-aliasing (Figure 3.1). These curves are used in a majority of vector graphics data, most importantly as the basis for resolution-independent text rendering using different fonts and sizes. These coverage masks, generated at run-time from network data such as web pages, are used billions of times on a daily basis (StatCounter, 2014). From a sampling of over 750,000 web pages, we have observed that 51% draw arbitrary paths of which 19% are anti-aliased requiring dynamically generated textures. Of the paths that require coverage information, most of the web page rendering time is spent drawing the coverage mask of the path on the CPU prior to uploading it to the GPU.

In this chapter, we show that coverage masks generated at run-time by the CPU can be compressed efficiently for GPU-based rendering with little loss in rendering fidelity. We present a way to augment the scan conversion process of non-convex path rendering to directly output compressed textures for use on the GPU. We demonstrate encoding into a variety of different compression formats in order to show

[1] Much of this chapter appeared as a paper by Krajcevski and Manocha (2016)

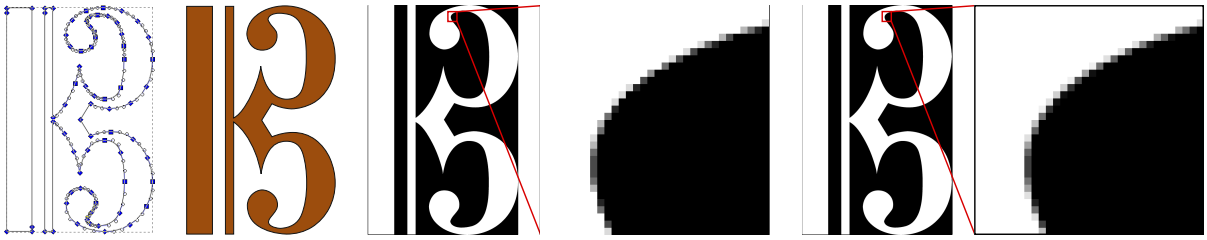


Figure 3.1: *(left)* The piece-wise anti-aliased cubic curve used as input. *(middle-left)* The final rendered curve. *(middle-right)* The uncompressed coverage mask passed to the GPU to determine the amount each pixel is covered by the curve. *(right)* The compressed coverage mask using our method. On the far right is a zoomed in comparison of the compressed and uncompressed masks. Although only a few pixels differ, using our method, these masks are compressed in real time and save time and memory during the rasterization of these curves.

applicability to a widespread range of commodity graphics hardware. In particular, we show that even with general 32-bit hardware, efficient coverage mask compression can be performed to target the DXTn, ETC, and ASTC texture compression formats (Iourcha et al., 1999)(Ström and Pettersson, 2007)(Nystad et al., 2012). Finally, we demonstrate a speedup of up to 2X in rendering speed using compressed coverage masks on current mobile platforms (e.g. tablets and smart phones). This savings in rendering speed is in addition to the GPU memory gains of 2X up to 9X depending on the compression format. Our method is integrated into the Skia[2] two-dimensional rendering library (Google, 2016). This library is the rendering backbone in the popular Google Chrome and Mozilla Firefox web browsers currently being used by billions of people (StatCounter, 2014). Additionally, we test our results against a suite of benchmarks, correctness tests, and web-page data. Overall, our approach aligns well with the current hardware and software trends. The mobile GPU market is growing at a considerable rate with more than a billion sales per year (Shebanow, 2014). To address this trend and develop higher performance on mobile GPUs, hardware vendors are developing more aggressive compression formats that are designed specifically for GPUs (Nystad et al., 2012). In particular, energy savings during rendering are becoming more important. Using a few extra CPU operations in order to decrease the texture bandwidth by 2-3X likely produces significant energy savings for texture-heavy mobile applications. Texture memory accesses are almost three orders of magnitude more expensive than standard ALU operations (Shebanow,

[2] `https://www.skia.org/`

2014). Our method for compressing coverage masks leverages these trends and becomes increasingly useful as hardware advances.

3.1 Background

One of the major problems in computer graphics has been to determine the amount that geometry covers a given pixel during rasterization (Barros and Fuchs, 1979)(Fiume et al., 1983). This problem, also known as *pixel coverage*, is used to reduce aliasing artifacts caused by the discrete nature of our display devices and memory layouts. More recently, coverage masks have been used for more than simply anti-aliased rasterization. Zhang et al. (1997) use occlusion maps, a variation of coverage masks, to quickly cull geometry during the rendering of large scenes. Kautz et al. (2004) use coverage masks to cache hemispherical visibility information in order to perform efficient self-shadowing of objects. Coverage information has also been used to accelerate shading operations in the GPU pipeline, although these methods are more suited to hardware implementations than software (Aila et al., 2003)(Fatahalian et al., 2010).

Coverage masks are used extensively to render 2D images from geometric primitives. In particular, coverage information is necessary when rasterizing anti-aliased polygons independent of the color and shading information. In order to render these polygons, first the pixel coverage mask is generated, and then the color of the polygon is modulated by the intensity of the pixel in the coverage mask. This technique is used in the 2D rendering library Skia (Google, 2016) for GPU rasterization of non-convex anti-aliased paths.

Resolution-independent rendering is important for many objects in graphics such as the arbitrary cubic and quadratic curves used to represent glyphs in most modern fonts. Until recently, these curves have been rendered using software rasterization algorithms. Given the recent advances in GPU development, there has been considerable groundbreaking work to use GPUs to perform resolution-independent rasterization (Loop and Blinn, 2005)(Kilgard and Bolz, 2012)(Qin, 2009). As pioneers in this work, Loop and Blinn (2005) devised a method to rasterize Bézier curves by assigning values to the texture coordinates of triangles derived from the control points of the curve. These values were used to calculate the distance from the curve in the given triangle, which was used for proper anti-aliasing. Kokojima et al. (2006) improved the efficiency of this method by exploiting the stencil buffer. Qin (2009) presented a method to exploit the texture storage of a graphics processor to store curve information using approximate

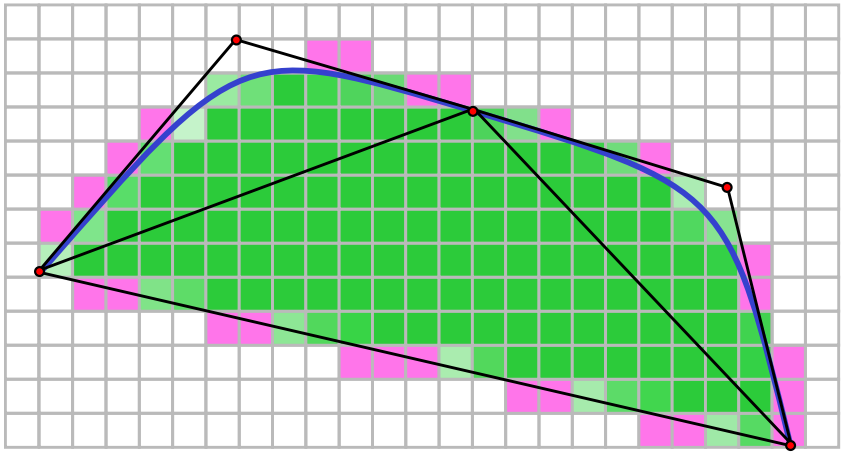


Figure 3.2: A piece-wise quadratic curve is filled with green using the Loop-Blinn method. The pixels (pink) whose centers are not covered by the triangles circumscribing the curve will not be drawn if the GPU is not using a hardware anti-aliasing method. For power constrained GPUs, such as those on mobile devices, MSAA is prohibitively expensive due to the large number of fragment shader invocations. When the curve is non-convex, it is often more correct to default to software rendering of the pixel coverage in these situations.

circular arcs. Finally, Kilgard and Bolz (2012) describe an approach that transmits control points directly to the GPU to render the curve. Although this method renders vector graphics very quickly, it requires additional proprietary hardware features. Further approaches using signed distance fields have been used by Green (2007) for artist generated vector graphics.

Despite recent advances in using GPUs to accelerate vector graphics rasterization, certain classes of vector graphics still remain slow on mobile hardware. Of the techniques mentioned, the Loop-Blinn method is among the fastest techniques for rendering resolution-independent vector graphics from arbitrary path data. The GPU-based method introduced by Kilgard and Bolz (2012) builds upon the Loop-Blinn method by implementing a conservative approach to determining coverage information in hardware. Most notably, as shown in Figure 3.2, for paths that generate smooth curves but are comprised of multiple control points, the triangles that conjoin quadratic and cubic pieces of a curve may not cover all necessary pixels. When these triangles are rasterized by the GPU, the centers of some pixels covered by the path may not be covered by the triangles. For GPUs that do not support hardware-based anti-aliasing, or where such anti-aliasing is too expensive due to power constraints, pixels that should have partial coverage from the path will not be drawn. This can cause aliasing artifacts when rendering curves whose details are on the order of a single pixel.

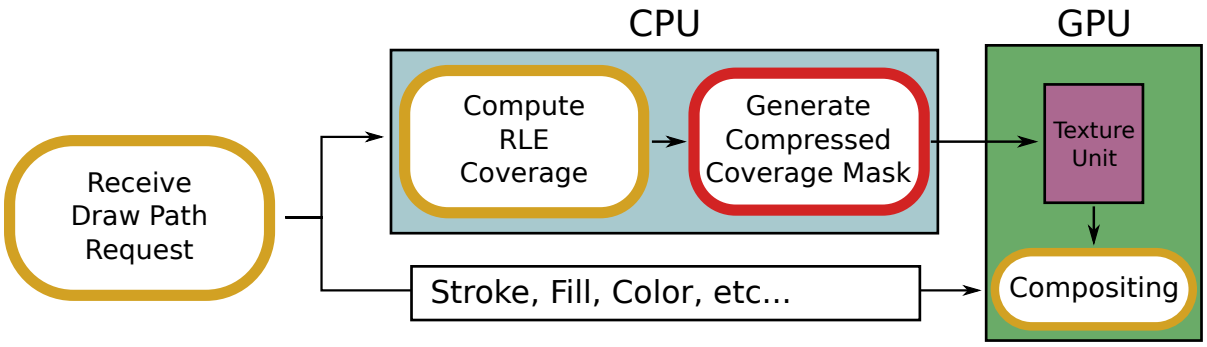


Figure 3.3: The different stages in GPU-based rendering of filled 2D regions using coverage masks. The only part that takes place on the GPU is the compositing. Our contribution in this modified pipelineis the stage outlined in red, where compressed textures are generated directly from the run-length encoded coverage information. In doing so, we avoid both writing a full resolution texture into CPU memory and uploading a full resolution texture to GPU memory, providing savings on both ends.

To support many different use-cases, the 2D rendering library Skia chooses different rendering paths dependent on the path being rendered. For non-convex paths without anti-aliasing, Skia approximates a path using line segments and then uses their endpoints as input to a triangle fan drawing both front and back facing triangles. Using the stencil buffer, pixels can be turned on or off based on whether they are inside or outside the path. However, line segments create significant aliasing artifacts during rendering, and this technique cannot be used for anti-aliased paths.

To perform anti-aliasing, in certain cases Skia uses the Blinn-Phong method followed by extruding the triangles along the normal to the path by the amount required to cover all of the pixels covered by the path. However, for general non-convex paths, this presents artifacts in areas where the extruded polygons of two different curves overlap leading to double-blending and incorrect pixel coverage. As a result, the GPU-based renderer in Skia draws the coverage information in software prior to uploading the resulting grayscale texture to the GPU for shading. This rendering algorithm used to support the use of GPUs can become a significant bottleneck during the rendering of anti-aliased concave paths (Google, 2016). In this chapter, we show that the grayscale coverage information can be efficiently compressed to a texture format thereby significantly increasing the speed at which it is uploaded to the GPU.

3.2 Compressed Scan Conversion

In this section we describe our technique for encoding the coverage information into a GPU-based compressed texture format. Given a piece-wise two-dimensional curve, or *path*, we augment the scan

conversion algorithm on the CPU for generating coverage information. Our formulation is based on the assumption that the time spent writing the encoded coverage information into a GPU-specific format can be recovered during the time it takes to upload the texture to the GPU. Even if the time saved by uploading a compressed representation is lost during the encoding step, we still gain memory savings from using compressed textures.

The input to our algorithm is a list of 2D curves defined using Bézier control points. From this list, our goal is to generate an accurate two-dimensional grid of pixels that best approximate the curve along with a specified *paint*. The paint determines the color and opacity of the pixels that are covered by the curve along with any other special operations such as anti-aliasing and gradient dithering. For pixels that are partially covered, they will be painted proportional to the amount that they are covered by the path. In a GPU-based rasterization pipeline, the coverage information is first generated and then used as a texture along with the paint to write to the framebuffer.

There are two operations commonly used for rasterizing these paths. First, the path may be *filled* such that a single color is painted within the bounds defined by the path. In this case, the coverage information in conjunction with the paint opacity is used to determine how much of that color should be blended with the background color. If the path is being rendered using the GPU, the coverage information must be uploaded as a texture prior to determining the final color and blending. The other operation, known as *stroking*, draws an outline of a given thickness along the path. In this case, the Skia library computes a new path along the outline of the stroke. Rendering this new path filled with the stroke color is identical to rendering the original stroked path. We restrict our formulation to non-convex paths. Convex paths can be efficiently drawn on GPUs by using a triangle fan in conjunction with the stencil buffer in a modified Loop-Blinn method described in Section 3.1 (Google, 2016).

The texture uploaded to the GPU is the image that stores the pixel coverage information. We proceed by first describing a variety of compression methods that we use to encode grayscale information on commodity graphics hardware. We then describe how we augment the scan conversion process to rows of compressed texture data.

3.2.1 Compression Formats

Due to the large schism of hardware support for various texture compression formats, our goal is to develop an approach that is portable between different GPUs. Decoding algorithms tend to be

```
uint32_t BytesToDXTnIndices(uint32_t x) {
  // Collect and invert high three bits
  x = 0x07070707 - ((x >> 5) & 0x07070707);

  // Set mask if any bits are set
  const uint32_t mask = x | (x >> 1) | (x >> 2);

  // Mapping: 7 6 5 4 3 2 1 0 -> 8 7 6 5 4 3 2 0
  x += mask & 0x01010101;

  // Handle overflow:
  // 8 6 5 4 3 2 1 0 -> 9 7 6 5 4 3 2 0
  x |= (x >> 3) & 0x01010101;

  // Result: 9 7 6 5 4 3 2 0 -> 1 7 6 5 4 3 2 0
  return x & 0x07070707;
}
```

Figure 3.4: C code for converting an integer storing four 8-bit values into four three-bit indices corresponding to the proper layout of a DXTn block. Using branchless code without multiplies or divides yields extremely fast and pipelined code on modern CPU architectures.

relatively simple because of the necessity of hardware-based implementations of GPU-encoded textures. Our encoding algorithm exploits this simplicity inherent in all compression formats. As described in Section 3.2.2, neighborhoods of pixels in coverage masks usually contain either fully transparent or fully opaque pixels. This allows us to precompute many of the parameters for our compression formats prior to the actual encoding. However, the reconstruction of the coverage information from these formats is necessarily lossy, due to the nature of the random access constraints. The following is a detailed overview of the algorithm applied to the DXTn, ETC2, and ASTC families of compression formats.

3.2.1.1 DXTn

In the DXT family of texture compression formats, introduced by Iourcha et al. (1999), 4×4 pixel blocks are encoded by storing two pixel values per block and a two-bit index per pixel. The two separate pixel values stored in the block generate a palette of colors from which the per-pixel index selects the final color. The palette is based on intermediate values chosen by linearly interpolating the two stored pixels. For coverage information, we use the DXTn format designed specifically for grayscale known as LATC, or Luminance-Alpha Texture Compression (also known as RGTC, 3DC, and BC4). This format supports two eight-bit grayscale values and sixteen three-bit index values per pixel for a total of 64 bits per block, giving a compression ratio of two-to-one for grayscale images. In order to reach the full range

```
uint32_t BytesToETC2Indices(uint32_t x) {
  // Three high bits: 0 1 2 3 4 5 6 7
  x = 0x07070707 - ((x >> 5) & 0x07070707);

  // Negate: 0 -1 -2 -3 -4 -5 -6 -7
  x = ~((0x80808080 - x) ^ 0x7F7F7F7F);

  // Add three: 3 2 1 0 -1 -2 -3 -4
  const uint32_t s = (x & 0x7F7F7F7F) + 0x03030303;
  x = ((x ^ 0x03030303) & 0x80808080) ^ s;

  // Absolute value...
  const uint32_t a = x & 0x80808080;
  const uint32_t b = a >> 7;

  // M is three if the byte was negative
  const uint32_t m = (a >> 6) | b;

  // .. continue absolute value:
  // 3 2 1 0 1 2 3 4
  x = (x ^ ((a - b) | a)) + b;

  // Add three to the negatives:
  // 3 2 1 0 4 5 6 7
  return x + m;
}
```

Figure 3.5: C code for converting an integer storing four 8-bit values into four three-bit indices corresponding to the proper layout of an ETC2 block. Similar to Figure 3.4, we perform the conversion using only bitwise operations and without expensive multiplies or divides.

of grayscale values, we store 0 and 255 as endpoints for our coverage mask. Due to the indexing scheme of DXTn, the mapping of coverage values to interpolation indices can not be directly copied from the high three bits of each coverage value. We first quantize each grayscale value to three bits such that their reconstruction into eight bits by bit replication minimizes the error from the original grayscale value. Once these three bits are computed, we must use a mapping from the quantized bits to the proper DXTn indices

$$0, 1, 2, 3, 4, 5, 6, 7 \rightarrow 1, 7, 6, 5, 4, 3, 2, 0.$$

This mapping can be performed without branches on commodity hardware using eight bits per index. If we treat each block row as four 8-bit grayscale values, we can store an entire block row in a single 32-bit register. Furthermore, 32-bit integer operations can be used to perform byte-wise SIMD computations without requiring special SIMD hardware, as shown in Figure 3.4.

3.2.1.2 ETC2

One variant of the ETC2 compression format is a table-based compression algorithm that takes 4×4 blocks of grayscale pixels, and reconstructs 11-bit grayscale values from 64-bit encoded data in order to provide higher precision than traditional 8-bit textures. However, the 64-bit representation maintains a two-to-one compression ratio similar to DXTn. The procedure by which the coverage value for pixel c_i is reconstructed is

$$c_i = b \times 8 + 4 + (\mathbf{T}_v)_{t_i} \times 8,$$

where the encoded data stores an 8-bit base codeword b, a 4-bit multiplier m, a 4-bit modulation index v, and sixteen 3-bit indices t_i. $\mathbf{T}$ is a table containing sets of modulation values constant across all the encodings. This table has sixteen entries, indexed by v. Each t_i selects a final modulation value from the set $\mathbf{T}_v$. The result c_i is then clamped to the range [0, 2047].

To compress the grayscale coverage information, we first fix values for v, b, and m such that they generate the tightest bounds to the entire range of grayscale values. We compute these values by performing an exhaustive search through all possible combinations of v, b, and m offline. In order to compress the coverage information, we perform a quantization to three bits as described in Section 3.2.1.1.

However, due to the indexing method of ETC2, we must use a different mapping

$$0, 1, 2, 3, 4, 5, 6, 7 \rightarrow 3, 2, 1, 0, 4, 5, 6, 7.$$

This mapping is also has the same implementation advantages as DXTn, as shown in Figure 3.5, allowing branchless computation to be done in fixed 32-bit registers.

3.2.1.3 ASTC

Finally, we demonstrate fast compression of our coverage information using the ASTC format introduced by Nystad et al. (2012). This format has a variable block size that must be chosen prior to compression, and we have noticed that even at the highest compression rate, 12×12, rendering artifacts were negligible. This is possible due to the high compressibility resulting from the low entropy of the coverage mask described in Section 3.2.2.

ASTC encoded blocks may choose from many different compression options. One such option is whether or not to partition the block into separate subsets of pixels with different compression parameters. Similar to DXTn and ETC2, ASTC uses per-pixel indices to reconstruct the block of pixels. However, there may be fewer indices than pixels, in which case the indices are stored in a grid and interpolated across the block. Finally, similar to DXTn, ASTC reconstructs pixels by using generated indices to lookup palette entries. However, ASTC allows the block encoding to choose how many bits are allocated towards endpoint representation versus index representation.

In order to maximize the fidelity of the ASTC compressed coverage mask, we outline a list of the choices that we made for each 12×12 block of pixels. The main insight is to maximize the number of pixel index values and their bit depth. We are able to maximize the index size because the endpoints must cover the full range of grayscale values and hence require very few bits. For this reason, we are able to generate a valid ASTC encoding using the following choices:

- 6×5 texel index grid to maximize the number of samples in a 12×12 pixel block
- Three bits per texel index
- Single plane encoding (redundant due to single-channel input). This is chosen because we do not use multi-channel pixels

- Only one color endpoint mode: direct luminance
- Single partition encoding with two 8-bit endpoints: 0, 255

Using these constants for all coverage information, there is no special need for the base-three and base-five integer sequences supported by ASTC (Nystad et al., 2012). Since we know the dimensions of the grid versus the dimensions of the block size, we can precompute the amount that each pixel contributes to each index, and store this in a look-up table. During compression, for each texel grid index we store the top three bits of a weighted average of the pixels that are affected by the index. The final result is 144 grayscale pixels compressed into 128 bits, providing a compression ratio of nine to one. Although compression of ASTC is slower than DXTn and ETC2, the generated compressed textures are significantly faster to load into GPU memory.

3.2.2 Scan conversion

While the compression format chosen is dependent on the underlying hardware, the scan conversion of path data is computed independently on the CPU. In particular there are two main steps:

1. Determine the run-length encoded coverage information for each scanline of pixels
2. Convert multiple scanlines at once into the necessary compression format

From a given path, coverage information for each pixel is computed by sampling the path N times per pixel, commonly $N = 16$ with the samples arranged in a regular grid (Figure 3.6). Each sample is applied a boolean value $b_i \in \{0, 1\}$ such that the final coverage for a given pixel in image $\mathbf{I}$ is

$$\mathbf{I}(x, y) = \frac{1}{N} \sum_{i=1}^{N} b_i.$$

For a value corresponding to $N = 16$, this implies that $\mathbf{I}$ can take up to 17 possible values for any $(x, y) \in \mathbf{N} \times \mathbf{N}$.

In a scanline of samples, the edges of the curve can be computed analytically in order to properly set the corresponding b_i. As shown in Figure 3.6, the per-pixel coverage information, i.e. the number of samples covered by the path, is stored in a sparse run-length encoded (RLE) buffer. This buffer is updated for each new scanline of samples within a row of pixels. The sparsity of the buffer prevents unnecessary

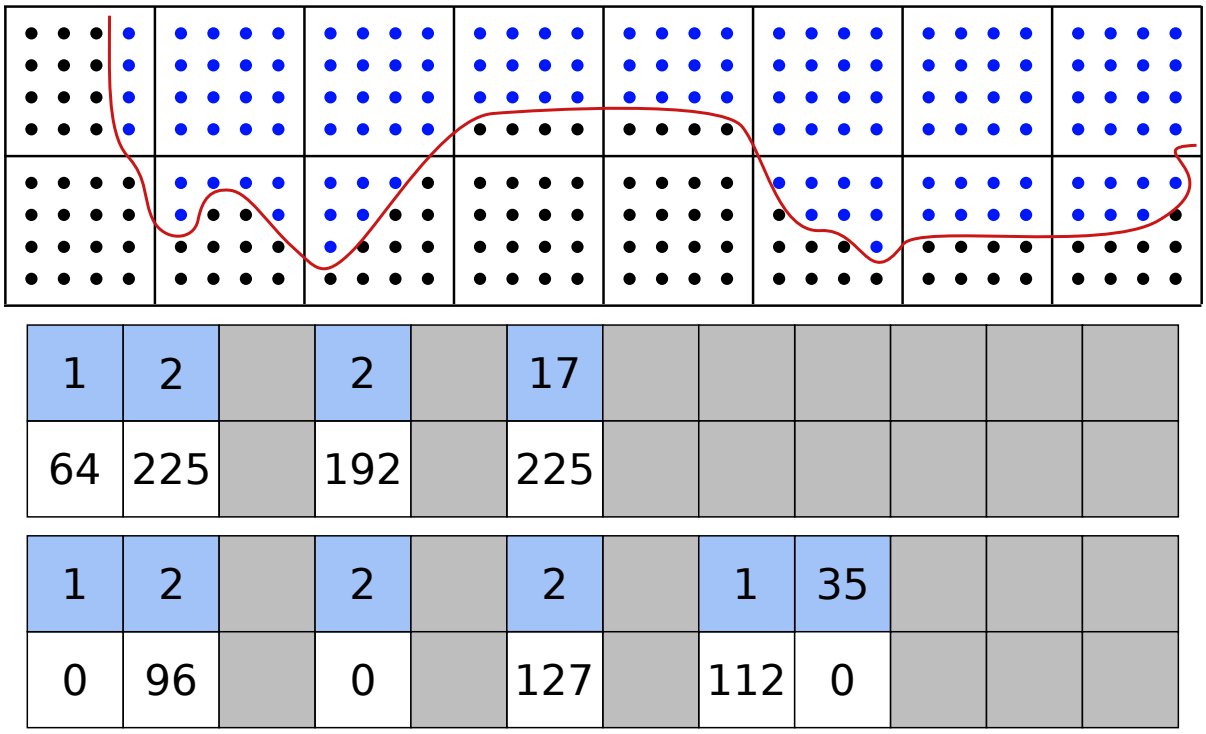


Figure 3.6: Sparse run length encoded (RLE) buffers. These buffers are used to store the coverage information for a row of pixels prior to writing them into the coverage mask. For each pixel row, the RLE buffer is allocated to contain as many RLE entries as there are pixels. The scan converter operates on rows of super-sampled pixels, shown here as a 4×4 grid within each pixel, and updates the corresponding RLE buffer. In this figure, the blue entries contain the number of runs of the corresponding pixel value. Grey entries are uninitialized and never written to nor read. Samples which contribute to the coverage of the red curve are drawn in blue and samples that are uncovered are drawn in black.

allocation when an initial scanline of samples is altered by a subsequent scanline. In this situation, the samples within a pixel may be identical in the first scanline of samples but different in the second.

The pixels containing intermediate values, i.e. those that are neither fully opaque (covered) or transparent (uncovered), are only found along the boundaries of the 2D path. For this reason, a majority of the pixels in a coverage mask take extremal values (0 or 255) and very few, along the edges of the path, tend to have intermediate values. This means that most of the image can be stored as a binary image, producing an entropy close to one (Shannon, 1948). This extremely low entropy property of coverage masks makes them highly compressible.

In order to generate compressed textures, we must adhere to the random access requirements in texture representations. Random access ensures that the renderer has equal access to all pixels regardless of when they are needed. This requirement implies a fixed block size for each compression format: 4×4

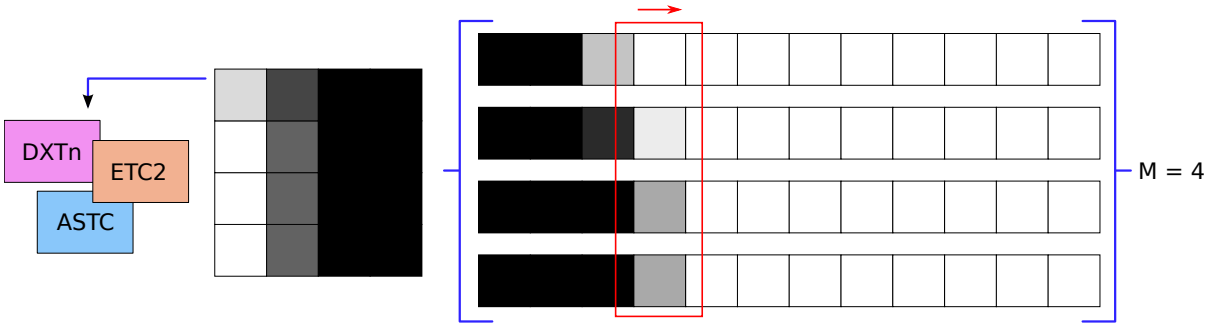


Figure 3.7: Our scan conversion pipeline augmented to output GPU-compressed blocks. For $M \times M$ compressed block sizes, our pipeline operates on M sparse RLE buffers in parallel (Figure 3.6). Once M columns are processed, they are compressed into the target compressed format. For a given column, we read from the entries in the associated sparse RLE buffers. If any of the row values have changed, we update the corresponding pixel for the current column (outlined in red). Otherwise, we simply copy the previous column. For 8-bit coverage values and 4x4 compressed block sizes, each column fits in a single 32-bit register.

for DXTn and ETC, and 12×12 for ASTC. Once a scanline of pixels is computed, it can be stored in a row of an 8-bit grayscale texture. We generate compressed representations of the grayscale textures by consuming M rows of run-length encoded data at a time, where M is the dimension of the (square) block size of the texture compression format. As shown in Figure 3.7, we read the leftmost column of grayscale values and update the corresponding byte as we walk down our M RLE buffers. At each step, we advance to the column with the earliest ending run length. Once we advance past M columns, we efficiently compute a compressed representation of the $M \times M$ block that we have read from the RLE buffers, as described in Section 3.2.1. For the most common case, $M = 4$, the four grayscale values are represented as a 32-bit integer, and we can perform SIMD byte-wise operations using integer shifts and adds. As an optimization, if we advance the current column farther than M pixels at once due to the RLE encoding, we can copy the previous block encoding into its neighbor to the right.

3.3 Results

To test our results, we have integrated our real-time compression pipeline into the 2D graphics library Skia (Google, 2016). This library is used as the backbone to many cross-platform 2D programs and operating systems including Android, Google Chrome, and Mozilla Firefox. In order to maintain performance and regression tests across all platforms, Skia includes two types of comprehensive tests. For any given change to the implementation, Skia tests the new rendered image against existing baseline

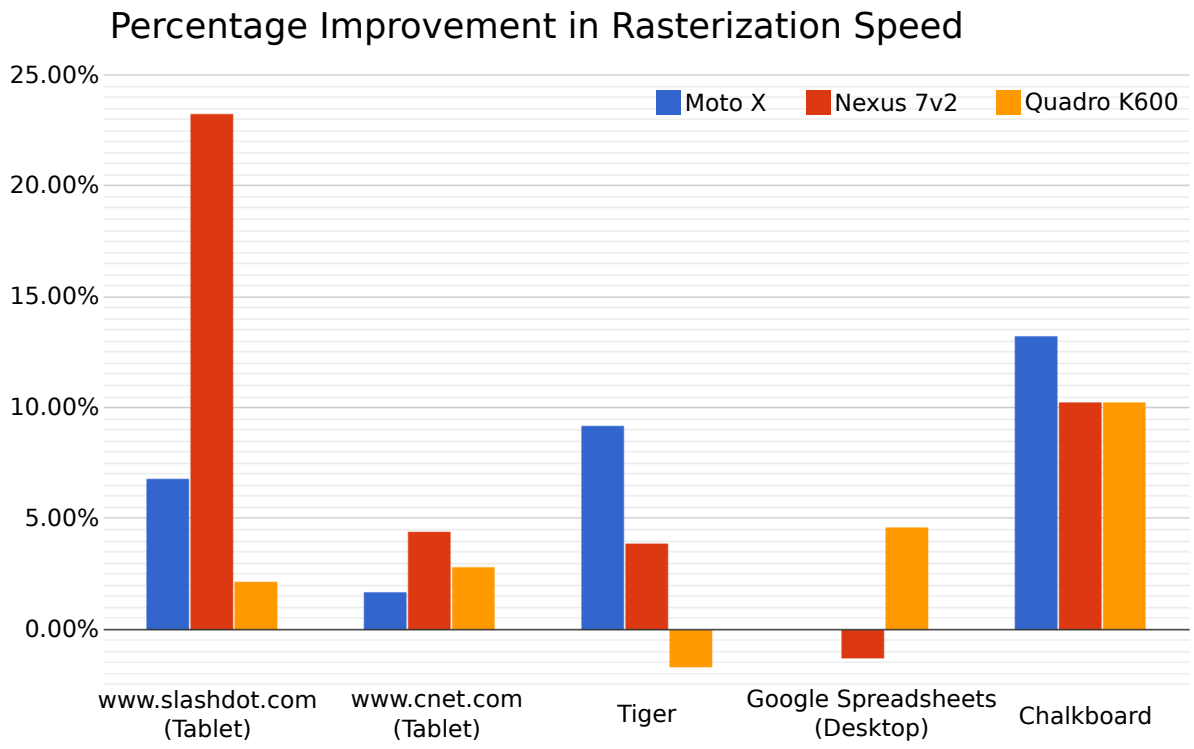


Figure 3.8: Performance improvements using compressed textures on a variety of different benchmarks. Two of the tests performed were on tablet versions of popular websites. The Google Spreadsheets benchmark data was gathered from the desktop version of the site using many stroked paths. The other two were the vector images in Figure 3.9.

images. If any pixels differ by a significant amount, these tests fail and the change is invalid. The second test measures performance against a suite of microbenchmarks and a suite of rendering commands that are invoked during the rendering of common web pages. In order for these tests to pass, their running time must be within a small threshold of the previously passed test. In each of our examples, we have maintained both correctness and performant code with respect to the existing implementations.

First, we must show that our implementation runs fast on modern hardware. In Figure 3.8, we show different classes of benchmarks that have been run on a variety of different mobile GPUs. In each case, we see a general increase in the rendering speed of certain web pages and common vector graphics benchmarks. As we can see, the desktop GPU does not receive as much of a benefit from the compression routine as the mobile GPUs. We conjecture that mobile GPUs are more sensitive to transmitting large amounts of data from the CPU to the GPU due to power restrictions and hence receive more benefits. Mobile GPU performance increases are better demonstrated in Table 3.1 where various mobile GPUs

Mobile Platform	CPU	GPU
Moto X	1.7 GHz Qualcomm Krait	Qualcomm Adreno 320
Galaxy Note 3	1.9 GHz ARM Cortex-A15	ARM Mali-T628
HTC One M8	2.3 GHz Qualcomm Krait 400	Qualcomm Adreno 330
Galaxy Note 10.1	1.9 GHz ARM Cortex-A15	ARM Mali-T628
Galaxy S5	1.3 GHz ARM Cortex-A7	ARM Mali-T628

Mobile Platform	Uncompressed	Compressed	Texture Format	Memory Benefit
Moto X	163ms	137ms	ETC2	2:1
Galaxy Note 3	171ms	161ms	ETC2	2:1
HTC One M8	114ms	102ms	ETC2	2:1
Galaxy Note 10.1	171ms	136ms	ASTC	9:1
Galaxy S5	311ms	157ms	ASTC	9:1

Table 3.1: The rendering times for the polygon benchmark (Figure 3.9) from Skia using both compressed and uncompressed texturing on a variety of CPU/GPU combinations. The polygon benchmark generates a large sequence of thin, concave polygons and stores them as piece-wise 2D paths on the GPU. These polygons are then both stroked and filled to generate a large amount of paths that must be rasterized. From these results, we notice an increase in rendering speed of the heavily optimized Skia library on all mobile devices. Most importantly, the increase in memory efficiency from ETC2 (2:1 ratio) to ASTC (9:1 ratio) provides significant improvements in rendering time. These results were generated from the mean runtime of 100 executions.

render the polygon image (Figure 3.9) from the Skia performance tests. From this table, we observe that both CPU speed (Galaxy Note 10.1 vs Galaxy S5) and compression ratio (Galaxy Note 10.1 vs Galaxy Note 3) play a vital role in rendering performance on mobile devices.

In order to test accuracy, we perform both a visual comparison against the reference images (without compression) and measure the difference using the *Peak Signal to Noise Ratio*, or PSNR:

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^2 \times w \times h}{\sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2 \right)} \right)$$

In Figure 3.10, we compare the various use cases of rendered paths and the difference in their rendering. We observe that only pixels along the borders of the paths are affected by the compression scheme. This homogeneity in the coverage masks is the primary reason why they are highly compressible. From the zoomed in comparisons, we notice that there is little to no quality loss in the final images. However, the pixels that differ do so by a non-trivial amount. This difference causes the relatively low PSNR values calculated for the images.

From the performance and quality results, we observe a benefit to compressing coverage masks prior to usage, with little visible loss in quality. The method described in Section 3.2 that yields these results relies heavily on 32-bit integer operations and is otherwise portable to a wide variety of platforms. These performance metrics also do not take into account the possible benefits from multi-threading approaches. Although these methods are highly parallelizable, the main benefit is reducing the latency of uploading the coverage masks to the GPU. Hence, any GPU compression method that would require the data uploaded prior to compression would lose this benefit. However, if the coverage information is generated on the GPU, then our method could be used to compress the mask very quickly using only a handful of low-latency integer operations.

3.4 Error Analysis

The methods for compressing coverage masks outlined in Section 3.2 are designed for speed and with the assumption that coverage masks will be mostly coherent. For any given coverage mask, the rendering time will be dependent on the resolution of the coverage mask. However, the quality is fixed due to the precomputed compression parameters for each format. As a result, it is possible to find the worst-case texture quality for data compressed into each format. In this section we investigate such failure cases and show scenarios that our method might not handle particularly well.

Due to the nature of coverage masks, the only areas of high detail are border regions where pixels may end up being partially covered. As a result, the error bounds reported in this section do not reflect the quality of the final compressed coverage masks. They are used to demonstrate the limitations of our compression scheme against general-purpose data and highly incoherent coverage masks. In practice, compressed coverage masks do not contain any visible artifacts. In contrast to the artifacts that are most commonly noticeable in low-resolution coverage masks, such as aliasing, or “jaggies”, the most noticeable artifacts in compressed coverage masks tend to be blurring caused by the interpolation described in Section 3.4.2.

3.4.1 DXTn and ETC2 Compression Formats

In both DXT and ETC2, we generate a fixed color palette into which we compress our coverage masks. For both formats, the palette is precomputed based on what the anticipated data in the block will

be. Our compression parameters are chosen such that we represent values ranging from fully transparent, or zero, to fully opaque, or $2^8 - 1$.

As described in Section 3.2.2, our input texture has at most 17 values, ranging from zero to sixteen, which counts the number of samples covered by our path. When uploading uncompressed coverage masks, each of these seventeen values gets quantized to a value from zero to $2^8 - 1$. However, since both DXT and ETC2 use three-bit indices, each compressed block contains only eight possible choices which vary depending on the format. For DXT, the available values are

$$\{0, 36, 73, 109, 146, 182, 219, 255\}$$

while for ETC2, the values are

$$\{0, 51, 78, 105, 149, 176, 203, 255\}.$$

Figure 3.11 shows the amount of absolute error each of the original 17 values incurs when compressing to the respective formats.

3.4.2 ASTC Compression Format

Unlike DXT and ETC, ASTC blocks interpolate their indices from a low-resolution index grid to determine per-pixel index values. For this reason, determining the proper ASTC representation requires more processing than converting pixels to index values. As in Figure 3.12, each pixel in the input block contributes to the final value of four surrounding indices.

In order to maintain real-time performance of coverage mask compression, we must precompute many of the parameters for each block, as described in Section 3.2.1.3. This optimization has implications on texture compression quality when dealing with high-frequency data. In particular, data that has a large variance between our index locations can become distorted. As we can see in Figure 3.13, blocks that have high frequency are very difficult to compress using our chosen parameters. In particular, the checkerboard block, which is simply alternating black and white pixels, results in the most artifacts due to high index averaging of all nearby pixel values.

In order to properly select the indices for our ASTC block, we may solve a linear system of the form $\mathbf{A}x = b$, where $\mathbf{A}$ is a 144×30 matrix corresponding to the contribution of each pixel in a 12×12 block to each index in a 6×5 grid. By virtue of $\mathbf{A}$ being fixed, we can precompute the pseudo-inverse $\mathbf{M} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ in order to find the index values

$$x \approx \mathbf{M}b$$

for any block b. Furthermore, we can use this method to determine what the error is for any given block b,

$$E(b) = \|\mathbf{A}\mathbf{M}b - b\|_2 .$$

We use a least-squares formulation in order to minimize the appearance of noisy pixel values. However, this error function is highly non-convex due to the nature of the quantization of each valid value of b. For this reason, we cannot analytically derive a global maximum or minimum. In pursuit of a numerical solution, we can calculate the gradient for $E(b)$,

$$\nabla E(b) = \left\|\hat{\mathbf{M}}b - b\right\|_2 \left(\hat{\mathbf{M}}^T\hat{\mathbf{M}}b - (\hat{\mathbf{M}} + \hat{\mathbf{M}}^T)b - b\right),$$

where $\hat{\mathbf{M}} = \mathbf{A}\mathbf{M}$. Using this gradient, we use gradient descent to find the worst-case blocks that can be encoded with our method. Due to the large number of local maxima within the search space, we seed our optimization routine with random blocks. The resulting block can be seen in Figure 3.12.

