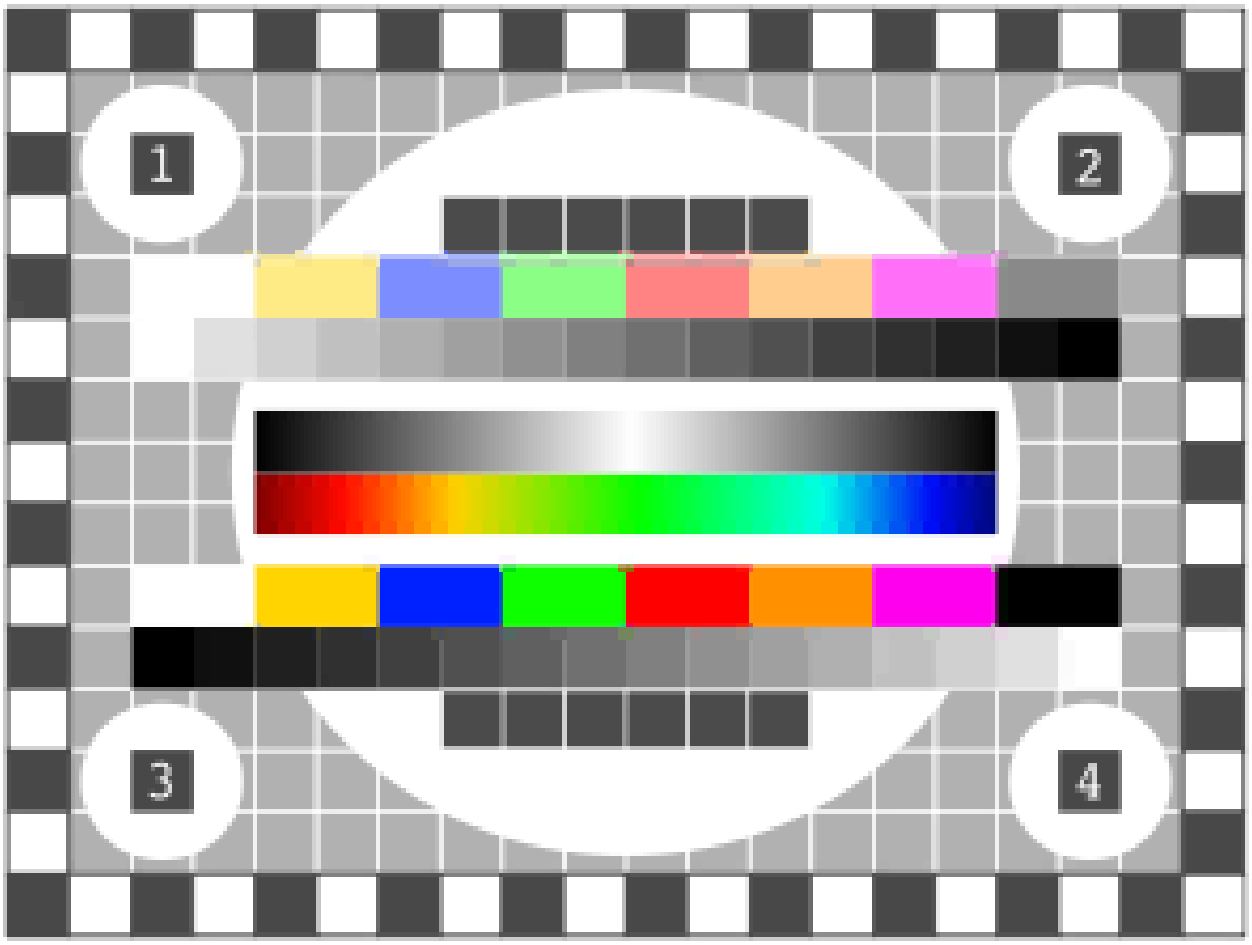
CHALMERS UNIVERSITY OF GOTHENBURG



Real-Time Adaptive Scalable Texture Compression for the Web

Master's thesis in Computer Science and Engineering

DANIEL OOM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2016

Real-Time Adaptive Scalable Texture Compression for the Web
DANIEL OOM

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
An example picture that have been encoded and decoded using the presented algorithm, showing some of the encoding error.

Göteborg, Sweden 2016

Real-Time Adaptive Scalable Texture Compression for the Web
Master's thesis in Computer Science and Engineering
DANIEL OOM
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

This thesis investigates using Adaptive Scalable Texture Compression in real-time to improve memory utilization in web browsers. A fast encoding method that uses heuristics and avoids the commonly used branch and bound method is presented. Performance benchmarks and quality comparisons against other commonly used encoders show that it is possible to compress textures with reasonable quality in real-time.

Keywords: Web Browser, Texture Compression, Real-time, Adaptive Scalable Texture Compression, Heuristics

Abstract

Detta examensarbetet undersöker användandet av texturkomprimering i realtid för att förbättra minnesanvändandet i webbläsare. En snabb metod baserad på heuristik som undviker den mer vanligen använda "branch and bound"-metoden. Prestandamätningar och kvalitetsjämförelser gentemot andra populära texturkomprimeringsprogram visar att det är praktiskt möjligt att komprimera texturer i realtid med rimlig kvalitet.

Acknowledgements

Special thanks to Christian Kinndal, for mentoring at Opera Software; Erik Sintorn, for helping me keep on track as my supervisor at Chalmers; and Ulf Assarsson, for agreeing to be my examiner. I also thank my girlfriend, Ingrid, for supporting me whenever my motivation was waning.

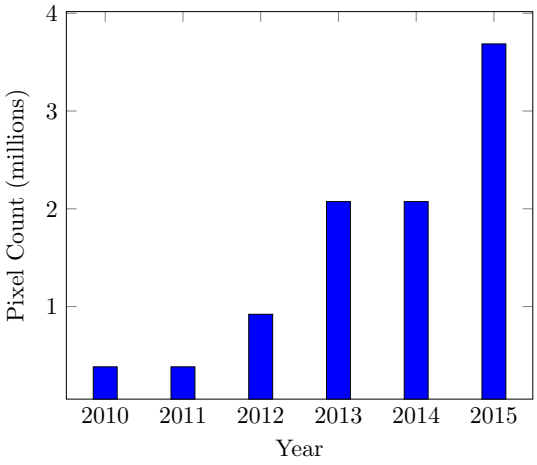
Contents

1 Introduction

1.1 Background

Over the years 2010 to 2016, the resolution of hand-held devices increased significantly. For example, the resolution of Samsung Galaxy S up to Samsung Galaxy S6 is shown in figure 1.1, similar advancement can be observed for most other mobile devices as well. This increase in pixel count causes a significant increase in memory usage for buffered screen rendering. Storing a buffer using a 32-bit RGBA format requires four bytes per pixel, for a Quad HD display with $2560 \times 1440 \approx 4$ million pixels that means a single full-screen buffer requires almost 15 megabytes of memory. Depending on how much render buffering an application uses, this can add up to a significant amount of the total memory usage.

Figure 1.1: *Samsung Galaxy device resolution*



Any web browser based on the rendering engine in Chromium uses a significant amount of memory for render buffers due to how web page rendering is implemented. The rendering engine in Chromium is designed to handle everything from the simplest piece of black text on white background to very complex websites. This means that rendering can be very expensive depending on what content is being displayed. To support smooth scrolling, among other things, the browser divides the web site (including non-visible parts) into parts referred to as *tiles*. Thus, tiles that are already rendered do not have to be rendered again if the page is scrolled by the user. Also, when the page is scrolled it is beneficial if the tiles that are scrolled into view are pre-rendered so that they appear instantly to the user and not after a delay due to rendering. However, buffering the tiles can require a significant amount of memory.

Opera Software ASA who develops several Chromium based products, among them Opera for Android, have made attempts at optimizing the memory used for tiles by using compression. Compression helps directly with the memory usage as the data is stored with fewer bytes, but it comes with its own set of problems. For example the previous work by Opera has problems related to quality due to the use of lossy compression and performance problem due to the design of the used compression formats. The purpose of this project is to continue Opera's previous work and evaluate to what extent it is feasible to compress the buffered tiles in real-time to save memory.

1.2 Problem

For compression to be usable in the browser it must not compromise the user experience. Thus, no human noticeable loss in quality is tolerated and the performance must be high enough such that there is almost no noticeable delay before displaying a page compared to using no compression. This poses a challenge since data compression in general is a computationally expensive problem. Another problem is that the web contains most of all conceivable types of graphics such as: text, gradients, solid colors, sharp and soft edges, chromatic variance, transparency, low frequency and high frequency. Since image compression relies on being able to

represent repeating patterns with fewer bits, it is difficult to maintain both high compression ratio and high quality for all possible patterns.

1.3 Limitations

To limit the scope of this project, one compression method have been selected, namely texture compression. The reason for using texture compression is that texture compression designed for high performance graphics applications and could thus provide the rendering performance we require. Further, only one texture compression format is going to be examined in detail, namely the Adaptive Scalable Texture Compression (ASTC) format developed by ARM [EN12]. Though, the many similarities between texture compression formats allows for conclusions to be drawn about issues with the other formats as well. An encoder will be implemented and the focus will be on reasonable quality and very high performance.

ASTC was selected because it is a newer format that have taken the ideas from previous texture compression formats and improved upon them. Thus, it should provide the most useful features for the varied types of images that this project is targeting. To decide what is constituted as good enough compression quality is a thesis in itself and thus for this project we will simply state that the reference encoder for ASTC provides sufficient quality when run in its fastest compression mode. As for performance, the encoder needs to be fast enough to compress the image with no noticeable delay for the user. That means that the delay should ideally not be more than a few tenths of a second for compressing all visible tiles for a web page. Non-visible tiles can take more time as long as they are not immediately needed.

2 Previous Work

This section presents the research and texture compression formats that Adaptive Scalable Texture Compression builds on. Additionally, previous work on fast encoding of different texture compression formats is also presented.

2.1 Texture Compression Formats

The first block based image compression method was Block Truncation Coding (BTC), presented by Delp and Mitchell [DM79]. BTC compresses single channel images by splitting them up into fixed size blocks and compressing each block independently. Texture compression is not discussed directly here but virtually all texture compression formats use the proposed block based compression scheme.

Campbell et al. [Cam+86] discuss compressing color images by using BTC on each color channel independently. They also introduce Color Cell Compression (CCC), which uses a color map scheme with lower bit-rate than channel independent BTC. Using CCC for texture mapping was introduced by Knittel et al. [Kni+96], where they present a hardware rasterizer that renders textures directly from the compressed state. Beers et al. present a compression method based on vector quantization together with another hardware rasterizer [BAC96]. Both vector quantization and the palette methods requires additional memory lookups during the decoding stage, incurring hardware complexity and lower performance.

S3 Texture Compression (S3TC) was introduced by Iourcha et al. [INH99]. They introduced the endpoint interpolation method which does not require extra memory lookups in the decoder. Additionally, S3TC also supports color images with and without an alpha channel. Thanks to inclusion in DirectX and OpenGL, S3TC has been very popular in computer graphics applications. S3TC is a set of five different formats (in DirectX referred to as DXTC). The first format (DXT1) encodes images with three color channels while the other four formats extends DXT1 with different methods for encoding an alpha channel. POOMA, presented by Akenine-Möller and Ström [AS03], uses the ideas from S3TC but changes the format to suit the hardware architecture they designed.