The error analysis for ASTC blocks allows us to quickly determine whether or not a given block is suited for compression. If compressing the block will introduce a significant amount of unacceptable compression error, we may abort the compression procedure and try alternatives such as a different format or reverting to uncompressed textures. Additionally, this technique of determining error can aid content authors in creating paths that can be compressed well at various resolutions.

3.5 Conclusion, Limitations, and Future Work

In this chapter we have shown that coverage masks used for rendering 2D anti-aliased non-convex paths are perfect candidates for real-time compression. Their low-detail properties make compression

algorithms very efficient and the masks themselves highly compressible. We have also shown that these masks can be compressed in real-time often speeding up the rendering of 2D curves and saving valuable GPU memory.

Limitations: Although the coverage masks can be compressed effectively, GPU-based methods for rendering arbitrary 2D-curves with anti-aliasing are still slower than their CPU-based counterparts. In general, generating the coverage mask is by far the most expensive operation of the rasterization procedure. During CPU-based rendering, the rasterizer can perform the shading directly from the RLE buffer discussed in Section 3.2.2. This limitation can be observed from the time it takes to run the polygon benchmark from Table 3.1 on different platforms using the software renderer:

Rendering time for convex path benchmark *strokedrects*

Platform	GPU	CPU
Moto X	6.9 µs	37.6 µs
Galaxy Note 10.1	3.76 µs	15.5 µs

Rendering time for non-convex path benchmark *polygon*

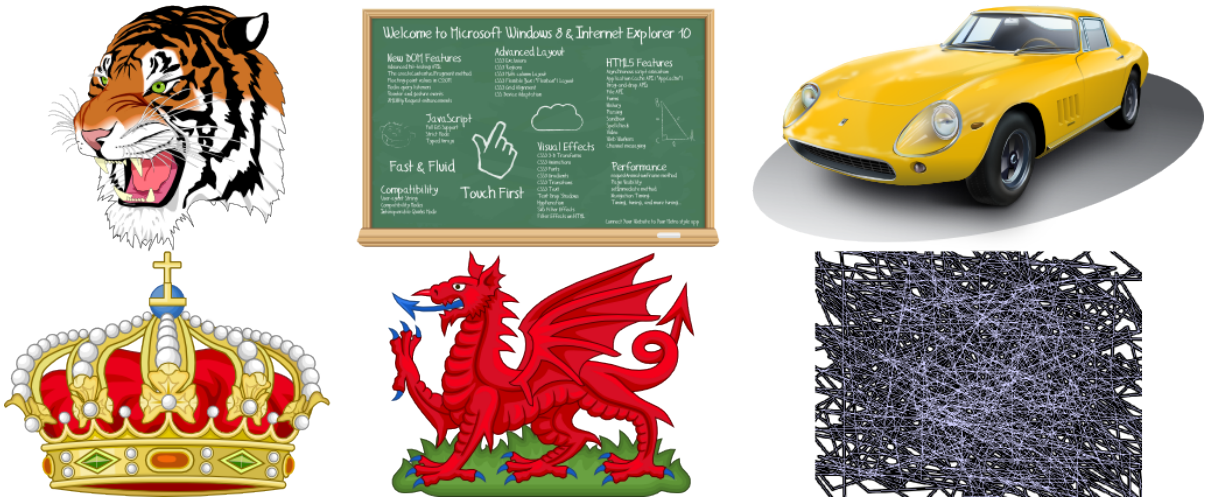
Platform	GPU		CPU
	Uncompressed	Compressed	
Moto X	163ms	137ms	83ms
Galaxy Note 10.1	171ms	136ms	46ms

However, many of the applications that require 2D rendering operate on many more primitives than non-convex 2D curves. In the table above, the GPU-based convex path rendering operation still outperforms its CPU counterpart. For this reason, it is advantageous to use a GPU-backed framebuffer. As such, our method provides benefits to the least efficient aspect of GPU-based resolution independent graphics rendering.

Additionally, as we described in Section 3.4, our method does not create high fidelity texture compression for general purpose grayscale images. We assume coverage masks to be highly uniform with little variation along primitive edges. If these assumptions are maintained, as we see in Figure 3.10, then rendering using our compression technique maintains acceptable perceptual quality.

Future Work: We have shown that coverage masks are very amenable to compression. Due to the very high fidelity of the rendered images even at the highest available compression ratios (12×12

ASTC) there is ample room for even more aggressive compression formats. Encodings that support block dimensions up to 32 or 64 may still produce nice results. The compression algorithms in Section 3.2.2 can be extended to support even better compression ratios, which will increase both the rendering speed and memory usage. Another direction for research is the ability to generate coverage information on the GPU itself. If such a technique existed, the compositing procedure using the coverage mask could be performed at the same time as generating the coverage information itself. However, if the coverage mask were generated on the GPU and then used as input to a second compositing pass, compressing the GPU-generated coverage masks using this technique would incur trivial cost. Due to the random-access restrictions of compressed texture formats, they are perfect candidates for massively parallel encoding. Furthermore, to combat the original artifacts from the Blinn-Phong method, conservative rasterization may be used to cover every pixel touched by the bounding triangles (Akenine-Möller and Aila, 2005). Such a solution could eliminate the need for CPU-side rendering entirely. Finally, the error analysis in Section 3.4 opens up the possibility for additional compression algorithms that may do a better job of compressing both coverage masks and general purpose data.



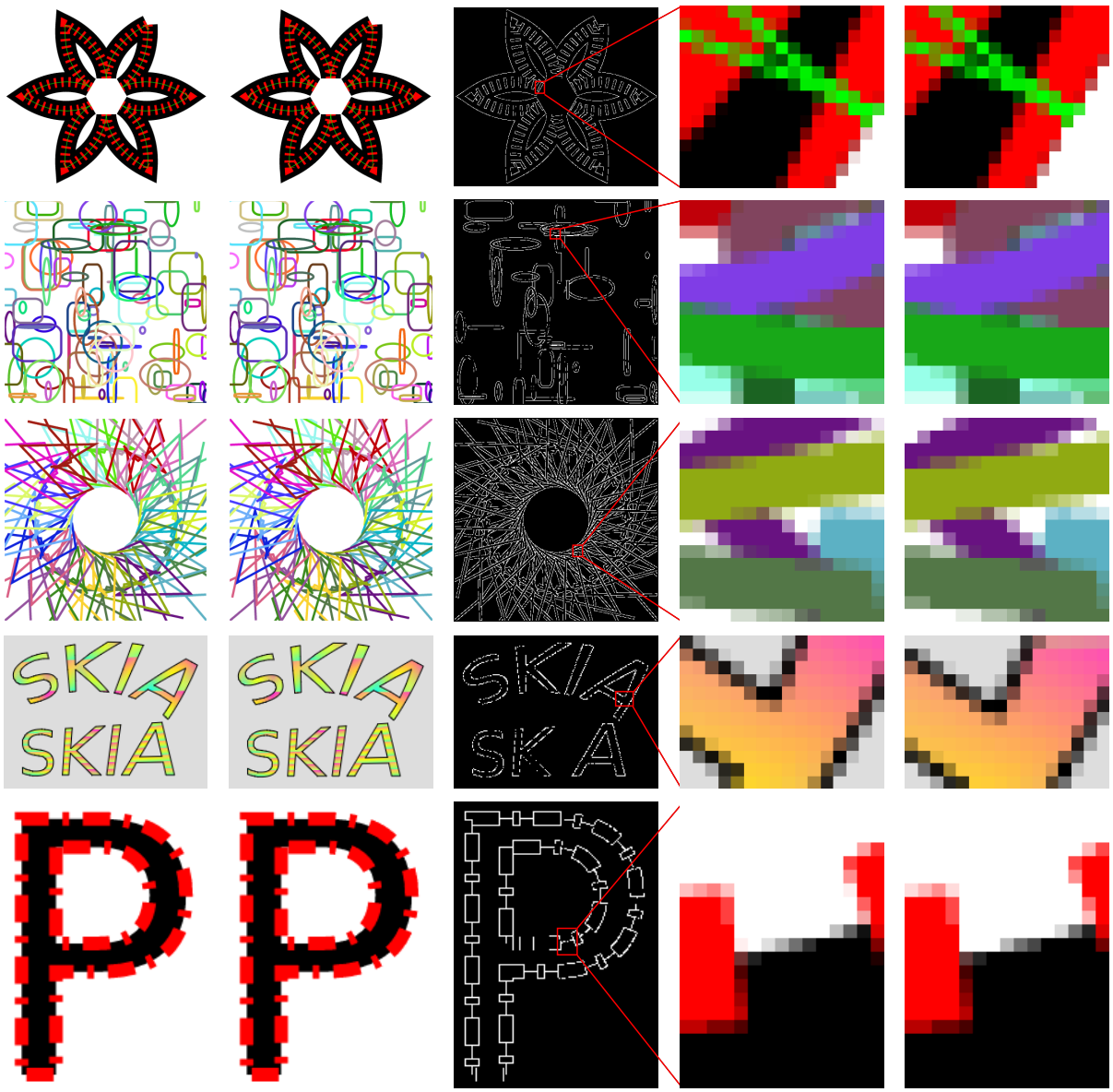
Uncompressed

Image	Min	Median	Mean	Max	σ
Tiger	95.3ms	96.6ms	97.8ms	109ms	3ms
Chalk	358ms	370ms	371ms	473ms	5ms
Car	368ms	385ms	385ms	403ms	2ms
Crown	121ms	127ms	137ms	200ms	15ms
Dragon	92ms	94.3ms	96ms	140ms	7ms
Polygon	149ms	152ms	154ms	208ms	5ms

Compressed

Image	Min	Median	Mean	Max	σ
Tiger	81ms	83ms	83ms	93ms	2ms
Chalk	339ms	349ms	350ms	495ms	5ms
Car	364ms	387ms	386ms	424ms	3ms
Crown	106ms	109ms	111ms	168ms	9ms
Dragon	87ms	92.2ms	101ms	156ms	19ms
Polygon	133ms	134ms	137ms	194ms	7ms

Figure 3.9: Rendering times of the following images on a first generation Moto X (1.7 GHz Qualcomm Krait, Qualcomm Adreno 320) from 100 runs. From left to right the images are labeled Tiger, Chalk, Car, Crown, Dragon, Polygon.



Original Compressed Difference Original Detail Compressed Detail

PSNR	dashed	rounded	poly	text	strokep
	35.457	41.028	35.457	37.736	53.876

Figure 3.10: Detailed analysis of correctness tests within Skia most heavily affected by changes to anti-aliased non-convex path rendering. From top to bottom, the images are labeled as 'dashed', 'rounded', 'poly', 'text', and 'strokep'. We observe very few artifacts due to compression. Although the pixels along the anti-aliased edges in the rendered images do contain different pixel values contributing to the relatively low PSNR values, the detail in the edges remains. Pixels in the difference image are on if the shaded values in the corresponding original and compressed images differ. Most noticeable in 'strokep', the low detail of the coverage masks causes pixel differences only in those along the edges of the filled paths.

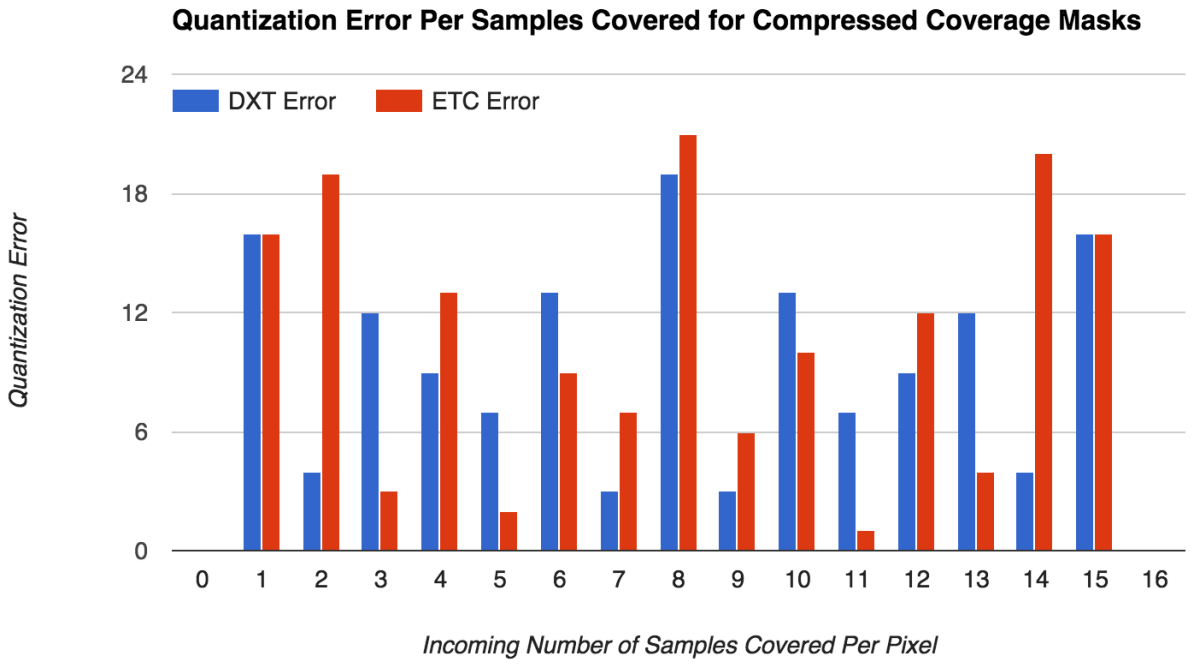


Figure 3.11: Quantization error when converting the incoming number of samples covered per pixel to the final value stored in the compressed format. We show absolute error for both DXT and ETC formats with respect to the original quantized values. For fully opaque and fully transparent pixels we have no error as designed. For intermediate values, discrepancies in error arise from the way values are quantized in adherence to the two texture formats.

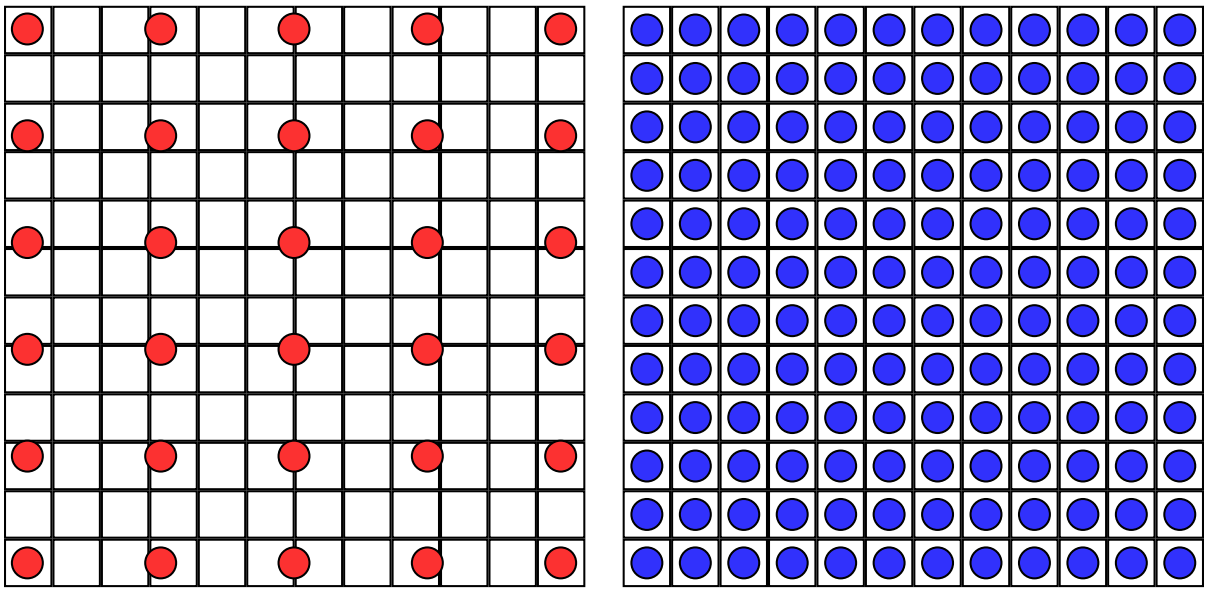
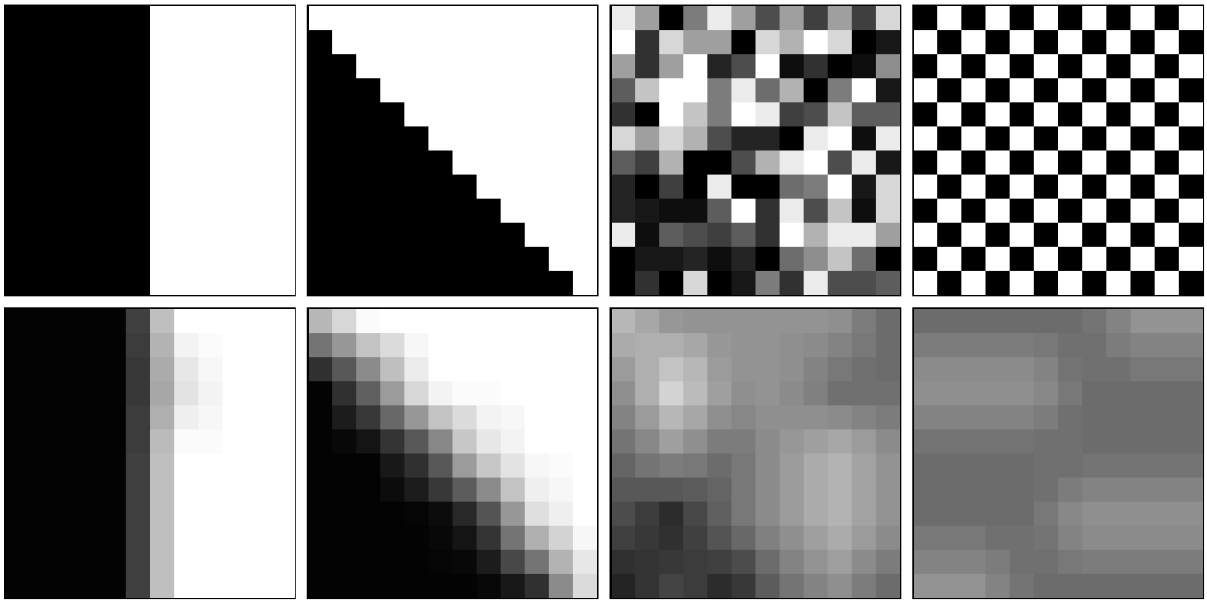


Figure 3.12: For a 12x12 ASTC block, we maximize the number of samples we store in order to get the finest granularity of control possible over the resulting pixels. Physical limitations of the ASTC format restrict us to a 6x5 index grid stored on disk (red samples). During decompression, these indices are interpolated to each texel (blue samples) to compute the final index used for selecting from the precomputed palette.



19.36dB 14.98dB 10.08dB 6.00dB

Figure 3.13: (Top row) Uncompressed failure cases for certain 12x12 blocks. (Bottom row) Our ASTC compression method applied to each block. Due to the interpolation of index coordinates in ASTC blocks, certain blocks will be compressed much more poorly than others. In particular, blocks that have many uncorrelated neighboring pixels, while able to be represented using ASTC, are not particularly well suited for our method. However, such blocks are very rare in coverage mask textures.

CHAPTER 4: ACCELERATING TEXTURE COMPRESSION [1]

One key aspect of using compressed textures for rendering is the notion that fast encoding speed is useful but not necessary (Beers et al., 1996; Fenney, 2003). In Chapter 2.6 we showed how fast texture compression can provide benefits to textures generated at run-time. Modern-day games use hundreds, if not thousands, of textures generated offline for everything from lightmaps to character albedo maps. These textures are usually generated by artists and then compressed as a secondary step to be used at run-time. The game development process relies heavily on constant iteration over all included assets, making the development of fast, high-quality texture compression methods important.

The hardware texture compression formats described in Section 2.3 apply a variety of techniques to compress data. In order to provide random-access pixel lookup and cache coherency, each format encodes a fixed size block of texels separately. Of significance are the *endpoint* compression formats that store two endpoints per block that are linearly interpolated to generate a palette of colors, and per-pixel *indices* that select from this palette. More recent encoding formats allow data points to be partitioned into separate subsets that each have their own endpoints.

Encoding for texture compression formats is usually performed offline. Algorithms for encoding textures vary on the spectrum of quality versus performance. Some publicly available and widely used compressors for older formats use cluster analysis techniques to achieve good compression quality (Brown, 2006; Donovan, 2010). However, the simplicity of these formats restricts the quality of the compressed image. Newer compressors use the same kind of analysis to achieve the quality available by the format but result in long compression times for the best quality (Nystad et al., 2012). It is not uncommon for some high quality texture encoders to take hours to compress hundreds of textures for a single scene (see Figure 4.1). In practice, texture compression is regarded as one of the most expensive stages of asset compilation.

Additionally, GPUs are increasingly being used for applications other than desktop-based 3D games, but continue to make heavy use of textures. These include geographic information systems and mapping

[1] Much of this chapter appeared previously Krajcevski et al. (2013); Krajcevski and Manocha (2014b,a)

[2] Unity3D engine vailable at *http://www.unity3d.com/*. Bootcamp demo available through the Unity Asset Store.

Figure 4.1: The Bootcamp demo from Unity3D[2]using uncompressed textures (top) and using textures compressed with FasTC-64 (bottom). The visual quality of the scene is only slightly altered and no visible artifacts appear. The scene uses 156 textures which were compressed in a total of 8.75 minutes by our method. The same textures are compressed by the BC7 Compressor in the NVIDIA Texture Tools in a total of 13.27 hours.

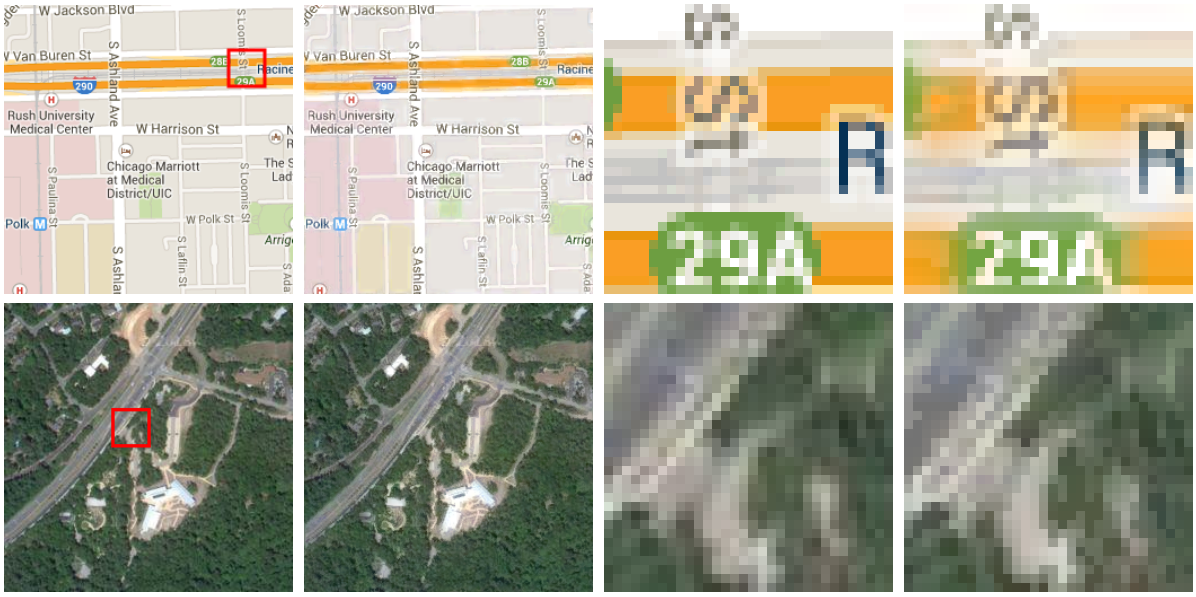


Figure 4.2: Real-time texture compression using intensity dilation applied to typical images used in GIS applications such as Google Maps. Each of these 256x256 textures was compressed in about 20ms on a single Intel® Core™ i7-4770 CPU. In each pair of images, the original texture is on the left, and the compressed version is on the right. A zoomed in version of the detailed areas is given on the right. Images retreived from Google Maps.

tools (e.g. Google Maps) that use textures rendered on-the-fly based on level of detail and other factors, as shown in Figure 4.2. Cloud-based game services are also emerging where the user experiences high-quality graphics from the lightweight rendering nature of a browser window. Many of these applications require uploading texture data across a network and then to the GPU very frequently. As a result, it is becoming increasingly important to develop real-time high quality texture compression algorithms. Finally, another recent trend is to support different texture formats for different GPUs or devices. With fast texture compression schemes, developers only need to store a single texture and compress it on-the-fly based on hardware capabilities of the client, significantly saving on storage space.

We present three new algorithms for compressing textures. In the first algorithm, we focus on a widely used texture compression method known as Low-Frequency Signal Modulated Texture Compression (LFSM) (Fenney, 2003). Up until very recently, this texture compression technique was the only technique supported on popular iPhone and iPad devices. LFSM leverages the cache-coherent worst-case scenario of block-based texture compression techniques such as DXT1 and BPTC (Iourcha et al., 1999; OpenGL, 2010). It has been pointed out that LFSM texture compression formats, such as PVRTC, provide better quality than formats with similar compression ratios (e.g. DXT) on certain classes of textures (Fenney, 2003). However, due to the structure of LFSM formats, fast or real-time compression algorithms are not

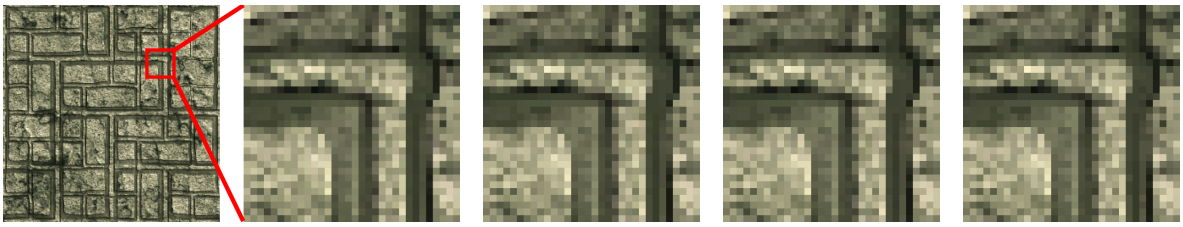
as available compared to other formats (Schneider, 2013). We present a novel, fast texture compression algorithm based on intensity dilation for LFSM formats to be explained in Section 4.1.2. Our technique finds the intensity values of a 8-bit RGBA image that contribute to high-contrast areas sensitive to the human visual system during compression (Aydin, 2010). We use these intensity values to represent high detail regions of the texture. We have evaluated our technique on a variety of benchmark images and observe 3 – 3.5X speedup over prior PVRTC algorithms and implementations. We also measure the quality of compression using both raw energy metrics and human perception, and notice considerable benefits over prior schemes.

The second, FasTC, is used for partition-based texture formats providing orders of magnitude improvements in speed over previous compression methods while maintaining comparable visual quality. Our approach uses a coarse approximation scheme to estimate the best partition. Due to the quantization artifacts in the compressed data, this coarse approximation gives a substantial increase in speed while avoiding a severe penalty in quality. Next, we use a generalized *cluster-fit* (Brown, 2006) to find the best encoding for the partition. The benefit of this approach is that we perform linear regression in the quantized space rather than in the continuous or even discrete RGB space. Optionally, we allow the user to refine the data using simulated annealing which provides the ability to set the speed versus quality ratio within a single algorithm. We test FasTC on low-frequency game textures and the canonical high-frequency Kodak Test Image Suite against popular compressors provided by both NVIDIA and Microsoft (KODAK, 1999; Donovan, 2010; Microsoft, 2010). Our method performs orders of magnitude faster than these methods while maintaining visually similar results. Furthermore, we demonstrate parallel scalability with the number of processor cores.

The third and final algorithm, SegTC, also targets partition-based texture formats, such as ASTC (Nystad et al., 2012) and BPTC (OpenGL, 2010). To avoid the exponential increase in the number of partitionings with respect to the block size, these formats choose from a restricted set of common partitionings. The per-block task of an encoding algorithm becomes twofold:

- Select a partitioning out of a predefined set.
- Choose parameters for each subset of the partitioning.

With FasTC, we estimated the optimal parameters for each partitioning in order to select it. With SegTC, we present a new method for choosing predefined partitionings for partition-based compression formats.



Original Context Exhaustive FasTC Our Method

Figure 4.3: Our partition-based texture compression algorithm applied to a standard wall texture. The full original texture is shown on the far left, followed by a zoomed in investigation of the region outlined in red. Our method compresses the texture into the BPTC format. The resulting image quality, measured in Figure 4.20, is comparable to prior methods. This texture is 256×256 pixels large and was compressed using an exhaustive method (64 seconds (Donovan, 2010)), FasTC (567 milliseconds (Krajcevski et al., 2013)), and our method (143 milliseconds) on an Intel Core i7-4770k 3.80GHz processor using a single core without vector instructions.

Our method first computes a segmentation of the image into *superpixels* to identify homogeneous areas based on a given metric. This segmentation defines borders between areas whose pixels share common characteristics. To select predefined partitions, each block uses the segmentation boundaries to determine the best partitioning for that block. Next, we use a *vantage point tree* to find the nearest matching partition based on Hamming distance. Using this selection scheme, we test our technique on low-dynamic range textures. We observe up to a 6x performance increase on existing single-core compression implementations and approach interactive rates for 256×256 sized textures.

Current codecs employ a metric on possible partitionings on a per-block basis. Most BPTC codecs choose a partitioning by estimating the amount of error per partition (Krajcevski et al., 2013; Dufresne, 2013). The ASTC reference codec computes an ideal partitioining for a given block and then finds the best match to an existing partitioning (ARM, 2012). The ASTC codec computes an optimal partitioning for every block. SegTC computes an optimal global partitioning (segmentation) and then chooses partitions based on the labels covered by compression blocks. The advantage here is that we can use the same segmentation for multiple different block sizes, allowing better performance when compressing textures for a variety of compression ratios, such as the adaptive method described in Chapter 4.3.3.

4.1 Speed-up for LFSM Texture Formats

4.1.1 Low Frequency Signal Modulated Texture Compression

In recent years, low frequency signal modulated texture compression has been widely adopted by many mobile devices. Prior to OpenGL ES 3.0, it has been the only technique available on Apple's iPhone and iPad (Apple, 2013). Despite its popularity, there has not been much work in improving the speed of associated compression techniques. In this section, we give an overview of LFSM texture compression.

4.1.1.1 LFSM compressed textures

Like other texture compression formats, LFSM compressed textures are stored in a grid of blocks, each containing information for a $n \times m$ grid of texels. As shown in Figure 2.1, each block contains two colors along with per-texel modulation data. Each of these colors, referred to as the *high color* and *low color*, is used in conjunction with neighboring blocks to create two low resolution images: the *high image* and *low image*, respectively. In order to lookup the value for a texel, the high image and low image are upscaled to the original image size using bilinear interpolation. Once upscaled, modulation data from the block that contains the texel in question is used to compute a final color. This bilinear interpolation avoids the worst-case scenario with respect to memory lookups. By filtering textures across block boundaries, information from four blocks is required to decode any texel value.

A good LFSM data compression algorithm needs to determine for each block $\mathbf{b}$ both the best high color, $\mathbf{b}_H$, and low color $\mathbf{b}_L$ and the modulation data w for each texel. These values must be chosen such that when decoding a texel $\widetilde{\mathbf{p}}$ surrounded by blocks $\mathbf{b}_A$, $\mathbf{b}_B$, $\mathbf{b}_C$, $\mathbf{b}_D$, the resulting color given by

$$\begin{aligned}\widetilde{\mathbf{p}} = \quad & w\left(l_A\mathbf{b}_{H,A} + l_B\mathbf{b}_{H,B} + l_C\mathbf{b}_{H,C} + l_D\mathbf{b}_{H,D}\right) \\ & +(1-w)\left(l_A\mathbf{b}_{L,A} + l_B\mathbf{b}_{L,B} + l_C\mathbf{b}_{L,C} + l_D\mathbf{b}_{L,D}\right)\end{aligned}$$

best matches the original texel $\mathbf{p}$, where l_k is the appropriate bilinear interpolation weight and $0 \leq w_i \leq 1$.

In order to better understand the problem of compressing into LFSM formats, we present an analytic formulation of the global optimization computation needed for this compression algorithm. Given a source RGBA image of dimensions $n_s \times m_s$ and pixels $\mathbf{p} = \{\mathbf{p}_R, \mathbf{p}_G, \mathbf{p}_B, \mathbf{p}_A\}$, we need to generate a

compressed image with dimensions $n_c \times m_c$ where $(n_c, m_c) = (n_s/r_n, m_s/r_m)$ for some $r_n, r_m \in \mathbb{N}$ (typical values are $r_n = 4$ and $r_m = 4$ or 8). The high and low colors $\mathbf{b}_H$ and $\mathbf{b}_L$ of the compressed image will be treated as $1 \times n_c m_c$ vectors. We must also determine modulation values w that correspond to the interpolation weights between the two bilinearly interpolated endpoints. Each channel of an image reconstructed from compressed data can be described by a matrix equation as follows:

$$\widetilde{\mathbf{p}_k} = WQ\mathbf{b}_{H,k} + (\mathbf{I} - W)Q\mathbf{b}_{L,k}, \tag{4.1}$$

where Q is the $n_s m_s \times n_c m_c$ matrix corresponding to the bilinear weights of each pixel given by

$$Q_{i,j} = \frac{1}{r_n r_m} \begin{cases} 0 & \text{if } |x_i - r_n x_j| \geq r_n \text{ or } |y_i - r_m y_j| \geq r_m, \\ (r_n - l_x)\,(r_m - l_y) & \text{if } r_n x_j \leq x_i \text{ and } r_m y_j \leq y_i, \\ l_x\,(r_m - l_y) & \text{if } r_n x_j > x_i \text{ and } r_m y_j \leq y_i, \\ l_y\,(r_n - l_x) & \text{if } r_n x_j \leq x_i \text{ and } r_m y_j > y_i, \\ l_x l_y & \text{if } r_n x_j > x_i \text{ and } r_m y_j > y_i, \end{cases}$$

$$\begin{aligned} x_i &= i \bmod n_s & y_i &= \left\lfloor \frac{i}{n_s} \right\rfloor \\ x_j &= j \bmod n_c & y_j &= \left\lfloor \frac{i}{n_c} \right\rfloor \\ l_x &= x_i \bmod r_n & l_y &= y_i \bmod r_m. \end{aligned}$$

W is the unknown diagonal matrix of dimensions $n_s m_s \times n_s m_s$ containing the modulation values w_i. The product WQ corresponds to both applying modulation weights and bilinear interpolation between the values $\mathbf{b}_H$. The values $\mathbf{b}_L$ are similarly weighted with the product $(\mathbf{I} - W)Q$. We have four of these systems, one corresponding to each channel $k \in \{R, G, B, A\}$ that are all coupled by W. This formulation gives us $n_s m_s + 8 n_c m_c$ unknowns (one for each w_i and one for each channel in each color $\mathbf{b}_H$, $\mathbf{b}_L$) and $4 n_s m_s$ equations. Since $(n_s, m_s) = (r_n n_c, r_m m_c)$, we will have as many equations as unknowns when $4 r_n r_m \geq r_n r_m + 8$. For any reasonable compression format, $r_n, r_m > 2$, so a solution exists in the continuous domain at least.

4.1.1.2 High Complexity

The first thing to notice about Equation (4.1) is that it is a non-linear system, which is typically solved using iterative solvers. Furthermore, for n_s, $m_s = 256$ and $r_n, r_m = 4$, the size of our solution vector will have dimension

$$n_s m_s + 8 n_c m_c = 2^8 2^8 + 2^3 2^6 2^6 > 2^{16}.$$

This is too large for any non-linear solver to find a solution to, especially in real-time.

We can reduce the complexity of the system by first using an approximation for W. In this case, we can bundle up the problem in Equation (4.1) by combining W and Q to get an equation of the form:

$$\mathbf{p} = A \begin{bmatrix} \mathbf{b}_H \\ \mathbf{b}_L \end{bmatrix}. \tag{4.2}$$

This becomes a linear system where the resulting matrix A is large, non-square, and high rank, since each row has $8 n_c m_c$ elements and thirty-two non-zero elements. The non-zero elements in row i are the elements corresponding to the channels in the high and low colors of the blocks that affect pixel i.

Finally, the biggest impediment to using this formulation to compute an efficient solution is the fact that the solution to our problem must be stored within an LFSM data format, which necessitates discretizing our values. Quantization of the solution to the $Ax = b$ problem from the real numbers to integers does not provide an optimal solution in general. This means that the aforementioned problem becomes an integer programming problem, which is known to be NP-Complete (von zur Gathen and Sieveking, 1978). As a result, computing the optimal solution is impractical.

4.1.2 LFSM compression using Intensity Dilation

Given the high complexity of computing the optimal solution of Equation (4.1), we present an alternative technique for real-time texture compression. The basis for our approach resides in the well studied foundations of the human visual system's sensitivity to contrast (Aydin, 2010; Thompson et al., 2011). In particular, our algorithm takes advantage of localized areas of an image that have high contrast

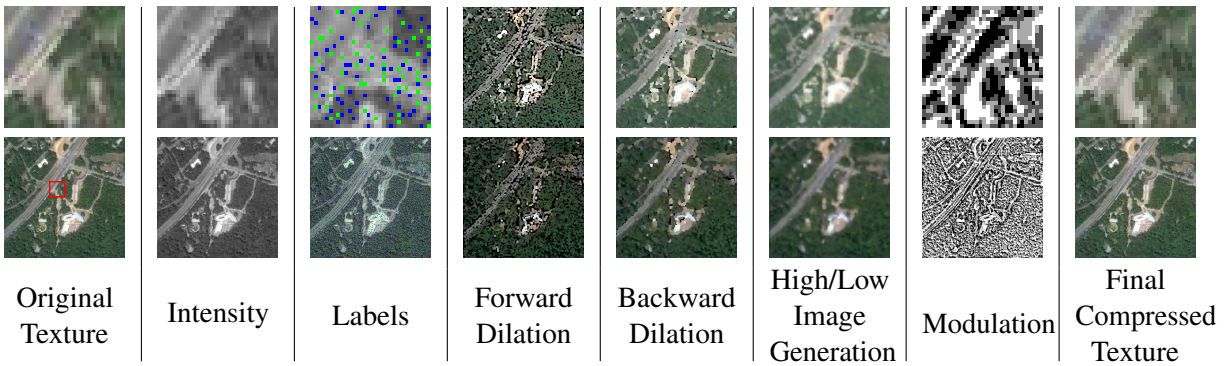


Figure 4.4: The different stages of the algorithm. Original texture: the texture we are compressing explicitly marked with an area of interest which is depicted in the zoomed in versions. Intensity: original image and zoomed in region in grayscale. Labels: labeled image and zoomed in region of texels with intensity values larger than their neighbors (green) and lower than their neighbors (blue). Forward dilation: after the first pass of the algorithm, both the high image containing local intensity maxima (top) and the low image containing local intensity minima (bottom) have been dilated forward. Backward dilation: after the second pass of the algorithm, both of the images have been completely dilated. High/Low image generation: Downscaled images that resulted from averaging all of the texels in a block of the dilated images. Modulation: computed optimal modulation values for the original image and the zoomed in region, given the computed high and low images. Final compressed texture: The resulting compressed texture and the corresponding zoomed in region. Original image retreived from Google Maps.

ratios. For most textures, these areas are those that contain edges between high intensity and low intensity regions.

Due to the way that LFSM compressed textures store the compressed data, as in Figure 2.1, there is an inherent filtering procedure that takes place during retrieval of texel data. During the bilinear upscale of the colors stored per block, adjacent blocks must maintain the extreme values in order to preserve the edges. In Figure 4.5, the optimal compression scheme to preserve the edge would be to store each color, red and blue, in all four blocks that cover the edge. The modulation can be used to choose the appropriate pixels from either of the two images. If any of the blocks have either red or blue as both high and low colors, then the result would never be able to fully encode the edge because the edge pixels would be blurred from the bilinear upscale. In order to fully encode areas of high contrast, such as edges across very different colors, the high and low intensity texels must be represented in all blocks that influence that region during decompression. In this section, we cover the basic principles behind the full intensity dilation algorithm, and present a two-pass approximation algorithm that performs the encoding.

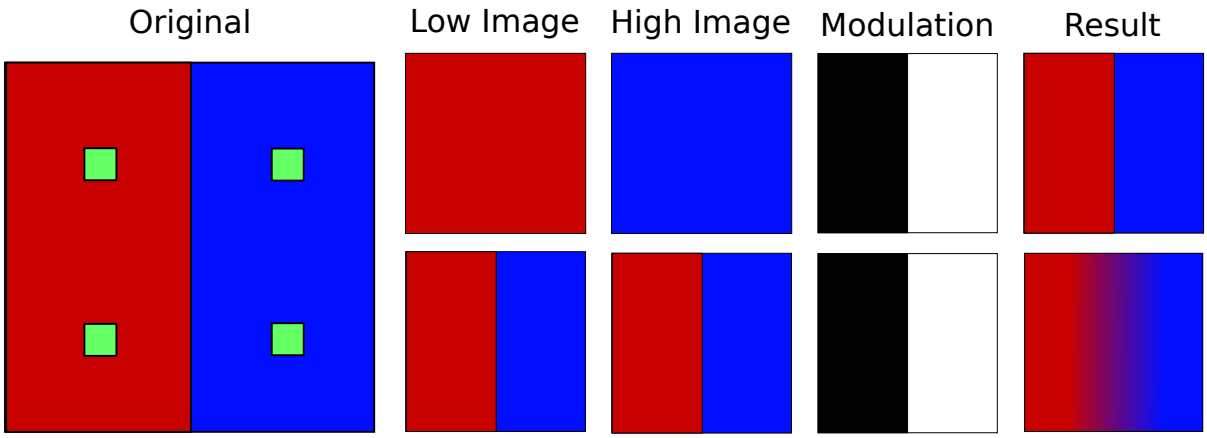


Figure 4.5: A red and blue texture is compressed using LFSM compression. Green positions are block centers. Since the border between blue and red areas aligns with block borders, the optimal compression is to store one color as the high color of each block, and another color as the low color of each block. The modulation data is used to reconstruct the original image.

4.1.2.1 Intensity Labeling

In order to preserve the contrast within textures, the first step in our compression scheme is to determine the high and low intensity values that produce the contrast. We start by using the definition for luminosity derived from the Y value of the CIE XYZ color space due to its speed and simplicity of calculation (on Illumination, 2004)

$$I(x) = R_x * 0.2126 + G_x * 0.7152 + B_x * 0.0722.$$

Other luminance values, such as the L channel of CIE L^*a^*b are also viable alternatives for computing the luminance. For textures with alpha, we premultiply the alpha channel across each color channel before performing the luminance calculation.

There are many ways to determine the local minima and maxima of intensity, including searching for a near-zero magnitude gradient or evaluating the eigenvalues of the Hessian. A simple alternative is to simply look at the intensity value of each of the neighboring pixels. If all of the neighbors have higher intensity values or all of the neighbors have lower intensity values, then the pixel in question is a local minimum or local maximum, respectively. Once we have determined these local minima and local maxima, we can separate them into two images, one representing all local minima, and the other representing all local maxima.

4.1.2.2 Intensity Dilation

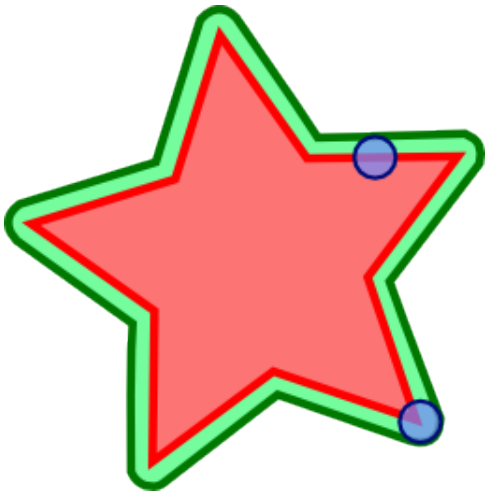
In order to capture the contrast features of an image, we propose the use of a technique from mathematical morphology known as *dilation* (Serra, 1983). Usually applied to binary images, dilation is the use of a small kernel shape, such as a 3×3 pixel box, to expand a region of pixels. Figure 4.6 shows how a star can be dilated by using a small disk to create a larger star with rounded corners. In LFSM texture compression, the input textures have at least three 8-bit channels that must be dilated. When an empty pixel is adjacent to two or more non-empty pixels there must be a strategy for how to perform the dilation. In our method, as shown in Figure 4.6, we have chosen to average adjacent pixel values in order to preserve the color range that corresponds to a block. This reduces the amount that noise affects our choice of block colors. One alternative is to take the texel with the higher or lower intensity based on the image being dilated, but this causes problems with noisy images.

In order to completely capture the important features of a texture, the intensity labels of the image (Section 4.1.2.1) must be dilated until they influence neighboring block values. In LFSM, blocks cover $r_n \times r_m$ pixel regions. This implies that any pixel $\mathbf{p}$ at location (p_x, p_y) affected by a block $\mathbf{b}$ centered at (b_x, b_y) is at most d units away, where d is defined as

$$d = \sup_{p_x, p_y} \left\{ ||\mathbf{b} - \mathbf{p}||_1 : |b_x - p_x| < r_n \text{ and } |b_y - p_y| < r_m \right\}.$$

In order to properly influence the colors of a block that covers a given labeled pixel, we must dilate each of the extrema d times.

Once dilated, each block will represent the major local influences of either low or high intensity depending on the image. The resulting block color will be the average of the intensities within the block boundaries. Certain areas, such as color gradients, contain very few local minima or maxima and may not have any dilated texels. In order to prevent these areas from being influenced by texels relatively far from the block center, we fill empty texel values with the corresponding extrema color. Once we have the high and low colors corresponding to a given block, we are free to compute optimal modulation values to match our original pixel colors. We compute the modulation values by locally decompressing the high and low colors and bilinearly upscaling them to get the proper interpolation extremes. We then



2	2	2	2	2	2		
2	2	2	2	2	$\frac{38}{25}$	1	1
2	2	2	2	$\frac{8}{5}$	1	1	1
2	2	2	2	1	1	1	1
2	2	2	$\frac{7}{5}$	1	1	1	1
2	2	$\frac{37}{25}$	1	1	1	1	1
		1	1	1	1	1	1
		1	1	1	1	1	1

Figure 4.6: Examples of dilation. Left: a red star is dilated by a smaller circle into the green star with rounded corners. Right: Two pixels, denoted in green, are dilated three times using a 3×3 pixel box. If an empty pixel $\mathbf{p}$ is to be filled during dilation from multiple pixels $\mathbf{q}_i$ of different values, then the value stored for $\mathbf{p}$ will be the average of the $\mathbf{q}_i$. The picture is labeled with the values that the pixels would take after dilation of the initial pixels. The pixels that have fractional labels denote the value that they would have taken between labels one and two.

choose the optimal interpolation weight based on the restrictions imposed by the format. For a complete overview of the algorithm, see Figure 4.4.

4.1.3 Two Pass Algorithm

In the previous sections, we present an approach which requires eight stages with a simple implementation. One to convert the image to intensity values, one to label the maxima/minima, and three to perform the dilation for each image containing intensity minima and maxima. In this section, we propose a scheme to approximate this pipeline using two passes: a forwards and a backwards pass. For each pixel $\mathbf{p}$, we store a per-pixel cache that lazily stores an intensity value, a high label, and a low label. In other words, we do not compute the intensity value until it is needed, at which point we store it for future use. Each label has a distance value $dist\,(\mathbf{p})$, and a list of indices into the pixels that correspond to the maxima or minima that the current pixel is dilated from.

Forward Pass: We traverse pixels from left to right starting at the top-left corner of the image. At each pixel $\mathbf{p}$, we determine whether or not it is an extrema by looking at neighboring intensity values. Whenever an intensity value is computed, it is subsequently cached to avoid further computation. If the pixel is a local maximum or local minimum, we set $dist(\mathbf{p}) = 0$, and continue. If the pixel is not a

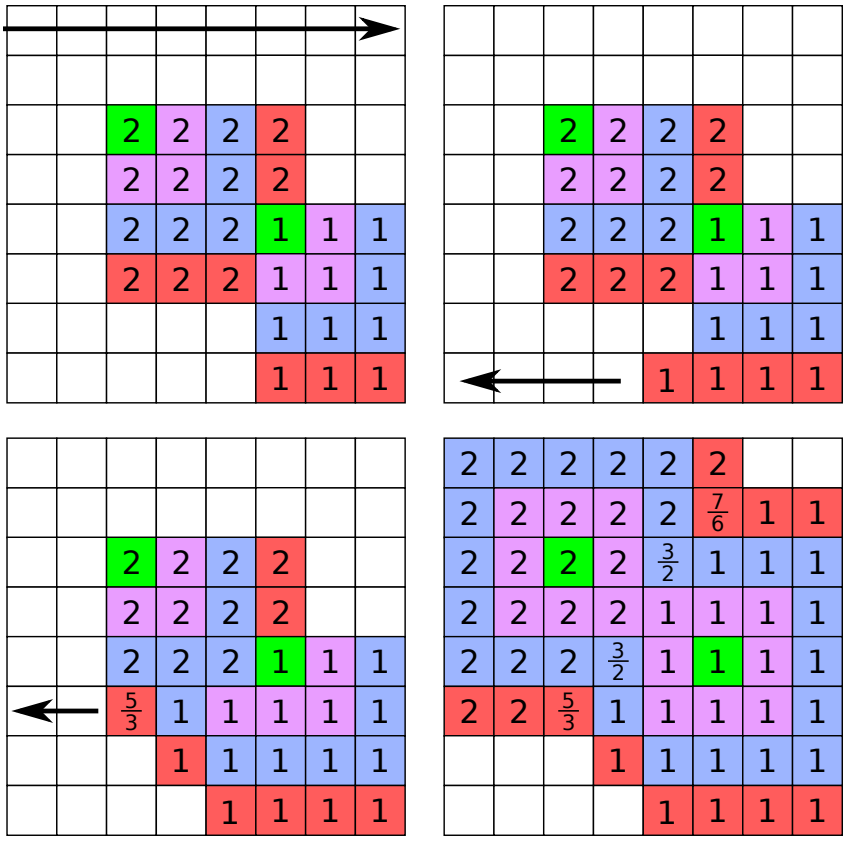


Figure 4.7: Our fast approximate dilation strategy. We perform the extrema calculation and dilation in two passes. Top Left: First pass, traverse the pixels from left to right, top to bottom labeling and dilating extrema in the order of traversal as we encounter them. Top right, bottom left, bottom right: Second pass, traverse the pixels from right to left, bottom to top. At each pixel, assign the label corresponding to the average of the pixels with the lowest distance to their respective labels.

local extrema, we investigate the values to the left and above the pixel to determine its distance from a local extrema. We need not look at any other neighbors due to the direction of this iteration. We also assume the painter's algorithm, so that if two or more local extrema conflict, they will be overwritten (see Figure 4.7).

Backward Pass: After we have labeled the pixels with their local extrema in one direction, we may proceed to dilate the pixels in the opposite direction. We dilate backwards by starting in the bottom right corner and proceed from right to left. At each pixel $\mathbf{p}$, both labeled and unlabeled, we find the set of neighbors of $\mathbf{p}$, $\{\mathbf{c}\}$, that have the least value $d = dist(\mathbf{c})$. If d is already the maximum number of dilations or $dist(\mathbf{p}) < d + 1$, then we ignore this texel. If $dist(\mathbf{p}) = d + 1$, then we concatenate $\{\mathbf{c}\}$ to the list for $\mathbf{p}$. Otherwise, we simply change the list for $\mathbf{p}$ to be $\{\mathbf{c}\}$ and set $dist(\mathbf{p}) = d + 1$.

After both passes, the list of indices stored at each pixel are approximately those pixels that would contribute to the final color of the pixel during a decoupled dilation of the extrema. This approximation can be seen in the difference between the final labels in Figures 4.6 and 4.7. During backwards dilation of a pixel $\mathbf{p}$, we do not have the proper information yet about whether or not pixels have dilated to the left above $\mathbf{p}$. This problem is most noticeable by the missing pixels in the bottom row of the bottom-right image in Figure 4.7. This can be mitigated by handling the special case whenever we place a non-maximally distant label above a label with a larger distance.

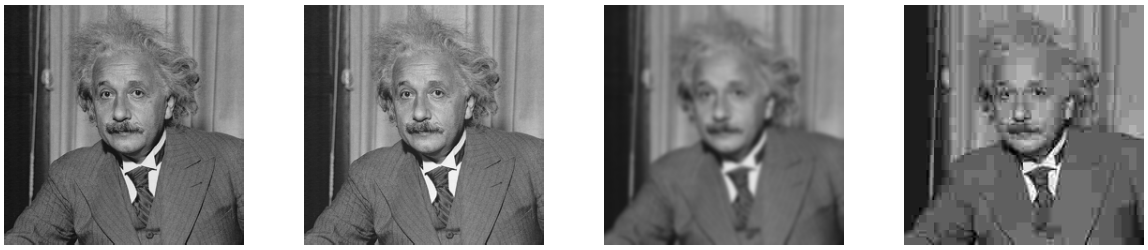
Once both passes are complete, at each pixel we have stored the intensity, and the closest minimum and maximum intensity pixels. When averaging the pixels in each block of the high and low images, we can simultaneously find the minimum or maximum intensity for the block. For a $n \times m$ block with N non-labeled pixels, we store as block colors the sum of each averaged label weighted with $\frac{1}{nm}$ and the pixel corresponding to the minimum or maximum intensity weighted with $\frac{N}{nm}$.

4.1.4 Results

The only LFSM texture compressor known to the authors is Imagination's PVRTexTool, which we use to compare the speed and quality of our algorithm. It incorporates the two stage compression technique described by Fenney (2003). The following comparisons all use the fastest setting for the compressor and are not focused on quality compression. They do not represent the best possible quality achievable by PVRTC or LFSM in general. Also, our results focus on the 4bpp version of PVRTC, but similar methods should be useful for both 2bpp and future iterations of PVRTC. Although the compressor is closed source, the decompressor provided with the SDK was used to verify the results (Imagination, 2013).

4.1.4.1 PSNR vs SSIM

Classically, the quality of texture compression techniques have always been measured with *Peak Signal to Noise Ratio* (PSNR) (Fenney, 2003; Ström and Pettersson, 2007; Nystad et al., 2012; Krajcevski et al., 2013). This metric originates from signal processing and corresponds to the amount of absolute difference between pixel values. When compressing textures, such a metric can be useful, such as when we need to encode a 2D function as a texture. However, in LFSM compressed textures, decompression



PSNR: ∞ SSIM: 1.0 | PSNR: 26.547 SSIM: 0.9884 | PSNR: 26.550 SSIM: 0.6940 | PSNR: 26.609 SSIM: 0.6640

Figure 4.8: Problems with using PSNR as the only metric. Each image above has a similar PSNR to the original image on the far left. Images courtesy of Wang et al. (2004).

focuses on a filtered representation of the compressed data and is mostly designed for textures that will be consumed visually. As shown in Figure 4.8, PSNR does not correlate with visual fidelity.

For this reason, we also include SSIM, a metric developed by Wang et al. (2004) that captures differences of two images as perceived by the human visual system. The metric is defined as

$$SSIM(I_x, I_y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)},$$

where μ is the mean intensity and σ is the standard deviation, and σ_{xy} is defined as

$$\sigma_{xy} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)(y_i - \mu_y).$$

C_1 and C_2 are application-defined constants to avoid numerical instability. One limitation of SSIM is that it only measures a single channel. In the subsequent comparisons, we measure SSIM by first converting both the original and compressed image to grayscale.

4.1.4.2 Compression Speed

The main benefits of using intensity dilation over previous techniques is in compression speed. Looking at Table 4.1, we observe a 3.1x speedup over the previous fastest implementations. Similar to other texture compression algorithms, we optimize away areas of homogeneous pixels with precomputed lookup tables (Waveren, 2006a; Krajcevski et al., 2013). Furthermore, textures that contain a lot of homogeneity such as the 'streets' texture in Table 4.1 gain a small benefit from the instruction cache since intensity calculations will reuse texel values. However, as we will see in Section 4.1.4.3, we suffer


streets
256×256


gradient
256×256


satellite
256×256


mountains
256×256


bricks
256×256


gametex
512×512

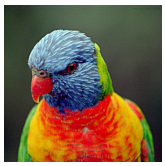

lorikeet
512×512

Image	SSIM		Peak Signal to Noise Ratio	
	Our Method	PVRTexTool	Our Method	PVRTexTool
streets	0.9666	0.9850	33.63	32.35
gradient	0.9797	0.9673	30.17	30.97
satellite	0.9488	0.9180	32.09	30.43
mountains	0.9138	0.9620	30.01	34.47
bricks	0.9476	0.9331	27.44	26.35
gametex	0.9531	0.9225	30.20	29.80
lorikeet	0.9455	0.9111	30.75	31.37

Image	Compression Speed (ms)		Entropy
	Our Method	PVRTexTool	
streets	17.76	63.74	1.546
gradient	20.30	65.11	7.528
satellite	20.93	64.92	6.963
mountains	21.77	66.63	3.960
bricks	21.91	65.45	7.468
gametex	97.25	264.68	6.552
lorikeet	97.39	263.14	7.386

Table 4.1: Various metrics of comparison for LFSM compressed textures using intensity dilation versus the existing state of the art tools. All comparisons were performed using the fastest quality settings of the February 21st 2013 release of the PVRTexTool (Imagination, 2013). For both metrics, higher numbers indicate better quality. The above results were generated on a single 3.40GHz Intel® Core™ i7-4770 CPU running Ubuntu Linux 12.04. Images courtesy of Google Maps, Simon Fenney, and `http://www.spiralgraphics.biz/`

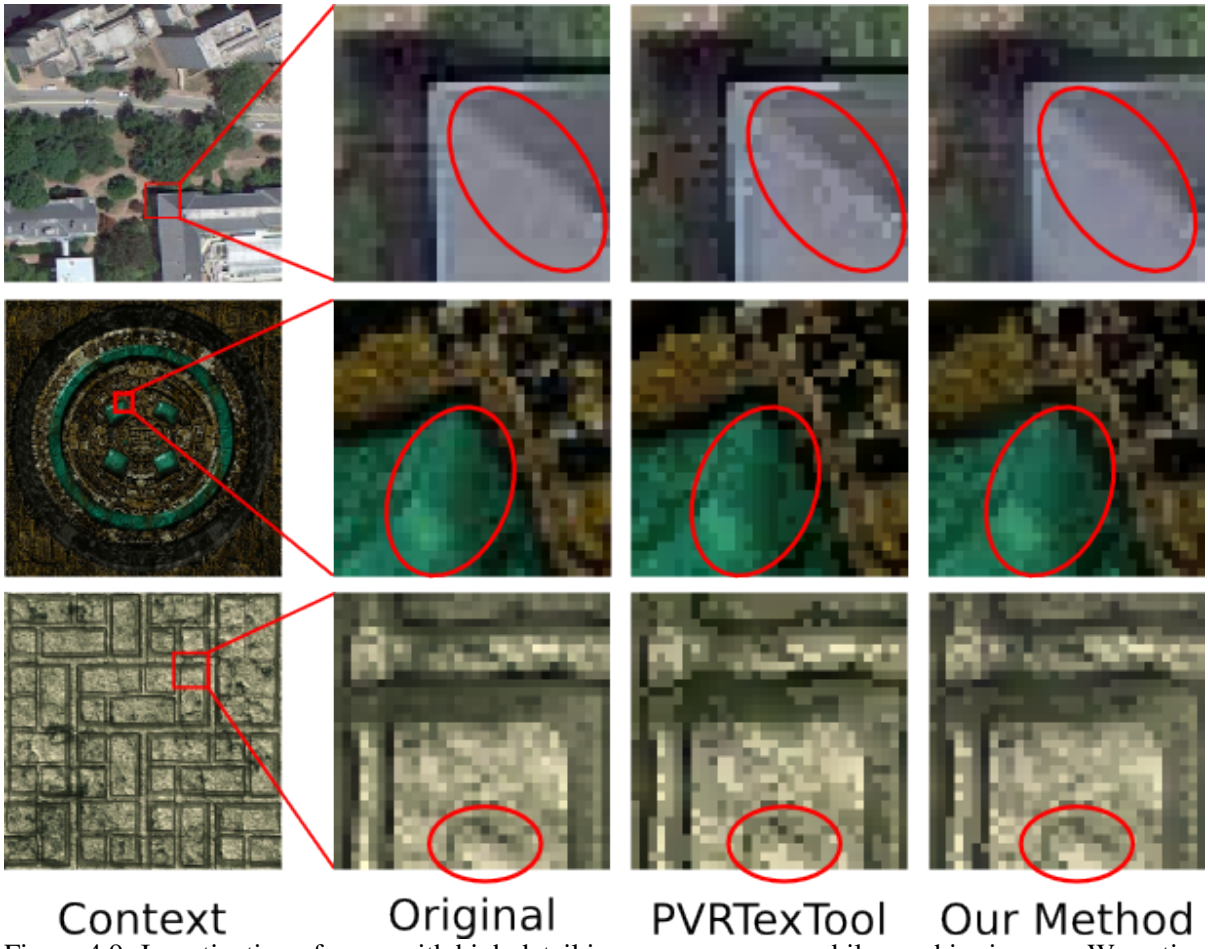


Figure 4.9: Investigation of areas with high detail in some common mobile graphics images. We notice that the texture compressed using intensity dilation maintains the smoothness of many image features, while the original PCA based approach leaves blocky streaks. Images courtesy of Google Maps, Simon Fenney, and `http://www.spiralgraphics.biz/`.

from aggressive averaging artifacts in these areas. Most images do not have large homogeneous areas, and consequently compression speed is tightly correlated with the size of the texture.

A majority of the speedups in our method come from minimizing the number of times that we traverse the entire texture. In doing so, we minimize the number of penalizing cache misses. Furthermore, during the optimized dilation step described in Section 4.1.3, the per-pixel cache that stores list of *indices* to pixels means that we are not averaging pixels until the very end. This also has the benefit of being cache friendly by avoiding costly memory lookups during the dilation process.

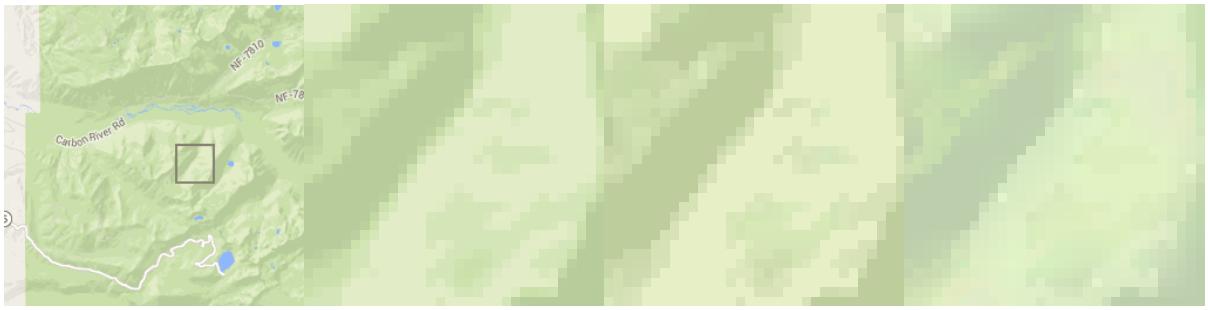
4.1.4.3 Compression Quality

Compressing textures using intensity dilation, we observe an increase in the SSIM index for a majority of textures and maintain similar results in PSNR. Most notably, we can see that certain low frequency features are retained in the compressed versions of many textures with high entropy. In Figure 4.9, the differences between the two methods are noticeable. Due to intensity dilation, the averaging during dilation around the edges of the roof prevents compression artifacts from arising due to local extrema. This is noticeable across all images that have low frequency features, such as photographs or billboard textures.

Although our technique is useful for this class of textures, we also observe a class of textures that perform poorly with intensity dilation. These textures correspond to the relatively low entropy texture 'mountains' (Table 4.1) generated from vector graphics and used in some modern day geomapping applications. We measure entropy using the common formula from 8-bit intensity values (Shannon, 1948):

$$E = -\sum p_i \log p_i,$$

where p_i is the number of pixels with intensity i divided by the total number of texels. This is not a steadfast metric of when our algorithm performs poorly due to the metric's lack of spatial coherence, but it does provide a good intuition for when intensity dilation may not produce favorable results. Many spatially correlated areas of moderate homogeneity result in overaggressive extrema labeling. The problem arises from the fact that in homogeneous regions of pixels, there is no maximum or minimum. In these instances, either no maximum or minimum exist, and the high and low images will take the maximum and minimum intensity pixel, which is the same value, or every pixel is a maximum and a minimum, so the dilation aggressively eliminates small scale image features. In the worst case, this problem occurs when there are very few colors in a block's region: on the order of two or three. Then every pixel becomes labeled as both a maximum and a minimum, and blurring occurs which removes image detail, as shown in Figure 4.10.



Context Original PVRTexTool Our Method

Figure 4.10: Detailed investigation of areas with high pixel homogeneity. Unlike the images in Figure 4.9, we notice that the texture compressed using intensity dilation suffers from artifacts arising from aggressive averaging of nearby intensity values, while the PCA based approach has relatively good quality compression results. Original image retreived from Google Maps.

4.2 Advanced Endpoint Formats

The main consideration for a texture compression algorithm is to efficiently produce a stream of bits, or encoding, that when decompressed recreates the original image as accurately as possible. In this section, we formally define the problem of compressing low-dynamic range data for endpoint texture compression formats. We split the problem into three parts: *partition selection*, *endpoint estimation*, and *endpoint refinement*. We present an algorithm that selects partitions using the real-time technique described by Waveren (2006a) and uses a generalized cluster fit along with simulated annealing to determine the optimal endpoint selection (Brown, 2006; Kirkpatrick et al., 1983). In the following sections, we assume that all textures are encoded using eight-bit RGB channels, although the precision of the input may vary.

4.2.1 Background

Currently, the most popular texture compression formats, including S3TC, BPTC, and ASTC, encode RGB data by defining a line segment in RGB space and assigning an index value to each texel. This index is used to interpolate between the endpoints of the line segment in order to reproduce the original color value. The formats require the compressor to accurately select values for the endpoints given a block of texels, as shown in Figure 4.11. Since low dynamic range color values are usually described with eight-bit color channels, they can be interpreted as points along a 256^3 lattice. The endpoints for each set of texels are usually encoded with a lower per-channel precision, meaning that they exist on a

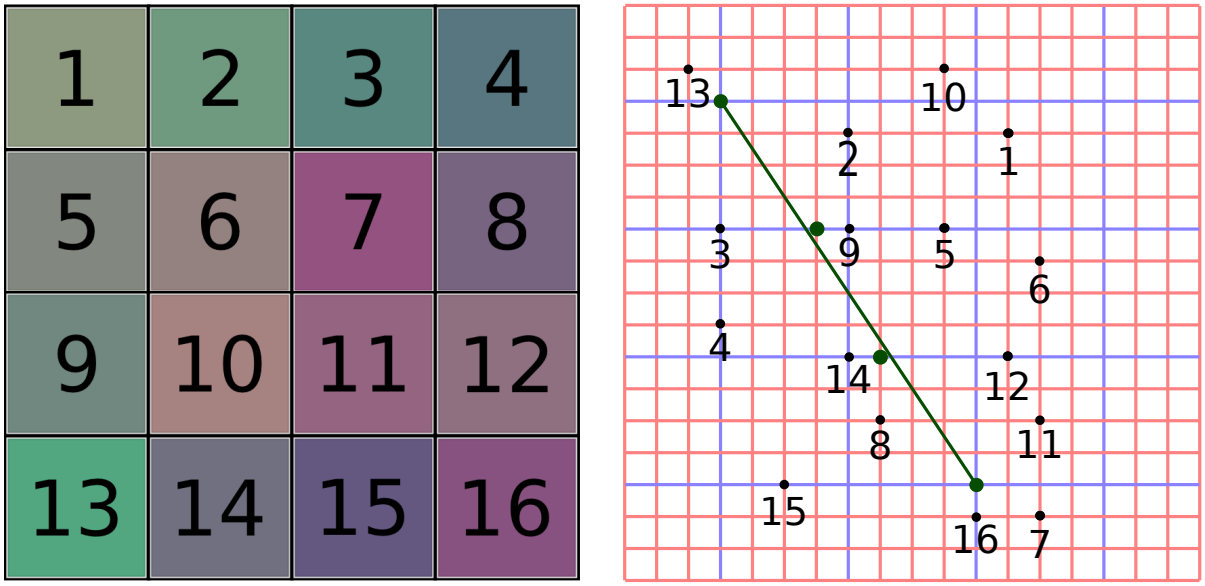


Figure 4.11: *(right)* A 4×4 block of texels. *(left)* The texels approximated with two-bit indices. The texels are interpreted as points on a lattice defined by the precision of the source texture (red). The endpoints approximating the texels are on a sparse lattice (blue) and the interpolation points are in green. For two bits per index we have $2^2 = 4$ interpolation points. Note: The internal interpolation points do not lie on the line segment due to quantization.

sparse lattice overlaid on the original. The goal of a compression algorithm is to compute the best points that lie on this lattice and per-texel indices that together reconstruct the original texel values.

Since S3TC is among one of the most popular compression formats, it has also been extensively investigated in terms of endpoint compression. S3TC operates on blocks of size 4×4, and stores compressed data with a single line segment whose endpoints use 5, 6, and 5 bits for red, green and blue respectively with two bits per index. Simon Brown's *libsquish* uses a method known as a cluster-fit. In this method, the 16 texel values $\mathbf{p}_i$ are first ordered along their principal axis (Brown, 2006); then each 4-clustering that preserves this ordering is used to solve the following least-squares problem

$$\min_{\mathbf{a},\mathbf{b}} |(\alpha_i \mathbf{a} + \beta_i \mathbf{b}) - \mathbf{p}_i|$$

where (α_i, β_i) are determined by the cluster that $\mathbf{p}_i$ belongs to:

$$(\alpha_i, \beta_i) \in \left\{(1,0), (\tfrac{1}{3}, \tfrac{2}{3}), (\tfrac{2}{3}, \tfrac{1}{3}), (0,1)\right\}$$

The endpoints $\mathbf{a}$ and $\mathbf{b}$ are then snapped to the lattice induced by the endpoint precision, and the result with the smallest error, as described in Section 4.2.1.1, is chosen. This algorithm is also the basis of Castaño's real-time GPU implementation (Castaño, 2007).

A CPU-based real-time algorithm was first introduced by J.M.P. Van Waveren that computes the diagonal of the axis-aligned bounding box (AABB) of the texels in color space, and uses the resulting diagonal as endpoints (Waveren, 2006a). Although this algorithm does not produce results as high in quality as the NVIDIA Texture Tools, it is fast enough to support compression of textures generated at run-time, such as the frame buffer (Donovan, 2010). Our approach expands upon these ideas in order to allow content pipeline designers to compress textures at a higher quality.

4.2.1.1 Problem Formulation

Formally, a compression method for endpoint-based texture compression takes as input n texels $\mathbf{p}_i = \left(p_i^r, p_i^g, p_i^b\right)$, a triplet $\zeta = (\zeta_r, \zeta_b, \zeta_g)$ that denotes the number of bits per channel in the compressed endpoint data, and an integer $\mathbf{I}$ specifying the number of bits per index. The output of the compression method is a pair of endpoints $(\mathbf{p}_a, \mathbf{p}_b)$ and n index values d_i, with $0 \leq d_i < 2^{\mathbf{I}}$. Here p_a^k and p_b^k, with $k \in \{r, g, b\}$, are specified with ζ_k bits. The goal of a compression method is to minimize the total error of the compressed texels, i.e.

$$\min_{\mathbf{p}_a, \mathbf{p}_b} \Phi(\mathbf{p}_a, \mathbf{p}_b),$$

where $\Phi(\mathbf{p}_a, \mathbf{p}_b)$ is the endpoint compression error defined as

$$\Phi(\mathbf{p}_a, \mathbf{p}_b) = \left[\sum_{i=0}^{n} \sum_{k \in \{r,g,b\}} \left| \frac{p_a^k \left(2^{\mathbf{I}} - d_i - 1\right) + p_b^k d_i}{2^{\mathbf{I}} - 1} - p_i^k \right| \right].$$

The values of d_i are inferred from the given endpoints $\mathbf{p}_a$ and $\mathbf{p}_b$ and the input data $\{\mathbf{p}_i\}$ by assigning each input value to the closest interpolation point between $\mathbf{p}_a$ and $\mathbf{p}_b$. This minimization problem is a special case of the quadratic integer programming problem, and it can be reduced to computing the shortest vector on a lattice (SVP). SVP is known to be NP-Hard (van Emde Boas, 1981; Ajtai, 1998) and most techniques use approximation techniques to estimate the optimal solution. We refer to this specific instance of the problem outlined above as the *endpoint optimization problem*. The triplet ζ is known as

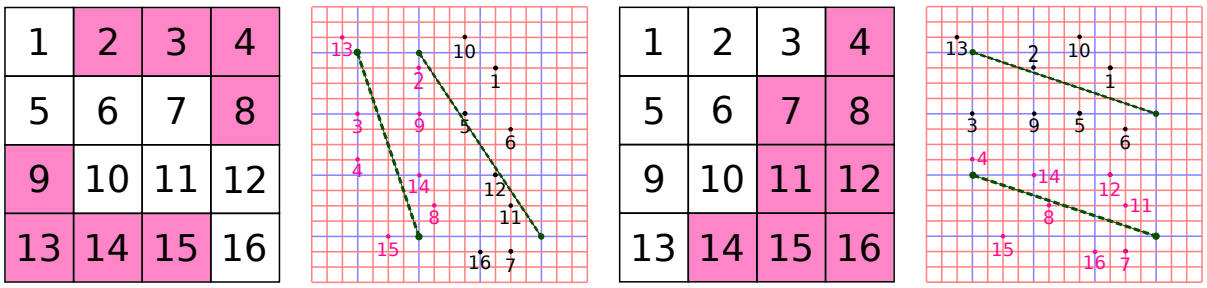


Figure 4.12: A 4×4 block partitioned by different P-shapes into two subsets from the BPTC format. P-shape partitioning is determined based on a lookup into a table of common partitionings. The texels marked with a pink background belong to one subset and the unmarked texels belong to another. Each subset is approximated with its own line segment (in green). *(left)* P-Shape #31 *(right)* P-Shape #4

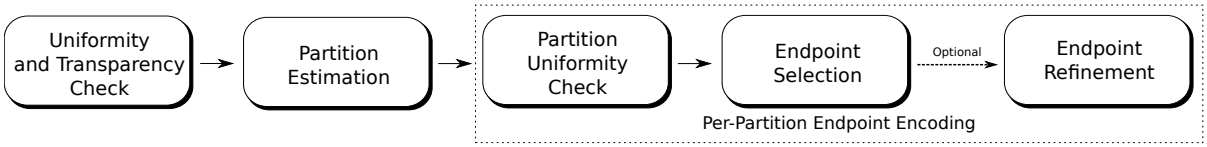


Figure 4.13: Overview of FasTC which is applicable to all endpoint based compression formats that support partitioning.

the *endpoint precision*, and $\mathbf{p}_a$, $\mathbf{p}_b$, and $\{d_i\}$ are collectively known as a *palette*. The number of bits per index $\mathbf{I}$ is known as the *index precision*.

4.2.1.2 Choosing a Partitioning

As described in Section 2.3, some modern texture compression formats support partitioning the texels into subsets as illustrated in Figure 4.12. Due to the limited number of bits that are allocated for partition selection, compressors must choose from a fixed set of preselected partitionings called *P-shapes*. For example, the BPTC format has 64 separate P-shapes for two and three subset partitionings for a total of 128 (OpenGL, 2010). In order to properly select a P-shape, many compressors chose from a partial ordering and then perform a full compression on each P-shape to chose the best one (Nystad et al., 2012)(Dufresne, 2013). For certain compression formats, such as 12×12 ASTC, the P-shape space contains 3123 unique P-shapes (Nystad et al., 2012). For large textures, P-shape selection becomes a very expensive part of the algorithm.

4.2.2 Partition Estimation

FasTC operates on the BPTC format. We have chosen this format because of the availability of compressors and hardware support for decompression in current GPUs. We expect all of the methods described in this chapter to be equally applicable to other block-based compression formats such as ASTC. Figure 4.13 gives an overview of our algorithm.

We consider the input to be a square block of texels. The first step of the algorithm is to check whether or not all of the texels are uniform or transparent. If not, we estimate the best partitioning for the block as described in Section 4.2.1.2. Next, for each subset in the partition, we perform another uniformity check and approximate the best endpoints to use (Section 4.2.4). We discuss an optional refinement step that searches for better solutions using simulated annealing (Section 4.2.5).

Blocks of a constant color can be handled as a special case. For a given index and endpoint precision, most eight-bit values can be compressed with an index value of one. For example, with six-bit endpoint precision and two-bit index precision, the value 73 is exactly encoded as the first index between endpoints (64, 92) after integer truncation. For a uniform block, if we assume that each texel will have an index value of one, we can find the optimal endpoints for this block by saving the optimal endpoints per channel in a lookup table with 256 entries. In this manner, we can effectively discard uniform blocks and partitions. This procedure corresponds to the uniform and transparency checks in Figure 4.13

The quality of a P-shape with respect to the block's texels is determined by the interrelationships of the texels in each subset and the input parameters to the compression algorithm. If we ignore endpoint precision, the best P-shapes are those whose subsets provide clusterings of the data that line up along interpolation points on a line segment. Mathematically, the collinearity of the texels in RGB space would be a good measurement of their ability to be approximated in this way. Collinearity of a point cloud is measured by taking the eigenvalues $\{\lambda_i\}$ of the covariance matrix and comparing the first and second maximal eigenvalues. If the second eigenvalue is small in comparison to the first, then the variation from the principal axis of the point cloud is minimal. Based on this observation, we can posit that the best P-shape estimation would be one whose average ratio λ_1/λ_0 is small for each subset, where λ_0 and λ_1 are the first and second largest eigenvalues, respectively. However, this method for P-shape estimation has the disadvantage that we must compute eigenvalues for each of the hundreds of possible P-shapes

in our compression format for each block. As we demonstrate in Section 4.2.7, this eigenvalue method incurs significant performance penalties and may not provide good estimates due to quantization artifacts.

A modification of this approach is to calculate the Euclidean distance from points to the principal axis. Instead of computing the first and second eigenvalues of the covariance matrix, one only needs to compute the principal eigenvector, define a line segment with the extremes of the points projected onto this axis, and use this segment as an estimated solution to the endpoint optimization problem for each subset. In fact, some encoders, such as NVIDIA's compressor, perform this step in order to estimate the quality of a given P-shape (Donovan, 2010).

Due to the discrete nature of endpoint optimization, any approximation that relies on the continuity of the domain may introduce inaccuracies. Instead of performing principal component analysis, we propose an approximation to each subset to be the amount of error it would create if it were encoded using the real-time CPU based method introduced by Waveren (2006a), which we will refer to as *bounding-box estimation*. Bounding box estimation uses the diagonal of the axis-aligned bounding box (AABB) as an estimate for the line segment that solves the endpoint optimization problem. It also provides an efficient check for uniformity by measuring the length of the diagonal. We present the following metric for approximating the quality of a given P-shape:

$$e\left(\{\mathbf{p}_i\}\right) = \Phi\left(\psi_+\left(\{\mathbf{p}_i\}\right), \psi_-\left(\{\mathbf{p}_i\}\right)\right)$$

where

$$\psi_+\left(\{\mathbf{p}_i\}\right) = \left(\max_i(p_i^r), \max_i(p_i^g), \max_i(p_i^b)\right),$$

$$\psi_-\left(\{\mathbf{p}_i\}\right) = \left(\min_i(p_i^r), \min_i(p_i^g), \min_i(p_i^b)\right)$$

The index precision $\mathbf{I}$ and endpoint precision ζ which influence the value of Φ should be chosen based on the encoding format. In our implementation, we use $\zeta = (8, 8, 8)$ and $\mathbf{I} = 2$ for three-subset P-shapes and $\mathbf{I} = 3$ for two-subset P-shapes. This approximation provides the main speed-up for our algorithm. For each block it must be performed twice per each of the 64 P-shapes in BPTC (once for two-subset P-shapes, and once for three-subset P-shapes). The ramifications of this approximation, with respect to compression quality and speed, are given in Section 4.2.7.