Akenine-Möller and Ström presented another texture compression format, PACKMAN, which uses an entirely different luminance modulation based scheme [SA04]. They also presented iPACKMAN [SA05] which improves encoding of blocks with small, continuous chromatic changes. In iPACKMAN, a new differential coding is introduced that can be used instead of the direct coding from PACKMAN, and the coding can be chosen on a per-block basis. Later, iPACKMAN was also referred to as Ericsson Texture Compression (ETC). Blocks containing several distinct chromatic values are not encoded well by iPACKMAN. Pettersson and Ström presented a new format, THUMB, which introduces a method that can encode such blocks [PS05]. However, THUMB is a separate texture compression format and does not work well for the kind of blocks where iPACKMAN excels. With Ericsson Texture Compression 2 (ETC2), Ström and Pettersson presented an encoding where iPACKMAN or THUMB can be chosen on a per block basis [SP07]. Based on the ordering technique, first used in S3TC and later named by Munkberg et al. [MAS06], ETC2 uses invalid bit sequences to encode more options without requiring more bits.

Microsoft renamed the DXTC formats in Direct3D 10 to Block Compression (BC). For Direct3D 11 Microsoft developed two new formats, BC6H and BC7 [Mic09]. These formats are based on the endpoint interpolation method from S3TC, together with the introduction of partitioning and support for high dynamic range textures. Partitioning is another solution for the problem solved by THUMB, where the texels of a block is divided into sets and each set is encoded with its own pair of endpoints. This allows encoding blocks where colors are distributed in distinct clusters with differing extents. NVIDIA also developed an OpenGL extension for BC6H and BC7 [ARB11].

Finally, the most recent established texture compression format is Adaptive Scalable Texture Compression, presented by Nystad in 2011 [Nys+12]. This new format builds on S3TC, ETC2, BC6H and BC7 while introducing high configurability allowing to choose between higher compression ratio or better visual quality.

2.2 Encoders and Performance

At the same time as the texture compression formats have grown in complexity, the encoding processes have suffered in performance. Many encoders for the S3TC formats exists [Cas10; AMD08; Bro06b], but most of

them are designed for offline usage and are thus not usable for real-time texture compression. Waveren [Wav06] have researched real-time S3TC compression on the CPU side and Castaño has worked on real-time GPU encoding [Cas07]. Waveren and Castaño also presented a real-time S3TC compressor that uses a different color space [WC07]. Even more extreme S3TC encoding performance have been presented by Peter Uličiansky [Uli10]. For the newer texture compression formats Krajcevski, Lake and Manocha [KLM13] have presented a fast encoding algorithm that can do partitioning. Krajcevski and Manocha also developed a fast partitioning method using a full image segmentation preprocessing stage [KM14].

3 Texture Compression

This chapter introduces the theory behind texture compression. We will begin with a section introducing color theory and texture mapping, which is then followed by an overview of the ideas used in texture compression. Following that is a detailed description of the Adaptive Scalable Texture Compression format. Finally, the last section describes different algorithms and methods used in texture compression.

3.1 Color Theory

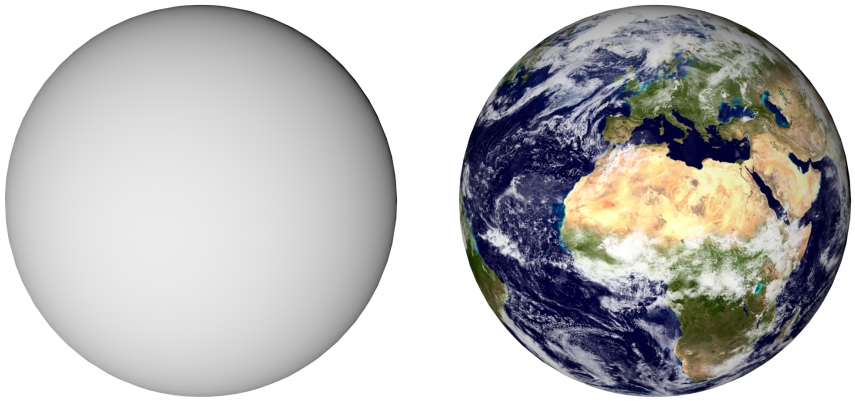
To understand the intrinsics of texture compression we need to understand what colors are and how they can be represented in a computer. The definition of *color* is derived from how the human visual system interprets the part of the electromagnetic spectrum that humans can see. A *color model* mathematically describes how to construct colors. Color models are usually defined using tuples of numbers with three or four components. Further, a *color space* is defined by describing how a color model is interpreted by the human visual system.

One of the most commonly used color spaces in computer graphics is the RGB color space. The design of RGB is based on human vision. Human vision uses three light receptors that detects different parts of the electromagnetic spectrum. Each receptor roughly corresponds to red, green and blue light. RGB is designed for light transmitters such as computer screens. Three emitters are used, each targeting one receptor in the human vision system, constructing colors in an additive fashion. The three components of RGB are thus red, green and blue.

Since the colors are described with three components, the color space can be interpreted as a volume in three dimensional euclidean space. This allows the use of mathematical concepts applying to euclidean spaces, such as linear algebra, something that is extensively used in all texture compression formats.

3.2 Texture Mapping

Surface textures are images used in computer graphics to add detail to graphic objects, a method referred to as texture mapping. Originally texture mapping only added color to surfaces through color mapping, as shown in figure 3.1. However, today textures are used for graphical features such as geometrical displacement of surfaces, shadows and reflections [AHH08]. Texture mapping is done by assigning a two-dimensional coordinate, referred to as a texture coordinate, to every vertex in a model. Coordinates between vertices are interpolated and then the screen pixel is calculated from the texture pixels (texels) using texture filtering. Texture filtering describes a method to find the color from one or more texels as the rendered pixels belonging to the textured surface may not match with the texture image pixels.



(a) *A sphere without textures.* (b) *The same sphere with color mapping.*

Figure 3.1: *Example of color mapping making a sphere look like the earth. Earth map image from the Visible Earth project by NASA.*

The amount of memory available to the graphics processor is the factor that limits the amount of textures that can be used. Higher-resolution textures allows more detail to be added but also uses more memory. The amount of detail graphics applications can add through textures is thus limited by the amount of available memory. For color mapping with raster images, the textures can be stored in RGB format where each channel is described by one byte. This means that a 1024 by 1024 texture would use around three megabytes of memory. That may not seem like much, but considering that modern graphics applications may use thousands of textures, available memory is easily exhausted. This is the motivation behind the development and usage of texture compression.

3.3 Compressing Textures

Texture compression methods is a subset of image compression methods that can be used to make each texture occupy less memory, thus enabling the usage of more and higher-resolution textures. Many texture compression formats, with different features, have been developed and all of them share the following four properties:

1. Fast decoding
2. Random access
3. High visual quality
4. Slow encoding

Firstly, fast decoding, or more specifically the possibility to implement a decoder in hardware with few components, is essential. Fast hardware decoders allow texture decompression to happen in real-time as part of the rendering pipeline. This means that the textures never have to be stored uncompressed, which saves bandwidth between the memory and the processor. Additionally, sending compressed data can actually increase performance when bandwidth is the bottleneck since more texture data can be sent simultaneously. Secondly, random access is required as the GPU is inherently parallel and the order in which the texture will be accessed is random. Thirdly, high visual quality is also very important as compression artifacts may negatively impact the visual experience of the graphical application. Finally, as texures are mainly created at development time of the application, there is sufficient processor time available which allows the compression to take much longer time than the decompression.

3.3.1 Fixed Rate and Block Division

To create an encoding scheme that accommodates the four parameters described above, we start with dividing the image into a grid of equally sized blocks and then encoding each block independently with a fixed number of bits. Then for each block, two pieces of information needs to be encoded: *color spaces* and *color specifiers*. Each pixel is through a color specifier tied to a point in one of the chosen color spaces.

This encoding satisfies the random access property because locating the block which a specific pixel belongs to can be done with just algebra and only one block needs to be decompressed to find the color of one specific pixel. To satisfy the third property it is assumed that pixels close to each other in coordinate space are also related in color space, which enables visual quality to be maintained even for high compression ratio. The first property is satisfied by making sure few operations are needed to go from the encoded data to the decoded pixels. Finally, the fourth property is an unfortunate consequence that stems from the underlying computational problem the encoder has to solve. Choosing the color spaces and color specifiers such that the resulting error of encoding and decoding is as small as possible turns out to be very difficult.

3.3.2 Block Definitions

A few different definitions for the color spaces and the color specifiers have been presented. This section explains a few of the ideas behind the definitions.

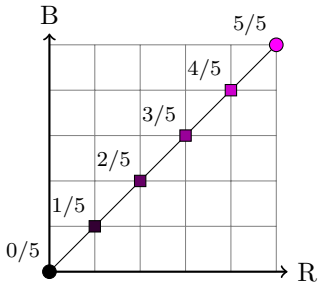
Quantization

Since each block is encoded with a fixed number of bits, it is not always possible to encode values with the same number of bits as they are represented with in the input. Thus, the original values needs to be mapped to representations using fewer bits, this process is called *quantization*. Representation with fewer bits is just an example of quantization, which in more general terms means to constrain one set of values to another set with fewer values. Quantization is the source of compression error, as information is by definition lost during the process. Another mathematical example is quantizing the set of real numbers to the set of integers through the use of rounding.

For texture compression there are two quantization processes employed. The first process is color quantization, which maps a set of input colors to colors in a smaller set of output colors, often referred to as a palette. Each color value in the original set is mapped to a color in the palette with an index. The palette has to be chosen such that the representation of indexes for the original colors together with the palette itself is smaller than the original representation of colors, otherwise the data size is increased. The second process is integer quantization where integers represented with specific number of bits are quantized to fewer bits. This is used whenever the values that are to be encoded are represented with more bits than are available in the encoded format.

Endpoint Interpolation

A common color quantization method used to describe blocks is endpoint interpolation. It was made popular by the S3TC texture compression format. With endpoint interpolation blocks are encoded by storing two endpoint colors in a chosen color space together with interpolation weights for all texels. When decoding, the texel weights are used to linearly interpolate between the two endpoints. An example representation of endpoints and weights in a two dimensional red and blue color space is shown in figure 3.2, working with RGB or RGBA is analogous. When the colors of a block are distributed along a line, which for example is the case with gradients and one dimensional color spaces, this method gives good results. However, when the colors are spread out, which is more likely the larger the blocks are, compression artifacts are introduced.



(a) *One pair of endpoints in RB-space.*

2/5	3/5	4/5	5/5
1/5	2/5	3/5	4/5
1/5	1/5	2/5	3/5
0/5	1/5	1/5	2/5

(b) *Texel weights for a* 4×4 *block.*

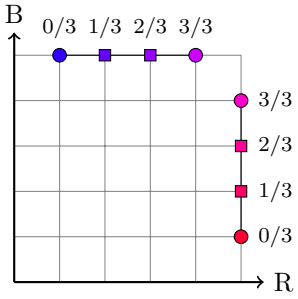
Figure 3.2: *Representation of endpoints and weights.*

Partitions

To handle cases where the colors in a block is not distributed along a line, some formats supports what is referred to as partitioning of each block. For blocks where partitioning is enabled the texels are divided in up to two or more sets, referred to as partitions. A specific set of partitions for a block can be referred to as a partitioning, shape, or pattern. Each partition is encoded using its own pair of endpoints and each weight is encoded with respect to which partition it belongs to. Additionally, a value describing which partition each texel belongs to is also stored together with the block. Figure 3.3 shows an example encoding of a 4×4 block with two partitions.