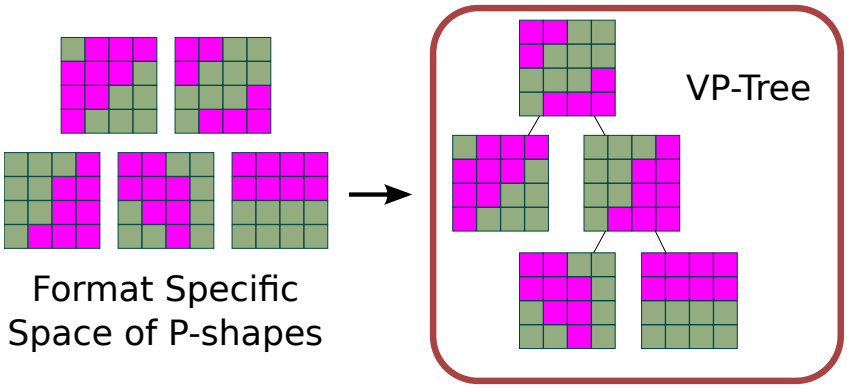
4.2.3 Partition Selection using Image Segmentation

We also demonstrate a new P-shape selection method, SegTC, based on image segmentation, as described in Figure 4.14. First, we segment the texture into *superpixels* as described in Section 4.2.3.1. As discussed in the beginning of this chapter, encoders for partition-based compression formats must implement two separate stages. The superpixels computed by the segmentation algorithm are used in the first stage of our approach for fast P-shape selection. The second stage of our approach computes compression parameters for each subset described in Section 4.2.4. During image segmentation, each pixel is labeled with a superpixel index. P-shapes are then chosen by considering the labels of the pixels within the block being compressed. As described in Section 4.2.3.2, this subset of labels is used as the target for a nearest-neighbor search of the available P-shapes. The P-shapes with the closest matching partitioning are used in the cluster-fit algorithm.

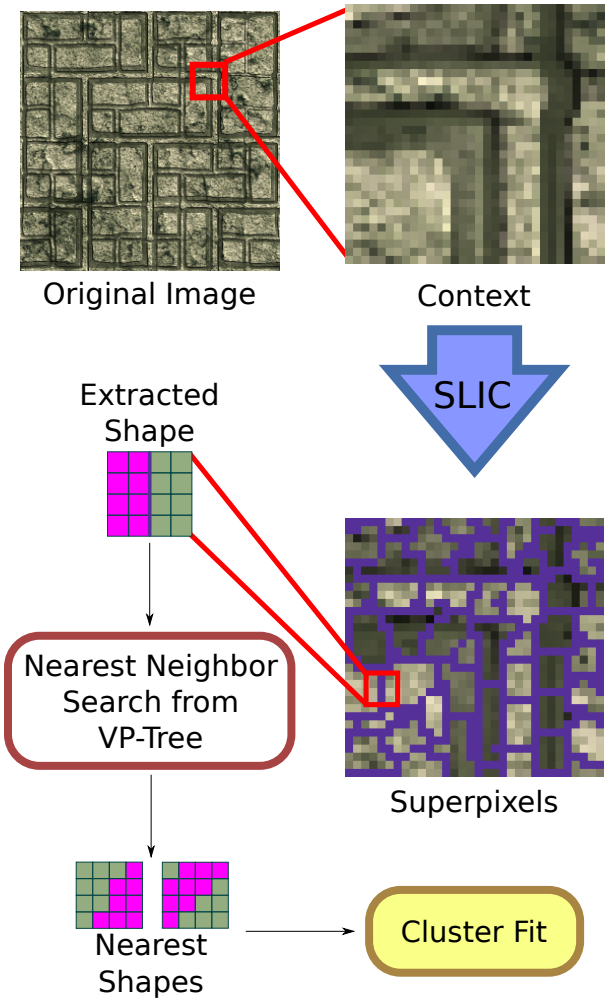
4.2.3.1 Segmentation

In this section, we describe the segmentation algorithm that underlies the P-shape selection method. Image segmentation has been heavily studied in computer vision and image processing (Achanta et al., 2012). The goal of image segmentation is to apply a labeling to each pixel such that pixels sharing a common label all share a common property or visual characteristic. This property is often the minimization of an metric used in distinguishing image features. Recently, there has been much work in segmenting the image into large contiguous regions of pixels known as *superpixels* (Felzenszwalb and Huttenlocher, 2004; Veksler et al., 2010; Achanta et al., 2010). Such a partitioning provides a classification into areas of pixels that admit certain coherence properties. We find that partitioning blocks by superpixel boundaries is suitable for texture compression.

We use a superpixel segmentation method known as SLIC, or Simple Linear Iterative Clustering (Achanta et al., 2010). In order to maintain encoder performance we chose this method for its simplicity and speed versus other methods (Achanta et al., 2012). SLIC takes as parameters either the number or desired size of the superpixels. Using this parameter, SLIC uses equally spaced kernels over the texture as the initial cluster centers for a k-means clustering algorithm. Once the clustering is computed, any pixels that are not contiguously connected to their cluster centers then 'push' the superpixel border to connect the components. The error metric chosen to determine the distance between



(a) Preprocessing



(b) Runtime

Figure 4.14: An overview of the SegTC compression algorithm. (a) The VP-Tree is constructed from format-specific P-shapes as a preprocessing step. (b) For each image, we perform SLIC segmentation. For each block, we extract the corresponding partitioning that matches the superpixel boundaries and find the nearest P-shapes using the VP-Tree. The closest P-shapes are used with the cluster-fit algorithm to produce the final compression parameters.

Original Context SLIC

Figure 4.15: (left) The original image. (center) An investigation of the area highlighted in teal. (right) SLIC superpixels: the image is segmented into small regions that adhere to feature boundaries (Achanta et al., 2010).

two pixels is a key issue with respect to SLIC. For each pixel p, we calculate the distance from the pixel coordinates (x, y) and the pixel value converted to CIE LAB space (L, a, b) as

$$d(p_1, p_2) = \begin{aligned} &\alpha \|(x_1, y_1) - (x_2, y_2)\|_2 \\ &+ \beta \|(L_1, a_1, b_1) - (L_2, a_2, b_2)\|_2 \end{aligned}$$

where α and β are values chosen to weigh the relative contribution(on Illumination, 2004). Although most compression formats operate in RGB space, we segment the image in CIE LAB space in order to leverage the fact that euclidean distance is correlated to perceived difference. Different error metrics may provide better compression values for specific textures because of the high variability of information types stored in textures.

4.2.3.2 P-shape Selection

Once the target partitioning has been selected from the segmented image, we proceed by finding the best P-shape defined by our compression format that matches it. The target partitioning rarely matches exactly to any of the predefined P-shapes. In order to properly compare one partitioning to another, we must define an error metric between partitionings. Furthermore, since the P-shape space does not change

between textures for a given compression format, we can accelerate the search by using a data structure to perform efficient nearest neighbor lookups using the metric described in Section 4.2.3.3

Some formats, such as ASTC, use blocks as large as 12x12 pixels, meaning that our partitionings would have up to 144 variables. The high dimensionality per P-shape prevents the use of classical data structures such as k-d trees because they are no better than brute force search. However, many well-studied data structures have been developed for performing nearest neighbor lookups (Athitsos et al., 2008; Yianilos, 1993; Bentley, 1975). The major requirement for most data structures is that the metric between two points satisfies the triangle inequality. Provided we can develop an adequate metric for partitionings that satisfies this inequality, we can use an existing data structure that supports high-dimensional queries.

4.2.3.3 Block partitioning metric

The main idea behind the metric is to determine whether or not a given partitioning is sufficiently different from any other. Partitionings are represented by a per-pixel labeling, but each label's compression parameters are computed independently. Hence, for a given block of $N \times M$ pixels, each partitioning can define labels $l_i \in \mathbf{N}$ with $i \in [0, NM-1]$. Two partitionings with labels p and q are identical if

$$p_i = p_j \iff q_i = q_j \; \forall \; i, j. \tag{4.3}$$

To determine the difference between two P-shapes, we consider their labeling as strings of length $k = NM$. Given labelings $p = p_0 p_1 ... p_k$ and $q = q_0 q_1 ... q_k$ we must first find a relabeling R from unique labels in p to unique labels in q such that the Hamming distance between strings $R(p) = R(p_0)R(p_1)...R(p_k)$ and q is minimized (Hamming, 1950). This distance is used as our P-shape metric.

The relabeling R is not necessarily bijective: one P-shape may have more unique labels than the other. We can define a subset of a P-shape p to be a set of all identical labels p_j. If two subsets of a P-shape p independently fulfill Property 4.3 with respect to a single subset of a P-shape q, then the compression parameters for the subsets of p may be duplicated for both subsets. However, in practice the number of subsets in a P-shape limits the number of bits that are allowed for compression parameters. Forcing R to be one-to-one, but not necessarily surjective, enforces the bit allocation constraint. Hence, we compute the optimal relabeling as a bipartite matching problem which can be done in polynomial time.

For performance, we approximate the optimal solution by relabelling each partitioning such that the pixel in the top left has label zero, and the labels increase from left to right. Using this method we observe a negligible decrease in compression quality (¡ 0.1 db) over full bipartite matching while observing a 2X performance increase.

4.2.3.4 Vantage Point Trees

In order to perform nearest neighbor lookups we use a *vantage-point tree* or VP-Tree (Yianilos, 1993). We use the VP-Tree because of its ability to handle high dimensional queries along with its $O(\log N)$ query complexity. The quality and speed of the VP-Tree ultimately depend on the breadth of P-shapes available for the compression format. The VP-tree is a binary tree where each node is a P-shape p, and the left child p_l contains all of the P-shapes within a radius r of p while the right child p_r contains all of the P-shapes outside of this radius. At each node, we can prune half of the remaining nodes if the candidate P-shape falls within the associated radius of that node. In practice, the discrete nature of the search space precludes asymptotic $O(\log n)$ behavior, but we still observe better performance than brute force search.

4.2.4 Endpoint Estimation

There are currently two main compression algorithms that are widely used to solve the endpoint optimization problem with respect to S3TC. One is AMD's Compressonator, which does not have any released source code(AMD, 2008). The other has been incorporated from Simon Brown's *libsquish* into NVIDIA's texture tools (Brown, 2006; Donovan, 2010). As described in Section 4.2.1, Brown's cluster-fit algorithm searches over all 4-point clusterings that preserve the points' ordering along the principal axis. The core of Brown's algorithm is to assign each texel of a cluster to an interpolation point along the line segment. In this way, we present the generalized cluster fit: Given n texels $\mathbf{p}_i$ and index precision $\mathbf{I}$, compute indices d_i and $2^{\mathbf{I}}$ clusters with centers at $\mathbf{c}_k$ such that

$$\|\mathbf{c}_{d_i} - \mathbf{p}_i\| \leq \|\mathbf{c}_k - \mathbf{p}_i\|$$

for all $0 \leq k < 2^{\mathbf{I}}$ and $0 \leq i < n$. We approximate the solution to the endpoint optimization problem as the pair $(\mathbf{p}_a, \mathbf{p}_b)$ that best approximates the overdetermined system of n equations $\alpha_i \mathbf{p}_a + \beta_i \mathbf{p}_b = \mathbf{p}_i$

where

$$(\alpha_i, \beta_i) = \left(\frac{2^{\mathbf{I}} - d_i - 1}{2^{\mathbf{I}} - 1}, \frac{d_i}{2^{\mathbf{I}} - 1} \right)$$

In S3TC, there are only two bits per index giving $\binom{16+2^2-1}{2^2-1} = \binom{19}{3} = 969$ possible clusterings. In BPTC, the texels can be encoded with up to four bits per index. This produces a total of $\binom{16+2^4-1}{2^4-1} = \binom{31}{15} \approx 3 \times 10^9$ possible 16-clusters making an exhaustive search infeasible. Charles Bloom presents an alternative method for S3TC that only uses a 2-means clustering along the principal axis instead of iterating through each possible clustering (Bloom, 2009). Similarly, we assume the best clustering to be along the direction of the principal axis. In FasTC, instead of iterating through all possible clusterings, we compute an appropriate k-means clustering where $k = 2^{\mathbf{I}}$. As shown in Algorithm 1, we initially project all points onto the principal axis. Then we generate $2^{\mathbf{I}} - 2$ equally spaced points along the axis between the extremes of the projection. We use these points as a seed to Lloyd's algorithm to compute a final clustering (Lloyd, 1982). Once we solve the least squares problem in $\mathbf{R}^3$, we choose the closest lattice endpoints as our approximation. Instead of projecting points onto the principal axis, we could take the bounding box diagonal as we did during shape estimation in Section 4.2.1.2. However, both the difference in quality and speed are negligible using this method.

4.2.5 Endpoint Refinement

After estimating the endpoints for a given block of texels, we have introduced errors due to quantization. The endpoint estimation algorithm mentioned in Section 4.2.4 must clamp the final endpoints to the sparse lattice defined by the endpoint precision ζ. Nearby lattice points to these endpoints can provide better approximations to the endpoint optimization problem. However, the initial approximation usually lands within a local minimum of Φ rendering methods that involve gradient descent ineffective. Furthermore, due to the high frequency nature of Φ, it is difficult to locate global minima for a given subset. Most compressors include a localized search around the estimated endpoints to improve their approximation. The size of this search space proves to be another limiting factor in the speed of compression algorithms.

Instead of using a local exhaustive search as in NVIDIA's tool, we use simulated annealing to do a local search for the best endpoints in each subset (Donovan, 2010; Kirkpatrick et al., 1983). Simulated annealing is particularly well suited for this problem due to the size and discrete nature of the search space.

Algorithm 1 FasTC-0: Generalized cluster fit for estimating a solution to the endpoint optimization problem

Require:
Texels $\mathbf{p}_i \in \mathbf{Z}^3$, $0 \leq i < n$
Index Precision $\mathbf{I}$

Ensure:
Endpoints $\mathbf{p}_a$ and $\mathbf{p}_b$, and indices d_i s.t. $0 \leq d_i < 2^{\mathbf{I}}$

$\mathbf{v} \leftarrow$ ComputePrincipalDirection($\mathbf{p}_i$)
$\mathbf{m} \leftarrow \frac{1}{n} \sum \mathbf{p}_i$

// Initialize the endpoints as extremes along the principal axis
$(q_a, q_b) \leftarrow (\max[(\mathbf{p}_i - \mathbf{m}) \cdot \mathbf{v}], \min[(\mathbf{p}_i - \mathbf{m}) \cdot \mathbf{v}])$
$(\mathbf{p}_a, \mathbf{p}_b) \leftarrow (\mathbf{m} + q_a\mathbf{v}, \mathbf{m} + q_b\mathbf{v})$

// Initialize cluster centers as the points along the segment
$\mathbf{C} \leftarrow \{\mathbf{c}_0, ..., \mathbf{c}_{2^{\mathbf{I}}-1}\}, \mathbf{c}_k = \frac{(2^{\mathbf{I}}-k-1)\mathbf{p}_a + k\mathbf{p}_b}{2^{\mathbf{I}}-1}$

// Compute an initial clustering
$d_i \leftarrow k$ s.t. $\|\mathbf{c}_k - \mathbf{p}_i\| \leq \|\mathbf{c}_j - \mathbf{p}_i\| \; \forall \; j \neq k \; \forall \; i$

// Perform k-means clustering to achieve a final clustering $\{d_i\}$
$(\{d_i\}, \mathbf{C}) \leftarrow Lloyd(\{d_i\}, \mathbf{C})$

// Setup and solve overdetermined system of linear equations
$(\alpha_i, \beta_i) \leftarrow \left(\frac{2^{\mathbf{I}}-d_i-1}{2^{\mathbf{I}}-1}, \frac{d_i}{2^{\mathbf{I}}-1}\right) \forall \; i$
$(\mathbf{p}_a, \mathbf{p}_b) \leftarrow (\mathbf{a}, \mathbf{b})$ s.t. $\forall \mathbf{a}_0, \mathbf{b}_0 \in \mathbf{R}^3$ and $0 \leq i < n$

$$\|\alpha_i\mathbf{a} + \beta_i\mathbf{b} - \mathbf{p}_{d_i}\| < \|\alpha_i\mathbf{a}_0 + \beta_i\mathbf{b}_0 - \mathbf{p}_{d_i}\|$$

Furthermore, the lack of known structure in the search space gives an advantage to a stochastically-based process. For each step of simulated annealing, we choose neighboring lattice points $\mathbf{q}'_a, \mathbf{q}'_b$ for each of the two endpoints. If the approximation to the endpoint optimization problem $\Phi(\mathbf{q}'_a, \mathbf{q}'_b)$ is better than the initial approximation, then we restart the annealing process with the improved approximation. Otherwise, if the approximation is close enough to the one we calculated in the previous annealing step, we continue the annealing with this approximation. In this situation, *close enough* is measured by some function $Accept$ whose value depends on how far along we are in the annealing process.

Special consideration should be taken when implementing the $Accept$ function. In general, if this function is too restrictive, then there is a high chance that the annealing process will get stuck in a local minimum. Conversely, if the function is too permissive, the annealing might go in directions that have

a catastrophically large amount of error. In our studies, we have selected a function with exponential decay:

$$Accept(\epsilon, \epsilon', \tau) = e^{\frac{\epsilon - \epsilon'}{10\tau}}$$

For the remainder of the chapter, we will refer to our compressor as FasTC-k, where k is the number of steps used for the simulated annealing portion of our endpoint refinement. We implement at least k steps of annealing for each subset for which we solve the endpoint optimization problem. Since we run the optimization on each block, this incurs a fairly large cost on the run-time efficiency of the total algorithm. However, if we assume that the initial approximation is close to the optimal solution, low values of k should be sufficient for generating high quality images.

4.2.6 Multi-Core Parallelization

Since fixed-rate formats specify texture encoding on a block by block basis, they are inherently parallel. The most common approach is to spawn as many threads as the host machine has cores and divide the number of blocks evenly across all threads. However, the algorithm that we have presented uses uniformity checks and other tests to attempt to resolve the encoding of a block as quickly as possible. An application parallelized in this way will thus not have every thread finish its work at the same time. For example, one such scenario is for a texture atlas that contains subtextures for a character or scene. In this case, all of the clothing, face, arm and leg textures for a character may contain areas of uniform texture values to separate the components.

In order to fully utilize the multiple CPU cores, we use a worker queue for processing textures. In this scheme, we spawn as many threads as we have cores, but for each thread we only process as many blocks as will fit in L1 cache. In practice, this approach results in faster performance on processors with a large number of cores. Using this method, we obtain parallel scaling proportional to the number of CPU cores.

4.2.7 Results

In order to test all acceleration features, we use the BPTC format due to its availability in both hardware and existing toolsets. For this format the two compressors that were tested were the one developed by Walt Donovan provided with NVIDIA's Texture Tools (Donovan, 2010), which we will refer

to as NVTC, and the one distributed with the June 2010 release of Microsoft's DirectX SDK (Microsoft, 2010), which we will refer to as DX-CPU. The tests were performed using both game textures and the standard Kodak Test Image suite (KODAK, 1999) where each texture is 512×768 pixels. Most game textures, such as character maps, are not as high frequency as the images in this test suite and generally are more amenable to compression. We use the Kodak Test Image Suite in order to provide a worst-case scenario in terms of compression quality.

4.2.7.1 FasTC

In our comparisons, we assume NVTC to be the baseline for BPTC. We make this assumption because this tool exhaustively explores an extremely large portion of the solution space, and in general produces very high quality results. Although a good metric for overall perceptible compression quality is the structural similarity metric introduced by Wang et al. (2004), the fixed-rate nature of compression formats will always introduce block artifacts in compressed images which artificially skew this metric. Instead, we use the canonical metric of *Peak Signal to Noise Ratio* (PSNR), using the following formula given by Ström and Pettersson (2007)

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^3 \times w \times h}{\sum_{x,y} \left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2 \right)} \right)$$

As we can see in Figure 4.16 our algorithm provides competitive results in terms of quality even without simulated annealing. Moreover, it provides better quality than DX-CPU. As a commonly accepted rule of thumb, 0.25 dB of difference in PSNR is noticeable to the human eye. Although our results average less than one decibel worse in quality over NVTC, the relatively high values of PSNR make even this difficult to notice. The performance gains achieved with our method are deomonstrated in Table 4.2. We observe order-of-magnitude increases in speed over previous implementations without simulated annealing. Figure 4.17 displays a closeup of the areas that produce the highest error in our algorithm for various images.

In Section 4.2.1.2 we discussed various different methods for estimating the proper shape to choose when encoding a block. One of those methods was to compare the magnitude of the first and second eigenvalues of the covariance matrix and use their ratio as a measurement of linearity. Figure 4.18 compares the difference in PSNR between using this method and using bounding box estimation. From

kodim13	atlas	small-char	big-char
512×768	512×512	512×512	1024×1024

Peak Signal to Noise Ratio				
Image	FasTC-0	FasTC-256	DX CPU	NVTC
kodim13	41.53	41.68	40.27	42.27
atlas	45.16	45.34	43.77	46.32
small-char	47.84	48.03	46.20	49.38
big-char	47.10	47.37	45.02	48.05

Compression Speed in Seconds				
Image	FasTC-0	FasTC-256	DX CPU	NVTC
kodim13	5.2	48.8	264.4	783.0
atlas	2.7	25.2	118.5	381.7
small-char	3.2	29.4	145.6	376.5
big-char	13.4	125.6	544.1	1760.8

Table 4.2: Average compression speed for various compression algorithms. We use a selection of both low and high frequency textures. FasTC easily outperforms all of the other algorithms in terms of speed while maintaining comparible quality. Tests were performed using a 3.0 GHz quad-core Intel Core i7 workstation.

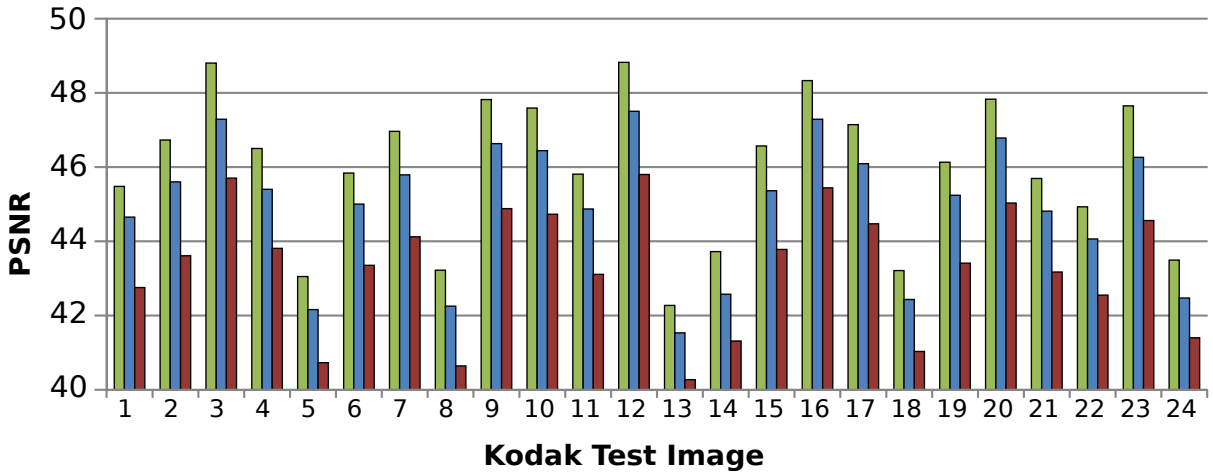


Figure 4.16: Peak Signal to Noise Ratio for various compression algorithms. NVTC, the tool provided with NVIDIA's Texture Tools (green). FasTC-0, our algorithm without simulated annealing (blue). DX CPU, the tool provided with Microsoft's DirectX SDK (red). Our algorithm (FasTC-0) provides similar quality to existing implementations.

this figure, we can see the effect quantization has on shape estimation. Furthermore, the speed of image compression using this method without simulated annealing was 10.7 seconds slower on average.

In order to further demonstrate the effects of quantization on shape selection, Figure 4.19 shows the difference in NVTC when we use bounding box estimation versus measuring distance from the principal axis. The average compression time of a single texture from the Kodak Test Image Suite reduces to 181.3 seconds from 1384.6 seconds. Moreover, the minimal difference in compression quality suggests that the largest gain in quality comes from an exhaustive search around the coarse approximation to the endpoint selection. This reinforces the assumption that taking a better approximation to the endpoints as outlined in Section 4.2.4 will accelerate texture encoding by reducing the need for this kind of search.

4.2.7.2 SegTC

In order to test SegTC, we also compare it against existing encoders. We compare it against all software implementations that can be run on a single core irrespective of specialized hardware, although implementations that use GPUs or vector instructions achieve faster compression speeds. However, the presence of such features is not guaranteed on all platforms, such as embedded devices. We test our algorithm against the existing reference encoder that performs an exhaustive search of the compression parameters and against FasTC (Krajcevski et al., 2013; Donovan, 2010).

Context Original NVTC FasTC-50

Figure 4.17: Detailed investigation of areas with high noise in the Kodak Test Images that produce lowest PSNR. We notice that the visual quality of FasTC is comparable to NVTC and close to the original texture.

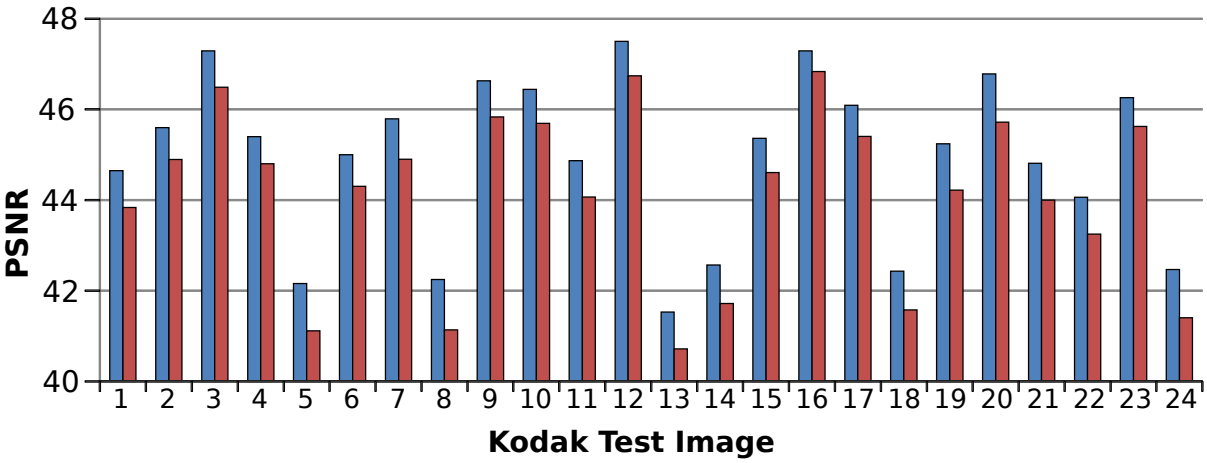


Figure 4.18: Peak Signal to Noise Ratio for FasTC-0 using bounding box estimation (blue) and eigenvalue comparison (red). In this experiment, we replaced the shape estimation technique from FasTC-0 with one that uses the ratio of the first and second eigenvalues of the covariance matrix. We can see that due to quantization errors, using bounding box estimation produces better results.

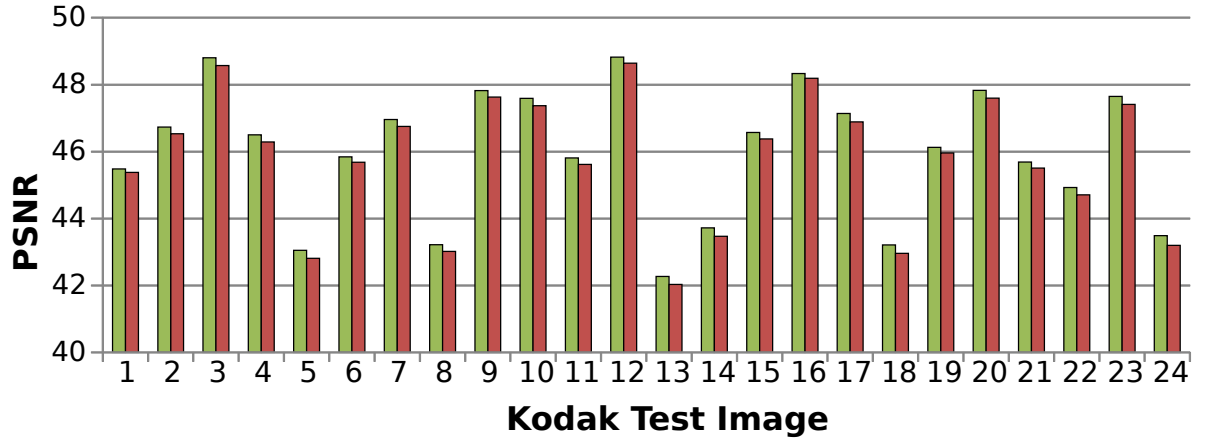


Figure 4.19: We compare the compression quality of the original NVTC (green) with a modified version that measures shape quality using bounding box estimation (red). In general, the difference in PSNR is very small, but we avoid a costly eigenvector computation during shape estimation giving us up to 10x in performance gains.

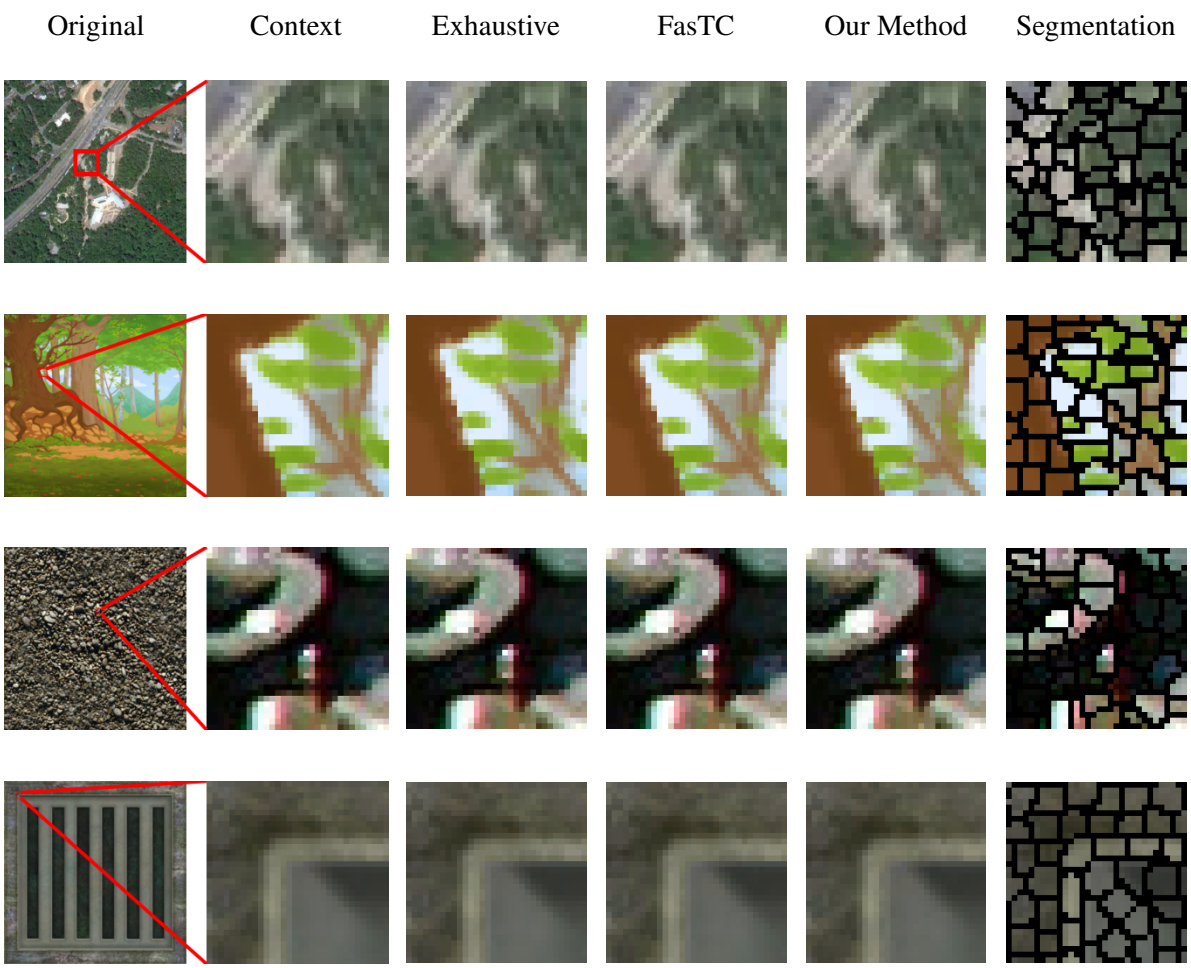


Figure 4.20: From top to bottom we compare encodings of 'satellite', 'colorsheep', 'pebbles', and 'crate'. To complement our segmentation algorithm we have chosen a representative sample of textures that are meant to be consumed visually, and report errors using both peak signal-to-noise ratio and the structural similarity image metric. Additionally, we notice that the choice of segmentation is very important because we lose some detail in parts of 'colorsheep' where the segmentation is too large to catch fine details. To contrast, we maintain the visual detail of 'pebbles' and 'satellite' very well. The texture 'colorsheep' is provided courtesy of Trinket Studios, Inc. The texture 'satellite' is provided courtesy of Google, Inc. The remaining textures are public domain from `www.opengameart.org`

As Griffin et al. have shown, it is difficult to choose a single quality metric for compressed textures (Griffin and Olano, 2014). Even classical PSNR is an unreliable metric (Griffin and Olano, 2014; Wang et al., 2004). However, for reproducibility and comparison with prior work, we present comparisons using both PSNR and the structural similarity image metric (SSIM) in Figure 4.20 and Table 4.3 (Wang et al., 2004). As SSIM is only a single channel metric, we first convert the textures to

Peak Signal to Noise Ratio (PSNR)

	brick	satelite	colorsheep	pebbles	crate
Ex.	43.14	45.30	49.50	39.07	49.25
FasTC	42.49	44.61	47.30	38.14	47.99
SegTC	41.60	43.55	44.75	36.00	47.09

Structural Similarity Image Metric (SSIM)

	brick	satelite	colorsheep	pebbles	crate
Ex.	.9987	.9975	.9976	.9970	.9969
FasTC	.9985	.9970	.9953	.9963	.9956
SegTC	.9981	.9966	.9925	.9947	.9954

Table 4.3: Quanititative assessment of the compression quality for the textures presented in Figure 4.20.

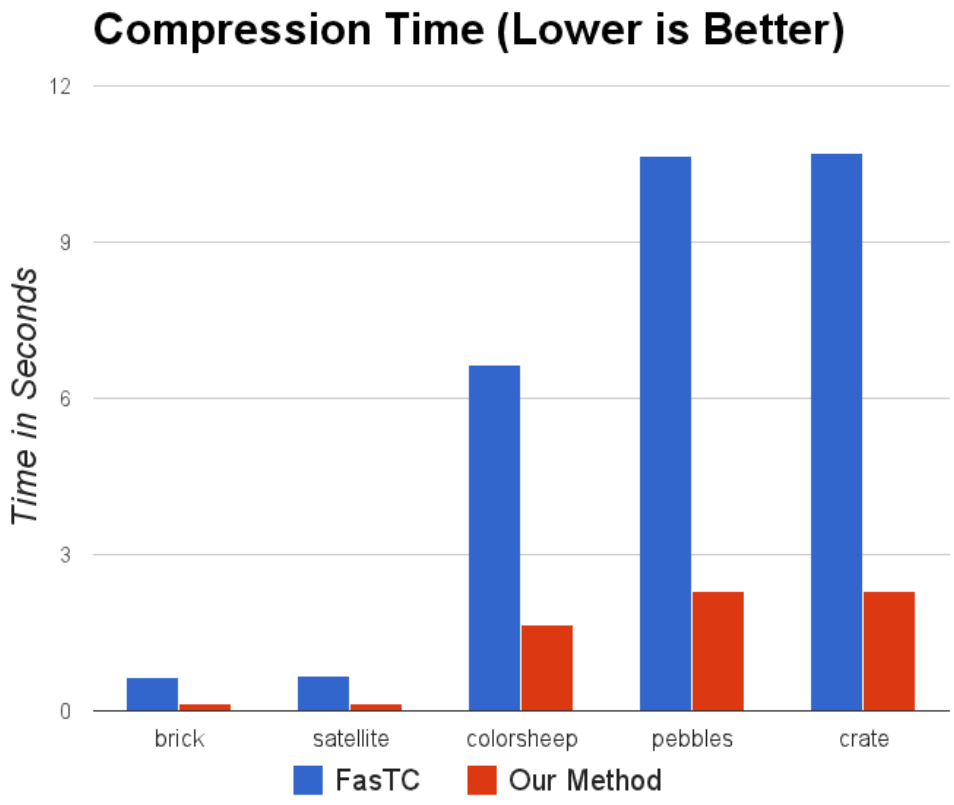


Figure 4.21: The run-time of SegTC against FasTC. Images used are labeled in Figure 4.20 'brick' refers to the texture displayed in Figure 4.3. The exhaustive algorithm is not displayed in the performance graph because it is two orders of magnitude slower than FasTC. We observe an increase in encoding speed over existing implementations while maintaining a similar quality level. All timings are performed on a single core Intel Core i7-4770 CPU 3.40GHz without vector instructions.

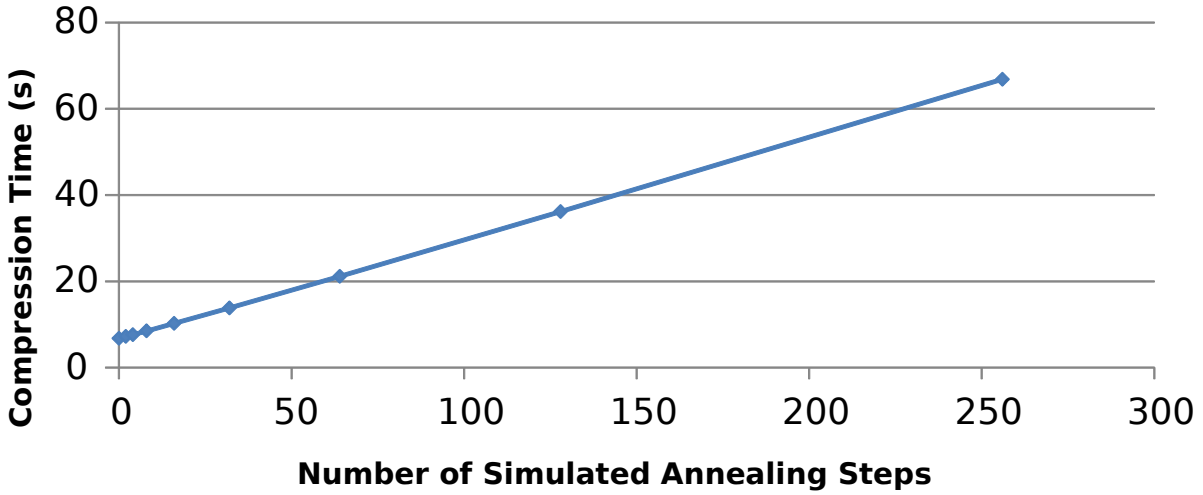


Figure 4.22: Average compression speed in seconds of the images in the Kodak Test Image suite for various different amounts of simulated annealing (FasTC-0 to FasTC-256). Since a constant amount of simulated annealing is applied once for each texel block, the increase in compression speed is linear. Tests were performed using a 3.0 GHz quad-core Intel Core i7 workstation.

grayscale prior to using the reference implementation. PSNR is calculated using the same formula as FasTC (Krajcevski et al., 2013). Compression performance is demonstrated in Figure 4.21.

4.2.7.3 Simulated Annealing

We discussed simulated annealing in Section 4.2.5 as an alternative to an exhaustive search of the neighboring area. Figure 4.22 demonstrates the performance impact of different amounts of simulated annealing. As expected, we observe a linear increase in compression time with respect to the number of annealing steps. Figure 4.23 demonstrates the increase in quality from using simulated annealing. In general, we see a sublinear increase in compression quality as more steps of simulated annealing are applied. This means that small amounts of simulated annealing are beneficial, but because of the nature of the search space, excessive application of the annealing process does not produce better results. The cause of such tapering is twofold. First, if the simulated annealing takes place over a long period of time, it allows the procedure's endpoint approximations to wander away from the optimum early on due to the loose restrictions of the *Accept* function. Second, once the annealing process passes a certain point, the *Accept* function will only accept errors that are very close to the best error, causing the algorithm to loop in a local minimum. We recommend no more than 64 steps of annealing to achieve the best ratio between performance and quality.

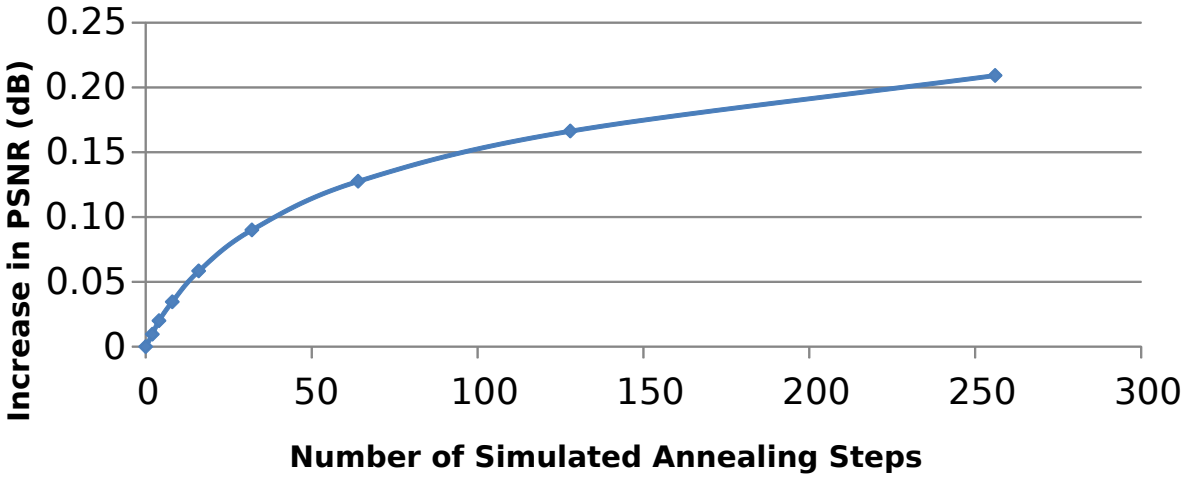


Figure 4.23: Average increase in Peak Signal to Noise ratio for the images in the Kodak Test Image suite for various different amounts of simulated annealing. The increase in quality is sublinear due to the nature of the solution space.

4.2.7.4 Parallelization

Each compression format that supports fixed-rate encoding operates on separate blocks of data independently making compressors inherently parallelizable. Our compression method is no different and observes speedups that scale proportionally to the number of cores in a machine. This massive amount of parallelization lends itself to high end work-stations in content pipelines, and likely GPU optimization. Moreover, the relatively small amount of data that must be read for each block means that the entirety of the computation is cache resident and hence scales with computing power. Figure 4.24 represents the gains in compression speed for various configurations.

4.3 Limitations and future work

4.3.1 LFSM formats

Although intensity dilation provides good results at 3.1 times the speed of conventional LFSM compression techniques, there are still some problems to contend with. Recently, trends in mobile devices are supporting multiple compression formats, such as DXT1 and ETC, where much faster, higher quality texture compression techniques may be available, as shown in Table 4.4.

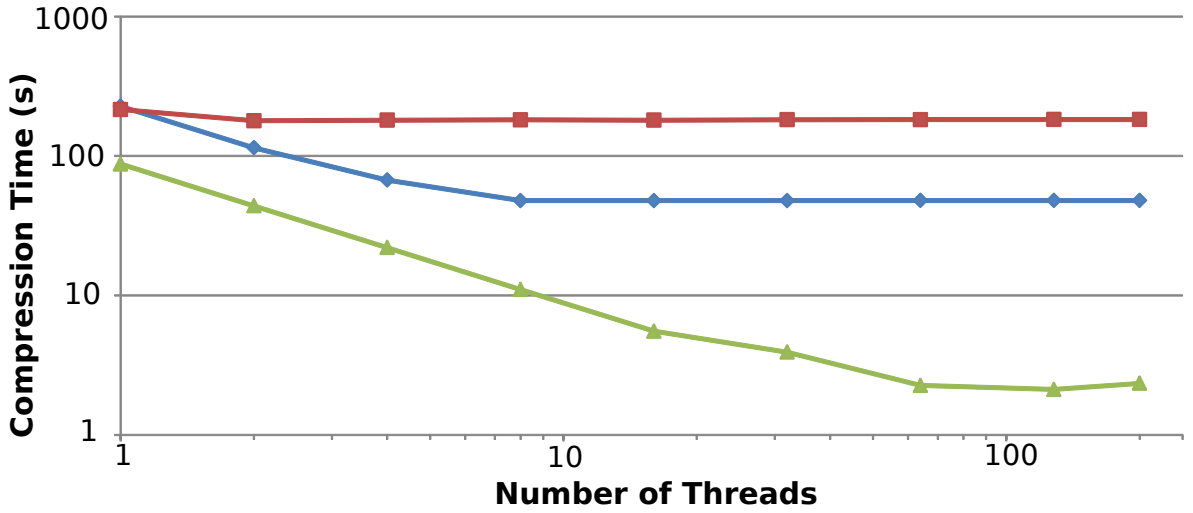


Figure 4.24: Compression time in seconds for FasTC-0 of a 2048^2 sized texture on different multi-core configurations using different numbers of threads. Tests were run on a Single-Core 3.00 GHz Pentium 4 running 32-bit Ubuntu Linux 11.10 (red), Quad-Core 3.50 GHz Intel Core i7 running 64-bit Windows 7 (blue) and 40-Core 2.40 GHz Intel Xeon running 64-bit Ubuntu Linux 11.04 (green). We observe a linear speedup with the number of cores.

Image	Speed (ms)			Quality (PSNR)		
	DXT1	PVRTC	ETC1	DXT1	PVRTC	ETC1
satellite	0.5	20.9	21.7	32.1	30.4	33.9
mountains	0.5	21.8	18.3	33.3	30.0	36.6
gametex	2.1	97.3	90.3	31.2	30.2	33.2

Table 4.4: Fastest available compression speeds (including our intensity dilation for PVRTC) for a variety of formats with similar compression ratios.

Using intensity dilation for LFSM formats should be used to focus on devices that exclusively support LFSM texture compression, such as Apple's iPhone and iPad. However, we have bridged the gap between fast texture compression techniques for certain formats, such as PVRTC and ETC1 (Geldreich, 2013). These times do not reflect any multi-threaded or GPU based techniques. Devoting an entire GPU to compress a texture will likely have certain benefits, but will also likely consume more power on mobile devices, which is ultimately undesirable.

Although intensity dilation is a good technique for fast PVRTC compression, it does not try to optimize the amount of compression quality afforded by LFSM formats. For example, additional investigation is required to determine the effects of gamma-corrected images versus raw RGB. Furthermore, in most compression techniques, an initial approximation is refined to gain better quality. We believe that intensity dilation serves as a better initial approximation to these refinement techniques than the previous state of the art and that developing fast techniques for refinement is still a ripe area of research. Additionally, we can use the multi-pass formulation of intensity dilation, as introduced in Sections 4.1.2.1 and 4.1.2.2, to come up with a parallelizable algorithm that exploit both SIMD and multiple cores, such as consumer GPUs.

We have presented two new methods, FasTC and SegTC, for the acceleration of encoding textures for formats that employ variations of Block Truncation Coding. They offer up to orders of magnitude increases in compression speed while maintaining high compression quality in terms of PSNR. Moreover, we present a flexible paradigm that gives the content pipeline designer a mechanism to choose between encoding time and compression quality.

4.3.2 FasTC

The approximations we have presented use heuristics that have not been fully explored. In the simulated annealing step of the optimization, instead of choosing neighbors randomly, it would be better to weigh neighbors that are likely to produce better endpoints. Also, if we are in a sufficiently severe local minimum then we will cycle through all of the nearby endpoints without proceeding. Furthermore, the generalized cluster fit is based off of a continuous representation. There may be methods that operate in the discrete solution space that could avoid quantization errors and be leveraged to require fewer annealing steps. We believe that there implementation details that would improve the performance of the methods introduced in this chapter. The massively parallelizable aspect of block truncation coding

lends itself to both SIMD and GPU implementation. We believe that with these enhancements such implementations would be able to achieve rates that are viable for real-time encoding.

4.3.3 SegTC

We have presented a new algorithm for selecting P-shapes for partition based texture compression formats. We use image segmentation to designate superpixels of an image and use them to select the ideal partitioning for each block. We expect this algorithm to be the basis for future research in fast P-shape selection methods. Efficient representations of images that quickly convert to GPU-based formats open up an entire area of research devoted to efficient GPU-oriented image representations.

The segmentation algorithm at the core of our method is crucial to providing fast compression speeds. The SLIC algorithm used in our method performs k-means clustering to group pixels based on both spatial and perceptual proximity. However, compression formats and partitionings do not meet these specific constraints in general. We believe that there is valuable future work to be done in terms of using segmentation algorithms that can group pixels amenable to compression formats. Similarly, the choice of error metric can provide better segmentations based on what the texture is used for. For example, a normal map may be segmented such that pixels that share a label reconstruct to a similar unit normal. Furthermore, it should be possible to store segmented images along with per-label compression parameters more efficiently than bare GPU-specific formats. We discuss such an approach for DXT textures in Chapter 5.4.

Limitations: Our algorithm also suffers from a few problems due to the limitations of the underlying segmentation algorithm. Most notably, it may not work well compressing textures with alpha. Also, it is very sensitive to the parameters used to perform the segmentation. The parameter that chooses the size of the superpixels must be small enough to capture fine details but not large enough such that we lose the benefits of the segmentation. Finally, the formats that support multiple subsets per block also support high-quality single-subset compression. For these formats, the acceleration gained from spending less time calculating an optimal P-shape pushes the multi-subset encodings of a single block behind the single-subset encodings with respect to image quality. This limitation ultimately diminishes the benefits of formats that support block partitoning. For this reason, calculating optimal P-shapes remains an active area of research.

CHAPTER 5: VARIABLE BLOCK SIZE TEXTURE COMPRESSION [1]

In the previous chapters, we described how texture mapping has become a standard feature for consumer-level desktop and mobile GPUs (nVidia, 2014, 2015; Imagination, 2016). The cost of adding texture mapping capabilities on graphics hardware includes additional memory for storing textures and the data bandwidth therein for transferring and accessing these textures. These additional features result in a significant increase in data access energy leading to reduced battery life. Fetching a byte of data from modern DRAM incurs 74-200 picojoules (pJ) (Keckler et al., 2011; Ross, 2012) while performing a floating point operation incurs only 5-10 pJ (Keckler et al., 2011). As a result, reducing memory bandwidth for texture accesses has been an important design metric, especially for mobile GPUs.

In Chapter 1.5 we describe a number of standards to allow lossy compressed textures to be used efficiently on GPUs with dedicated hardware decompression circuitry (Iourcha et al., 1999; Fenney, 2003; Ström and Akenine-Möller, 2005; Ström and Pettersson, 2007; OpenGL, 2010; Nystad et al., 2012). To improve hardware decompression performance and reduce energy usage, these formats work with blocks of pixels, 4×4 being the historically popular choice, commonly compressed down to eight or sixteen bytes. These formats follow both *fixed-rate compression*, and *fixed-rate addressing*. Fixed-rate compression requires that each block of pixels be compressed to the same number of bytes, while fixed-rate addressing implies that the physical memory location of a block can be computed purely from the coordinates of the block within the texture. These two features allow GPU hardware to fetch and decompress compressed texture blocks with high throughput and low latency, both of which are essential to obtain high GPU performance and energy efficiency.

One artifact of using compressed textures is that fixed-rate compression introduces an inherent quality versus size tradeoff. Since a fixed number of bytes are used to represent varying amounts of local image detail, higher compression rates may be obtained at the cost of lost texture detail. In cases where these details are localized to small portions of the texture, this requires the user to reduce compression ratios for the entire texture to maintain the necessary detail. With increasing photorealism in computer

[1] Much of this chapter appeared as a paper by Krajcevski et al. (2016a)

graphics, high-detail textures occur with increasing frequency (Griffin and Olano, 2014). In the context of mobile graphics, penalizing the compression ratio for an entire texture due to sparse, localized regions of high detail is less acceptable than on desktops. Recent industrial approaches that take steps towards addressing this problem include the Nvidia Maxwell GPU (nVidia, 2015; Imagination, 2016), which implements framebuffer compression although the actual details are not public.

In this chapter, we present a practical approach to support variable bit-rate texture compression on mobile GPUs. This includes a variable bit-rate compression algorithm as well as modified hardware architecture that can support real-time decompression. Our approach is designed to reduce the impact of the quality versus size tradeoff for textures with sparse high-detail regions. We present two adaptive block size techniques with varying levels of granularity. The crux of our approach lies in varying the block size dynamically throughout the texture, as opposed to a static choice. To illustrate the practicality of our method, we utilize the block types proposed by the new Adaptive Scalable Texture Compression (ASTC) standard available on all modern mobile devices compatible with the Android Extension Pack (OpenGL, 2014). This existing hardware choice results in incremental changes to the current texturing architecture, small modifications to the address computation block, while using the same texture block decompression that is also used for decompressing ASTC textures.

Variable bit-rate compression in our approach is achieved by adding one level of indirection in the decompression path - adding a metadata dictionary defining the location of the desired block. The form of this dictionary differs among our two proposed variations, but in both it allows the compressor to perform de-duplication of texture blocks, allowing only one copy of a unique compressed block to be stored. This helps in terms of the memory capacity and any caches used in the memory subsystem.

Furthermore, we show that variable bit-rate compression with hardware decompression and random-access for textures on GPUs are not mutually exclusive. In this scenario we are able get the best of both worlds: our textures are small so we load faster and reduce the memory bus congestion. On the other hand, they are still stored as compressed textures in GPU memory so the number of cache lines fetched during rendering remains low. We gain the benefits of using more color information bits where the textures are detailed and eliminate duplication by using a dictionary style reuse of compressed data. The key contributions of this chapter include:

- A novel variable bit-rate texture compression method with efficient hardware decompression for current GPUs. Compared to ASTC, our new adaptive schemes can provide a significant reduction

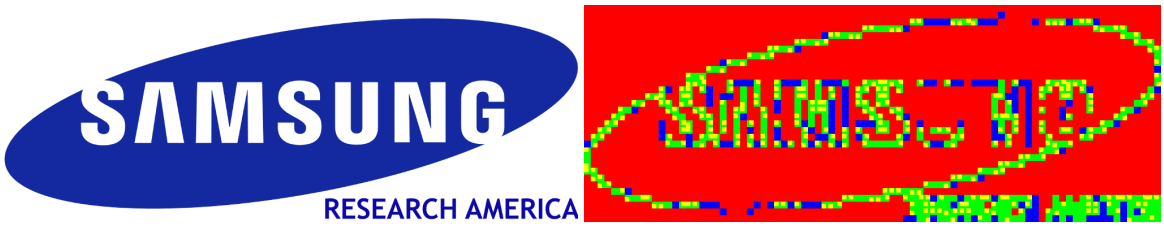


Figure 5.1: Demonstration of the subdivision of the Samsung logo. The constant color regions of the texture are low in detail and can accurately be approximated using 12×12 blocks while the blocks along the edges are subdivided to provide additional detail. For 4×4 metadata, we can reuse the low-detail 12×12 blocks by referencing them in the low-detail 4×4 and 8×8 blocks. The subdivision visualization colors match those in Figure 5.2.

in texture access energy. Results on standard texture benchmarks show that data access energy can be reduced by as much as 33-50%

- A flexible dictionary-based block addressing scheme taking advantage of redundant compressed blocks and maintaining low latency overheads

- A compression algorithm to perform variable bit-rate texture compression with a local image quality threshold

For a wide variety of textures, our method provides comparable compression ratios, some with higher quality, as compared to the reference ASTC compressor (ARM, 2012). We also provide a reference decompressor design. Our codec generates textures compressed into our proposed format that can be decompressed using modern GPU architectures providing lower memory bandwidth usage with low hardware cost.

5.1 Variable bit-rate Texture Compression

Much of prevailing literature in the field of texture compression assumes a single memory lookup operation (Nystad et al., 2012; OpenGL, 2010) per texel, utilizing fixed-rate compression schemes where each texel uses exactly the same number of bits. This approach has certain downsides, the primary shortcoming being that it is agnostic to the natural variation of detail within a texture. Most textures demonstrate a variation of detail within the image by possessing regions of high and low detail.

Consider Figure 5.1 for example, where a fixed-rate compression scheme would utilize the same quality representation for the entire image. To represent these regions in a compressed format while

preserving salient features requires a varying number of bits-per-pixel – large numbers for high-detail regions *like tiles with the edges of the character 'S'*, and smaller for low-detail ones *like the constant color white or blue regions*. An end-user using a fixed-rate compression format must make a tradeoff between image quality and compression – either choose a large texture to preserve high-detail regions, or compromise on quality in these regions and get higher compression ratios. The second shortcoming of these formats is that regions within a texture which are duplicates of each other cannot be mapped to the same region, which leads to missed compression opportunities for the texture.

5.1.1 Two-level Texture Layout

To remedy the aforementioned shortcomings, we propose a two-level compressed texture layout to enable variable bit-rate texture compression. This approach is a dictionary-based scheme using a *metadata* dictionary for:

- Addressing and fetching a particular *block* of texels
- Describing the method of compression for the given block

To minimize the amount of required hardware changes for supporting this scheme, we utilize existing hardware decompressors for decoding blocks of texels. In particular, we use block types proposed by Nystad et al. (Nystad et al., 2012) as the underlying block storage formats – changing only the data fetch portion of the pipeline.

The metadata used here is fixed-rate, simplifying address calculations for metadata fetch. In addition, hardware designs can be optimized to ensure that a small metadata cache will reduce the number of memory accesses per texel from the theoretical maximum of two to an effective rate very close to one. The size of metadata presents a tradeoff between the overhead of storing additional metadata, and the compression ratios it enables. To illustrate this tradeoff, we present two possible metadata definitions along with the flexibility they provide - one minimizing metadata overhead, and the other providing increased flexibility to enable higher compression ratios.

5.1.2 Adaptive Compression with Metadata per 12×12 block

The first proposal maintains metadata at a 12×12 block granularity, representing it as one of the following choices:

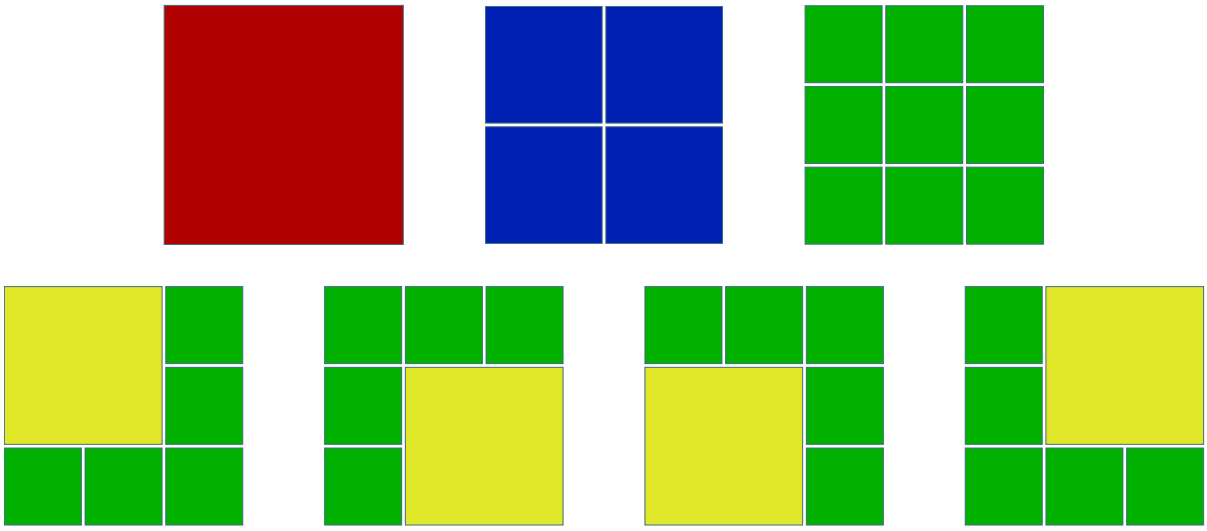


Figure 5.2: The seven different configurations of a 12×12 ASTC block. It can be subdivided into four 6×6 blocks, nine 4×4 blocks, or any one of four different ways to store a single 8×8 block and five 4×4 blocks.

- a 12×12 block
- a combination of one 8×8 sub-block, and five 4×4 sub-blocks
- four 6×6 sub-blocks
- nine 4×4 sub-blocks

These 7 configurations, described in Figure 5.2, can be encoded using a 3-bit code, augmented with a 21-bit block offset from the texture base address. For decompressing any 12×12 block, the decompressor reads 3 bytes of metadata, followed by at least 16 bytes of data. The metadata will be followed by at most 144 bytes if the 12×12 block is comprised of nine 4×4 ASTC blocks.

Modern computing systems interface with the memory subsystem in chunks of data - cache lines - commonly sized to 32 or 64 bytes - dictating that all accesses will fetch that amount of data irrespective of the amount of data actually needed. Since metadata for a 12×12 block consists of 3 bytes, a 64-byte cache line will contain data for at least 21 blocks, i.e. $21 \cdot 144 = 3024$ texels. Matching the metadata memory layout to the expected access patterns, such as using a 2-dimensional Z-order curve, one cache line of metadata can service metadata requests for a significant percentage of texture requests. This helps the effective data fetch rate remain well below 2 per texel block.

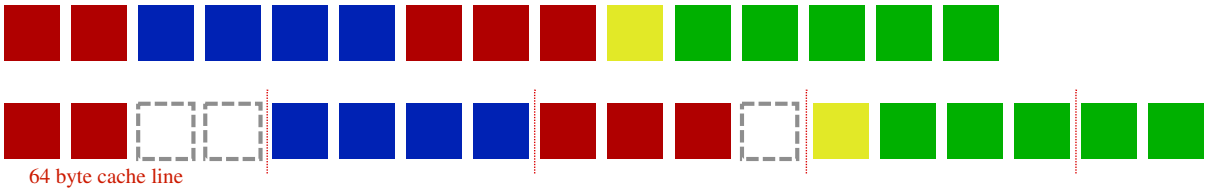


Figure 5.3: (Top Row) Seven 12×12 blocks compressed adaptively and packed on disk using the same color scheme as Figure 5.2. (Bottom Row) The same seven blocks expanded out to cache-line granularity to decrease the number of cache lines required to access an entire 12×12 block.

With cache-coherent access patterns in mind, it is important to note that the physical memory layout of the texture data allows for different layouts on disk and in memory. As in Figure 5.3, the ideal layout of texture data in memory would be to align sub-blocks for a parent 12×12 block to cache line granularity, ensuring that two blocks never partially occupy the same cache line. Two or more blocks may still occupy the same cache line, as long as only one of them spans multiple cache lines. However, on disk, the layout does not need to have such fragmentation, and can instead be packed. When loading the texture from disk to memory, data can be repacked to ensure optimal cache performance.

5.1.3 Adaptive Compression with Metadata per 4×4 block

Our proposed metadata layout of the previous section constrains the possible configurations of ASTC sub-blocks to minimize metadata overhead, at the cost of possibly higher compression ratios. An alternative approach is to store metadata for every 4×4 block - the finest possible granularity in ASTC. In this paradigm, each 4×4 may belong to one of the following:

- flat/constant block: all pixels within the block have the same color value
- 4×4 block
- one of 4 sub-blocks of an 8×8 block
- one of 9 sub-blocks of a 12×12 block

These 15 configurations can be expressed using a 4-bit code, augmented with a 20-bit block offset to maintain byte aligned data. Note that each metadata entry is 3 bytes long, now corresponding to a 4×4 block, implying that fixed overhead of the metadata size is nine times larger than the formulation described in Section 5.1.2. However, this layout lends more flexibility for larger blocks (8×8 or 12×12)

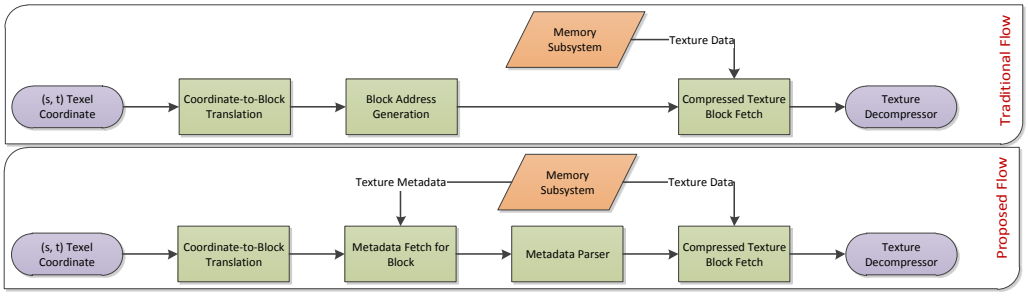


Figure 5.4: Texture fetch flow in a traditional fixed-rate pipeline (top), and our proposed variable bit-rate method (bottom). Note that the block address generation step in a traditional pipeline – a function of the texel format and memory layout – is now replaced by a metadata table lookup

to be placed within the texture, making higher compression ratios more likely depending on the texture data.

This layout also allows for more flexible data packing, particularly for *flat* blocks, in which all texels have the same color value. Two of the 4-bit code values (using the 16th available configuration) indicate flat blocks, the first indicating storage in the first half of a 16-byte compressed block, the second indicating the latter half. An offline compressor can utilize this data layout to make similar blocks point to the same memory location, improving the hit rate of any caching mechanism for texture data.

5.1.4 Unified Adaptive Compression and Decompression

The two manifestations of metadata shown above can be unified into a common metadata decoder akin to multiple kinds of texture formats commonly implemented in a texture pipeline. The input to such a block can be an offset within the uncompressed texture or texel coordinates, the output being the resulting compressed data to be decompressed by an existing ASTC decoder, or in the case of constant blocks, the color data itself.

Figure 5.4 highlights the process of translating a texture request into a compressed block which can be processed by the decompressor. In a traditional pipeline, this begins by mapping the requested texel to a block of texels within the texture – 4×4 in most cases. Since the traditional pipeline has fixed-rate addressing, this block coordinate can be translated into a memory address for the compressed block using the fixed size of each compressed block, and the layout of blocks in memory.

In our proposed method, we replace the block generation logic with a metadata table lookup. Once the (s, t) coordinate is translated to a block – 4×4 or 12×12 in our case – the address generation logic

can be used to address into the metadata table instead of compressed texture data. Once fetched, the metadata provides a map into compressed texture space to fetch the appropriate texture block. Though at first glance this seems to imply two memory accesses per texture access, this is not the case in practice. Given that data is accessed at cache-line granularity, one metadata request fetches data for multiple blocks, which can be cached in a small metadata cache within the texture fetch hardware.

As described in Section 5.1.2, we can exploit the fact that most texture accesses are spatially coherent, meaning that a request for a specific texel will be followed and preceded by requests for its neighbors. A 64-byte cache-line holds metadata for 21 blocks, and ordering metadata blocks in a Z-order can help this cache achieve high hit rates, reducing the penalty of metadata access. In cases where an additional cache exists for uncompressed texture blocks, the metadata parser can also be used to improve its hit rates by remapping multiple compressed blocks to the same data. For constant blocks, the parser can directly return uncompressed block data, or pass the color value to the decompressor with a flag set to denote constant block data. These two optimizations provide energy savings in addition to those already provided by higher compression ratios.

With block-based addressing, each offset in both the 4×4 and 12×12 metadata refers to an ASTC-compressed block. The worst-case subdivision criteria for both schemes is to have every block use 4×4 ASTC blocks. It follows that the block addressing for both schemes requires offsets at this granularity. In each compression scheme we either use 20 or 21 bits for block offsets, meaning 2^{20} or 2^{21} total 4×4 blocks. This implies a texture dimension limitation of 4096×4096 for 4×4 metadata and a limitation of 8192×4096 for 12×12 metadata.

5.2 Offline Compression

Following the requirements of texture compression formats from Beers et al. (1996), we use an offline compressor to encode our images. The compression problem can be posed as a constrained optimization problem to minimize compressed texture size, while satisfying the following constraints:

- A user-specified error threshold ϵ
- Block sizes and configurations belong to an *allowed* subset of possible options

The second constraint differs for the two metadata layouts proposed in Section 5.1.2 and Section 5.1.3, since allowed block sizes and configurations differ between them.

5.2.1 Compressor Structure

In order to perform offline compression we rely on the ASTC reference codec implementation (ARM, 2012) as a black-box for compressing a block of texels. The codec exposes a variety of settings that control the quality versus speed of the compressor. The settings used to perform the compression for each block are chosen to match those used to generate the comparisons demonstrated in Section 5.3.

The compression process then iterates through the possible block configurations from most to least efficient with respect to the compression size. The available block configurations are dependent on the block-size granularity of the metadata. For each iteration, if a certain configuration provides adequate compression quality with respect to the user-specified error threshold ϵ, then that configuration is chosen. After tests with the L_1, L_2, and L_∞ norms, we selected the L-infinity norm to determine the feasibility of a compressed ASTC block. We chose this norm based on a subjective analysis of the results provided. More formally, a block of size $N \times M$ pixels can be treated as a value of an NM dimensional Euclidean vector space. From this definition, the L_∞ norm provides a very useful metric given a block $\mathbf{x}$ and a decompressed block $\mathbf{y}$

$$|\mathbf{x} - \mathbf{y}|_\infty = \max_i |x_i - y_i| ,$$

where x_i and y_i are individual texels within the block. Other candidates included the L_2 norm, which has the downside of rejecting blocks that have low absolute error but high aggregate error. This property of the L_2 norm proved to be more difficult to tune due to the nonuniformity of blocks across a texture.

One of the main benefits of variable rate encoding is the ability to reuse compressed blocks by duplicating the offset stored in the metadata. In order to effectively search for matching blocks within the user-specified error threshold, we use *vantage-point trees* (Yianilos, 1993) to effectively store decompressed block representations. Prior to compressing any new block, we first search for an existing block representation in the VP-tree, as in (Krajcevski and Manocha, 2014b).

5.2.2 Compression for 4×4 metadata

For any compression scheme based on ASTC, 12×12 blocks are the largest possible sized block that can serve as a basis. In our compressor, we begin by first dividing the entire texture into 12×12 blocks. We keep the blocks that fall within ϵ of the original texture based on the metric described in Section 5.2.1. For each 12×12 block that is kept, we insert the appropriate decompressed blocks into

various VP-Trees. One 12×12 block creates nine new 4×4 entries, four 8×8 entries, and one 12×12 entry into three separate VP-trees. When considering subsequent 12×12 blocks, we first check these VP-tress for any already compressed block accurately approximating the current block.

After processing the 12×12 blocks that provide adequate compression quality, we investigate the blocks that need to be subdivided. We proceed by looking at each of the possible 8×8 configurations of the uncompressed 12×12 blocks (Figure 5.2). For each configuration, we first check the corresponding VP-trees to see if any existing 12×12 blocks already approximate it. If not, then we proceed to investigate each configuration against our threshold and insert any sufficiently compressible blocks into the proper 8×8 or 4×4 VP-trees. For every remaining uncompressed 12×12 block, we fall back to a 4×4 representation. We do not use 6×6 blocks with 4×4 metadata because excess metadata bits would be required to denote 4×4 blocks that split 6×6 block boundaries.

We take full advantage of the offset in the metadata by reusing the existing compressed blocks and looking them up in a VP-Tree. However, the greedy strategy described is by no means an optimal solution. Indeed, the problem of determining the best compressed representation for a given texture is a constrained optimization problem and is at best a special case of the set-cover problem, which is known to be NP-complete (Karp, 1972). For example, there is no reason that two 4×4 blocks in separate areas of the texture cannot belong to the same 12×12 block. Furthermore, it is not necessary that the entire 12×12 block be used in order to provide compression benefits, such as when there is only details in the corners. In this case, it would be useful for most of the blocks to be included in the 12×12 block while using 4×4 blocks for the corners. Another limitation of this approach is that it only considers 12×12 blocks on boundaries that are a multiple of 12. The hardware decoder does not have this restriction and introducing it only hinders the possible results.

5.2.3 Compression for 12×12 metadata

For textures using 12×12 metadata entries, the problem simplifies considerably. While we still have the ability to reuse compressed data, we only need to remember decompressed 12×12 blocks instead of all available dimensions. For each 12×12 block, the compressor must choose one of the seven configurations in Figure 5.2. The most straightforward way to do this is also the optimal with respect to the metric used to determine block error as described in Section 5.2.1. The compressor goes in order

from least to most expensive configuration in terms of bitrate, and the first one to provide an adequate error threshold is the one chosen to represent the block.

The choice between using a 12×12 granularity metadata versus a 4×4 granularity, as described in Section 5.2.2, depends on the content of the texture. For certain textures with very high repetition of details, such as animated sprites in a game, the repetitions can be hidden in the metadata using the 4×4 metadata entries. However, with textures that have sharp contrasts in the amount of detail of a given area, such as coverage masks, the 12×12 metadata compression scheme will likely produce better results. Most importantly, however, is that due to the metadata overhead, there are textures that still perform better with simple ASTC compression because of the lack of coherence between different areas and their uniform distribution of texture details, such as natural images.

5.3 Results

We test our method against a few representative images, the 128 textures recently distributed by Pixar (2015), KODAK (1999), and Rubinstein et al. (2010). Using our scheme, application developers can choose a baseline quality level for their textures rather than a bitrate. We have run evaluations using two main metrics for compression quality: *Peak Signal to Noise Ratio* (PSNR) and the *Structural Similarity Image Metric* (SSIM) (Wang et al., 2004). For grayscale images, we use $||\Delta B||_\infty < 4$ and for the color results, we use $||\Delta B||_\infty < 8$. We present images with which our method is both suited for (`android` and `alto` from Figure 5.10) and incompatible (Figure 5.6). Figure 5.8 compares a few of these images compressed with various algorithms.

The efficiency of compression schemes can be measured by two metrics:

- Memory bandwidth reduction, computed as a ratio of the total adaptive texture size (including metadata) compared to the traditional ASTC compressed texture size in kilobytes (KB). The total adaptive texture size is the sum of the variable rate compressed texture size plus the total size of the metadata. We investigate this size by measuring the number of bits used per pixel.

- Energy reduction, computed as a ratio of the energy cost incurred for fetching the adaptive texture compared to the energy cost of fetching the traditional ASTC texture. The energy cost of the adaptive texture comprises of the following components: energy cost of fetching the compressed

block data, and the energy cost of fetching and decoding the metadata. We investigate this value by measuring the number of times we miss a 1 KB L1 cache with 64 byte cache lines.

In mobile systems, the energy cost of fetching a byte of data from DRAM (LPDDR3 or LPDDR4) is in the range of 75 pJ (Ross, 2012) to 155 pJ. For the purpose of this study, we will employ a median energy cost of 115 pJ/byte, inline with LPDDR4 memories that are expected to be used in current and future systems. There is a metadata entry per 4×4 or 12×12 block and each block is 3 bytes wide. In addition to the cost of fetching metadata, additional hardware is required to decode and compute addresses (3-5 multiply-add operations) for the various blocks. We will employ an average energy cost of 30 pJ for decoding and computing compressed addresses for one block of metadata.

The majority of energy costs are incurred by fetching the data for a given texture. In order to measure the energy efficiency of our algorithm, we simulated a direct-mapped cache with a least recently used replacement scheme to determine how many times we would incur a cache miss for various cache sizes. We tested using various cache sizes each assuming a cache line size of 64 bytes. We measured the number of cache misses incurred by reading out an entire texture by three different access patterns: morton (z-curve), raster, and random access. The results for general texture images are displayed in Figure 5.8 and Figure 5.9. We use this analysis to show the benefits of our algorithm on mixed-detail images in Figure 5.10.

The key avenues by which our proposed method may provide increased compression are the use of coarser block sizes, and removal of redundancy within the image data. In the following two sections, we analyze the compression benefits shown in Figure 5.10 to demonstrate their efficacy.

5.3.1 Compression vs. Image Complexity

Figure 5.5 shows the distribution of block sizes used in compressing the test images shown in Figure 5.10. Correlating the bitrates of these images with the block distribution provides some useful insights.

Firstly, as expected, higher percentages of 12×12 leads to lower bitrates and higher compression. Due to the quality threshold, the occurrence of such blocks varies with the content of the image – the `android` image shows higher percentages of coarser blocks like 12×12 due to it's large expanse of

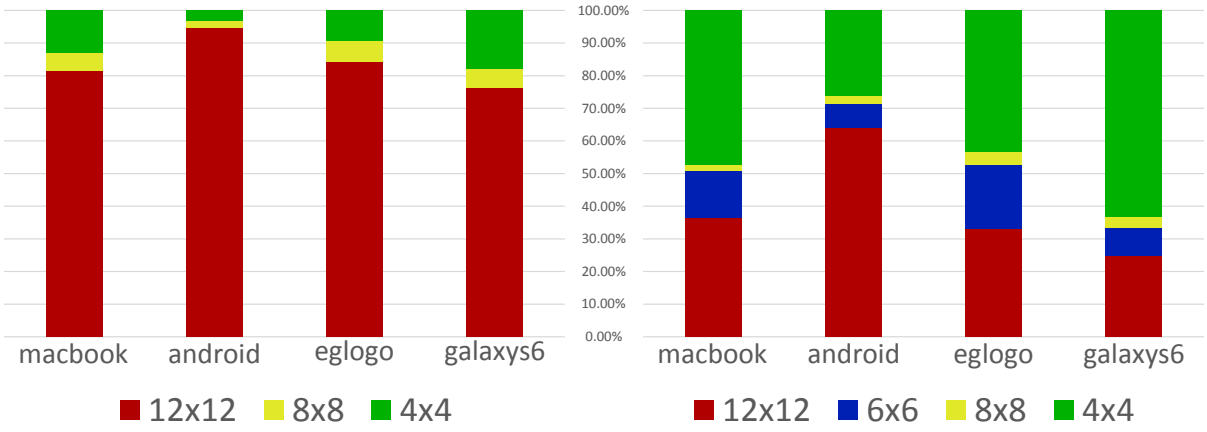


Figure 5.5: Block distribution with (left) 4x4 metadata and (right) 12x12 metadata. Comparing these results to Figure 5.10, we observe that higher percentages of 12×12 blocks leads to higher compression rates.

Figure 5.6: Images with which our algorithm performs relatively poorly. In these images tuning the local subdivision criteria proves difficult. We observe this in images that have uniform low detail with noise, such as bump maps. For comparison with ASTC, we observed (left) 43.8 PNSR (8 bpp) against 40.8 PSNR (9 bpp) with adaptive 4x4 metadata, (middle) 55.27 PSNR (8 bpp) against 40.4 PSNR (7.58 bpp) with adaptive 12x12 metadata, and (right) 39.43 PSNR (5.12 bpp) against 32.94 PSNR (6.32 bpp) with adaptive 4x4 metadata

white, while the `galaxys6` image has the lowest occurrence of the same owing to its natural gradient of colors. A comparison of the compression artifacts introduced by each scheme is illustrated in Figure 5.11.

Secondly, the usage of finer metadata (4×4 vs. 12×12) leads to a higher percentage of coarser 12×12 blocks used in the image – an increase of nearly $2x$. However, as can be seen by the quality per bits per pixel in Figure 5.10, this does not translate into more compression due to the metadata overhead. This observation also suggests that a metadata representation that combines the flexibility of 4×4 metadata with the size of 12×12 metadata would be ideal and thus a promising direction of future research.