Luminance Modulation

Another color quantization method for encoding a block is to store a base color and then for each texel store a luminance modifier. This is the method introduced by the PACKMAN format, later extended by iPACKMAN, THUMB, and ETC2. PACKMAN only considers RGB textures and uses 2×4 sized blocks. For each block



(a) *Two endpoint pairs in RB-space.*

0	0	0	0
0	0	0	0
1	1	1	1
1	1	1	1

(b) *Partitioning mask.*

0/3	1/3	2/3	3/3
0/3	1/3	2/3	3/3
0/3	1/3	2/3	3/3
0/3	1/3	2/3	3/3

(c) *Texel weights.*

Figure 3.3: *Representation of endpoints, weights and partitioning for two partitions.*

a base color is stored with four bits per channel (RGB444). For each texel a two bit index into a luminance modulation table is stored. In total 32 bits are used per block, leaving four bits that selects between different modulation tables. The tables are derived from real image data. This method works well when the color points only vary along the luminance axis. If there is chromatic variance in the block it is impossible to represent it well with this format.

Redundant and Invalid Combinations

To maximize the use of the available bits, several techniques have been used in different formats. For example, the encoding order of a pair of endpoints does not matter, the list of interpolation weights can simply be reversed. Thus, the order of the encoded endpoints can be chosen such that the first is smaller than the second or the other way around. When decoding, comparison of the endpoints gives an additional bit which can be defined as a configuration option. This was first used in S3TC but later named the *ordering technique* by Munkberg et al [MAS06].

An different but similar technique was introduced in ETC2 which is based on invalidation rather than redundancy [SP07]. When many configuration options are added to a format, certain combinations of these options can be considered as invalid in that they can not be decoded in any meaningful way. Thus, the bit sequences that represent invalid combinations can be re-defined and given some new and meaningful interpretation, essentially using the available bits more efficiently.

3.4 Adaptive Scalable Texture Compression

Adaptive Scalable Texture Compression is a texture format developed by ARM and was first presented in 2012[Nys+12]. ASTC builds on the ideas introduced by previous texture compression formats while also introducing a few new ideas. Configurability is the main benefit of ASTC, offering a trade off between high visual quality and high compression ratio.

3.4.1 Block sizing

ASTC allows choosing the block size on a per texture basis and also supports both 2D textures and 3D textures. Regardless of the chosen block size, the blocks are always encoded with 128 bits. Thus, choosing a larger block size results in a lower bit rate, that is, fewer bits used per pixel. For video game textures this offers the ability to control compression ratio and visual quality on a per texture basis. The available 2D block sizes are listed in table 3.1 and a similar table is available for 3D blocks in the ASTC specification[EN12].

Block size (pixels)	Bit rate (bits per pixel)	Compression ratio
4×4	8.00	25.00%
5×4	6.40	20.00%
5×5	5.12	16.00%
6×5	4.27	13.33%
6×6	3.56	11.11%
8×5	3.20	10.00%
8×6	2.67	8.33%
10×5	2.56	8.00%
10×6	2.13	6.67%
8×8	2.00	6.25%
10×8	1.60	5.00%
10×10	1.28	4.00%
12×10	1.07	3.33%
12×12	0.89	2.78%

Table 3.1: Block sizes with corresponding bit rates and compression ratios.

3.4.2 Block encoding

ASTC uses the endpoint interpolation method for encoding. The big difference compared to previous usage of endpoint interpolation is that ASTC allows configuring the number of interpolation points and the encoding of color endpoints on a per block basis. Since different blocks benefit from different encoding schemes, allowing to chose the scheme for every block makes it possible to encode with better quality.

Since the number of encoded endpoint values and interpolation points may vary the bit layout of a ASTC block is not trivial. The bits are divided into three categories: configuration bits, endpoint bits and weight bits; and each category may vary in size. The endpoint values and weight values are quantized to fit within the available bits. The bit layout is shown in figure 3.4 and the configuration bits always start at the least significant bit and grows towards the most significant bit. The endpoint bits follows after the configuration bits and grow in the same direction. Weight bits instead start at the most significant bit and are stored in bit-reversed order, growing towards the least significant bit. The layout is designed to make it easier to construct a hardware decoder, which is easier when the different categories starts at known positions.

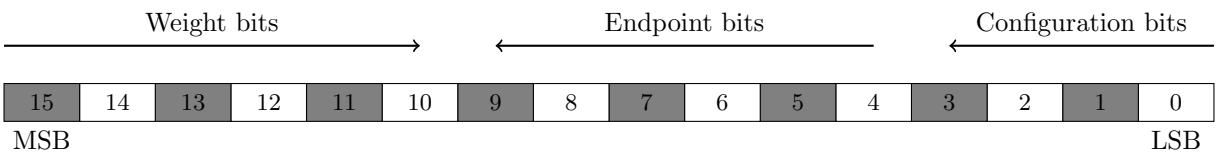


Figure 3.4: *ASTC bit layout for encoded block*

3.4.3 Endpoint Modes

The color spaces have differing component count, thus they require different number of values for the encoded endpoints. For example, for the luminance color space (L), only one component is used and thus only two values are needed for the encoded representation. For RGB there is three channels and thus six encoded values are needed for two endpoints. This enables grey scale blocks to be encoded with higher quality than color blocks, as there are more bits available per endpoint value for the luminance mode, compared to the RGB mode.

Further, ASTC provides three different methods for encoding the two endpoint values. The first is the direct encoding where the two endpoints are encoded independently. The second encoding is a differential encoding where the first endpoint is encoded directly while, the second endpoint is encoded using an offset value (as presented in [SA05]). Thus, the second endpoint is decoded by adding the first endpoint together with the offset. This improves accuracy when two endpoints are close together as quantization otherwise may result in that the endpoints are encoded as the same color in direct mode. The third encoding is a multiplicative encoding which is similar to the differential encoding but uses a scale value instead of an offset value. To decode the second endpoint, the first endpoint is multiplied by the scale value. This is useful when the endpoints vary only along the luminance axis.

	Dynamic Range	Color Space	Encoding Method
1	LDR	L	Direct
2	LDR	L	Differential
3	HDR	L	Large range
4	HDR	L	Small range
5	LDR	LA	Direct
6	LDR	LA	Differential
7	LDR	RGB	Multiplicative
8	HDR	RGB	Multiplicative
9	LDR	RGB	Direct
10	LDR	RGB	Differential
11	LDR	RGB	Multiplicative plus two alpha
12	HDR	RGB	Direct
13	LDR	RGBA	Direct
14	LDR	RGBA	Differential
15	HDR	RGBA	Direct, LDR alpha
16	HDR	RGBA	Direct, HDR alpha

Table 3.2: ASTC endpoint modes. L refers to the luminance color space and LA refers to luminance with alpha.

3.4.4 Partitions

ASTC also supports partitioning as introduced by BC6H and BC7 [Mic09]. In the BC formats each block is encoded according to a specific mode and the mode determines if the block is encoded with partitioning or not. For partitioned blocks a six bit value is used to select one of 64 pre-determined partitions. The partitionings are hard-coded as a lookup table and used both in the encoder and the decoder to tell which partition each texel belongs to.

In ASTC, partitioning is implemented in a slightly different fashion. Firstly, ASTC supports up to four partitions and two configuration bits are used in each block to tell the number of partitions used for that block. Secondly, for each partition count there are 1024 possible partitionings chosen by a ten bit index. Having 1024 partitionings hard-coded in a lookup table would be expensive in a hardware implementation, as it would take up a significant part of the physical size of the circuit. To overcome this problem, the partitionings are actually computed on the fly using a pseudo-random number generator (PRNG). A specialized generator have been developed for this purpose that is much simpler to construct in hardware than a large look-up table. The trick is to seed the PRNG by the partition count, the partition index, and the texel coordinate within the block. This ensures that the same result is given for each run of the generator, essentially using it as a replacement for a look-up table. The returned value is a number telling which partition the texel belongs to. The generator is designed to give good partitionings and also facilitate hardware implementation.

3.4.5 Dual Planes

Dual plane mode allows encoding one color channel independently from the other channels. This enables better visual quality when one channel has low correlation with the other channels, which can happen with for example alpha channels or in non-color textures. The second plane is encoded with an additional weight for every texel which during interpolation is used for the chosen channel.

3.4.6 Bounded Integer Sequence Encoding

ASTC also uses an integer sequence compression trick to further optimize the usage of available bits. With binary encoding the number of values m that can be encoded with n bits is 2^n. However, if m is less than 2^n but more than 2^{n-1}, using binary encoding means that more values can be represented than necessary. In theory, the lower bound of the number of bits is $n = \log_2 m$. Bounded Integer Sequence Encoding (BISE) introduces two additional encodings, *trinary* and *quintary*, that for longer sequences of numbers can approach the theoretical limit for more values of m than regular binary encoding.

Trinary encoding groups integers in groups of five and splits each number into a two-bit trinary value, t, and a binary number b. The trit, t, is taken from the most significant bits and b is the remaining bits. Figure 3.5 shows an example grouping of the 5-bit numbers v_0 to v_4. The trits t_0 to t_4 in the original binary encoding requires ten bits. Since they are trinary and can only take on three different values, they can together only represent $3^5 = 243$ values and thus it is possible to encode them in a binary number with only eight bits. The trits are packed into an eight-bit number and are, together with b_0 to b_4, representable with only 23 bits compared to the original 25 bits. Effectively, the number of bits per trit becomes $\frac{8}{5} = 1.6$ which is less than the original two bits.

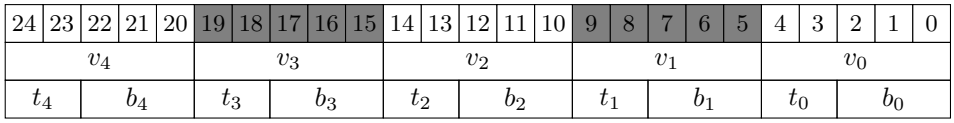
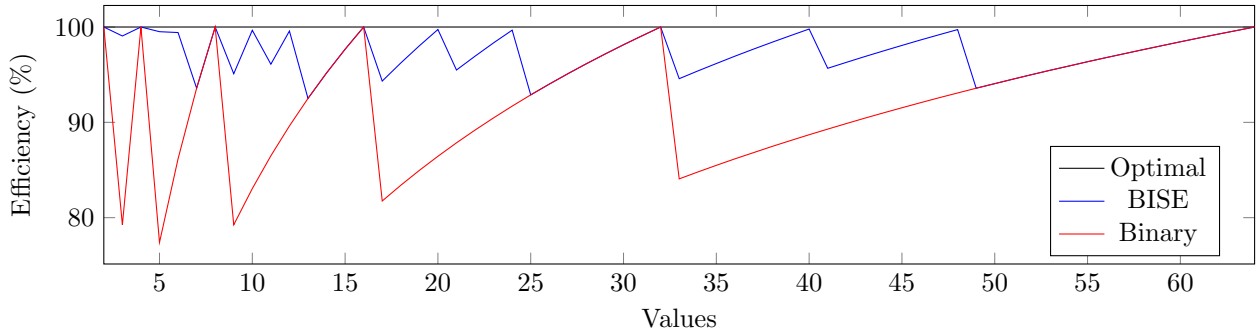


Figure 3.5: *Trinary encoding of 5-bit numbers*

Quintary encoding works analogously to trinary, the difference being that the sequence is divided into groups of three numbers and the three most significant bits are treated as a quint, that is a number that can take on five values. Three quints can represent $5^3 = 125$ values and can thus be encoded with only seven bits. The effective number of bits per quint thus becomes $\frac{7}{3} = 2.33$ which is less than the original three bits.