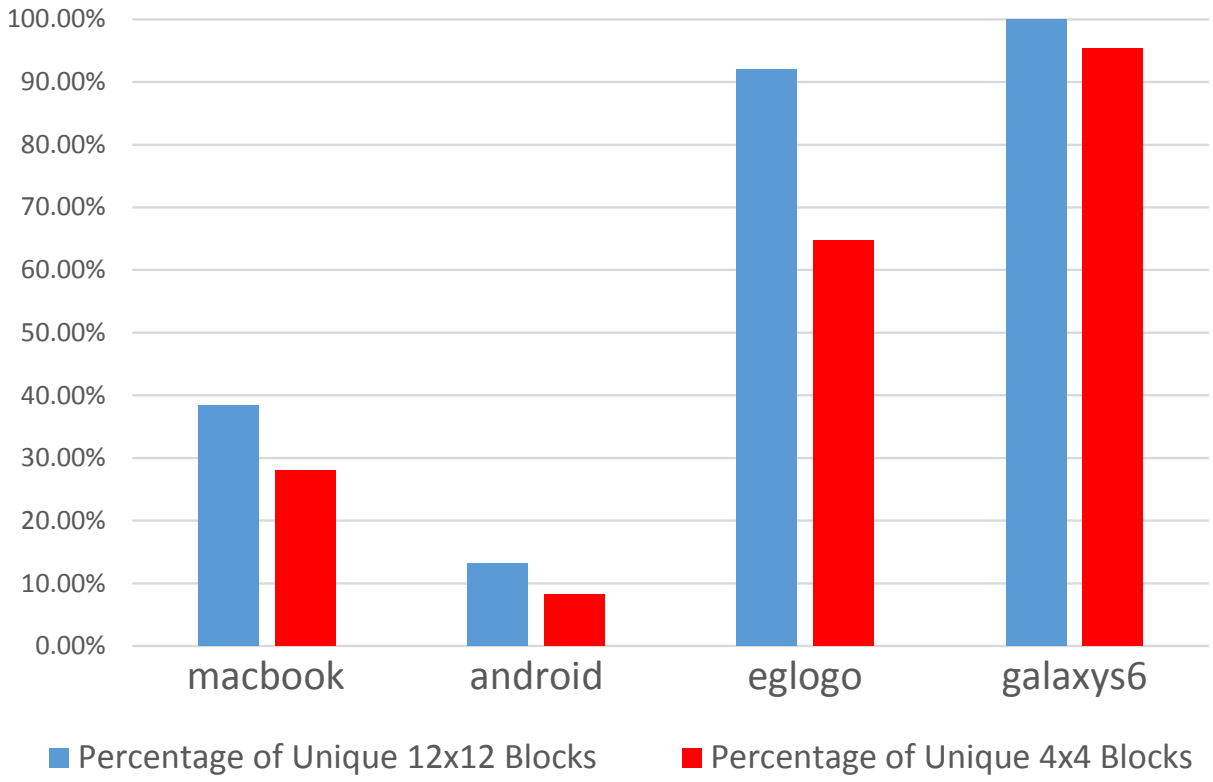


Figure 5.7: Percentage of unique blocks in the four sample color images from Figure 5.10. We show results for both 4×4 and 12×12 metadata. Higer values indicate more unique detail and correlates with larger bitrates for the final compressed texture.

5.3.2 Compression vs. Redundancy

The second avenue for compression is exploiting the redundancy of blocks within the image. Figure 5.7 shows these statistics for the color images in Figure 5.10. Again, a higher redundancy in pixel blocks has a positive correlation with compression. The key fact to note is that most images – barring photographs of natural scenes with lighting variations – demonstrate a significant redundancy (as high as 90%) which can be exploited by our proposed method. On the other end of the spectrum, images demonstrating a wide spread of detail expectedly present the worst case results. This is expected due to the high information content in these images which cannot be compressed without lowering the specified quality threshold.

The `android` image represents the ideal case for our algorithm, with concentrated details surrounded by redundant simple blocks. The compression obtained in such a case is high enough that the metadata size begins to dominate, as noted in the nearly $3x$ increase in bitrate when moving from 12×12 metadata to 4×4.

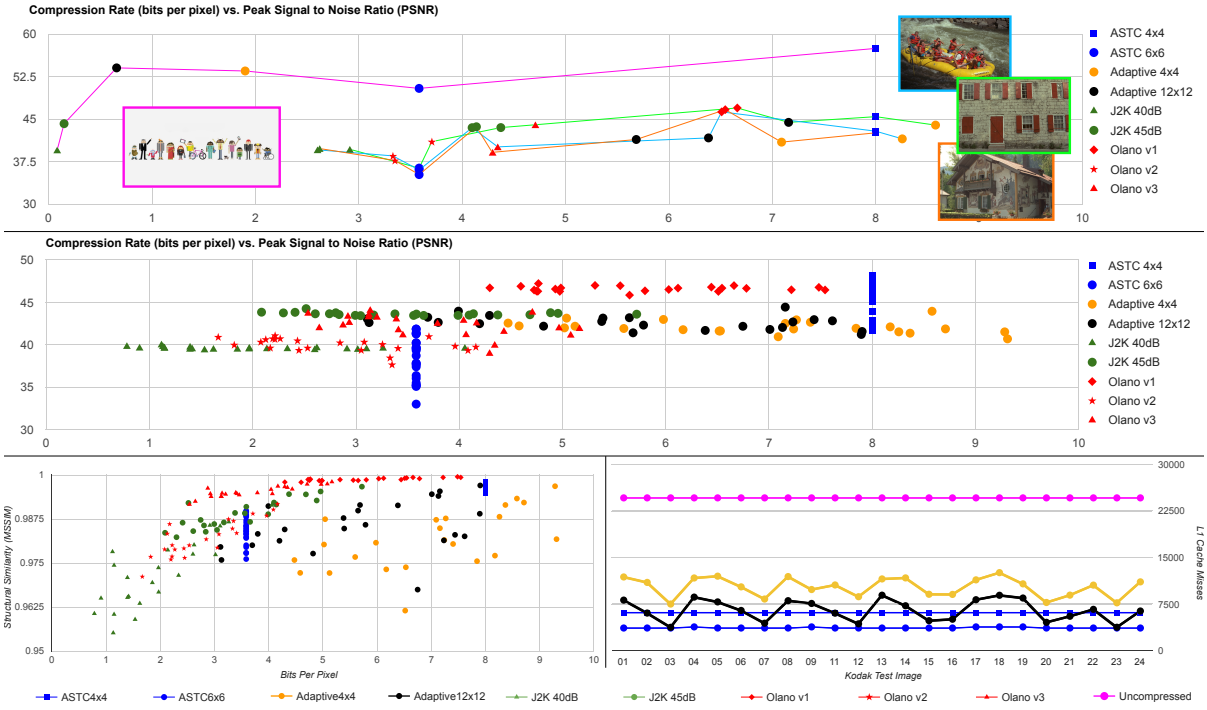


Figure 5.8: A comparison of different compression algorithms. (top) We select a few images to compare PSNR against compression size measured in bits per pixel. We compare against ASTC, JPEG2000 (J2K) and Olano et. al. (Olano et al., 2011). The designations *v1*, *v2* and *v3* are used to match those presented in the paper (Olano et al., 2011). (middle) The same comparison across all images from the Kodak Test Image Suite(KODAK, 1999). (bottom-left) Comparison of MSSIM(Wang et al., 2004) across the Kodak images. (bottom-right) Comparison of cache coherency measured in total cache misses for hardware formats. As we can see from these results, our algorithm performs favorably on images with non-uniform distribution of details. We can contrast the natural images from the KODAK Image Suite against the `android` image from Figure 5.10, where our adaptive 12×12 variant performs significantly better than ASTC approaching bitrates similar to J2K. Furthermore, we can observe the effect of the metadata overhead in our approach with some images with a larger bitrate than 4×4 ASTC. From the cache coherency graph, we can see that for certain images, we are within the same number of cache misses as ASTC.

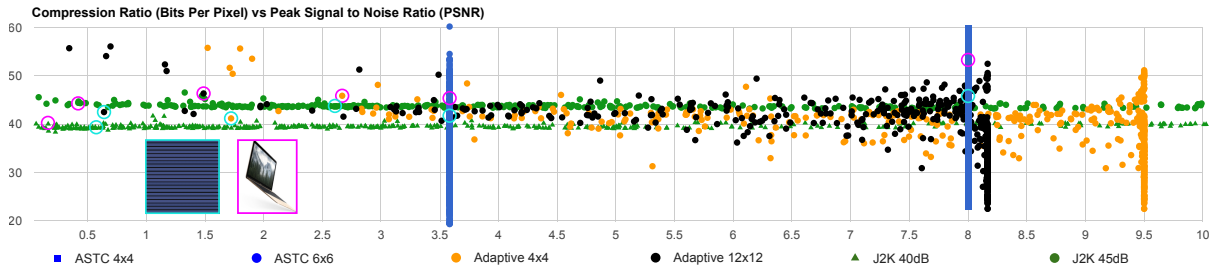
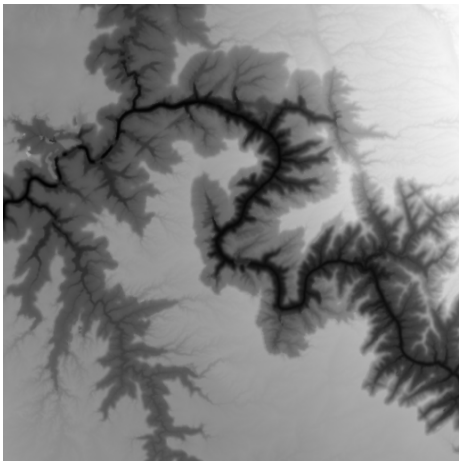


Figure 5.9: Similar to Figure 5.8, we compare a large suite of images with bitrate versus PSNR. The images used were from the Kodak Image Suite(KODAK, 1999), the 128 Pixar Textures(Pixar, 2015), the Retargetme Image Suite(Rubinstein et al., 2010), and the images from Figure 5.10. In this plot, we notice many of the images here are high detail bump and normal maps, for which our algorithm performs poorly. These images usually contain very uniform detail and require accurate compression. Our method subdivides these images to full 4×4 ASTC, creating clusters around at 9.5 bits per pixel for 4×4 metadata and 8.1 for 12×12 metadata. Similarly, for some images, such as those in the lower-left portion of the plot, their high repetitive nature or large areas of low detail make them ideal candidates for our method. Additionally, we observe a significant difference between non-GPU based variable bitrate algorithms. In particular, J2K has a much more tunable quality threshold that is apparent in the bitrate distributions of images.

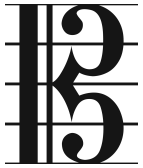
5.3.3 Energy Efficiency

To understand the data fetch energy improvements, the `grandcanyon` image is used for explaining the energy results below. `grandcanyon` is a 512×512 texture compressed using ASTC 4×4 blocks. There are 128×128 ASTC blocks in the texture, thus requiring 256 KB of total storage with ASTC 4×4 block based compressed. Fetching this texture from memory incurs a total of 30 microJoules or 0.03 milliJoules (mJ) (115 pJ * 256 * 1024) of energy. We assume that our adaptive 4×4 scheme delivers an additional

compression rate of 2.3X over 4×4 ASTC. The texture compressed using our scheme comprises of approximately 111 KB of compressed data and 48 KB of metadata. Using these numbers, total compressed data fetch energy is 0.013 mJ. Total metadata fetch energy is 0.0056 mJ. Total metadata fetch and decode cost is approximately 0.495 μJ or 0.0005 mJ (128*128 blocks * 30 pJ). Total cost of fetching adaptive ASTC 4×4 data is a total energy of 0.019 nJ. Compared to



grandcanyon

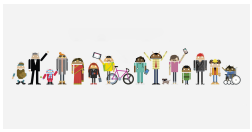


alto
960 × 1152

eglogo
1920 × 915

macbook
1346 × 1686

galaxys6
624 × 416



android
3200 × 1618

Peak Signal to Noise Ratio (PSNR) per bit per pixel

Image	alto	eglogo	macbook	galaxys6	android
ASTC 4x4	8.68	6.34	6.77	5.67	7.19
ASTC 6x6	14.35	12.66	12.70	11.34	14.19
ASTC 8x8	22.53	21.09	20.44	18.89	23.17
ASTC 12x12	42.92	42.23	40.29	38.24	45.64
Adaptive 4x4	30.92	14.66	17.18	10.50	28.15
Adaptive 12x12	80.70	22.19	31.15	15.50	82.23

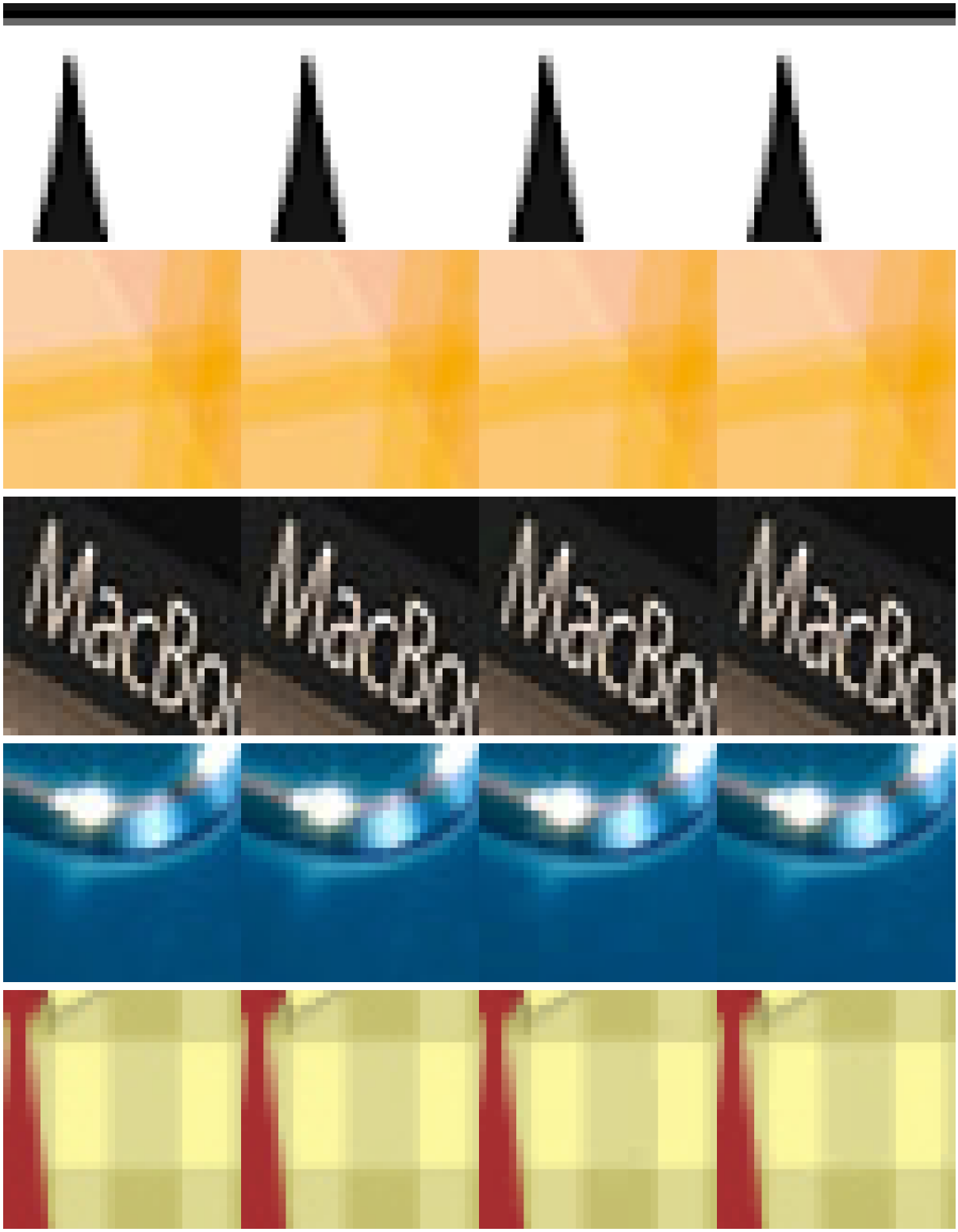
L1 Cache Misses Per Texture (Raster Order)

Image	alto	eglogo	macbook	galaxys6	android
ASTC 4x4	41126	109440	141120	15912	321399
ASTC 8x8	20424	54720	70560	8135	161450
Adaptive 4x4	3518	117034	64026	13975	86701
Adaptive 12x12	7257	82471	77941	16535	69750

L1 Cache Misses Per Texture (Morton Order)

Image	alto	eglogo	macbook	galaxys6	android
ASTC 4x4	13337	27360	35280	3978	90224
ASTC 8x8	2553	6840	8820	1118	26250
Adaptive 4x4	5952	36056	26524	5094	45709
Adaptive 12x12	1700	12096	11264	2403	12618

Figure 5.10: An analysis of our method for compressing textures against Adaptive Scalable Texture Compression (Nystad et al., 2012). We observe that sparse textures, such as alto and android, take advantage of the redundancy inherent in dictionary encoding and produce significant gains. A variety of metrics using ARM (2012) are included. In our adaptive compression schemes we require the error for each grayscale block to be within $||\Delta B||_\infty < 4$ and each color block to be within $||\Delta B||_\infty < 8$. We compare the compression quality per bit per pixel and the number of times we would miss an 1KB L1 cache with 64B cache lines for raster and morton access patterns.



Original	ASTC 4×4	Adaptive 4×4	Adaptive 12×12

Figure 5.11: A comparison of the compression artifacts generated by each algorithm. We compare our method against Adaptive Scalable Texture Compression (Nystad et al., 2012). As in Figure Figure 5.10, our adaptive compression schemes require the error for each grayscale block to be within $||\Delta B||_\infty < 4$ and each color block to be within $||\Delta B||_\infty < 8$.

ASTC 4×4 compression, the adaptive ASTC 4×4 reduces
energy by 36% for the grand canyon texture.

It can be observed that the adaptive scheme reduces
overall data fetch energy by up to 24-55% depending on the settings. However, as we can see in Figure 5.9, the general case observes a net efficiency decrease in energy consumption. This decline is due to the expansion that occurs when compressing the image. The uniform high-complexity of the details of the image allow little room to exploit both redundancy and our adaptive technique. As a result, the image is larger than all ASTC variants and requires more energy to decode.

5.4 Conclusion and Future Work

In this chapter we have proposed a novel variable-rate texture compression format, which provides reductions in memory usage and memory bandwidth usage. For a certain class of images, our technique generates compressed textures with higher quality with smaller bitrate compared to current fixed-rate formats. In addition, the changes to the hardware architecture and decompression logic have a low impact on overall hardware complexity as well as texture fetch latency.

Our proposed approach has certain limitations. For textures with an even distribution of details, which are ideally suited for fixed-rate compression schemes, our method performs poorly, as is evident from certain results in Figure 5.6. This is expected as the metadata overhead increases the size of a compressed texture with minimal benefit. Further investigation is needed to determine the optimal metadata configuration - for all textures in general, as well as optimizations for specific classes of textures. Further progress can be made by improving the algorithm for compressing textures using the 4×4 metadata formulation.

One interesting avenue of future work that this approach enables is user-controlled compression of art assets during game production. A painting tool could be easily designed that allows artists to mark specific regions of the texture in which compression should maintain quality, while not prioritizing compression in other regions. Such a tool could reduce the number of iterations used in compressing art assets for games.

CHAPTER 6: COMPRESSED TEXTURE STORAGE [1]

For over a decade, commodity graphics hardware has shipped with dedicated compressed texture decoding units. Classically, these units decode a fixed number of bits into a block of pixels of predetermined dimension for use with the texture sampling pipeline. Storing compressed textures with respect to these hardware capabilities reduces the amount of bandwidth needed to transfer a texture into dedicated video memory, and the compressed representation allows for significantly more texture data to be resident on the GPU.

Hardware texture compression formats map nicely to GPU architectures by allowing random-access to texture data. However, random access requires the texture to be encoded using fixed compression rates. In contrast to image compression formats such as PNG and JPEG, which provide up to 50:1 compression, GPU texture formats commonly provide lower quality for file sizes at 6:1 compression (Wallace, 1992). As an example, a 4K video frame (19MB uncompressed) requires 345KB of storage as a JPEG, whereas the same video frame requires 3.21MB as a pure DXT compressed texture. This discrepancy is largely due to random access hardware requirements preventing the use of entropy encoding techniques that are used in image compression. The result is that application developers must choose between optimizing their data for streaming and optimizing for run-time efficiency, as image compression formats such as JPEG decode into fully uncompressed textures in memory. This trade-off becomes even more troublesome for applications that stream their texture assets over a low-bandwidth channel such as network-enabled GIS applications (e.g., Google Maps) and video game streaming services. Additionally, the latest advances in virtual reality have made streaming texture data significantly more important (Pohl et al., 2014).

In order to tackle these limitations of fixed-rate compression, recent work has focused on *supercompressing* the textures (Geldreich, 2012)(Ström and Wennersten, 2011). In other words, an additional layer of compression is used in order to encode the already compressed representation in preparation for storage on disk. These methods typically process the compressed texture representations in preparation for an entropy encoding step, such as Huffman or arithmetic encoding providing an additional 2-3X compression

[1] Much of this chapter appeared in a paper by Krajcevski et al. (2016b)

to regain the advantage of compressed image sizes on disk. However, decoding the texture on the CPU eschews the main benefits of compressing textures: the gained bandwidth across the CPU-GPU bus. This bandwidth is even more important in mobile devices that have power constrained GPUs (Leskela et al., 2009).

Although entropy decoding algorithms are inherently serial, we may use multiple decoders in parallel. Duda (2013) presented asymmetric numeral systems (ANS) that maintains an internal state consistent between the encoder and decoder. This property allows multiple encoding streams to be interleaved into a single data stream. Giesen (2014) uses this property to demonstrate how to create data streams that can be decoded in parallel using single-instruction multiple-data (SIMD) architectures, such as GPUs. We give an overview of the range variant of ANS encoding and its use in our method in Section 6.1.3.

In this chapter, we present a new supercompression algorithm GST, pronounced *jist*, for decompressing textures on the GPU into hardware-compressed formats. Our three main contributions include:

1. A new supercompressed texture representation for endpoint-compressed formats;

2. A method for encoding textures into this format;

3. A parallel decoding algorithm suitable for SIMD architectures such as GPUs.

The basis of our algorithm is a state-of-the-art entropy encoding technique known as ANS that allows multiple compression streams to be interleaved and decoded in parallel on the GPU (Giesen, 2014). We exploit the underlying structure of commonly used endpoint compression formats in a way that increases the internal redundancy of the texture data, allowing for efficient static context modeling for the entropy encoder. Our approach saves both streaming and CPU-GPU bandwidth by providing compressed texture data to be decompressed by the device that will use it. Furthermore, one of our main benefits is the increased decoding speed realized by using massively parallel architectures.

We target the class of endpoint compression formats as described in Chapter 1.5. We first re-encode the per-pixel palette indices from a compressed representation into per-block dictionary entries. To improve redundancy between successive index blocks, we store the differences in sequential dictionary entries, similar to differential pulse-code modulation (DPCM). Next, we treat the separate palette-generating endpoints of each block as two low-resolution images for which we use a wavelet transform. Finally, both of these parts are written to disk using entropy encoding. Our current implementation rivals the state-of-the-art CPU codecs in compression size and quality with up to 3X improvement in

decompression speed. This translates to about a 2-3X improvement in compression size over the original hardware-compressed formats, which is realized as additional gains in CPU-GPU bandwidth when the supercompressed texture data is sent to the GPU to be decoded. Our algorithm is designed from the ground up to target current desktop and mobile GPU architectures, and we show benefits to loading 4K video frames and large numbers of textures.

6.1 Compression Pipeline

In this section we present the GST encoding scheme and discuss the techniques used to prepare data for entropy encoding. Our supercompression algorithm is designed to be compatible with many widely used texture formats and to map well to current GPU architectures. Our approach is based on fulfilling the following design goals:

- The supercompressed texture representation should be directly decodable on SIMD architectures, such as GPUs, without additional processing.
- The final decoded result should be a compressed texture in GPU memory.
- The supercompressed texture representation should be agnostic to the underlying endpoint compression formats.

Given an endpoint compressed texture representation, our compression pipeline is organized in three stages, one for each of the constituent parts of a compressed texture as described in Figure 6.1, and one for the final entropy encoding. Only the first stage introduces a minimal amount of error while the last two stages are lossless. In the first stage, starting with the original texture, we generate an initial target endpoint compressed representation. This representation attempts to reuse indices from successive blocks of pixels in order to efficiently generate a dictionary similar to vector quantization (VQ). In doing so, we also generate a new set of endpoints per-block. In the second stage, we independently process these endpoints as separate low-resolution images in preparation for entropy encoding. Finally, we combine similar data streams and encode each using the range variant of ANS before storing to disk (Duda, 2013). The final output of our compression pipeline is four ANS streams to be decoded as described in Section 6.2.

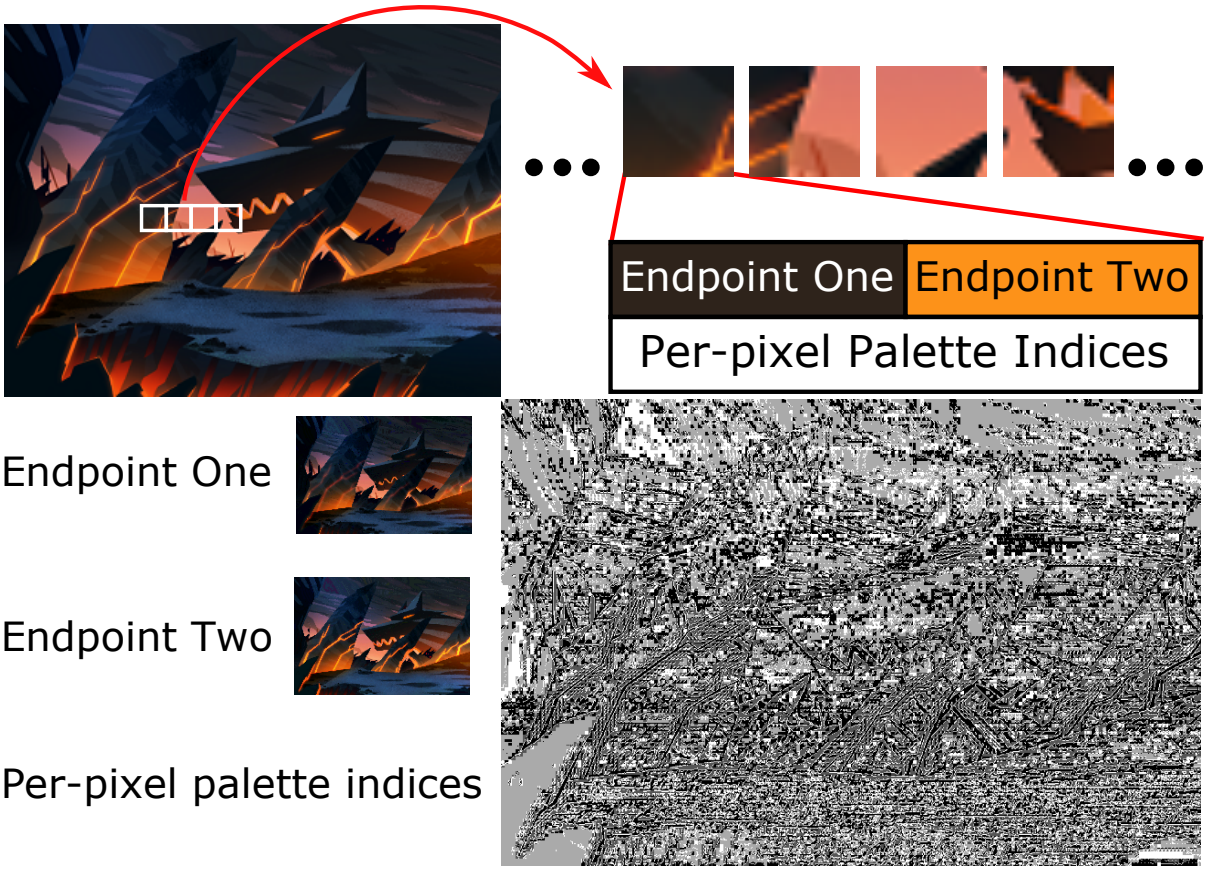


Figure 6.1: The constituent parts of a compressed texture. Each endpoint compressed texture represents a sequence of equally sized blocks. Each block contains a fixed number of bits containing two endpoint colors that generate a palette and per-pixel index data. Here we show the endpoints separated into individual images and visualize the per-pixel indices. We re-encode the indices using VQ-style dictionary compression and transform the endpoint images using a wavelet transform prior to encoding the final texture using an entropy encoder.

6.1.1 Index Block Dictionary Generation

The per-pixel palette indices are classically the most difficult piece of information to compress (Ström and Wennersten, 2011)(Waveren, 2006b). The first step in our compression pipeline is a re-encoding stage as described in Figure 6.2. The purpose of this step is to recompute palette indices for blocks of pixels in a way that is conducive to dictionary construction, similar to VQ. The endpoints for each block are then optimized to fit these newly assigned indices. Our goal is to build a dictionary of *index blocks* representing $N \times M$ blocks of indices as described in Section 2.3. As an example, a 4×4 block of pixels that uses two-bit palette indices will have a dictionary of thirty-two bit index blocks. The goal of

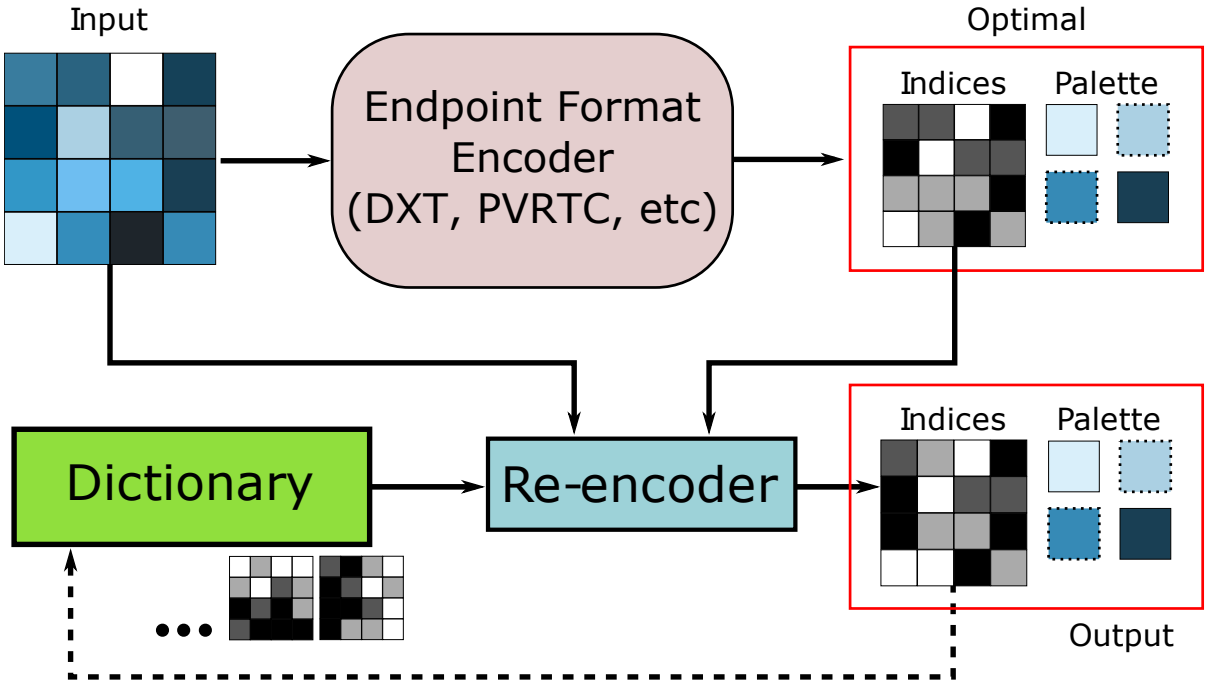


Figure 6.2: The first stage of our encoding pipeline. We process each block in raster-scan order while maintaining a dictionary of recently added index blocks. For each index block, if we find an existing index block in the dictionary that closely matches the original, we reuse that index block. If significant error is introduced, then we add this index block to the dictionary.

the dictionary is to have many redundancies in order to map well to the final entropy encoding step of pipeline as described in Section 6.1.3.

We begin by using an existing codec such as DXT or PVRTC as a black box for providing the original compressed representation of a given texture. This codec is assumed to process each $N \times M$ block of pixels to produce two RGB endpoints defining a palette, and $N \times M$ per-pixel palette indices as described in Section 2.3. We define an error threshold $\mathcal{E}$ that determines the amount of additional mean-squared error that can be introduced for a given block. Using the original compressed representation, we proceed by searching for recently added dictionary entries that correspond to acceptable index blocks for reuse. If no such dictionary entry is found, then we add the index block corresponding to the original indices for that block to the dictionary. The amount of overall error introduced in the compressed representation is directly related to the choice of $\mathcal{E}$. This error threshold is a simple way to affect the rate versus distortion properties of our compression method. By choosing a higher value for $\mathcal{E}$, we get more redundancy in our dictionary, as more recently used index blocks become acceptable, but introduce additional error resulting in more noticeable compression artifacts.

Similarly to VQ, we replace each index block in the texture with a dictionary entry. In order to decrease the number of bits required to store the entries, we only consider the last k dictionary entries. This allows us to represent a given block's dictionary entry as a delta in the range $[-k, k]$ from the previous block's entry. The entries can then be reconstructed using a parallel prefix-sum. By increasing the size of k, we obtain significantly better compression performance, as we have a larger history to consider. Although we process blocks in raster-scan order, different images may provide better delta compression using different orderings, such as a Z-curve. In our experiments, the compression performance of each ordering is highly dependent on the texture, and can be specified in a small per-file header. In our implementation, however, we choose raster-scan order and $k = 127$ in order to represent each entry delta using one byte.

In order to determine the amount of error introduced by deviating from the optimal index block, we use an optimization technique for the endpoints found in many existing endpoint texture encoders (Fenney, 2003; Brown, 2006; Castaño, 2007; Krajcevski et al., 2013). For endpoint-based texture compression each reconstructed pixel comes from a palette generated by two endpoints $\mathbf{p}_A$ and $\mathbf{p}_B$. The number of palette entries $\mathbf{p}_i$ is determined by the number of bits b allotted to each pixel index in an index block,

$$\mathbf{p}_i = \frac{(2^b - 1 - i)\mathbf{p}_A + i\mathbf{p}_B}{2^b - 1},$$

with $i \in [0, 2^b - 1]$. Using this formulation, for a given index block we can construct a $NM \times 2$ matrix $\mathbf{B}$ such that the optimal endpoints $\mathbf{p}_A$ and $\mathbf{p}_B$ are found by minimizing the least-squares error of the equation

$$\left\| \mathbf{B} \begin{bmatrix} \mathbf{p}_A \\ \mathbf{p}_B \end{bmatrix} - [\mathbf{p}_x] \right\|,$$

where each $\mathbf{p}_x$ corresponds to the RGB value of the x-th pixel in the original block.

6.1.2 Endpoint Processing

The second stage of our compression pipeline handles the endpoints themselves. Once the index block dictionary is generated, each block in the texture contains two RGB endpoints that define the palette for that block. Similarly to the PVRTC algorithm, we consider these endpoints independently

as two separate low-resolution images that approximate the final image (Fenney, 2003). Each of these images can be treated independently as a separate image using traditional compression techniques.

Our endpoint encoding step processes the images in two steps prior to entropy encoding, similar to JPEG2000 (Skodras et al., 2001). The first step is a decorrelation step in order to improve the redundancy of neighboring values and to collect the visual information into a single channel. We chose the lossless YCoCg transform in order to avoid additional loss in the final texture and for its simplicity of implementation (Malvar et al., 2008). The lossless property of this color transform is important because any additional error is magnified by the block dimensions in the final reconstructed image. The second step applies a wavelet transform to each color plane after the YCoCg transform. This step alters the total distribution of values in order to skew their probability distribution in preparation for entropy encoding. In our experiments, the choice of wavelet basis does not significantly affect the resulting compression size. However, in order to preserve lossless compression of the endpoints, we use the CDF 5/3 wavelet as in JPEG2000 (Cohen et al., 1992).

6.1.3 ANS Entropy Encoding

The final stage of our compression pipeline combines the output of the two previous stages into a single data stream. Each previous stage produces two symbol streams with different probability distributions requiring a separate context model for each. The index block dictionary and entries comprise the two streams from the first stage, and the second two are the separate Y and CoCg streams for the combined endpoints (Figure 6.3). Each of the four streams are encoded separately and the results are concatenated and saved on disk along with the associated probability distributions. In the rest of this subsection we describe the entropy encoding technique known as Asymmetric Numeral Systems (ANS), first introduced by Duda (2013).

6.1.3.1 Introduction

Entropy encoding is a general term used for any method that converts a sequence of values, or *symbols*, chosen from an *alphabet*, into a sequence of bits based solely on the probability of each value appearing in the input stream. The earliest such method, known as Huffman coding, directly assigns a pattern of bits to each possible input symbol (Huffman, 1952). The length of each bit pattern corresponds

to the probability of that symbol appearing in the input stream. Compression occurs when the probability of a few symbols is far larger than the probability of others.

In Huffman encoding, since each symbol is represented by $b \in \mathbb{Z}$ bits, the corresponding probability of seeing the symbol in the input stream becomes $\frac{1}{2^b}$. As a result, we are not able to represent non-power-of-two probability distributions (or *models*) of symbols used with the input sequence. To rectify this limitation, a technique known as arithmetic coding takes a different approach (Rissanen and Langdon, 1979). Both encoder and decoder take as input an alphabet of symbols $\mathcal{A} = \{0, ..., n-1\}$ with probabilities $p_0, ..., p_{n-1}$ such that $1 = \sum_{s=0}^{n-1} p_s$. The encoder maintains two values, or *states*, L and H, that describe the range of numbers that encode all previously seen symbols. Initializing $L = 0$ and $H = 1$, for each symbol s received as input, the encoder alters the states by the following formula:

$$L_{new} = L + (H - L) \sum_{i=0}^{s-1} p_i$$

$$H_{new} = L_{new} + p_s \, (H - L)$$

The final result written to disk can be any number in the range $[L, H)$. This number uniquely determines the input sequence for the given probability distribution of symbols in $\mathcal{A}$. Compression occurs when the numbers L and H are relatively far apart, and we can choose a number that requires few bits to represent within that range. In particular, we can see that for a sequence of symbols $s_0, s_1, ...$, the range $H - L$ gets smaller at the rate of $p_{s_0} p_{s_1} ....$ This implies that the number of bits needed to represent a number in this range grows at the rate $O(\log \frac{1}{p_i})$, which matches the optimal theoretical limit established by Shannon (1948). By knowing the final result, the decoder can follow the same procedure as the encoder and stop upon reaching the end of the bit stream.

6.1.3.2 Asymmetric Numeral Systems

ANS is similar to arithmetic encoding in that it approximates the theoretical limit to compression size, but has certain properties that make it amenable for implementation on SIMD architectures. The input to the compression algorithm is an alphabet $\mathcal{A} = \{0, ..., n-1\}$, a stream of input symbols $s \in \mathcal{A}$, and a probability distribution $\{p_s\}$, $\sum_s p_s = 1$. Commonly, this probability distribution is discretized with the approximation $F : \mathcal{A} \to \mathbb{N}$ such that $F(s)/M \approx p_s$, where $M = \sum_s F(s)$. We use the common

approach of using the notation F_s to represent $F_s = F(s)$. Given a symbol s and a *state* $x \in \mathbb{N}$ that encodes all of the previously seen symbols of a given stream, ANS provides an encoder C and a decoder D such that

$$\begin{aligned} C(s,x) = x' \qquad & = M \left\lfloor \frac{x}{F_s} \right\rfloor + B_s + (x \bmod F_s) \\ D(x') = (s,x) \qquad & = \left(\mathbf{L}(R), F_s \left\lfloor \frac{x'}{M} \right\rfloor + R - B_s \right), \end{aligned}$$

where

$$R = x' \bmod M \qquad \text{and} \qquad B_s = \sum_{i=0}^{s-1} F_i.$$

The function $\mathbf{L}$ is a lookup function that determines the symbol s such that

$$\mathbf{L}(z) = \max_{B_s < z} s.$$

It follows that C and D are direct inverses of each other, a property that arithmetic coding does not satisfy. Furthermore, C and D are monotonically increasing and decreasing, respectively, with respect to the state x. The more interesting property is that our state grows at a rate similar to that of arithmetic encoding, requiring $O(\log \frac{M}{F_s}) \approx O(\log \frac{1}{p_s})$ bits per symbol.