Figure 3.6 shows the storage efficiency, that is the ratio of the number of requested values to the number of possible values, of binary encoding and BISE. It shows that BISE is optimal two times in between every optimal binary encoding and the worst case is improved significantly.

Figure 3.6: *Efficiency of storage*



3.5 Encoding Algorithms

Encoding a given input image to a specific texture compression format can be seen as an optimization problem where the optimal solution is the encoded configuration which when decoded as closely as possible matches the input image. In the specific instance of endpoint interpolation formats, it is possible to formulate the encoding as an integer programming problem, which is a specific class of optimization problems. We will refer to this as the *endpoint optimization problem*. Integer programming problems are expressed as a set of integer variables that should be optimized with respect to some specified constraints. It has been shown that the endpoint optimization problem is NP-hard [KLM13]. Thus, it is impossible to write an encoder that gives an optimal solution and an approximate method will have to be used instead.

The most commonly used approximations for endpoint interpolation formats will be described in the rest of this section.

3.5.1 Dominant Axis

The first step is to find a dominant axis and then search for endpoints along that axis. This significantly simplifies the search space as we only have to consider points along a line rather than in the entire three dimensional color space. Note that since there always are at least one pair of endpoints that is optimal there is also an axis passing through one of those pairs. There are several methods for finding a dominant axis.

Bounding Box

Waveren presents several methods aimed for performance [Wav06]. The first and quality-wise best method is finding the two color points furthest away from each other with respect to euclidean distance and taking the line between them as dominant axis. This requires that the distance is calculated between all points making it a quadratic algorithm but with quite good quality. Waveren suggests comparing the distance along the luminance axis only for a speed up with some loss in quality. The second and now more commonly used method is to compute the bounding box for the color points, that is the component-wise minimum and maximum, and taking its diagonal as dominant axis. Computing the bounding box is linear in time complexity as we only have to compute the minimum and maximum for each color channel. The problem with the bounding-box method is that the loss in quality is high since the bounding box diagonal does not necessarily correlate with the actual distribution of points sometimes giving entirely wrong results.

Principal Component Analysis

Another method for determining a dominant axis is Principal Component Analysis (PCA). PCA is a statistical method for finding patterns in data with any dimension and is thus suitable for use with 3-dimensional RGB or 4-dimensional RGBA data. Compared to the bounding box computation PCA is much more complicated. Firstly, the input data has to be normalized such that the mean is zero. Secondly, the covariance matrix is computed for the normalized data. Thirdly, the last step is to compute the eigenvectors for the covariance matrix. The eigenvectors describe the data points' distribution and if they are orthogonalized and normalized the data points can be described as linear combinations of the eigenvectors. The eigenvector with the largest absolute eigenvalue (also referred to as first eigenvector) points in the direction of a best fit line. Together with the mean value the first eigenvector can be used to describe a dominant axis.

There is an iterative numerical method for determining the first eigenvector, namely the *power method*. The power method calculates the first eigenvector for a matrix, A. The algorithm is initialized with an initial vector x_0, each iteration is given by equation 3.1.

$$\hat{x}_n = \frac{Ax_{n-1}}{||Ax_{n-1}||} \tag{3.1}$$

The algorithm is usually run until the result does not change more than a predetermined epsilon value. For some matrices the power method converges slowly and it does not work at all when the data is uniform.

3.5.2 Selecting Endpoints and Interpolation Weights

Once a dominant axis has been established the endpoints are searched for along this axis. There are several methods for finding a good solution with trade-offs between speed and quality.

The simplest method for selecting endpoints is to project the color points onto the axis and select the minimum and maximum projections as endpoints. Interpolation weights are then selected by calculating the distance from each texel color point to every interpolated color and the weight giving the shortest distance is chosen. For extra speed the weights can be selected by rounding the projected scale value directly to the nearest quantized weight value, as presented by Uličiansky [Uli10]. This avoids costly comparison between all possible interpolation points but is not mathematically correct and introduces additional compression errors.

Simon Brown introduces a method referred to as *cluster fit* [Bro06a] that reverses the range fit procedure, that is, cluster fit starts with a set of interpolation points and then searches for the optimal pair of endpoints. This is done by finding all possible clusterings of the color points that respects the points' ordering along the dominant axis. Then the least squares method is used to compute the endpoints for each of these

Once a pair of endpoints and a set of interpolations have been chosen the encoding is mostly finished, what is left is quantization to the ranges used in storage and putting the bits together. Some encoders may add an additional optimization step that conducts a local search around the selected endpoints to see if they can be improved, of course at the cost of performance.

3.5.3 Partitioning

For the formats that supports partitioning, the best pattern needs to be selected as well, which requires solving another problem. The *partitioning optimization problem* is a generalization of the endpoint optimization problem. Instead of optimizing for one pair of endpoints, several pairs of endpoints needs to be found which is much more difficult. One solution is to perform the endpoint selection algorithm for each possible partitioning, compute the compression error and chose the pattern with the lowest error. Exhaustively searching all partitions gives the optimal solution but can be very slow, especially for formats such as ASTC that has many partitionings. A faster method is suggested in [Nys+12] which first computes a clustering using the k-means vector quantization algorithm. Each texel is labeled based on the clustering and the resulting pattern is ranked against the available patterns. The available partitions are then searched in order, starting with the best match and the algorithm can exit if the previous partitioning was better.

In [KLM13] it is suggested to use Waveren's bounding box method on each partition instead of doing full principal component analysis. This simplifies and speeds-up the check for each partition and thus may allow checking all partitions. Another completely different method was also presented in a later paper that suggest running an edge-detection algorithm as a pre-processing stage. The partitions are then selected based on the detected edges, or rather the areas in between the edges [KM14].

4 Implementation

In this chapter, a real-time algorithm, henceforth referred to as `astcrt`, for encoding ASTC is presented. The implementation is first outlined in pesudocode and then each part of the encoder is explained using more pseudocode, focusing on the used algorithms rather than the technical details of the implementation. This encoder relies on heuristics and tries to avoid computing results that will be unused later, unlike many of the encoders built previuosly. This means that many of ASTC's features are unused due to the difficulty of finding good heuristics.

4.1 Algorithm Pseudocode

The pseudocode for the encoding of a single block is shown in figure 4.1 and described in detail in the following sections. The ENCODE-BLOCK function is called on each block independently and returns an encoded ASTC block.

```
function ENCODE-BLOCK(texels)
    if IS-SOLID(texels) then
        return ENCODE-VOID-EXTENT(texels)
    else if IS-GREY(texels) then
        endpoints ← FIND-EXTREMES(texels)
        weights ← COMPUTE-WEIGHTS(endpoints, texels)
        return ENCODE-LUMINANCE(endpoints, weights)
    end if
    axis ← PRINCIPAL-COMPONENT-ANALYSIS(texels)
    if SHOULD-PARTITION(axis, texels) then
        clustering ← K-MEANS(texels)
        bitmask ← FIND-PARTITIONING(clustering)
        if bitmask ≠ 0 then
            set_1 ← texels where bitmask = 0
            set_2 ← texels where bitmask = 1
            axis_1 ← PRINCIPAL-COMPONENT-ANALYSIS(set_1)
            axis_2 ← PRINCIPAL-COMPONENT-ANALYSIS(set_2)
            endpoints_1 ← FIND-EXTREMES(axis_1, set_1)
            endpoints_2 ← FIND-EXTREMES(axis_2, set_2)
            weights_1 ← COMPUTE-WEIGHTS(endpoints_1, set_1)
            weights_2 ← COMPUTE-WEIGHTS(endpoints_2, set_2)
            return ENCODE-PARTITIONS(endpoints_1, weights_1, endpoints_2, weights_2)
        end if
    end if
    endpoints ← FIND-EXTREMES(axis, texels)
    weights ← COMPUTE-WEIGHTS(endpoints, texels)
    return ENCODE-RGB(endpoints, weights)
end function
```

Figure 4.1: *Real-Time algorithm pseudo code.*

4.1.1 Fixed Block Size

The first decision for the encoder was which block size should be used. Using the reference encoder [ARM15] a set of web page renders where investigated. The encoder has support for different performance modes ranging from very fast to exhaustive. The very fast mode does very little searching in the configuration space and encodes images quickly while exhaustive mode searches as much of the configuration space as is computationally viable. The exhaustive mode reaches a quality that is as close to the optimal encoding as possible. Thus, if the exhaustive mode does not provide adequate quality it would not be possible to achieve better quality in real-time.

It was concluded that using the exhaustive quality option for block sizes larger than 4×4 did not produce adequate results quality-wise. Thus, it was decided to only use 4×4 blocks in the real-time encoder. As for performance, the fastest mode was considered quick enough such that a specialized encoder would provide significantly better performance than the reference encoder.

4.1.2 Heuristics

The algorithm starts with testing two performance-related heuristics. The first, IS-SOLID, picks out blocks that constist of entirely one color. For such blocks, ASTC has the special void extent mode which can be encoded without any expensive computations making this code-path very fast. This provides an important optimization as solid colors are commonly used on the web, for example in margins around text or as filler. The function is implemented by comparing each color with the first color, returning false if any of them differ.

The second heuristic, IS-GREY, handles blocks where the red, green and blue channels are equal. This is also an important optimization since the problem of finding endpoints becomes one dimensional and we can use the fast minimum and maximum method without loosing much quality. The interpolation weights still has to be computed with respect to the selected endpoints which requires an additional loop over the texels. Greyscale is also common in the web even though black text on white background is not entirely greyscale due to the use of tricks to improve rendering with respect to how computer displays works. The function is implemented by testing the channels of each texel for equality.

4.1.3 Endpoint Selection

The PRINCIPAL-COMPONENT-ANALYSIS and FIND-EXTREMES functions are responsible for selecting the endpoints. Endpoint selection is done using the Principal Component Analysis method as shown in figure 4.2. The eigenvector is computed using the power method with eight iterations as suggested by Castaño [Cas07].

function PRINCIPAL-COMPONENT-ANALYSIS(*texels*)
    mean ← mean color value of *texels*
    A ← *mean* subtracted from *texels*
    B ← covariance matrix of *A*
    return (*mean*, POWER-METHOD(*B*))
end function
function POWER-METHOD(*B*)
    k ← normalization of (1, 3, 2)
    repeat 8 times **do**
        k ← *B* multiplied by *v* and then normalized
    end repeat
    return *k*
end function

Figure 4.2: *Principal Component Analysis pseudo code.*

The color points are projected onto to the principal axis with scalar projections. Then the minimum and maximum projections are taken as endpoints, which is shown in figure 4.3.

function FIND-EXTREMES(*axis*, *texels*)
    $a \leftarrow \infty$
    $b \leftarrow -\infty$
    for each *texel* **in** *texels* **do**
        projection ← *texel* projected onto *axis*
        $a \leftarrow \min(a, projection)$
        $b \leftarrow \max(b, projection)$
    end for
    return (point *a* on *axis*, point *b* on *axis*)
end function

Figure 4.3: *Endpoint selection pseudo code.*

4.1.4 Weight Selection

The COMPUTE-WEIGHTS function calculates the interpolation weights for the texels given an endpoint pair. This is done using vector projection, the vector from the first endpoint to each texel is projected onto the line going through both endpoints. Given the texel color c and the endpoints e_0 and e_1, let $k = e_1 - e_0$ and $m = e_0$. Thus the line starting in m with direction k goes through both endpoints and if we project the vector from m to c onto k we get the parameter for the point on the line closest to the texel color. That value is then extended to the range $[0, 1024]$ and rounded, this computation is shown equation 4.1 where the resulting value is named w.

$$w = \left\lfloor \frac{(c - m) \cdot k}{|k|} \times 1024 \right\rfloor \tag{4.1}$$

The value w is then used as index into a look-up table with the optimal quantized weight values based on euclidean distance.

4.1.5 Partitioning

The partitioning algorithm is made up of the following three steps:

1. Deciding if partitioning should be used or not.
2. Computing an ideal clustering of the texels.
3. Finding a good partitioning that matches the ideal clustering.

The following sections describes each of these steps in more detail.

Determining Partitioning Viability

The first part of the partitioning algorithm decides if it is worth to attempt it, which is done with a heuristic approach implemented by SHOULD-PARTITION. The heuristic is based on the fact that if many texels are far away from the principal axis, the encoding error is going to become large. Thus the euclidean distance can be computed from each color point to the principal axis and if enough of the distances are large enough we use partitioning. These conditions were determined empirically to be:

- A texel is far away if distance is larger than ten.
- A block should be partitioned if three or more texels are far away.

These chosen values are at the point where increasing or decreasing either constant would decrease the overall compression quality. Figure 4.4 shows pseudo code for the partitioning decision heuristic.

```
function SHOULD-PARTITION(texels, endpoints)
    count ← 0
    for each texel in texels do
        p ← texel projected onto line between endpoints
        if distance from p to texels > 10 then
            count ← count + 1
        end if
    end for
    if count ≥ 3 then
        return true
    else
        return false
    end if
end function
```

Figure 4.4: *Partitioning heuristic pseudo code.*

Computing a Clustering

The second part of the partitioning algorithm is computing an ideal clustering from the texels. This is done using the k-means clustering algorithm, also referred to as Lloyd's algorithm, for two clusters. The k-means algorithm starts with one initial center point for each cluster and then in an iterative fashion it assigns each data point to its closest cluster center and then recomputes the cluster centers by taking the average of each clusters assigned data points. For usage in the partitioning algorithm it was empirically determined that starting with the two color points farthest away from each other and then iterating only four times gave an usable result.

```
function K-MEANS(texels)
    centers ← the two points in texels furthest apart
    repeat 4 times do
        clusters ← two empty clusterings
        for each texel in texels do
            d1 ← distance from texel to centers1
            d2 ← distance from texel to centers2
            if d1 < d2 then
                add texel to clusters1
            else
                add texel to clusters2
            end if
        end for
        centers1 ← mean of texels in clusters1
        centers2 ← mean of texels in clusters2
    end repeat
    return clusters
end function
```

Figure 4.5: *k-means clustering pseudo code.*

Finding a Partitioning

To avoid having to search 1024 partition entries, a pre-computed look-up table is used. The table maps every possible partitioning for a 4×4 block, that is $2^{4\times4} = 2^{16}$ entries, to the available partitioning with the smallest edit distance. The edit distance is defined as the number of pixels in the two partitionings that are different. Additionally, the inverted partitioning is also considered and the smallest edit distance is chosen. Partitionings with a large edit distance is not considered as they likely would not improve the result anyway. If there are several available partitionings with a minimum edit distance, only one of them is selected in no particular order.

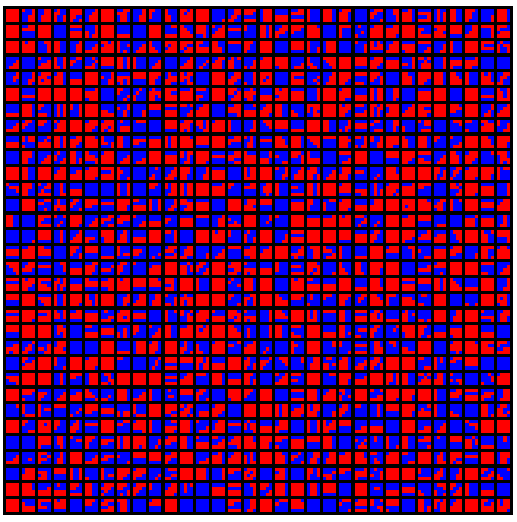


Figure 4.6: *All available 2-set partitionings for* 4×4 *blocks.*

4.1.6 Quantization

Quantization from color values and weight values to their respective storage ranges is done using lookup tables. Tables are again generated by implementing the unquantization steps as described in the ASTC specification [EN12] and then using brute force to reverse the tables.

4.1.7 Bounded Integer Sequence Encoding

The BISE implementation uses a table based method as suggested in the ASTC specification. A decoding table is computed as specified in the ASTC specification [EN12] that maps the packed trit and quint values to unpacked trits and quints. Encoding look-up tables that maps the unpacked trits and quints to the packed representation are computed via brute force.

5 Results

This section presents the quality comparisons and performance benchmarks that have been conducted to confirm that `astcrt` fulfills the goals of good-enough quality at high performance. The first part compares `astcrt` with other established formats and encoders while the second part presents examples of when `astcrt` works and when problems may arise.

5.1 Comparison with Formats and Encoders

This section outlines the performance and quality of the *astcrt* encoder in comparison with other established formats and encoders. A few popular encoders that supported the used testing platform was selected and are presented in table 5.1.

Encoder	Version	Source
`astcenc`	v1.3	`github.com/arm-software/astc-encoder`
`fastc`	commit 319b293	`github.com/mokosha/fastc`
`etcpack`	v4.0.1	`malideveloper.arm.com`
`etcpak`	v0.4	`bitbucket.org/wolfpld/etcpak`

Table 5.1: Tested encoders

The first encoder, `astcenc`, is the reference encoder for ASTC. It is developed by ARM and allows control over quality and encoding performance. Settings are mainly available through five modes: `veryfast`, `fast`, `medium`, `thorough`, and `exhaustive`. The modes control the performance versus quality trade off with `veryfast` providing the highest performance but lowest quality, and `exhaustive` providing the highest quality but the lowest performance.

The second encoder, `fastc`, is an implementation of the algorithm presented in [KLM13] for BC6H and BC7. This encoder aims to be very fast without sacrificing quality. Additionally, it provides control over an endpoint refinement process and have been tested with and without the refinement.

The third encoder, `etcpack`, supports both ETC1 and ETC2 while providing both fast and slow modes for lower and higher quality encoding, but here only the slow mode have been tested. It is developed by ARM and distributed as part of the Mali Texture Compression Tools.

The fourth and last encoder, `etcpak`, is a extremely fast encoder for ETC1 that disregards quality and only focuses on high encoding performance.

Additionally a set of test images is needed and traditionally when it comes to image compression (and texture compression) the Kodak image set [Kod] is used for comparison. The Kodak image set is suitable for comparisons due to its high quality and variety in image features.

Given these encoders and the differences in compression formats, the hypothesis is that `astcrt` will have better quality than the best ETC1 encoder. This is because the ETC1 format is fundamentally flawed since it can not represent several chromatic values in the same block. However, ETC2 together with ASTC and BPTC are similar in that they provide representations for more than one chromatic value per block. The best encoder for each format should thus outperform the real-time encoder quality-wise. For run-time performance it is expected that the `astcrt` provides a competitive speed for the resulting compression quality. Other encoders may be faster, but if they are faster they will most likely provide worse quality. For example `etcpak` is designed to be very fast but it is limited by the quality constraints of ETC1. Also, any S3TC encoder that implements Waveren's fastest methods [Wav06] would be faster than `astcrt`, but the quality would be worse because of the shortcomings of the bounding box method.

5.2 Quality Comparisons

This section presents how `astcrt` performs quality-wise with respect to other formats and encoders. First the numerical method used for measuring encoding error is presented. That is then followed by the measured results and a few excerpts of the encoded images.

5.2.1 Peak Signal-to-Noise Ratio

The metric that have been used is Peak Signal-to-Noise Ratio (PSNR). PSNR refers to the ratio between the maximum power of a signal and the power of the noise that affects the signal. For images, the signal is the sequence of pixels and the maximum power is the largest value we can represent, which in the case of RGB is the color white. The noise is measured as the mean squared error (MSE) of all pixels in the image. The squared error is the same as the squared euclidean distance between each original pixel and the corresponding encoded pixels. Given two pixels p and q, the euclidean distance is given in equation 5.1. Then, given two $m \times n$ RGB images A and B, the MSE is computed as shown in equation 5.2. Finally, the equation for computing the PSNR of the two images are given in equation 5.3.

$$d(p,q) = \sqrt{(p_r - q_r)^2 + (p_g - q_g)^2 + (p_b - q_b)^2} \tag{5.1}$$

$$MSE(A,B) = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} d(A_{ij}, B_{ij})^2 \tag{5.2}$$

$$PSNR(A,B) = 10 \cdot \log_{10} \left(\frac{3 \times 255^2}{MSE(A,B)} \right) \tag{5.3}$$

Some of the tested encoders does compute and present a PSNR value. However, as `astcrt` does not support an alpha channel, care has been taken to ensure that the RGB PSNR value is used in these comparisons, and not the RGBA PSNR value. The RGBA PSNR is slightly higher than the RGB PSNR when the alpha channels are equal since there are four components instead of three in equation 5.3. Thus, to ensure consistent results, the `astcenc` program's comparison function have been used for all PSNR computations which gives the RGB PSNR.

5.2.2 Results

The data is presented in figure 5.1 as a line graph although no linear relationship is present for the images used. It is simply easier to read compared to the more obvious choice of a bar graph. Additionally a few excerpts of the encoded images are shown in figure 5.2 for visual comparison. From the graph we can see that the presented encoder is better than the ETC1 encoder but slightly worse than the fastest mode of the ASTC encoder. Additionally, the encoding method presented by Krajcevski et al [KLM13] gives very good quality and competes with the best ASTC encoder mode. In theory, the ETC2 results should be closer to the results of ASTC and BPTC as it also supports encoding multiple chromatic values. This is probably a problem with the encoder used, but no better encoded was found.