In order to stream data into and out of bits, as is required for use with a physical machine, each intermediate state x must be *normalized*. In other words, we must write data to disk and decrease x in order to prevent it from growing out of physical memory bounds, typically a 32-bit register. Duda (2013) claims that this is possible by defining a normalized interval $[L, bL)$ such that $L = kM$ for some $k, b \in \mathbb{N}$. In this interval, b represents the divisor required to normalize the interval. In the encoder, C, whenever x grows larger than bL, then x is repeatedly normalized by x/b until $x \in [L, bL)$. For each required normalization, the compressor C first writes the corresponding $x \pmod b$ to disk. Common choices for b are powers of two in order to map nicely to integer shift and bitwise masking operations during encoding and decoding. Just as the encoder may exceed the normalized interval from above, a decoder may require normalization when $x < L$. In this case, the decoder will read a value of size b from disk and increase x by $x' = bx$ until x is in the normalized interval. In order to maintain reciprocity with the encoder, a decoder is required to read from disk at the same point in the data stream that the

corresponding encoder wrote to disk. However, it is not required that the data for the data stream remain contiguous in memory, which is a separate property from arithmetic coding.

6.1.3.3 Preparing ANS for Parallel Decoding

As with all entropy encoding, the variability of the length of the resulting data stream makes it difficult for decoders to start working in parallel, or for a single decoder to be parallelized. In general, parallel decoding approaches have required additional metadata to to the start positions in the datastream, increasing the overhead of the compressed data. ANS provides new properties for being amenable to GPU decoding.

The fact that C and D are inverses of each other, as defined in Section 6.1.3.2, and always read or write a value of size b to disk provides a significant advantage in terms of implementation on SIMD architectures. As Giesen (2014) shows, if we have N compressors C_i working in parallel, then all C_i can operate in lockstep and share a common data stream as long as they write to disk in a deterministic ordering. To maintain the reciprocity between the corresponding set of N decoders D_i, we must make sure that all decoders read from disk in the same order that the encoders wrote to. This reciprocity ensures that a lock-step SIMD implementation such as those found in GPUs will always require each decoder to read at the same machine instruction, and by masking out the threads that must read from disk, we can decode in parallel.

Furthermore, if we choose b such that $b \geq M$, then we know that each encoding step will at most write one value in the range $[0, b)$ to the disk during state normalization. This property implies that each encoder will write at most one value x mod b to disk per encoded symbol and hence each decoding thread (e.g. on a GPU) will read at most one such value per decoded symbol. Using a value of $b = 2^n$ we ensure an integer number of bits being written and read from disk. Once all encoders C_i finish writing to the shared stream, N final encoding states x_i will be used to seed the decoders during decompression. We discuss these trade-offs in more detail in Section 6.2. To map well to current GPU architectures, the rest of this paper will assume the settings $k = 2^2$, $b = 2^{16}$, $M = 2^{11}$, and $|\mathcal{A}| = 2^8$.

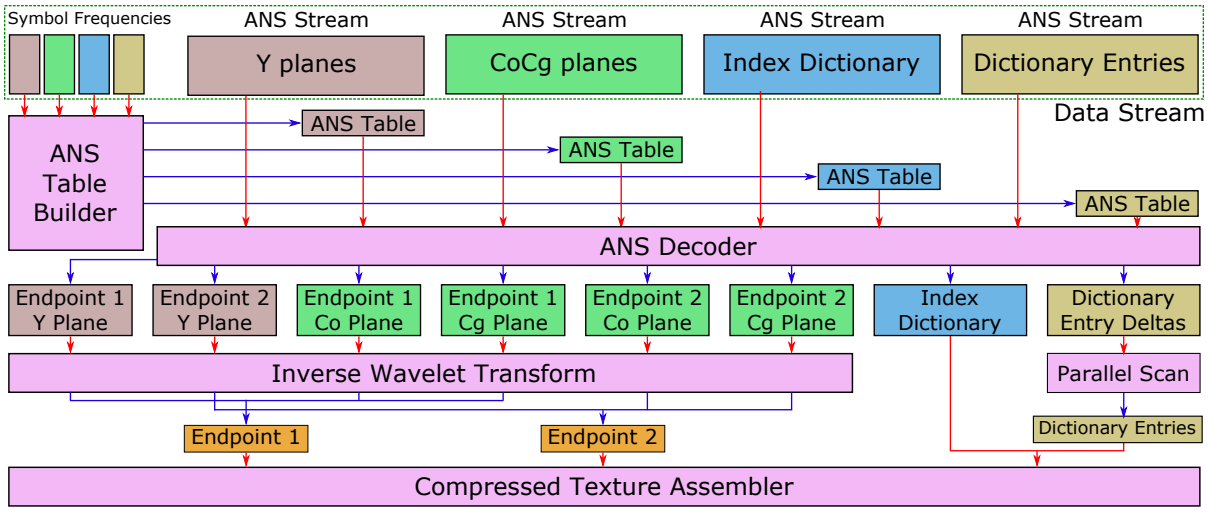


Figure 6.3: Our decompression pipeline. Pink boxes represent separate GPGPU executions for which red arrows are inputs and the blue arrows are the outputs. Per our design, the supercompressed texture data can be uploaded directly to the GPU for decoding. The input data and intermediate results all remain resident in GPU memory during decoding. Multiple texture streams can be interleaved between the symbol frequencies and the ANS streams to provide additional decoding parallelism.

6.2 Parallel Decoding

The design outlined in Section 6.1 facilitates the decoding of textures on SIMD architectures. We outline the overall structure of a decompressor in Figure 6.3. A key feature is that each step in the decoding process is very well suited for implementation on a SIMD processor. We have structured our encoders and decoders for use with commodity GPUs, but our algorithm can be appropriated to any SIMD architecture. In practice, the main trade-off of our method is between decompression speed and compression size. Our compressed representation contains a small header in order to properly construct the data pointers needed to begin the decoding process on the GPU. For a full implementation of both CPU encoder with GPU decoder, please refer to the source code included as supplementary material.

ANS is able to take advantage of SIMD hardware by interleaving many compression streams and decoding them in parallel. One of the biggest constraints is the number of streams that can be interleaved at a time. In order to determine the order in which the interleaved compressors read from the shared compression stream, each decoder thread must notify all the others that it is reading in order to advance their shared offset (Giesen, 2014). Although arbitrarily many encoders can be interleaved, we suggest using 32 or 64 in order to use the available 32-bit or 64-bit shared registers that map well to the warp size of certain GPU vendors.

Additionally, the number of symbols encoded per thread has significant implications on the decoding performance and compression size. First, each decoding thread must be initialized with the ANS state of the corresponding encoding stream. The fewer symbols encoded per thread will increase the total number of threads and hence will also increase the storage overhead of the encoder states. However, decreasing the total number of encoded symbols per thread increases overall parallelism by giving the GPU additional opportunity for scheduling work while waiting on reads and writes to global memory. Finally, the total number of symbols encoded per thread limits the resolution of the final texture. In our method, we use a fixed number of symbols per encoding stream in order to keep all threads busy during decoding. Due to each endpoint belonging to a block of pixels, the number of symbols per set of encoders must divide the total number of pixel blocks in the texture. In our approach, we choose to use 256 symbols per thread requiring the total number of pixel blocks to be a multiple of $256 \times 32 = 8192$. The ramifications of these trade-offs are shown in Figure 6.4.

Each ANS decoder relies on a context model given by the frequencies F_s described in Section 6.1.3. In particular, the decoder needs to know the values of F_s and B_s along with a fast implementation of $\mathbf{L}(z)$. Hence, each ANS encoded stream contains an additional set of frequencies F_s on disk. Because we know $z \in [0, M)$, we can construct a table of size M containing triplets (s, F_s, B_s) for every possible z. This table can be constructed from the set of F_s as a parallel prefix-sum to construct the B_s and a parallel binary search to find s for each value of z. Constructing this table is the first step of our parallel decoding process as outlined in Figure 6.3.

6.3 Implementation

Many of the limits of SIMD architectures require careful consideration of implementation details. Our method mainly focuses on further encoding *endpoint* compression formats as described in Section 2.3. We present an investigation of the compression pipeline presented in Section 6.1 with respect to the DXT and PVRTC compression formats. We have chosen these formats due to their simplicity and widespread usage (Iourcha et al., 1999)(Fenney, 2003). PVRTC and DXT differ only in the amount of bits allotted to store the endpoints and the manner in which their corresponding hardware reconstructs the compressed block. The compression quality of the texture is largely determined by the target hardware compressed format, although DXT and PVRTC usually provide similar quality encodings.

6.3.1 DXT

DXT (a.k.a. S3TC) has a number of variations in order to deal with textures containing alpha, single-channel, and two-channel textures. Here we will address the most common variation, DXT1, and note that additional variations involve adding or removing a pair of channels (DXT3/4) and possibly a separate re-encoding of additional index blocks (DXT5) (Iourcha et al., 1999).

DXT1 has two palette generation modes depending on which order the RGB endpoints are placed. If the 16-bit integer representation of the first endpoint yields a smaller value than the second endpoint, then only three palette colors are generated and the fourth corresponds to a solid black or transparent pixel. The optimal endpoint values described in Section 6.1.1 for a given DXT index block may be generated in either order. In the case in which these endpoints do not generate the expected four-color palette, we discard the index block as invalid.

During the color transform step as described in Section 6.1.2, we attempt to maintain the low bit depth of the pixel channels. Maintaining a low bit depth allows us to limit the number of symbols needed for entropy encoding following the wavelet transform. In the case of DXT1, endpoints are stored using five, six, and five bits for the red, green, and blue channels, respectively. Each 565 RGB value can be losslessly converted to 667 YCoCg (Malvar et al., 2008). After performing the wavelet transform on each channel of the YCoCg data separately, we need less than eight bits to represent the coefficients. In order to increase parallelism, we use 32×32 blocks of endpoints. As this wavelet transform is operating on endpoints per 4×4 pixel blocks, this implies that we currently limit the dimensions of each texture to be a multiple of 128. This block size was chosen to map well to the common limit of 256 threads per work-group on modern AMD GPUs.

6.3.2 PVRTC

PVRTC is similar to DXT in that it has the ability to choose between opaque and transparent textures in the compressed block representation. However, in the original PVRTC, the opacity of the color is determined on a per-endpoint basis rather than on a per-block basis. As a result, each color contains an extra bit to determine whether or not the color contains opaque RGB values or transparent RGBA values. A further bit is provided to alter the generated palette to provide a similar punch-through alpha as in DXT1. In contrast to DXT, these features of PVRTC are agnostic to the ordering of the endpoints,

so we can choose opaque endpoints every time to match the generated endpoints from the re-encoding described in Section 6.1.1 (Fenney, 2003).

Additionally, PVRTC uses lower-precision endpoints than DXT1. Where DXT1 stores three-channel endpoints with 565 precision, PVRTC stores endpoints using 555 and 554 precision. With 565 endpoints, the additional bit in the green channel requires additional bits in the resulting YCoCg transform. However, using 555 endpoints restricts the additional bits needed for YCoCg to an additional bit in the Co and Cg channels, meaning these endpoints can be represented using 566 YCoCg, i.e. two fewer bits per endpoint than DXT1.

6.4 Results

In order to compare our data against prior state of the art methods, we have restricted our testing to DXT1 based textures. We have used Barett's port of Giesen's DXT encoder as our ground truth for optimal DXT encoding due to its overall quality of compressed output and encoding speed (Barett and Giesen, 2009). We measure against raw images, stored as BMP, standard JPEG compression, raw DXT1 compression, and the Crunch library (Geldreich, 2012). For any given set of images, our method produces similar quality results as leading DXT1 compressors, as shown in the detailed analysis of Figure 6.8. Additional close-up comparisons can be found in the supplementary material.

For fair comparison, we have chosen the maximum quality settings for Crunch and have chosen settings for our method to be $\mathcal{E} = 30$, as described in Section 6.1.1. For these settings, on a typical 512×512 texture, our method takes about 0.76s to compress compared to the 6.32s for Crunch. However, as can be seen in Figure 6.6, our compression method has slightly larger variability than Crunch in terms of optimizing for rate-distortion. One of the main sources of this variability is the discrepancy in compressed index data, which is the least amenable to entropy encoding due to its incoherency (Figure 6.1). Our only dial for dealing with the rate-distortion properties is the error threshold $\mathcal{E}$ which is fairly coarse grained, as shown in Figure 6.5. The global dictionary of Crunch also gives it an advantage on hard-to-compress images since neighboring redundancies are generally hard to find. To our benefit, however, using a truncated dictionary does improve the compression size for many textures in the Kodak data set, as shown in Figure 6.7. We present a breakdown of the size of each of the parts of a GST texture in Figure 6.1. This benefit arises due to the assumption that neighboring blocks produce similar palette

indices during compression and as a result our method is dependent on the details of the image, similar to JPEG.

We use two main benchmarks for testing the performance benefits of our implementation. The first benchmark measures the average load time for all 600 4K resolution frames in a 360° video using a motion JPEG application similar to Pohl et al. (2014). The second benchmark measures the time required to load all 128 Pixar textures, each with dimensions 512×512, into GPU memory on a single CPU thread (Pixar, 2015). One of the main benefits of our method is the reduction in load times. We observe this benefit in both the batch load times for the Pixar dataset described in Table 6.2 and our 360° video benchmark in Figure 6.1. All results are measured on desktop PC running Windows 7 on an Intel Xeon 8 core CPU and AMD R9 Fury GPU. As demonstrated in these results, the raw DXT1 load times are significantly faster than the supercompressed textures. Disk seek times and actual disk reads provide a significant amount of inconsistency in disk load timings. In our measurements we make sure that each of the files are fully loaded in the operating system's page cache prior to doing any measurements in order to improve consistency. The long load times of high resolution DXT frames in the 360° video benchmark are due to the full saturation of this cache. Our method is particularly suited to high resolution textures due to the increased parallelism offered in the GPU decoder.

We also investigate some of the tradeoffs presented in Section 6.2. In particular, we show the impact of varying the number of symbols encoded per thread for a single large resolution texture. The performance implications are twofold. Fewer symbols per thread leads to increased parallelism from having more decoders in flight. On the other hand, more symbols per thread reduces the amount of on-disk overhead per group of interleaved decoders. The speed of using more or fewer symbols per thread also depends on whether or not we copy the ANS decoding table into local memory for each group of decoders (Figure 6.3). These tradeoffs are shown in Figure 6.4.

6.5 Discussion

We have presented a new algorithm for storing compressed textures on disk. The benefits of our algorithm show a significant improvement in decoding speed over state of the art CPU techniques. Furthermore, our algorithm provides a way to upload texture data directly to the GPU for decoding in

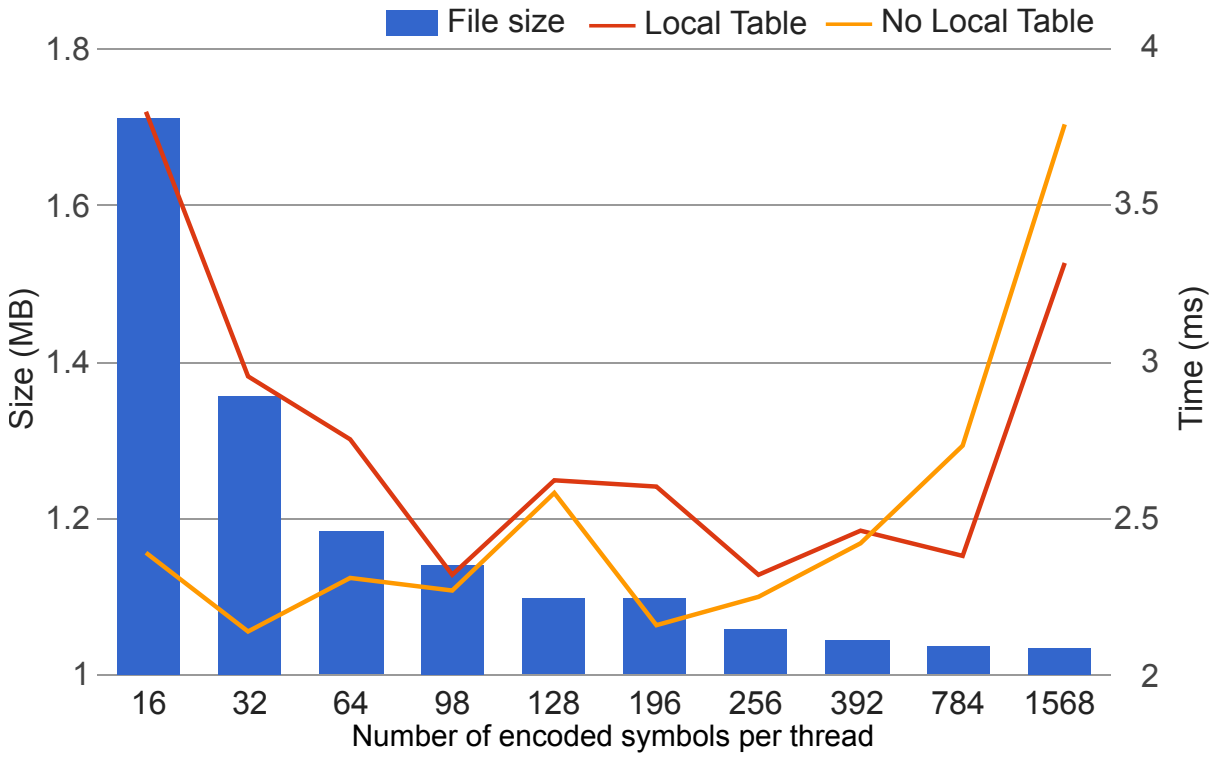


Figure 6.4: The affect of varying the number of symbols per decoding thread, and hence number of parallel decoders, as a comparison between average file size and decoding time of 600 frames of a 4K 360° video. When decoding few symbols per thread, size is dominated by storing many encoder states, although the increased parallelism helps decoding speed. Copying the ANS decoding table (Section 6.2) into local memory only benefits decoding speed when there is enough work per thread to benefit from fewer global reads.

Format	CPU Load	CPU Decode	GPU Load	GPU Decode	Total
JPG	0.1	51.9	2.8	0	54.8
DXT	3.0	0	0.4	0	3.4
BMP	116.3	0	2.2	0	118.5
CRN	0.4	7.7	0.4	0	8.5
GST	0.5	0	0.3	2.5	3.3

Table 6.1: Comparison of various timings in milliseconds for different compression schemes. We test our method against various formats rendering a set of frames from a 360° video at 4K resolution (3584×1792) similar to motion JPEG video (Wallace, 1992).

order to maintain the CPU-GPU benefits. As GPUs become more widespread, such as for rendering web pages, effective streaming solutions present a growing need.

Based on the results in Section 6.4, we observe significant benefits from using a GPU-based decoding algorithm in the general case. In particular, very large textures are the most susceptible to the serial

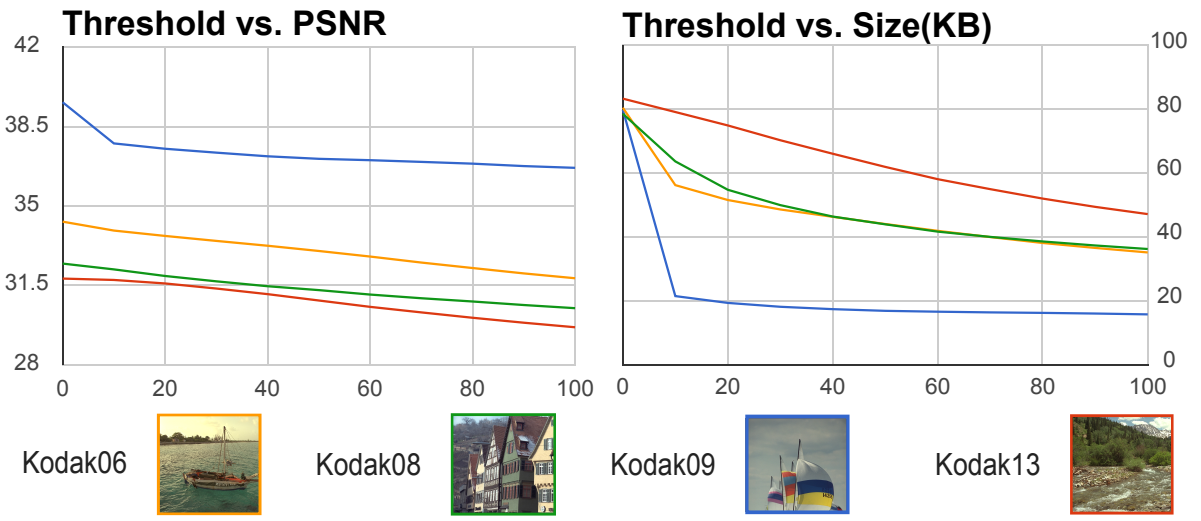


Figure 6.5: The difference in PSNR and compression size as a function of our error threshold for a few images from the Kodak test suite. As we increase our error threshold, we see a decrease in the size of our index data and a drop in our PSNR. Both of these metrics are sensitive based on the features of the encoded image. The PSNR stabilization after an increase in the error threshold supports the assumption of block-level coherency between indices.

Format	JPEG	PNG	DXT1	Crunch	GST
Time (ms)	848.6	1190.2	85.8	242.3	93.4
Disk size (MB)	6.46	58.7	16.8	8.50	8.91
CPU size (MB)	100	100	16.8	16.8	8.91

Table 6.2: Quantitative results of single-threaded loading of the 128 textures in Pixar (2015). The CPU size represents the size of all textures in memory after any decoding procedure and prior to uploading to the GPU. The disk bandwidth is sufficiently fast to make decoding textures the bottleneck.

decoding requirements of traditional entropy encoders. In other applications, a multicore machine may parallelize the decoding of multiple low-resolution textures, which may be beneficial in terms of reducing the overhead of interfacing with the GPU. We observed that a set of per-core Crunch decoders processed all 128 textures of the Pixar dataset at similar load times to our method.

Limitations: Although our method presents many advantages, there are a few limitations in practice. First, our current implementation requires additional scratch memory for intermediate results during decompression. Although the additional memory is minimal, about 3X the size of the final texture, it may be a limiting factor for streaming texture applications that try to exhaust the available GPU resources. However, this limitation may be overcome by using the final compressed texture as scratch memory, but this reduces efficiency by requiring unaligned memory reads and writes along with additional

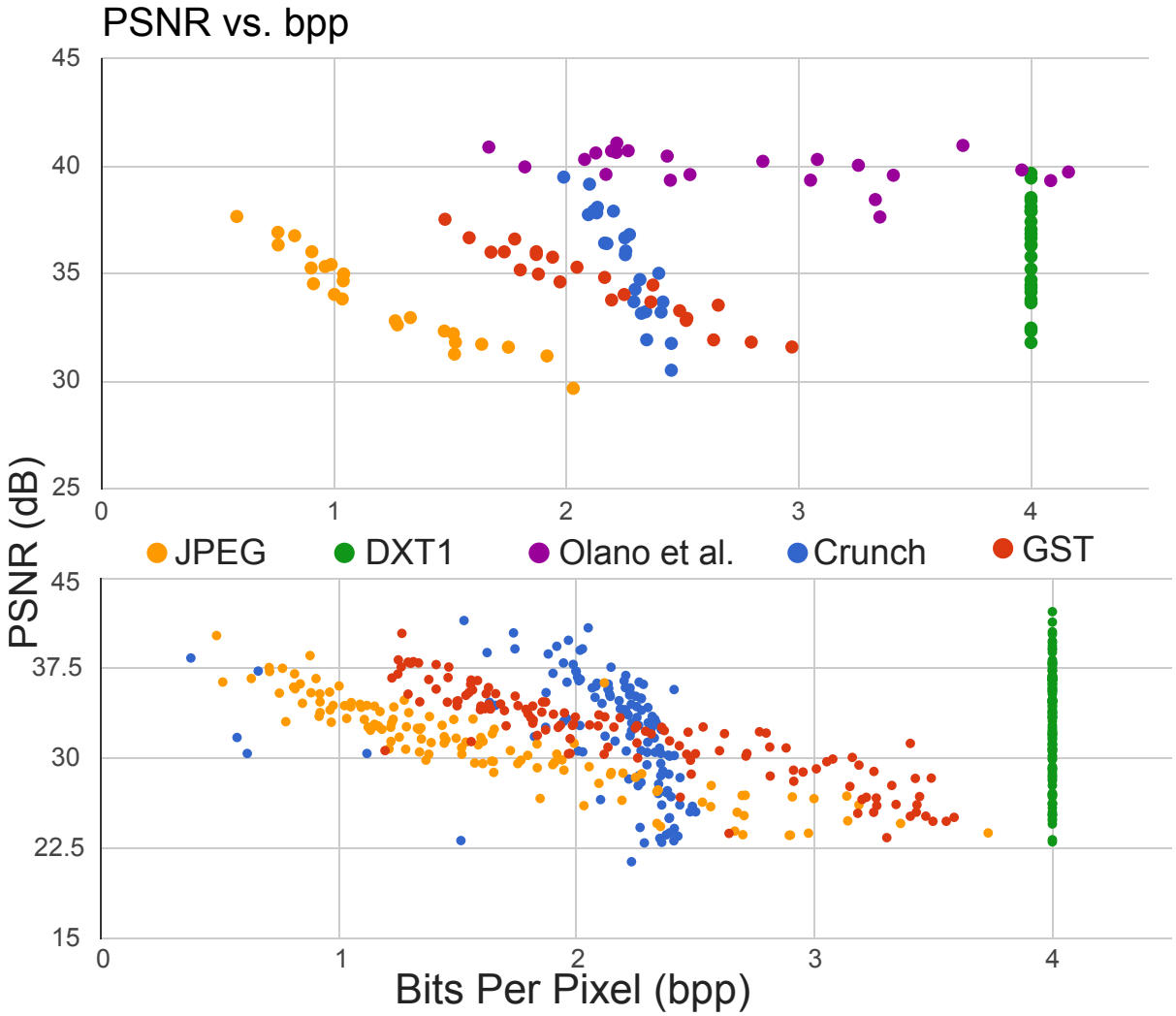


Figure 6.6: We show PSNR versus bitrate values for our method against other compression schemes. The data shows that our method provides bit rates and quality comparable to the state of the art supercompressed textures. Each data point is an image in the (top) KODAK (1999) and (bottom) Pixar (2015) datasets with dimensions 512×512.

synchronization requirements. This sort of 'in-place' decoding presents additional performance concerns by retrieving the values for individual channels in each of the endpoints as described in Section 6.3.

Additionally, the results in Section 6.4 are presented using our reference implementation written in OpenCL for portability. However, during our experiments, we noticed significant stalls on the GPU that were unaccounted for. Our implementation would benefit from additional fine-grained control over the GPU programming interface, such as those presented in the Vulkan API, and further optimization could go a long way to realizing additional performance gains in our application.

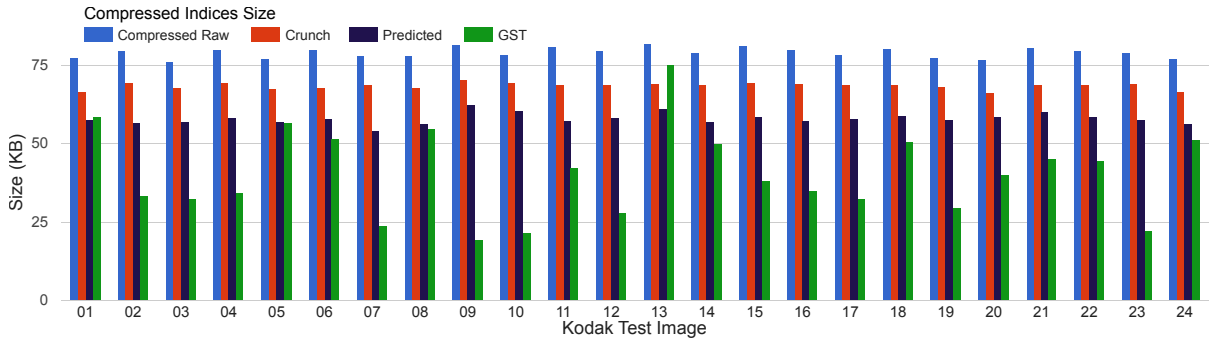


Figure 6.7: We demonstrate a comparison of the compressibility of various approaches to preprocessing index data. This graph demonstrates the size of the compressed index data for various algorithms against KODAK (1999) using the same Huffman encoder used in the Crunch textures. Palette indices are classically the most incoherent data in compressed textures. We show that by limiting the dictionary lookup to the k most recently added dictionary entries, we increase the entropy encoding capabilities of the index data significantly over Crunch at maximum quality settings. Compressed raw is Huffman encoding applied to the unprocessed index data while the predicted method is the same method used by Ström and Wennersten (2011) applied to DXT textures.

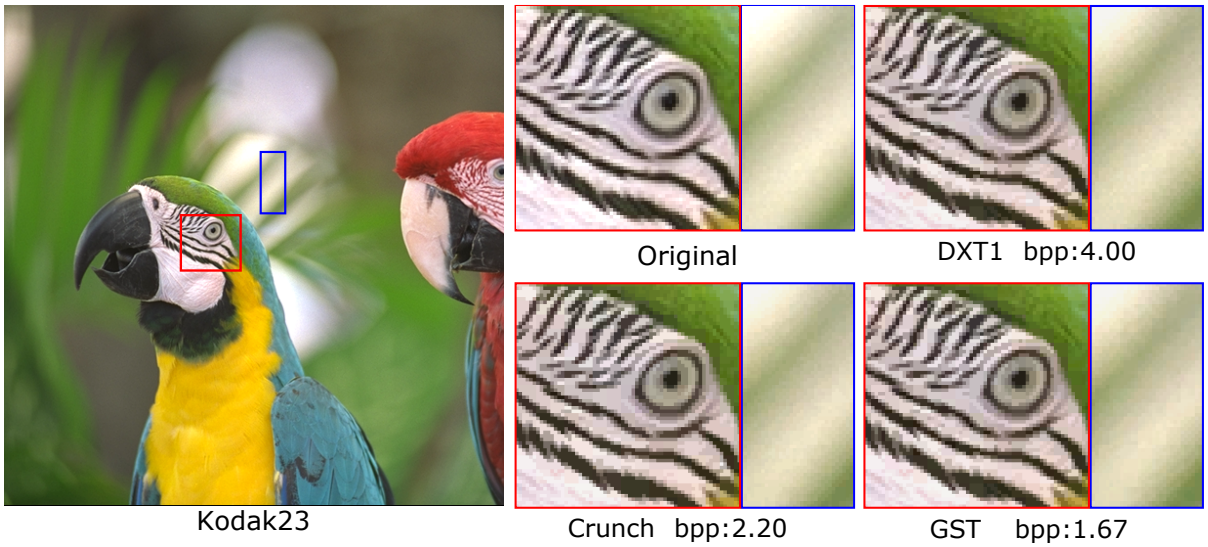


Figure 6.8: A zoomed-in view of the visual quality of various compressed formats. The only stage in our compression pipeline that may introduce additional error is the re-encoding stage. Here we show that the amount of error introduced is imperceptible with respect to other DXT compression formats.

Future Work: Although our implementation focuses on PVRTC and DXT1, more recent endpoint methods such as BPTC and ASTC have introduced increased complexity in the compressed representation of textures (OpenGL, 2010)(Nystad et al., 2012). They allow multiple palettes per block and variable bit depths for their palette indices. Additionally, ASTC provides variable index block dimensions that presents increased complexity to our re-encoding scheme. Although our method works with simple

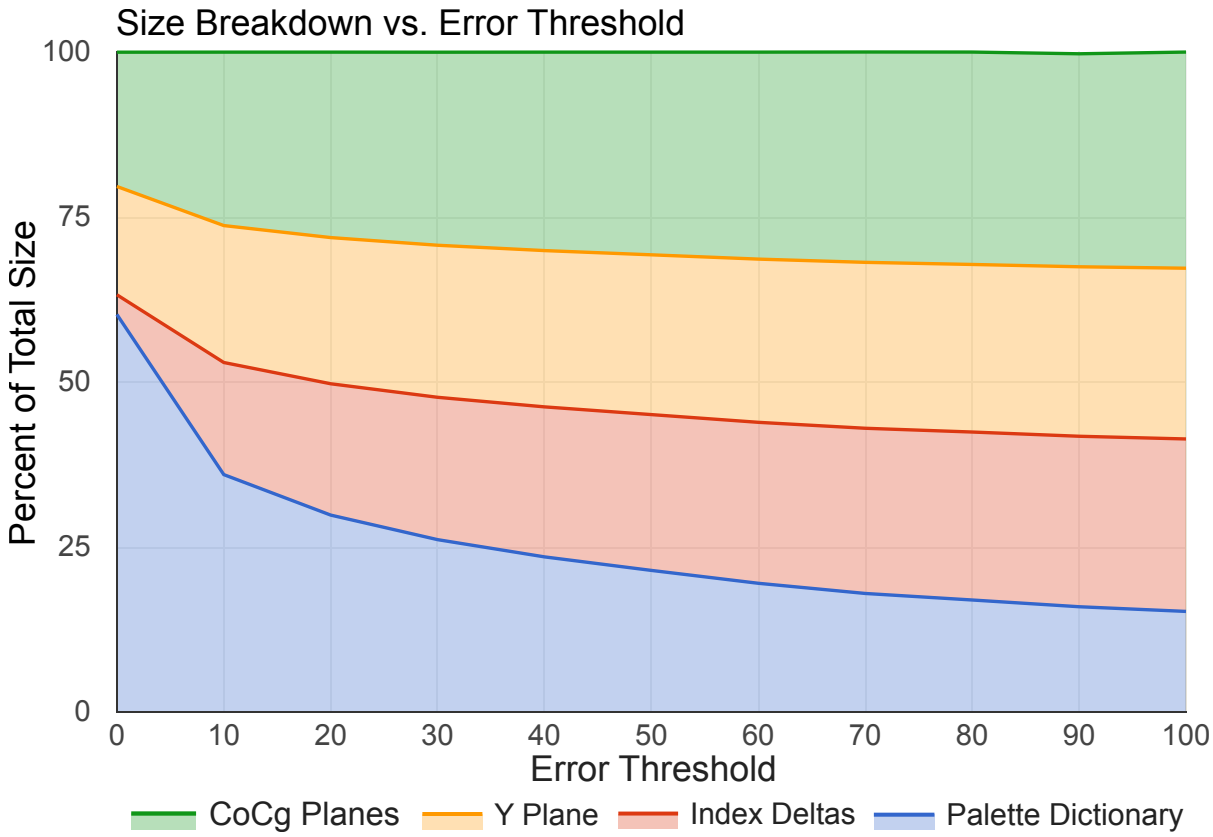


Figure 6.9: An average of the percentage-wise breakdown of each of the constituent parts of a GST encoded texture using various error thresholds. As we allow more error, the size of the dictionary decreases as a percentage of overall space consumed. We used both the Pixar and the Kodak data sets.

endpoint compressed formats, extensions to more complicated formats that preserve the additional quality may be possible.

HDR compression formats present another natural extension of our work. We also believe that the tabled compression formats such as ETC1 and ETC2 can be tackled using a variation of our algorithm. We do not expect our algorithm to emit compression rates as low as those presented by Ström and Wennersten (2011). However, interpreting the parameters of tabled compressed textures as separate low-resolution images may significantly reduce the overhead of this class of compressed textures.

Although our benchmark uses 360° video, our method is inherently designed for single-texture representations. With respect to mip-mapping, a method similar to Olano et al. (2011) is possible where an entire mip-map chain can be used to encode each of the individual endpoint images rather than an explicit wavelet transform. Furthermore, video used in interactive applications can extend many of the ideas presented in this chapter to an additional temporal dimension. For example, a common index

dictionary can be used in conjunction with motion vectors in order to provide minimal overhead between frames.

CHAPTER 7: FUTURE GPU TEXTURING ARCHITECTURES

A significant number of problems have been addressed in the preceding chapters. Apart from the current applications of compressed textures and their benefits to rendering applications, Chapter 4.3.3 shows that there is ample room for formats targeting mixed-detail images, and Chapter 5.4 shows how to apply an additional level of compression on top of the existing formats. In this chapter we cover some of the features of GPUs that are currently not exposed to the GPU programming model and discuss some of the architectural changes that could introduce better support for compressed representations. To avoid confusion, we will use the term *pixels* to denote the color values stored in a framebuffer and *texels* to denote the color values stored in a texture.

Modern GPU hardware has evolved to both give the programmer additional control for application specific purposes and to maximize the speed and power efficiency of the general case (NVIDIA, 2016). The largest growth trends have seen mobile GPUs become significantly faster while a focus on power has resulted in different programming trends (Shebanow, 2014; Imagination, 2016). The ubiquity of graphics hardware has also pushed for a focus on programmability beyond existing hardware units. The introduction of unified shading architectures[1] has allowed a variety of different applications to use the GPU to speed up general purpose tasks. Without the need for functional hardware units for many of the operations used in 3D rendering, the graphics hardware itself has generalized to allow additional flexibility in many interactive rendering pipelines. Some applications, such as the traditional 2D rendering done by the Skia library discussed in Chapter 2.6, have shown benefits from using this architecture in non-traditional ways (Google, 2016).

The interface to the texturing unit has resisted much of the changes directed towards increased programmability. Much like alpha blending, many of the functional units for performing texturing operations are still baked into the silicon of the GPU. These operations still exist due to the lack of diversity in applications that interface with these units. In almost all graphics architectures, apart from compressed texture formats, texture *access* has remained largely the same. Applications generally still

[1] https://msdn.microsoft.com/en-us/library/bb509580(VS.85).aspx

use textures in a way that maps spatial coherent areas of a texture to spatially coherent areas of the screen. Without a need for additional flexibility, the existing hardware units for accessing textures have provided significantly faster and lower-power access to texture data in comparison to the overhead needed for a fully programmable approach.

For two dimensional textures, the abstraction of the texturing hardware to the programmer is a way to convert texture coordinates, usually represented using single-precision floating point numbers, from $\mathbb{R}^2$ into colors in $\mathbb{R}^n$. This abstraction is used in the programmable shaders to offer a seemingly continuous domain from which textures can be sampled. For each pixel in the rendered image, all texture accesses must go through many steps prior to returning color values:

1. Coordinates are coverted from $\mathbb{R}^2$ to $[0, 1]^2$ based on the selected repeating mode for the given texture, which may be repeat, clamp, mirrored, etc.

2. Coordinates are then converted from $[0, 1]^2$ to $[w, h]$ based on the width, w, and height, h, of the source texture.

3. Based on the filtering mode and relationship between this texture access and the texture access of the neighboring rendered pixel, one, four, or eight source texels are selected.

4. For each source texel, the physical $(x, y) \in \mathbb{Z}^2$ coordinates are mapped to a memory address storing the texel data.

5. Depending on the source representation, this data is sent through a functional unit to decode the final texel values and convert them to values in $\mathbb{R}^n$.

6. Depending on the filtering mode, these texel values are linearly interpolated to get the final texel value for the given texture access.

In the previous scenario, the largest number of texel accesses (8) are required when trilinearly interpolating between two mip-map levels to collect the final texel value. Global memory access is among the most expensive operations in terms of power (Jiao et al., 2010). Fortunately, the access patterns of textures are usually optimized such that texels are requested in a spatially coherent manner with respect to pixels rendered on the screen. In other words, after rasterizing a given framebuffer pixel, neighboring pixels will usually sample a texture at similar locations. This access pattern avoids significant penalties since memory caches are able to collect neighboring texel values.

This commonly used texturing architecture makes many assumptions that are becoming increasingly difficult to defend in practice. For example, many of these applications assume that texture data will be reused over many multiple frames, such as detailed scenes in high-production 3D games. While these may be used as a benchmark for testing the performance of certain graphics architectures, many developers are increasingly using GPUs for accelerating non-3D graphics rendering tasks. As an example, some user interfaces are exploiting the parallelism of the GPU for compositing images on top of simple rendering primitives such as squares and circles. Others are using the texturing hardware to offer animated and stylized effects such as icons in popular mobile operating systems. Another example is to use graphics hardware for overlaying geographical data on satellite imagery for mapping applications such as Google Maps. For each of these examples, images are loaded into memory for a short amount of time and then discarded, many of them only appearing on screen for a few seconds. Additionally, the detail in many images is becoming significantly easier to compress even though the images themselves are becoming larger to accomodate the current growth in display size (Shebanow, 2014). In Chapter 4.3.3, we discussed ways to accomodate hardware to support such images within the existing hardware pipeline. However, this approach simply adds an additional layer onto an already complicated texturing system.

In order to map to texture hardware, texture data is expected to be organized linearly in memory. The underlying architecture might reorganize the data to allow for optimal texture access, but in general this process is opaque to the programmer. This data layout forces the application developer to limit how texture hardware is used. For example, the fixed function filtering stage can only be used once the texel values are fetched, but the only way to fetch them is by reading the data from memory. In other words, the hardware for bilinear filtering is coupled to the memory caching of the texture access. This inflexibility creates a significant limitation on the way that data can be represented in memory and may end up being a significant setback in providing compression techniques that exploit the application-specific redundancy of textures.