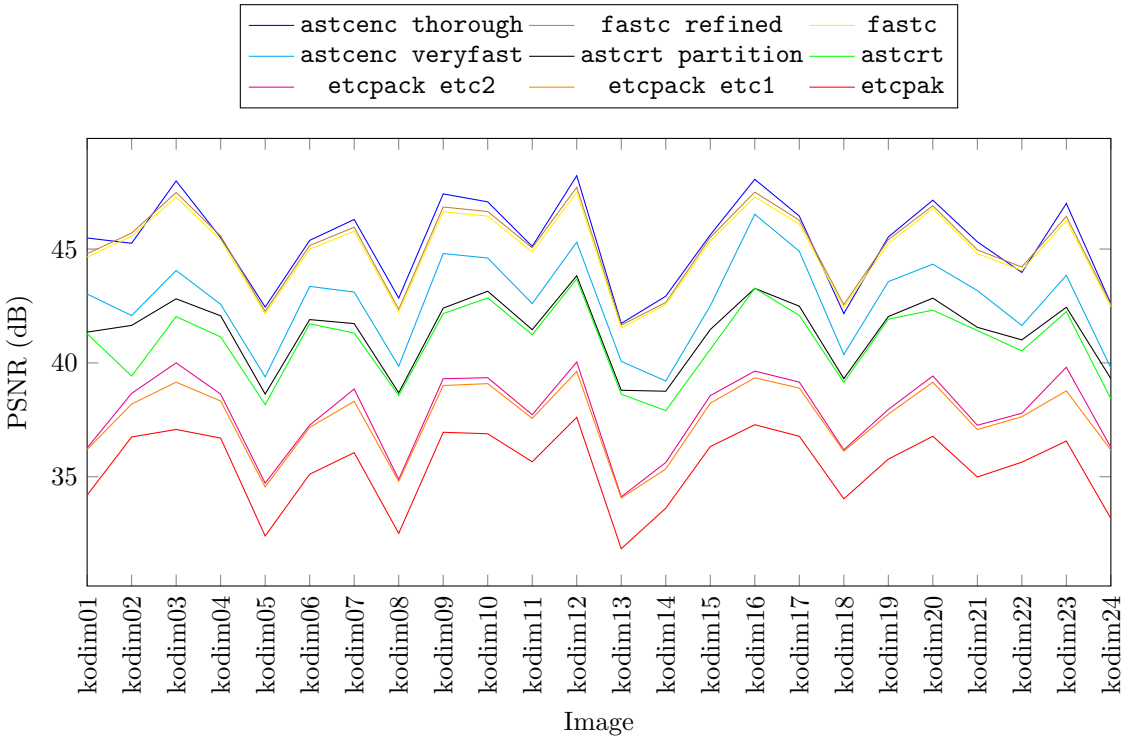


Figure 5.1: *PSNR for the Kodak image set.*

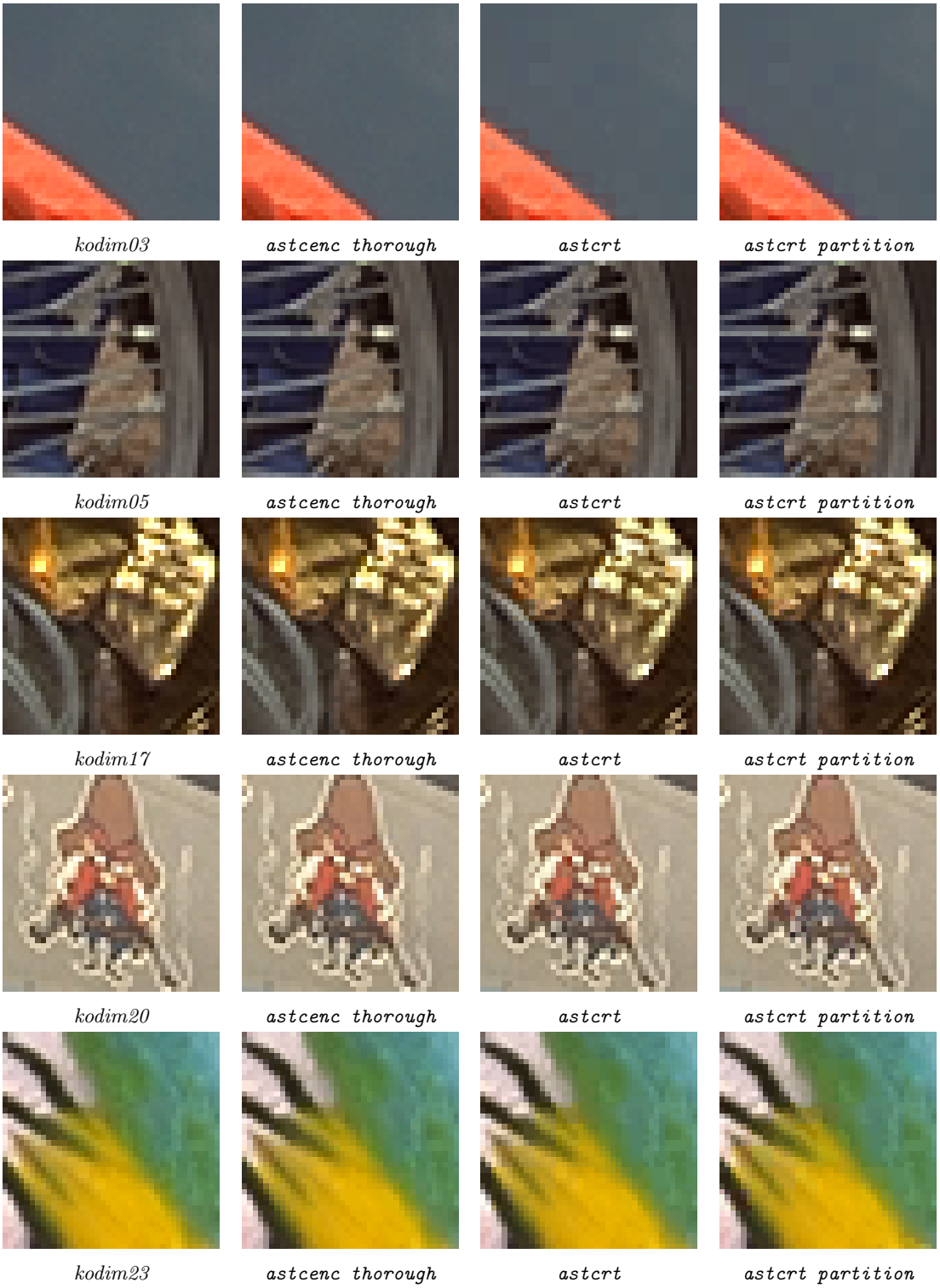


Figure 5.2: *Encoded excerpts of the Kodak image set.*

5.3 Performance Benchmarks

To compare the performance of different encoders, their compression rate is measured. The compression rate is calculated by measuring the time it takes to compress an image and then dividing the pixel count with the measured compression time as in equation 5.4.

$$\text{Compression rate} = \frac{\text{Pixel count}}{\text{Compression time}} \tag{5.4}$$

For fair comparisons, all encoders have been benchmarked in single-threaded mode as the implementation of `astcrt` is not parallelized. Further, when the encoder is implemented in the browser, there is virtually no initialization as the encoder is directly fed the correct input format. Thus, we are only interested in measuring the running time of the encoding algorithms and will avoid counting reading the input file, writing the output and other setup costs. All tested encoders provided the option to print a run time measurement, it is assumed that this measurement excludes the setup time and is thus used in these experiments. Also, each encoder is benchmarked by first running it a few times to warm up the process scheduler and all caches. Then, 50 samples are taken of the compression time and the average is computed to account for any variations that may occur. Note that some of the encoders were sampled only once as they were very slow, taking up to a minute per image. Since we are interested in real-time performance, accurate measurement of encoders that take a minute or more is not necessary.

5.3.1 Hardware

Three hardware platforms have been benchmarked as shown in table 5.2. Both smart phones are from the high-end segment and were chosen to give an upper limit on performance of today. The laptop was included to compare how hardware have improved in general over the years from 2011 to 2015. Both phones are running Android and the laptop is running Linux.

	Laptop	Phone 1	Phone 2
Vendor	Asus	Google	Samsung
Model	U46-SV	Nexus 5X	Galaxy S6
Year	2011	2015	2015
Processor	Intel i5-2410M	Snapdragon 808	Exynos 7 Octa 7420
Max Clock	2.9 GHz	1.8 GHz	2.1 GHz

Table 5.2: Benchmarked hardware

5.3.2 Compiler

It is important that the encoder is built by the same compiler and with the same compiler flags to generate comparable binaries. Thus, since no source code could be found for `etcpack`, only `astcenc`, `etcpak`, `fastc`, and `astcrt` have been benchmarked. For benchmarking on the laptop, version 5.3 of the GNU Compiler Collection was used with the following flags:

```
-O3 -march=native -DNDEBUG
```

The compiler was run on the laptop itself which ensures that the best optimization options for that specific processor is used.

On android, the choice of compiler is more limited as the Android Native Development Kit (NDK) must be used for compiling for the android platform. The Android NDK shipped with GNU Compiler Collection version 4.9 and two different builds were made. The first build is targets version 7 of the ARM architecture with the following flags:

```
-O3 -march=armv7-a
```

The second build enables optimizations for the NEON instruction set, which provides specialized instructions for working with multiple data values at the same time as a form of parallelism. This form of parallelism is called Single Instruction Multiple Data, or SIMD for short. To enable NEON, two flags are added:

```
-O3 -march=armv7-a -mfpu=neon -mfloat-abi=softfp
```

The `-mfpu` flag tells the compiler to enable NEON and the `-mfloat-abi` flag also enables the usage of hardware floating point instructions.

Since none of the other encoders have been designed to be compiled with the Android NDK, it would take some work to run them on Android. Thus only the presented `astcrt` encoder have been benchmarked on Android.

5.3.3 Results

The benchmark results for the Kodak image set is shown in table 5.3 and also on a logarithmic scale in figure 5.3. This data shows that both modes of `astcrt` and also `etcpak` are several orders of magnitude faster than `astcenc` and `fastc`. It also shows that `etcpak` is about three times faster than the partitioning mode of `astcrt`. This is in-line with the expectation that `etcpak` would be the faster of the two encoders.

Encoder	Samples	Rate (pixels/second)
`astcrt`	50	2.42×10^7
`astcrt partition`	50	1.77×10^7
`astcenc thorough`	1	6.88×10^3
`astcenc veryfast`	50	3.69×10^5
`etcpak`	50	6.62×10^7
`fastc refined`	1	5.44×10^3
`fastc`	50	3.48×10^4

Table 5.3: Benchmark results for Intel laptop.

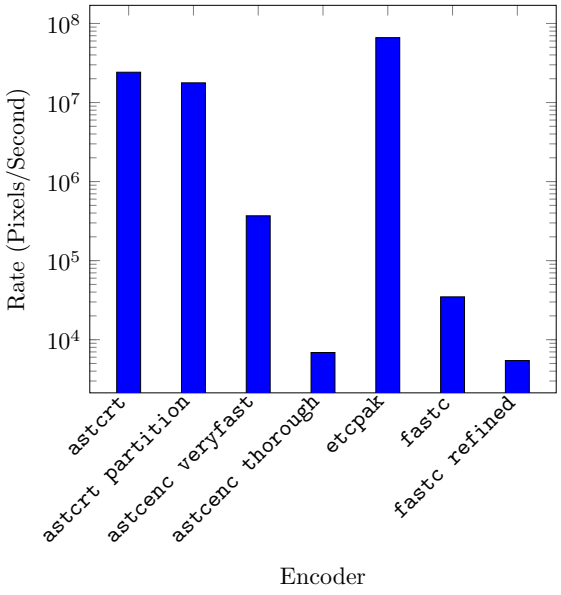


Figure 5.3: *Laptop benchmark compression rate on a logarithmic scale.*

The android benchmarks are also interesting and are shown in table 5.4 and figure 5.4. We can see that NEON and hardware floating point instructions improves the performance a lot, making the encoding almost four times faster. Also, a compression rate of 137 megapixels per second can be considered very good. For perspective, with 137 megapixels per second it would take (on average) 15 milliseconds to encode a Full HD image, which is fast enough to not be noticeable by a user.

Encoder	NEON	Rate (pixels/second)
`astcrt`	No	3.33×10^7
`astcrt`	Yes	1.65×10^8
`astcrt partition`	No	2.23×10^7
`astcrt partition`	Yes	1.37×10^8

(a) Benchmark results for Google Nexus 5X.

Encoder	NEON	Rate (pixels/second)
`astcrt`	No	3.87×10^7
`astcrt`	Yes	2.04×10^8
`astcrt partition`	No	2.57×10^7
`astcrt partition`	Yes	1.68×10^8

(b) Benchmark results for Samsung Galaxy S6.

Table 5.4: Android benchmarks results.

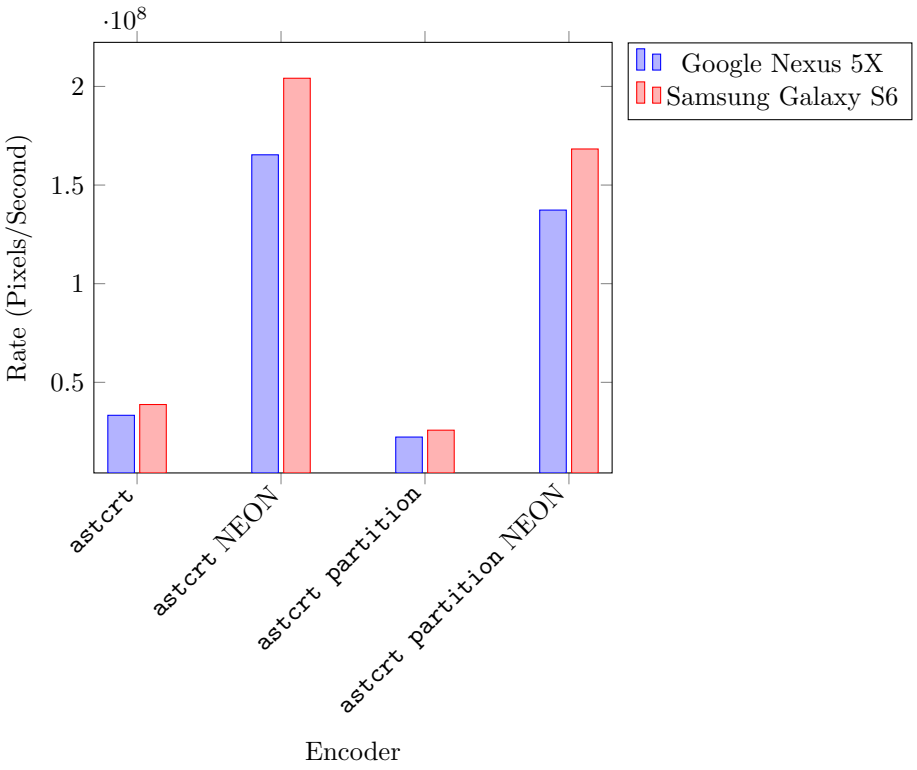


Figure 5.4: *Graph of android bench marks (linear scale).*

5.4 Problematic Blocks

While PSNR can help us gauge how `astcrt` performs in comparison to other encoders, it is still only a comparisons of *averages*. If a single block has a very large error in an image with an otherwise low average error, it is easy for a human viewer to spot the bad block. Though, the PSNR will not be affected much, if at all, since the average error is still low. For this reason, even though the partitioning improves the PSNR only slightly, it may be large improvement in quality for a human viewer. To illustrate the improvement partitioning brings, a few blocks have been chosen and are presented together with their compressed version. The example blocks can be seen in figure 5.5.

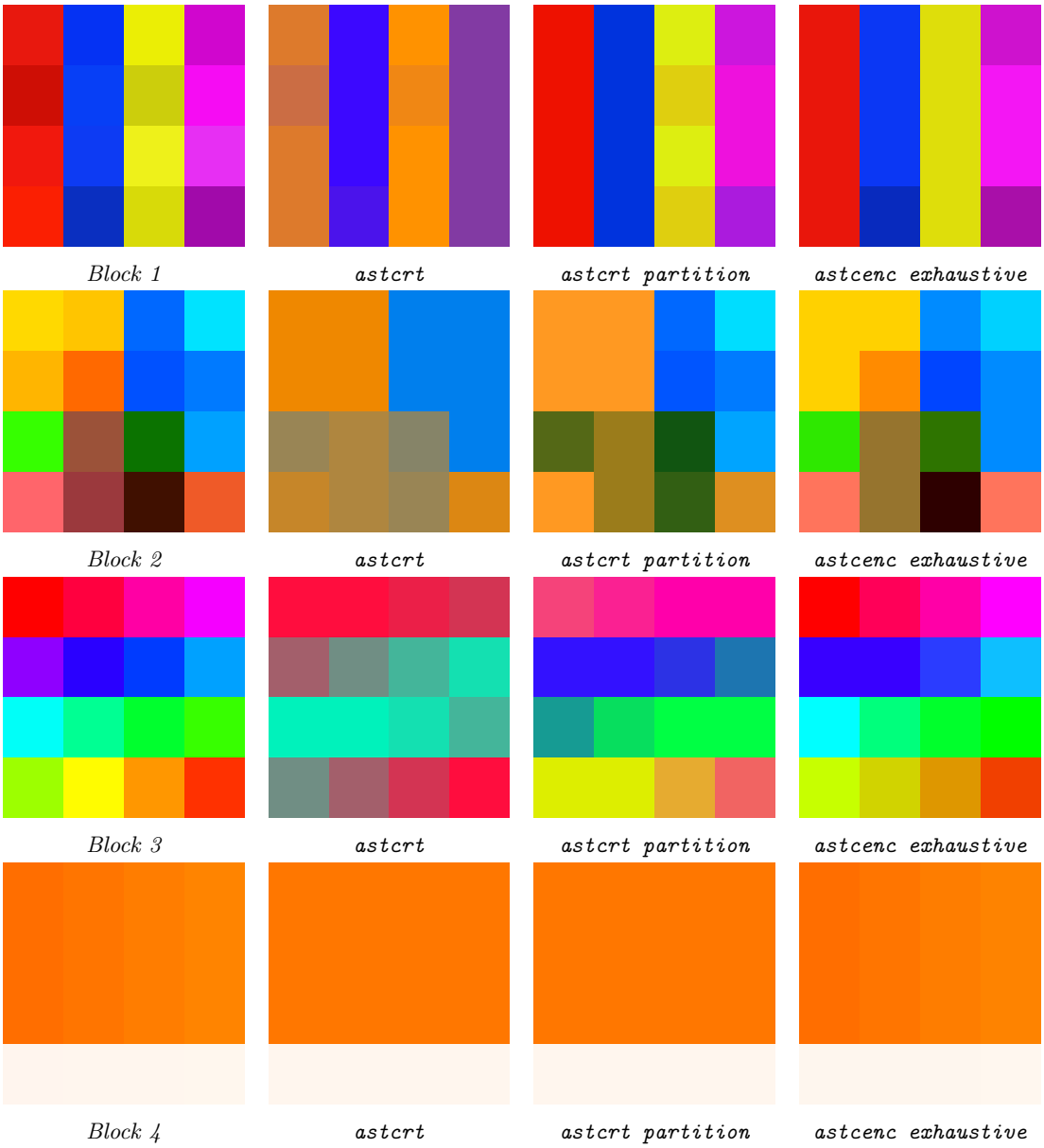
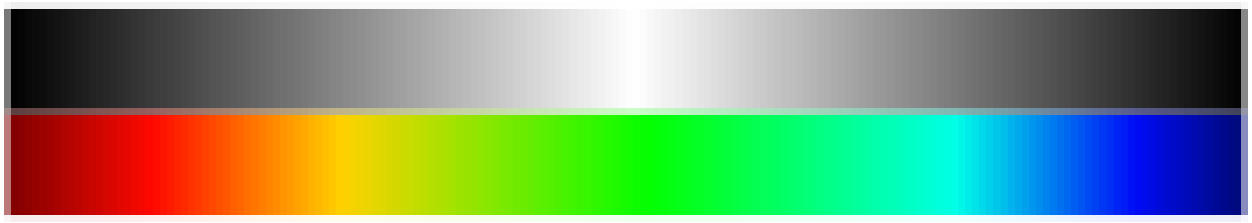
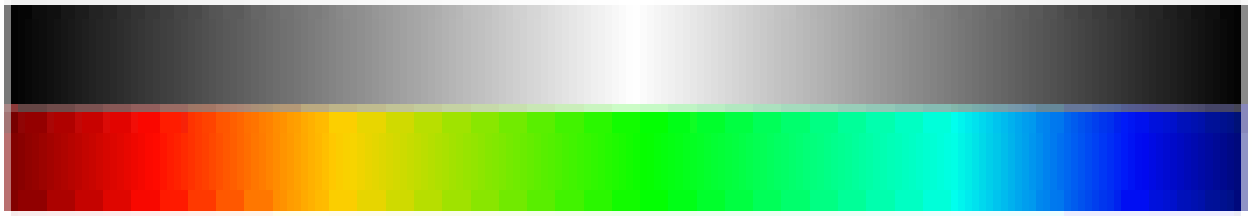


Figure 5.5: *Example blocks where partitioning improves the encoding error.*

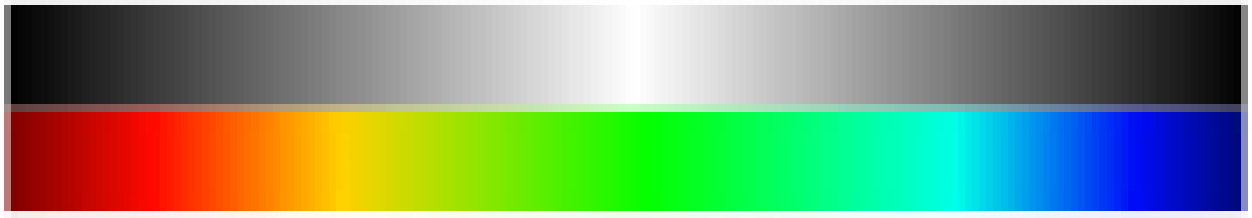
It is obvious that partitioning helps for the first three blocks and that `astcenc`, as expected, gives the best results in all cases. For the fourth block, it is difficult to spot the difference because of the low contrast. What has happened is that `astcrt` has encoded the original gradient as a solid color, which most likely happens due to quantization. The original block is quantized into a range that represents the slightly different shades of orange with the same shade of orange. It may seem like a small problem but if you put the block into the image context, as in figure 5.6, it is much easier to spot the compression artifacts.



(a) *Original image*



(b) `astcrt partition`



(c) `astcenc exhaustive`

Figure 5.6: *Block 4 context.*

6 Discussion

In this section the presented results and the presented implementation are discussed in the context of real-time texture compression.