In this chapter, we discuss a potentially useful addition to graphics hardware that decouples the texel access routines from the rest of the texturing hardware. In essence, we would like to propose support for *texture programs* that may provide additional flexibility to developers for representing texture data. These programs should live in between the filtering stage and the texture access stage such that each texture program itself describes the conversion from image coordinates $(x, y) \in \mathbb{Z}^2$ to texel values. Using such texture programs, compressed texture formats such as ASTC and DXT can be implemented by the

programmer, gaining the flexibility of application-specific compression benefits at the cost of increased energy consumption and latency from avoiding the fixed-function nature of the pipelines. However, these programs could even operate prior to the compressed texture hardware allowing supercompressed implementations (Chapter 5.4) and variable bit-rate formats (Chapter 4.3.3) to be implemented on an as-needed basis. The rest of this chapter will discuss some of the expected benefits and limitations of texture programs.

7.1 Address Abstraction

In general, most graphics architectures assume that textures are stored as a finite sequence of texels arranged in a two dimensional grid. In this case, for a given texture of dimensions (m, n), there is specialized hardware for looking up the proper address in memory for given a two dimensional texel coordinate $(x, y) \in \mathbb{Z}_m \times \mathbb{Z}_n$. For the simplest representation, where the texels are placed in row-major order, the address $A_{(x,y)}$ for a given location of a texel of size b bytes will be

$$A_{(x,y)} = b\,(my + x)$$

. For over a decade, GPUs have taken advantage of the fact that texture access is *spatially coherent*. In other words, if a texel is requested at location (x, y), it is likely texels at locations $(x \pm 1, y \pm 1)$ will be requested soon as well, such as during texture filtering. For this reason, when textures are loaded into the GPU, they are laid out in a manner such that accessing a texel at one location will bring additional texels into the L2 and L1 memory caches. One such layout that preserves spatial coherence is the Z-order curve, where the texels are ordered as shown in Figure 7.1. The address $A_{(x,y)}$ computation for this method is performed by interleaving the bitwise representations of x and y. While difficult to describe mathematically, the hardware for this operation is surprisingly simple to implement.

The addressing scheme for compressed textures is designed to take the format's block size into account. For example, DXT compresses 4×4 blocks down to 64 bits, or 8 bytes. Hence, for a naive raster-order layout of DXT blocks, the address is computed using the formula

$$A_{(x,y)} = 8\left(\left\lfloor\frac{y}{4}\right\rfloor\left\lfloor\frac{m+3}{4}\right\rfloor + \left\lfloor\frac{x}{4}\right\rfloor\right)$$

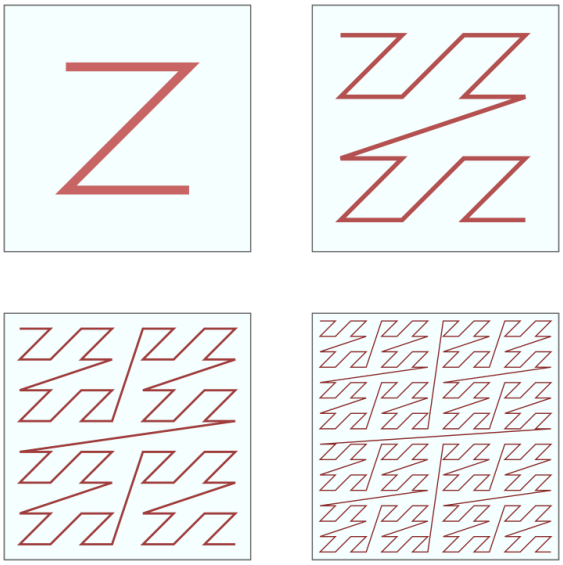


Figure 7.1: A Z-order curve at different resolutions. Each increased resolution follows a similar structure from the previous resolution. Image courtesy of David Eppstein.

Once fetched, the eight bytes corresponding to this texel are passed through the hardware DXT decoder and the local $(x \bmod 4, y \bmod 4)$ texel is returned. Since $p \bmod 2^k$ and $\frac{p}{2^k}$ can be performed efficiently in hardware by bitwise shifts and masks, these calculations are usually very fast. The cache locality is preserved in two ways, first by storing blocks themselves in a Z-order curve, and second by storing the full decompressed block in L1 cache. In the case of DXT, since a single address corresponds to sixteen texels, storing the entire uncompressed block in L1 cache is usually good for performance.

In Chapter 4.3.3 we showed that by allowing a two level addressing scheme, we can effectively reduce the amount of data needed for storing a compressed texture. In particular, most novel compressed texture architectures, such as the binary trees proposed by Inada and McCool (2006), take a unique and format-specific approach to texel addressing. In some situations, such as streaming of real-time assets, CPU-GPU transfer is significantly more expensive than the data access. Each individual program must specify their own method for accessing and reading texture data in accordance to the fixed-function addressing scheme currently imposed by graphics hardware.

With texture programs, the application has explicit control over how to access memory. Certain textures that may be split into constituent parts may allow for better compression of each part individually than for the texture as a whole. For such textures, some parts may be compressed to such a degree as to cover large blocks of texels, amortizing the memory cost across the entire rendering task. Instead of

assuming a data layout or offering a fixed number of available addressing units, texture programs would offer a small but targeted set of instructions for computing one or more memory addresses. The data stored at these memory locations would then be used to construct a texel value to return to the remainder of the texture pipeline. As a preliminary implementation, the existing SIMD compute units in the GPU can be used to generate these texel values, as described in Figure 7.2.

7.1.1 Caching Benefits of Texture Programs

Allowing programmable addressing for a single texture access provides the programmer many benefits. First, abstracting the addressing into texture programs allows the developer to maintain transparent texture access for the rest of the GPU. In other words, the texture access would be defined on a per-texture basis for all other shader programs (vertex shader, fragment shader, etc), reducing code duplication. Second, by restricting the set of operations that can be performed by this addressing unit, the architecture can be optimized for integer operations, boosting performance. Additionally, the developer's choice in layout could benefit the total memory costs of the entire application's rendering task with that texture. For example, a binary tree representation that describes when to look-up texels and when not to could fit almost entirely in cache, allowing significant performance and energy improvements to rendering (Inada and McCool, 2006).

One of the largest benefits of the data layouts for fixed function addressing schemes is the cache coherence of multiple texel accesses. In general, the latency for each texture fetch is expected to be amortized over the cost of subsequent accesses. For this reason, the data layout is constructed in a way that matches the expected access pattern for displaying images. However, this only assumes a spatial coherence of texel accesses and has no notion of spatial coherence of image data. In effect, an image containing random noise will have the same access patterns as any game texture or natural image. In order to improve the cache properties of texture data, texture units can take advantage of intra-texture redundancies by incurring a few additional memory fetches that themselves become amortized over large number of nearby texel accesses. For example, the organization of three-channel texture data can be split into different data streams and collected later for rendering. One such example is to store the R, G, and B (or transformed Y, U, and V) planes of transformed data separately. Then, applications do not have to re-interleave the data prior to uploading them to the GPU and can rather load the planes separately into memory, possibly at different levels of compression. The texture program would request addresses for

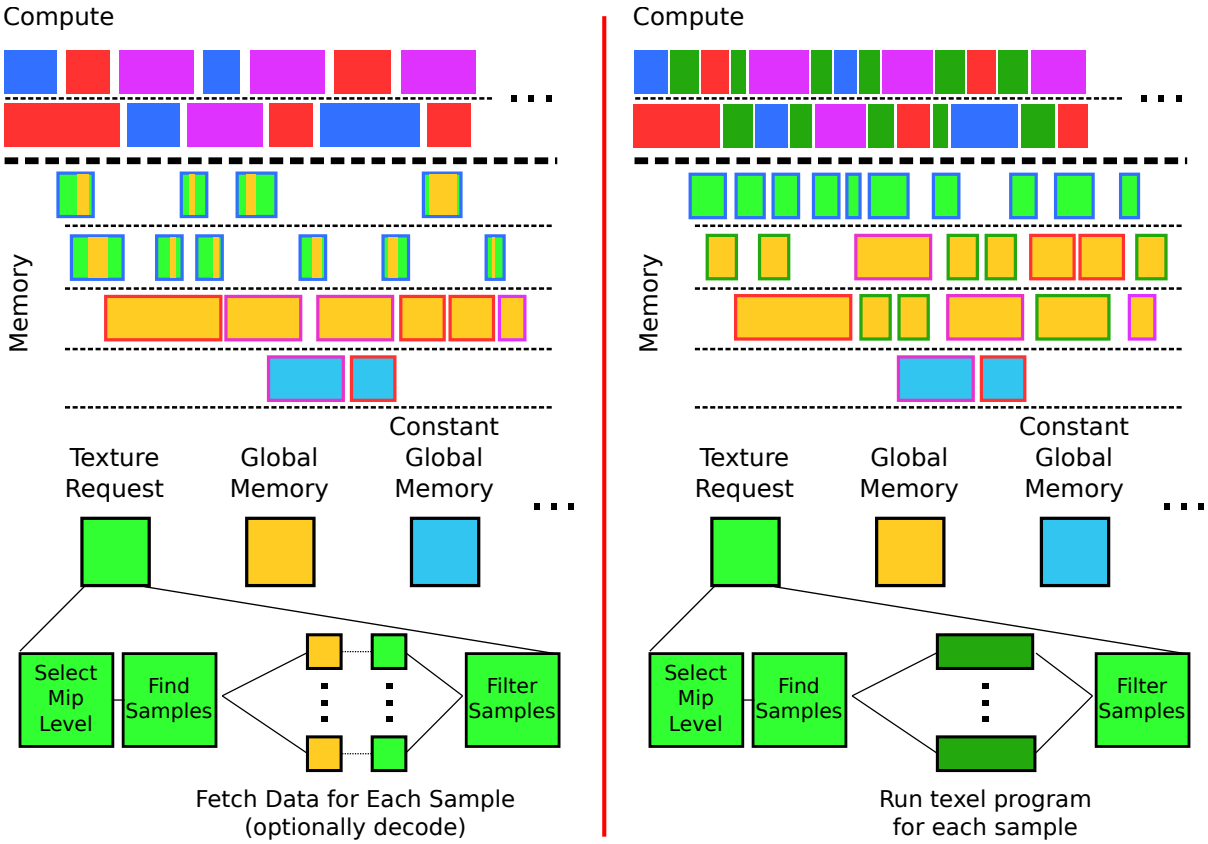


Figure 7.2: (left) The current GPU programming model with respect to texture accesses. (right) The proposed GPU programming model where texel values are generated by running small programs as part of the texture pipeline. Each GPU program (compute) that runs may make one or more memory requests (outlined) from different memory types usually handled in parallel. During each memory request, GPU programs are preempted to allow other parallel work to execute. As a result, the proposed architecture changes would be to allow an additional program to compute the texel samples needed as a part of each texture access. These programs would be free to make their own memory requests in order to compute the final pixel values. Each of the other existing caching mechanisms would remain in place, allowing the programmer to make a choice between optimizing caching behavior and compressed texture size.

the three separate channels and combine them into the final texel value. Certain applications that only require two of the three channels, such as compactly packed normal maps, would not have to pay the cache penalty of fetching all three. Although the first texel requested will incur three times the latency from separate memory accesses, the subsequent texel fetches will find all of the required data already loaded into cache. Waveren and Castaño (2007) show that even using a fragment shader to reconstruct YCoCg encoded DXT5 textures often times provides benefits.

Another advantage that texture programs may have on the caching properties of textures is that we can use a VQ-style dictionary compression for different texture features. In that case, large textures with areas of both low and high detail can be specified with less data. The VQ will replicate the low-detail

areas in the dictionary and many more of the dictionary entries will reside in cache for any given memory address. The resulting application will usually hit the cache for the less detailed areas and miss the cache in the high detail regions, but if a majority of the texture is low detail, the amortized cost over the entire texture may be much smaller. These kinds of cache savings would be very good for managing battery performance with current mobile GPUs.

7.2 Texture Programs for Compressed Textures

For compressed textures, there are a variety of methods that could benefit from texture programs. As in a simple two-level variable block size scheme as described in Chapter 4.3.3, the texture program would be very simple and collapse down to a two-level memory access scheme. Additionally, many of the older texturing architectures become available. For example, the architecture given by Inada and McCool (2006) works very well on textures that have large areas of constant color, but the scheme based on binary trees is expensive. By abstracting away the addressing, such a compression algorithm would be able to be implemented only for the textures that benefit from it.

For other compressed block formats, such as DXT, the hardware could provide addresses for each of the pieces of the compressed block and assemble them for decompression. In particular, each of the two endpoints and the set of indices could all come from different VQ streams. If each texture program was assigned similarly valued texture indices, the impact on cache would be minimal. This would also ameliorate some of the steps in the decompression of the supercompressed textures introduced in Chapter 5.4. Each of the constituent parts would be able to be stored independently, moving the texture assembly to runtime. Similar to the separate channels described in Section 7.1.1, the initial latency for the first texel would certainly be amortized over the course of accessing many of the neighboring texels. Additionally, instead of storing an entire DXT texture in memory, the storage could be optimized to the various pieces of the DXT stream and be reconstructed on the fly.

Finally, traditional image compression algorithms become more feasible for GPUs. If the texel can be quickly decoded, even if the decoder is not implemented in hardware, it might provide a benefit for certain classes of textures. Developers could provide a stream of offsets to entropy encoded data for each texel location. The texture program can then perform the texel decoding at runtime allowing the data to remain compressed in memory. Since JPEG achieves up to 50:1 compression rates, the overhead

of the data offsets and decode time may offset the need for storing the entire uncompressed texture in memory. For example, Olano et al. (2011) show that an entire uncompressed mip-map chain can be compressed down to approximately two bits per texel by using successive mip-map levels as predictors for texel values. Using this scheme, it is concievable that a similar approach could store the entire mip-map chain as compressed on disk and simply reconstruct the texel values as needed during texture access. Certain use-cases for texturing, such as sparse textures, become more feasible with this sort of approach (OpenGL, 2013).

7.3 Other Applications of Texture Programs

The applications of texture programs are more than just for compressed textures. In addition to providing a method for specifying dictionary lookups, texture programs could be used to synthesize large procedural textures from smaller 'seed' textures. For example, using a color palette in conjunction with a noise function to specify the color in the palette to use as a texel value. This is better than the current approach where such a palette is used in the shader because it would still take advantage of texture filtering. GPUs have classically supported the use of palettized textures, but they have become less popular recently in favor of the newer compressed formats(OpenGL, 2004).

Additionally, virtual textures become much more realistic. A texture could have very large dimensions but resolve down to a small number of data accesses. In this way, textures can introduce interesting tiling modes and simulate detail without needing to store each variation in memory. One such example is terrain that may need to decouple the terrain detail from the terrain geometry. Most terrain generation algorithms assign texture coordinates to triangles at runtime in order to access detail textures from a very limited set that are predefined to perform tiling. However, this set of textures could abstract the detail away into many textures which may be logically larger but physically take up a small amount of memory.

7.4 Texture Programs on Current GPU architectures

Current GPUs are organized as a collection of large vector processors that operate on one or more *lanes* of data in parallel, also known as single-instruction multiple-data processors (or SIMD). At run-time, each processor may work on as many as 32 or 64 lanes at once depending on the manufacturer. Each vector processor operates in lock-step, meaning that one instruction is shared across all lanes in the

processor. In general, this maps well to existing graphics pipelines, where many triangle vertices and framebuffer pixels may undergo the same operations over the course of rendering a single frame. Similar to traditional CPU architectures, each of these vector processors has a number of available registers in which to hold intermediate data used for computation. However, due to physical restrictions on the number of registers, the number of lanes in flight at any given time is limited. On some architectures, the number of registers is a severe limitation requiring significant optimizations from the programmer.

One problem introduced using this architecture is that of reading from and writing to data in multiple memory locations in parallel. Many algorithms currently read from many different memory locations at the same time, perform some operations, and then write them out to memory. In current GPUs, these concurrent memory reads and writes are handled by specialized *memory banks*. Memory banks are selected by the memory controller based on the physical address of the memory request. For example, on AMD hardware, there are roughly 256 memory banks that are assigned a memory operation based on the least significant eight bits in the memory address. SIMD algorithms that perform many spatially sequential memory reads at once can take advantage of the full parallelism of the architecture. However, if all memory requests read from an address that is a multiple of the number of memory banks, each memory read is done in sequence. The burden of properly selecting memory banks traditionally rests on the programmer. Data is expected to be laid out in such a way that multiple threads accessing the same memory bank, or *bank conflicts*, are minimized (Harris, 2007). In some situations, however, data is known where it is needed before hand. In this case, a small amount of data may need to be *scattered* or spread across a large memory block (He et al., 2007). In essence, a texture program is a restricted form of this scatter operation. Allowing the texture data to be laid out by the programmer to exploit better access patterns may lead to improved performance of the memory banks.

For image compression, many of the operations performed require defining many per-texel operations. The operations performed here are replicated at several orders of magnitude more times than the number of available SIMD lanes in a typical GPU. For example, one of the first steps in any compressed image format is the transformation from the traditional RGB color space to one in which the perceptually important information is consolidated. One such example is the reversible RGB to modified YUV transform used in Skodras et al. (2001)

$$Y = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor; U = B - G; V = R - G.$$

Converting an image from RGB to YUV would not incur any differences between all per-texel invocations. For a typical 1024×1024 image, we are performing over a million of these computations. The recently announced NVIDIA GTX 1080 has 2560 SIMD processors meaning we can do about 80,000 operations in parallel. By construction, image programs are essentially the same for each texel access, meaning that they already map well to the extreme parallelism afforded by current GPU architectures. Many GPUs currently employ latency hiding to switch out groups of SIMD threads when they wait for memory accesses regardless. Hence, if all threads are required to access a texel at the same time, most of the overhead will be in waiting for the memory accesses to return, and the additional computational load of each texture access should be minimal.

7.5 Potential Limitations

As we showed in Chapter 4.3.3, current texture compression formats are well suited for images of uniform detail. In particular, for certain images, a simple VQ approach to compressed blocks provided worse quality at worse compression rates. This seems to imply that for uniform detail textures, the current cabal of texture compression formats may be a sufficient compromise between compression size, compression quality, and decompression speed. As a result, the introduction of texture programs may actually incur additional decompression time over the previously well-suited fixed-function approach.

Additionally, many aspects of existing image compression techniques do not map well to graphics hardware. Even with increased flexibility in computing final texel values, there is no guarantee that the algorithms for decoding them will provide similar benefits as compressed image algorithms. Additionally, the performance cost of these algorithms is fairly high, so even with compression benefits as good as those shown in Chapter 5.4, the resulting representation may perform worse than existing hardware. However, the availability of the hardware is the biggest benefit, such that any existing limitations can be exposed to the developer giving them the choice based on their specific use case.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

The main issue with current approaches to texture compression is the requirement of the application developer to tailor their textures to the hardware. By coupling the formats to the hardware, the developer has no choice but to use the compressed structure and data layout as defined by the interface chosen to program their GPU. We have seen this limitation both in Chapter 2.6 and in Chapter 4.3.3. In Chapter 2.6, the benefits of the coverage mask compression depended largely on the formats available, both in terms of rendering speed and rendering quality. In Chapter 4.3.3, on the other hand, we demonstrated a simple change to the hardware to benefit an entirely new class of application-specific texture styles. In the remaining chapters, we have tried our best to focus on the general case by using standard data sets for running our evaluation metrics. However, even here many of the results may need to use different metrics for textures that are not consumed strictly visually.

In general, many of the problems with compressed textures stem from this coupling to hardware features. By allowing the texturing pipeline to be programmable as described in Chapter 6.5, we can eliminate some of these issues. This flexibility can certainly benefit the pipeline in some respects while introducing performance implications in others. However, the main benefit is in providing this flexibility to the developer to exploit the structure of their textures based on what their application needs. One very clear example is the use of a different compression method for normal maps and bump maps versus textures used strictly for color. Currently the compression methods are identical regardless of the purpose of the texture.

Overall, the current state of compressed textures is fundamentally similar to where it was a decade ago. Given the increasing complexity of the compressed texture formats, tools are still having a hard time keeping up with hardware improvements. The ideas presented in this dissertation address some of the main issues underlying the delivery of compressed textures, namely the speed at which current encoders work and how formats can be processed to provide better support for applications that may use them. In essence, however, most of the compression algorithms must be tuned for special cases and idiosyncrasies associated with each individual hardware compression format. The production-level

encoders for some formats are therefore still not where many developers would like them to be. Building a high-quality texture compressor still requires significant time and effort that many developers do not have, as it detracts from the main application that they are writing.

Future work: Most of the innovation in texture compression will likely come from a paradigm shift in terms of how we treat textures. In Chapter 6.5 we presented one framework that would provide a significantly new model for reading images on the GPU. Without a comprehensive approach to store and decompress textures given the architectural properties underlying the target applications, many of the current texture specifications are simply adding to the complexity. This complexity may be able to be abstracted away by an intermediate format that is applicable to all current compressed texture formats. In other words, an abstraction of both endpoint and table based compression formats could prove to be a useful delivery mechanism for compressed textures. One such intermediate format was described in Chapter 5.4, although it was only restricted to DXT and PVRTC and assumed that each of these compression formats had black-box encoders. To truly take advantage of the common elements of each format, a signficantly larger scale approach needs to be taken. In particular, for such an intermediate formst, texture compression algorithms should not be encouraged to optimize their approach for any one given format. Rather, textures should be stored in a way that encourages a fast run-time encoding or decoding into the target format for the given GPU. Given such storage, the architectural advantages of a method such as the variable block size method presented in Chapter 4.3.3 become much more feasible. However, this approach still hides the underlying problem of a fractured state of support for compressed textures within commodity GPUs.

As an example of the multi-format approach, the compression benefits presented in Chapter 3.5 are applicable to *classes* of compressed textures. They were not developed with any particular format in mind although the demonstration of their applicability was restricted to a single format. The reason for this limitation is the aforementioned complexity in developing compressors for each individual format. Additionally, the underlying benefits are not clear between existing texture compression formats. There are textures for which PVRTC does better and which DXT does not, and vice versa (Fenney, 2003). ASTC is a step in the right direction for supporting the variety of different compressed texture methods, but many of the good ideas from other formats, such as separating luminance and chrominance (ETC/ETC2) and interpolating endpoints across block boundaries (PVRTC) are not present. Extending

ASTC to support these features would likely increase its applicability and allow developers to focus on one single format.

Finally, there is still ample room for additional research into proper usage of texture data. The increasing influence of the mobile space and the convergence of virtual reailty (VR) will likely make GPU access to video data for the purposes of texturing arbitrary geometry much more significant in the coming years. Figuring out a new paradigm for compressed video encoding that emits compressed texture frames rather than full-resolution video frames may have significant benefits. Furthermore, many of the compressed formats can be encoded using GPU encoders. Subsets of ASTC are already targeted by some CUDA-based compression tools[1]. These tools can be used in conjunction with supercompression techniques described in Chapter 5.4 in order to create a run-time encoded format that does more than simply re-pack the compressed texture data after decompression. In doing so, the images can be preprocessed for storage using traditional compression techniques and then reconstructed on-the-fly at runtime. Although many of these problems are unsolved, there is still ample work to be done to make them better. As both power consumption and memory latency become more important, so will the need for efficiently compressed textures.

[1] `https://developer.nvidia.com/content/astc-compression-gets-cuda-boost`

BIBLIOGRAPHY

Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., and Ssstrunk, S. (2010). SLIC Superpixels.

Achanta, R., Shaji, A., Smith, K., Lucchi, A., Fua, P., and Ssstrunk, S. (2012). SLIC Superpixels Compared to State-of-the-art Superpixel Methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(11).

Aila, T., Miettinen, V., and Nordlund, P. (2003). Delay streams for graphics hardware. *ACM Trans. Graph.*, 22(3):792–800.

Ajtai, M. (1998). The shortest vector problem in l2 is NP-hard for randomized reductions (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 10–19. ACM.

Akenine-Möller, T. and Aila, T. (2005). Conservative and tiled rasterization using a modified triangle set-up. *J. Graphics Tools*, 10(3):1–8.

AMD (2008). The compressonator. `http://developer.amd.com/archive/gpu/compressonator`.

Apple (2013). Best practices for working with texture data. `https://developer.apple.com/library/ios/`.

ARM (2012). ASTC evaluation codec. `http://malideveloper.arm.com/develop-for-mali/tools/astc-evaluation-codec/`.

Athitsos, V., Potamias, M., Papapetrou, P., and Kollios, G. (2008). Nearest neighbor retrieval using distance-based hashing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 327–336.

Aydin, T. O. (2010). *Human visual system models in computer graphics*. Doctoral dissertation, Universität des Saarlandes, Saarbrücken.

Barett, S. and Giesen, F. (2009). DXTC encoder. `https://github.com/nothings/stb/blob/master/stb_dxt.h`.

Barros, J. and Fuchs, H. (1979). Generating smooth 2-d monocolor line drawings on video displays. *SIGGRAPH Comput. Graph.*, 13(2):260–269.

Beers, A. C., Agrawala, M., and Chaddha, N. (1996). Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 373–378. ACM.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Bloom, C. (2009). Oodle, distributed with the Granny3D package from RAD Game Tools. `http://www.radgametools.com/granny.html`.

Brown, S. (2006). libsquish. `http://code.google.com/p/libsquish/`.

Buccigrossi, R. W. and Simoncelli, E. P. (1999). Image compression via joint statistical characterization in the wavelet domain. *IEEE Transactions on Image Processing*, 8(12):1688–1701.

Campbell, G., DeFanti, T. A., Frederiksen, J., Joyce, S. A., and Leske, L. A. (1986). Two bit/pixel full color encoding. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 215–223. ACM.

Castaño, I. (2007). High Quality DXT Compression using CUDA. *NVIDIA Developer Network*.

Catmull, E. E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis. AAI7504786.

Chandler, J. E., Bunker, W. M., Economy, R., Fadden, J. R. G., and Nelson, M. P. (1986). Advanced video object generator. U. S. Patent 4727365.

Chen, H. (2016). From pixels to reality - thinking differently about graphics in games. In *Keynote Address of the ACM SIGGRAPH/EUROGRAPHICS conference on Interactive 3D Graphics and Games*, I3D '16. ACM.

Cohen, A., Daubechies, I., and Feauveau, J.-C. (1992). Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560.

Delp, E. and Mitchell, O. (1979). Image compression using block truncation coding. *Communications, IEEE Transactions on*, 27(9):1335–1342.

Donovan, W. (2010). BC7 export, distrubuted with NVIDIA texture tools. `http://code.google.com/p/nvidia-texture-tools/`.

Duce, D. (2003). Portable network graphics (PNG) specification (second edition). W3C recommendation, W3C. http://www.w3.org/TR/2003/REC-PNG-20031110.

Duda, J. (2013). Asymmetric numeral systems as close to capacity low state entropy coders. *CoRR*, abs/1311.2540.

Dufresne, M. F. (2013). Fast ispc texture compressor. `http://software.intel.com/en-us/articles/fast-ispc-texture-compressor`.

Economy, R., Fadden, J. R. G., and Steiner, W. R. (1987). Yiq based color cell texture. U. S. Patent 4965745.

Fatahalian, K., Boulos, S., Hegarty, J., Akeley, K., Mark, W. R., Moreton, H., and Hanrahan, P. (2010). Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph.*, 29(4):67:1–67:8.

Felzenszwalb, P. F. and Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. *Int. J. Comput. Vision*, 59(2):167–181.

Fenney, S. (2003). Texture compression using low-frequency signal modulation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 84–91. Eurographics Association.

Fiume, E., Fournier, A., and Rudolph, L. (1983). A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *SIGGRAPH Comput. Graph.*, 17(3):141–150.

Fränti, P., Nevalainen, O., and Kaukoranta, T. (1994). Compression of digital images by block truncation coding: A survey. *The Computer Journal*, 37(4):308–332.

Geldreich, R. (2012). Crunch. `http://code.google.com/p/crunch/`.

Geldreich, R. (2013). Fast, high quality ETC1 (ericsson texture compression) block packer/unpacker. `http://code.google.com/p/rg-etc1/`.

Gersho, A. and Gray, R. M. (1991). *Vector quantization and signal compression*. Kluwer Academic Publishers.

Giesen, F. (2014). Interleaved entropy coders. *CoRR*, abs/1402.3392.

Google (2016). Skia – 2d rendering library. `https://www.skia.org/`.

Green, C. (2007). Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 9–18, New York, NY, USA. ACM.

Griffin, W. and Olano, M. (2014). Objective image quality assessment of texture compression. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 119–126, New York, NY, USA. ACM.

Hakura, Z. S. and Gupta, A. (1997). The design and analysis of a cache architecture for texture mapping. *ACM SIGARCH Computer Architecture News*, 25(2):108–120.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160.

Harris, M. (2007). Parallel prefix sum (scan) with CUDA. In Nguyen, H., editor, *Gpu Gems 3*. Addison-Wesley Professional, first edition.

He, B., Govindaraju, N. K., Luo, Q., and Smith, B. (2007). Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 46:1–46:12, New York, NY, USA. ACM.

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Imagination (2013). PowerVR Insider SDK and Utilities. `http://www.imgtec.com/powervr/insider`.

Imagination (2016). PowerVR graphics. `http://imgtec.com/powervr/graphics/architecture`.

Inada, T. and McCool, M. D. (2006). Compressed lossless texture representation and caching. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '06, pages 111–120.

Iourcha, K. I., Nayak, K. S., and Hong, Z. (1999). System and method for fixed-rate block-based image compression with inferred pixel values. U. S. Patent 5956431.

Ivanov, D. V. and Kuzmin, Y. (2000). Color distribution - a new approach to texture compression. *Comput. Graph. Forum*, 19(3):283–290.

Jiao, Y., Lin, H., Balaji, P., and Feng, W. (2010). Power and performance characterization of computational kernels on the GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228.

Karp, R. (1972). Reducibility among combinatorial problems. In Miller, R., Thatcher, J., and Bohlinger, J., editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US.

Kautz, J., Lehtinen, J., and Aila, T. (2004). Hemispherical rasterization for self-shadowing of dynamic objects. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 179–184, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. (2011). GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17.

Kilgard, M. J. and Bolz, J. (2012). GPU-accelerated path rendering. *ACM Trans. Graph.*, 31(6):172:1–172:10.

Kim, C.-H. and Park, I.-C. (2007). Parallel decoding of context-based adaptive binary arithmetic codes based on most probable symbol prediction. *IEICE - Trans. Inf. Syst.*, E90-D(2):609–612.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

Klein, S. T. and Wiseman, Y. (2003). Parallel Huffman decoding with applications to JPEG files. *THE COMPUTER JOURNAL*, 46:487–497.

Knittel, G., Schilling, A., Kugler, A., and Straßer, W. (1996). Hardware for superior texture performance. *Computers & Graphics*, 20(4):475–481.

KODAK (1999). Kodak lossless true color image suite. available at `http://r0k.us/graphics/kodak`.

Kokojima, Y., Sugita, K., Saito, T., and Takemoto, T. (2006). Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA. ACM.

Krajcevski, P., Golas, A., Ramani, K., Shebanow, M., and Manocha, D. (2016a). VBTC: GPU-friendly variable block size texture encoding. *Computer Graphics Forum*, 35(2):409–418.

Krajcevski, P., Lake, A., and Manocha, D. (2013). FasTC: accelerated fixed-rate texture encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 137–144. ACM.

Krajcevski, P. and Manocha, D. (2014a). Real-time low-frequency signal modulated texture compression using intensity dilation. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 127–134, New York, NY, USA. ACM.

Krajcevski, P. and Manocha, D. (2014b). SegTC: Fast Texture Compression using Image Segmentation. pages 71–77, Lyon, France. Eurographics Association.

Krajcevski, P. and Manocha, D. (2016). Compressed coverage masks for path rendering on mobile GPUs. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1.

Krajcevski, P., Pratapa, S., and Manocha, D. (2016b). GST: GPU-decodable supercompressed textures. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*.

Krause, P. K. (2010). ftc–Floating precision texture compression. *Computers Graphics*, 34(5):594–601. CAD/GRAPHICS 2009, Extended papers from the 2009 Sketch-Based Interfaces and Modeling Conference, Vision, Modeling & Visualization.

Lane, J. M. and M. Rarick, R. a. (1983). An algorithm for filling regions on graphics display devices. *ACM Trans. Graph.*, 2(3):192–196.

Leskela, J., Nikula, J., and Salmela, M. (2009). Opencl embedded profile prototype in mobile device. In *2009 IEEE Workshop on Signal Processing Systems*, pages 279–284.

Levkovich-Maslyuk, L., Kalyuzhny, P., and Zhirkov, A. (2000). Texture compression with adaptive block partitions (poster session). In *Proceedings of the 8th ACM International Conference on Multimedia 2000, Los Angeles, CA, USA, October 30 - November 3, 2000.*, pages 401–403.

Lloyd, S. (1982). Least squares quantization in PCM. *Information Theory, IEEE Transactions on*, 28(2):129–137.

Loop, C. and Blinn, J. F. (2005). Resolution independent curve rendering using programmable graphics hardware. In *July 2005 Transactions on Graphics (TOG) Volume 24 Issue 3 (Siggraph 2005)*. Association for Computing Machinery, Inc.

Malvar, H. S., Sullivan, G. J., and Srinivasan, S. (2008). Lifting-based reversible color transformations for image compression. In *SPIE Applications of Digital Image Processing*. International Society for Optical Engineering.

Mavridis, P. and Papaioannou, G. (2012). Texture compression using wavelet decomposition. *Computer Graphics Forum*, 31(7(1)):2107–2116.

Microsoft (2010). Directx software development kit. `http://www.microsoft.com/en-us/download/details.aspx?id=6812`.

Munkberg, J., Clarberg, P., Hasselgren, J., and Akenine-Mller, T. (2008). Practical HDR Texture Compression. *Computer Graphics Forum*, 27(6):1664–1676.

Nasrabadi, N. M., Choo, C. Y., Harries, T., and Smallcomb, J. (1990). Hierarchical block truncation coding of digital HDTV images. *IEEE Trans. on Consum. Electron.*, 36(3):254–261.

Nixon, K. W., Chen, X., Zhou, H., Liu, Y., and Chen, Y. (2014). Mobile GPU power consumption reduction via dynamic resolution and frame rate scaling. In *Proceedings of the 6th USENIX Conference on Power-Aware Computing and Systems*, HotPower'14, pages 5–5, Berkeley, CA, USA. USENIX Association.

nVidia (2014). Maxwell Architecture. `https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/`.

nVidia (2015). The NVIDIA GeForce GTX 980 Review: Color Compression. `http://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/3`.

NVIDIA (2016). Nvidia Tesla P100 - the most advanced data center accelerator ever built. featuring Pascal GP100, the world's fastest GPU. `http://www.nvidia.com/object/pascal-architecture-whitepaper.html`.

Nystad, J., Lassen, A., Pomianowski, A., Ellis, S., and Olson, T. (2012). Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High Performance Graphics*, HPG '12, pages 105–114. Eurographics Association.

Olano, M., Baker, D., Griffin, W., and Barczak, J. (2011). Variable bit rate GPU texture decompression. In *Proceedings of the Twenty-second Eurographics Conference on Rendering*, EGSR '11, pages 1299–1308, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

on Illumination, I. C. (2004). *Colorimetry*. CIE technical report. Commission internationale de l'Eclairage, CIE Central Bureau.

Oom, D. (2016). Real-time adaptive scalable texture compression for the web. Master's thesis, Chalmers University of Technology. 39.

OpenGL, A. R. B. (2000). 3DFX_texture_compression_FXT1. `https://www.opengl.org/registry/specs/3DFX/texture_compression_FXT1.txt`.

OpenGL, A. R. B. (2004). EXT_paletted_texture. `https://www.opengl.org/registry/specs/EXT/paletted_texture.txt`.

OpenGL, A. R. B. (2010). ARB_texture_compression_bptc. `http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt`.

OpenGL, A. R. B. (2013). ARB_sparse_texture. `http://www.opengl.org/registry/specs/ARB/sparse_texture.txt`.

OpenGL, A. R. B. (2014). ANDROID_extension_pack_es31a. `https://www.khronos.org/registry/gles/extensions/ANDROID/ANDROID_extension_pack_es31a.txt`.

Pereberin, A. (1999). Hierarchical approach for texture compression. In *Proceedings of the International Conference Graphicon*, Moscow, Russia.

Pixar (2015). Pixar one twenty eight. `https://community.renderman.pixar.com/article/114/library-pixar-one-twenty-eight.html`.

Pohl, D., Nickels, S., Nalla, R., and Grau, O. (2014). High quality, low latency in-home streaming of multimedia applications for mobile devices. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 687–694.

Qin, Z. (2009). *Vector Graphics for Real-time 3D Rendering*. PhD thesis, University of Waterloo.

Rissanen, J. and Langdon, G. G. (1979). Arithmetic coding. *IBM J. Res. Dev.*, 23(2):149–162.

Roimela, K., Aarnio, T., and Itäranta, J. (2006). High dynamic range texture compression. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 707–712. ACM.

Ross, F. (2012). Migrating to LPDDR3. *LPDDR3 Symposium 2012*.

Rubinstein, M., Gutierrez, D., Sorkine, O., and Shamir, A. (2010). A comparative study of image retargeting. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 29(6):160:1–160:10.

Salomon, D. and Motta, G. (2010). *Handbook of Data Compression*. Springer London.

Schneider, J. (2013). GPU-friendly data compression. Presentation at GPU Technology Conference.

Serra, J. (1983). *Image Analysis and Mathematical Morphology*. Academic Press, Inc.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656.

Shebanow, M. C. (2014). The evolution of mobile graphics and the potential impact on interactive applications. In *Keynote Address of the ACM SIGGRAPH Symposium on Mobile Graphics and Interactive Applications*, SIGGRAPH ASIA '14. ACM.

Skodras, A., Christopoulos, C., and Ebrahimi, T. (2001). The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5):36–58.

StatCounter, I. (1999-2014). Global stats, top 5 desktop, tablet & console browsers from sept 2013 to sept 2014. `http://gs.statcounter.com`.

Ström, J. and Akenine-Möller, T. (2004). PACKMAN: texture compression for mobile phones. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, pages 66–. ACM.

Ström, J. and Akenine-Möller, T. (2005). iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 63–70. ACM.

Ström, J. and Pettersson, M. (2007). ETC2: texture compression using invalid combinations. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 49–54. Eurographics Association.

Ström, J. and Wennersten, P. (2011). Lossless compression of already compressed textures. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 177–182. ACM.

Sun, W., Lu, Y., Wu, F., and Li, S. (2008). DHTC: an effective DXTC-based HDR texture compression scheme. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 85–94. Eurographics Association.

Thompson, W., Fleming, R., Creem-Regehr, S., and Stefanucci, J. K. (2011). *Visual Perception from a Computer Graphics Perspective*. A. K. Peters, Ltd., 1st edition.

Torborg, J. and Kajiya, J. T. (1996). Talisman: Commodity realtime 3d graphics for the PC. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 353–363, New York, NY, USA. ACM.

van Emde Boas, P. (1981). Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report MI-UvA-81-04, Department of Mathematics, University of Amsterdam.

Veksler, O., Boykov, Y., and Mehrani, P. (2010). Superpixels and supervoxels in an energy optimization framework. In Daniilidis, K., Maragos, P., and Paragios, N., editors, *ECCV 2010*, volume 6315 of *Lecture Notes in Computer Science*, pages 211–224. Springer Berlin Heidelberg.

von zur Gathen, J. and Sieveking, M. (1978). A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):pp. 155–158.

Wallace, G. K. (1992). The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv.

Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E. (2004). Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612.

Waveren, J. M. P. v. (2006a). Real-time DXT Compression. *Intel Software Network*.

Waveren, J. M. P. v. (2006b). Real-time texture streaming and decompression. *Id Software Technical Report*.

Waveren, J. M. P. v. and Castaño, I. (2007). Real-time YCoCg-DXT Compression. *NVIDIA Developer Network*.

Waveren, J. M. P. v. and Castaño, I. (2008). Real-time Normal Map DXT Compression. *NVIDIA Developer Network*.

Wei, L.-Y. (2004). Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 55–63.

Wennersten, P. and Ström, J. (2009). Table-based alpha compression. *Computer Graphics Forum*, 28(2):687–695.

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Zhang, H., Manocha, D., Hudson, T., and Hoff, III, K. E. (1997). Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 77–88, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.