6.1 Quality Evaluation

Computationally evaluating the quality of compressed images is difficult. In addition to what is noted in the results, PSNR is only an approximation of human perception of encoding errors. For instance, a block with low compression error may look worse than a block with high compression error. The fourth example in figure 5.5 has a small error since there is only a small amount of variation among the original colors. However, it is still possible to spot the compression artifacts in the image. But for blocks in very noisy images, high compression error could be present without any easily spotted artifacts. This is because noise introduced by compression can be indistinguishable from the original noise in the image. It would have been interesting to use a more modern image comparison methods such as Multiscale Structural Similarity [WSB03] to see if the results would be different. This was unfortunately not done due to time constraints.

Also, when it comes to visual quality, an important consideration is that for displays with higher pixel density, compression artifacts are more difficult to discern. This is because each visual feature (for example a character symbol) will be smaller in visual size and thus it is harder to see small differences. A second reason is that with higher pixel density, visual features are usually rendered over a larger pixel area to maintain their visual size. This is important to for example make text large enough to be readable. In turn, this means that more blocks will have higher color correlation between their pixel and thus compress better with endpoint interpolation. But that could also mean that it might be possible to use a larger block size for a lower compression ratio.

6.2 Performance

While the performance of `astcrt` when optimized for NEON is very good, the common consensus is that hand optimization still beats compiler optimizations. Thus, performance could possibly be improved further. Also, it would have been interesting to compare `astcrt` with Intel's SIMD optimized texture compressor[Int15]. Since it was only distributed for Windows, testing it on a Linux machine was not possible.

6.3 Unused ASTC features

Many of the available features in ASTC are unused by `astcrt`. This is due to not having enough time to implement and test good heuristics for the those features. For example, the fourth block in figure 5.5 may benefit from the use of the differential endpoint encoding mode. It would also have been interesting to see how using additional modes would affect the overall quality and performance.

6.4 Power Method

Since the power method is only run eight times, and not to the point of convergence, this is a source of error. It would be interesting to investigate how much this source of error contributes to the overall result. Also, when the algorithm does not converge, the choice of initial vector matters. Different initial vectors could be tested on real images, such as the Kodak image set used in the results, and the vector giving the smallest error could be chosen.

6.5 Partitioning

The proposed partition selection method actually improves quality. However, it is not perfect and a few improvements could be investigated. Firstly, the usage of k-means does not optimize the two endpoint pairs, only the distance from each point to the clusters' centers. Thus, the resulting partitions computed from the

clustering may not be the best with respect to endpoint interpolation. One example is blocks with four or more distinct chromatic values. What could happen is that one cluster is created for one of the chromatic values, and then the other three is lumped together in the second cluster. Thus, the endpoint interpolation has to cover three chromatic values in that case. The ideal solution for this case would be to put two chromatic values in each cluster. Modifying the k-means algorithm to find better partitionings in the general case, or using a different clustering algorithm, could be investigated for better compression quality.

Secondly, the scoring method used for computing the look-up table is not ideal. Since the edit distance is primitive, partitionings containing widely different texels are considered equal. Figure 6.1 shows three example partitioning masks. The first is considered the ideal partitioning and the other two are the available partitionings. Both of the available partitionings only have a single difference from the ideal partitioning, meaning that they have the same edit distance and are considered equal by the partitioning choosing heuristic. However, the chosen partitioning can give very different results depending on the block. Test encoding of the found partitionings is an obvious way to mitigate this problem but would be very costly performance-wise. Another fix could be to change the distance metric such that each cell of each partitions is geometrically close to the other cells of that partition. That would mean that eventual compression errors are gathered together which may look better.

1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0

(a) *Ideal partitioning*

1	1	1	0
1	1	0	0
1	1	0	0
1	1	0	0

(b) *Available partitioning*

1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	1

(c) *Available partitioning*

Figure 6.1: *Example partitioning masks.*

Thirdly, the endpoints are computed from the actual chosen partitioning rather than the ideal computed partitioning. It could be possible to get better results by using the k-means clustering to compute the endpoints, and then use the actual partitioning only for texel weight computation. If we assume that the k-means clustering represents a good partitioning, introducing color points from a different part of the block to the endpoint computation stage will skew the result. Thus, using the ideal clustering instead would avoid the skew. However, the encoded color for the additional texel will come from an endpoint pair which it did not contribute to, meaning that it might stand out among its surrounding pixels. That may look worse than a slight skew of all colors in the partitions.

Fourthly, a general limitation is that this method is only feasible for two partitions and smaller block sizes. For 4×4 blocks with two partitions the table becomes 128 kibibytes in size which in general can be considered small. If the block size is increased to 6×6 the table becomes 128 gibibytes instead, which is too large for the general purpose computers today. Three partitions would also mean that the table becomes large, around 82 mebibytes. Thus, using a pre-computed look-up table for other than 4×4 blocks can be viewed as intractable.

6.6 Using another format

ASTC, BPTC, and ETC2 have in their publications shown that they are capable of encoding with high enough PSNR. Given the similarities between ASTC and BPTC, it should be possible to implement an encoder that is as fast as `astcrt` for BPTC as well. ETC2 is different enough to mandate a different encoding method.

Another consideration for the format is the flexibility of ASTC. This flexibility is not easy to utilize efficiently as previously explained. Even if `astcrt` were to be improved to utilize more of ASTC's features, many configurations would still be unused. For instance, it would suffice to use only one bit to indicate that there are one or two partitions in the block, rather than two bits. Another example is the configuration bits used to state how many interpolation weights there are in the block, that number is always fixed to sixteen in `astcrt`. There are many more examples and this essentially means that bits are wasted. These bits could have been used to increase the accuracy of the encoded endpoints and weights. Thus, a specialized format could be created with only the necessary configuration options, allowing more bits to be used for specifying endpoint

colors and weight values. This is something that may happen in the future as improving memory utilization and battery usage is very important for hand held devices.

7 Conclusion

Going back to the problem presented in the introduction: saving memory by compressing what the web browser renders. Using texture compression enables significant memory savings at the cost of processor time for encoding, and also quality loss due to the lossy nature of the encoding. It has been shown that the presented ASTC encoder almost provides quality as good as the fastest mode of the reference encoder, and at a much higher compression rate. While the quality is not as good as the targeted reference encoder, a few possible improvements have been suggested which may bring the quality closer to the goal and eliminate any significant compression artifacts. The performance benchmarks also shows that the compiler is good at utilizing the NEON instruction set for the given workload, and that the presented encoder can be applicable for real-time usage. Unfortunately, the low-end hardware that actually need the memory savings the most may be too slow for real-time texture compression. While the choice of hardware may matter, it is still an interesting result for future research and development of using compression to decrease the amount of memory used for image data in general.

References

[AHH08] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. CRC Press, 2008.

[AMD08] AMD. *The Compressonator*. 2008. URL: http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/the-compressonator.

[ARB11] O. A.R.B. *ARB_texture_compression_bptc*. 2011. URL: https://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt.

[ARM15] ARM. *ASTC-encoder*. 2015. URL: https://github.com/ARM-software/astc-encoder.

[AS03] T. Akenine-Möller and J. Ström. "Graphics for the masses: a hardware rasterization architecture for mobile phones". In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 801–808.

[BAC96] A. C. Beers, M. Agrawala, and N. Chaddha. "Rendering from compressed textures". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 373–378.

[Bro06a] S. Brown. *DXT Compression Techniques*. 2006. URL: http://www.sjbrown.co.uk/2006/01/19/dxt-compression-techniques.

[Bro06b] S. Brown. *libsquish*. 2006. URL: https://code.google.com/p/libsquish.

[Cam+86] G. Campbell et al. "Two bit/pixel full color encoding". In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 215–223.

[Cas07] I. Castaño. "High quality dxt compression using cuda". In: *NVIDIA Developer Network* (2007).

[Cas10] I. Castaño. *NVIDIA Texture Tools*. 2010. URL: http://code.google.com/p/nvidia-texture-tools.

[DM79] E. J. Delp and O. R. Mitchell. "Image compression using block truncation coding". In: *Communications, IEEE Transactions on* 27.9 (1979), pp. 1335–1342.

[EN12] S. Ellis and J. Nystad. *ASTC Specification*. 1.0. ARM Ltd. July 2012. URL: https://github.com/ARM-software/astc-encoder/raw/5d545016573317fefd6cebd77694d6123cd82b94/Documentation/ASTC%20Specification%201.0.pdf.

[INH99] K. I. Iourcha, K. S. Nayak, and Z. Hong. *System and method for fixed-rate block-based image compression with inferred pixel values*. US Patent 5,956,431. Sept. 1999.

[Int15] Intel. *Fast ISPC Texture Compressor*. 2015. URL: https://software.intel.com/en-us/articles/fast-ispc-texture-compressor-update.

[KLM13] P. Krajcevski, A. Lake, and D. Manocha. "FasTC: accelerated fixed-rate texture encoding". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM. 2013, pp. 137–144.

[KM14] P. Krajcevski and D. Manocha. "SegTC: Fast Texture Compression using Image Segmentation". In: *Eurographics Association, Lyon, France, I. Wald and J. Ragan-Kelley, Eds* (2014), pp. 71–77.

[Kni+96] G. Knittel et al. "Hardware for superior texture performance". In: *Computers & Graphics* 20.4 (1996), pp. 475–481.

[Kod] Kodak. *Kodak lossless true color image set*. URL: http://r0k.us/graphics/kodak.

[MAS06] J. Munkberg, T. Akenine-Möller, and J. Ström. "High quality normal map compression". In: *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware: Proceedings of the 21 st ACM SIGGRAPH/Eurographics symposium on Graphics hardware: Vienna, Austria*. Vol. 3. 04. 2006, pp. 95–102.

[Mic09] Microsoft. *Texture Block Compression in Direct3D 11*. 2009. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/hh308955(v=vs.85).aspx.

[Nys+12] J. Nystad et al. "Adaptive scalable texture compression". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association. 2012, pp. 105–114.

[PS05] M. Pettersson and J. Ström. "Texture Compression: THUMB—Two Hues Using Modified Brightness". In: *Proceedings of Sigrad, Lund* (2005), pp. 7–12.

[SA04] J. Ström and T. Akenine-Möller. "PACKMAN: texture compression for mobile phones". In: *ACM SIGGRAPH 2004 Sketches*. ACM. 2004, p. 66.

[SA05] J. Ström and T. Akenine-Möller. "iPACKMAN: High-quality, low-complexity texture compression for mobile phones". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM. 2005, pp. 63–70.

[SP07] J. Ström and M. Pettersson. "ETC 2: texture compression using invalid combinations". In: *Graphics Hardware*. 2007, pp. 49–54.

[Uli10] P. Uličiansky. *Extreme DXT Compression*. Cauldron, Ltd, 2010. URL: `http://www.cauldron.sk/files/extreme_dxt_compression.pdf`.

[Wav06] J. M. P. v. Waveren. "Real-time DXT compression". In: *Intel Software Network* (2006).

[WC07] J. M. P. v. Waveren and I. Castaño. "Real-time YCoCg-DXT compression". In: *nVidia Report, Sep* 14 (2007).

[WSB03] Z. Wang, E. P. Simoncelli, and A. C. Bovik. "Multiscale structural similarity for image quality assessment". In: *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*. Vol. 2. Ieee. 2003, pp. 1398–1402.