Compressing Dynamically Generated Textures on the GPU

Oskar Alexanderson Christoffer Gurell

Thesis for a diploma in computer science, 20 credit points,
Department of Computer Science, Faculty of Science, Lund University

Examensarbete för 20 p, Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet

Compressing Dynamically Generated Textures on the GPU

Abstract

In the area of computer graphics, texture mapping is often used to enhance the appearance of rendered objects. To fit more data into the graphics cards memory, and to speed up rendering, it is common to compress the images used for texture mapping. The process of compressing an image, with currently available tools, takes several seconds. This makes it impossible to use the benefits of compressed textures if the textures are somehow dynamically generated by the application for immediate use (e.g. dynamic environment maps or hardware accelerated window managers).

This thesis presents a method that makes it possible to very rapidly compress textures to the S3TC format. This is achieved by simplifying an available compression algorithm with speed in mind, and then adapting this simplified algorithm to take advantage of the incredible computational power of modern GPUs. The result is a compressor that compresses textures hundreds of times faster than available tools while maintaining comparable image quality.

Att komprimera dynamiskt genererade texturer på GPUn

Sammanfattning

Inom området datorgrafik används ofta texturmappning för att förbättra utseendet hos renderade objekt. För att få plats med mer data i grafikkortets minne och för att öka renderingshastigheten, är det vanligt att bilderna som används för texturmappningen komprimeras. Komprimeringen av en bild tar, med befintliga verktyg, flera sekunder. Detta gör det omöjligt att andvända fördelarna med komprimerade texturer om dessa texturer på något sätt dynamiskt genereras av applikationen för omedelbar användning (t.ex. dynamiska environment maps eller hårdvaruaccelererade fönsterhanterare).

Detta examensarbete presenterar en metod som gör det möjligt att väldigt snabbt komprimera texturer till S3TC-formatet. Detta möjligörs genom att en befintlig komprimeringsalgoritm förenklas med hastighet i åtanke, och sedan anpassas denna förenklade algoritm så att den enorma beräkningskraften i moderna GPUer utnyttjas. Resultatet är en komprimerare som komprimerar texturer flera hundra gånger snabbare än befintliga verktyg samtidigt som den bibehåller en jämförbar bildkvalitet.

Contents

1 Introduction **5**
1.1 Problem statement 6
1.2 Purpose 6

2 Background **7**
2.1 GPU architecture 7
2.2 Variable vs fixed rate image compression 9
2.3 S3 Texture Compression 9

3 Real-time texture compression **13**
3.1 Simplifying S3TC for speed 13
3.2 Algorithm in detail 14
3.3 Achieving real-time performance 17

4 Results **20**
4.1 Compression speed 20
4.2 Compressed image quality 21
4.3 Bandwidth savings 22

5 Discussion **23**

Bibliography **25**

A Visual image comparison **27**
A.1 Lena 28
A.2 Mandrill 29
A.3 Lorikeet 30

Acknowledgements

We would like to thank Tomas Akenine-Möller for, despite having a hectic schedule, taking the time to help and guide us through the completion of this thesis. We would also like to thank Jacob Munkberg, Jon Hasselgren and Petrik Clarberg for their feedback whenever it was needed.

Oskar Alexanderson and Christoffer Gurell

Lund, May 2006

Chapter 1

Introduction

A fundamental operation in real-time computer graphics is the drawing of a triangle. A triangle is represented by three vertices and several triangles define a mesh, representing a 3D object. The mesh is supplied by the user application to the graphics card, which then projects the vertices onto the screen and rasterizes the triangles. To enhance the coloring of triangles beyond that of a single color, textures are used. Textures are bitmap images stored in the memory of the graphics card. The user application supplies a texture coordinate for each vertex and the textures are then mapped onto the triangles using perspective-correct interpolation.

By compressing textures we gain a number of benefits [6, 2, 8]:

- Texture compression allows an application to use more textures:
  By storing compressed textures, an application can fit more textures into the limited memory of the graphics card, thus allowing more complex texturing.
- Texture compression can increase overall rendering quality:
  Because texture compression is lossy (Section 2.2), compressing a texture will decrease it's image quality. However, when using compressed textures, higher resolution images and more mipmap[1] levels can fit into the same amount of memory. This will increase overall rendering quality, and generally, this increase is more significant than the decrease due to lossy compression.
- Texture compression will reduce bandwidth usage:
  When mapping textures onto triangles, color values are read from the graphics cards memory. These color values are cached in a small on-chip memory inside the GPU itself. By using compressed textures, less data has to be read from memory, which will result in reduced memory bandwidth usage.
- Texture compression can increase performance:

[1]Mipmapping is beyond the scope of this thesis. See Williams 1983 [9].

Storing compressed color values in the cache allows more data to fit into the same amount of on-chip cache memory. This will increase cache hit rate and, since reading from the cache is extremely fast, performance will increase.

- Texture compression can reduce power consumption on mobile devices:
  On mobile devices, a memory access is costly not only in terms of time, but also in terms of battery power. Reducing memory bandwidth usage, e.g. through using compressed textures, is a good technique for reducing overall power consumption [1].

Modern graphics cards have the ability to use compressed textures in a number of different formats. They do not, however, have the ability to compress textures. Compression is done off-line using software tools such as ATI's Compressonator or NVIDIA's DDS Utilities, and compression times of several seconds are common. The textures are then uploaded in compressed form to the graphics card's memory, to be used when rendering.

1.1 Problem statement

The long compression times of traditional methods make these unusable when the textures are dynamically updated for immediate use. Examples include composite window managers (Mac OS X, UI engines on mobile devices, etc), as well as dynamically generated environment maps (dynamic reflections in games etc). To our knowledge, there is currently no method for doing real-time texture compression.

1.2 Purpose

In this thesis, we intend to present a method for doing real-time on-the-fly texture compression using the speed and programmability of modern graphics cards. The goal is to be able to produce compressed textures that are comparable in quality to those produced by current off-line texture compression tools. We also intend to show the memory bandwidth savings that are possible by compressing dynamically generated textures, and finally, indicate how texture casting and compression could be included as part of the OpenGL API.

Chapter 2

Background

In this chapter, we explain the properties of the modern graphics processing units (GPU), we use to accelerate texture compression. We describe the properties of texture compression in general, and the S3 texture compression (S3TC) format [6] in detail. The S3TC format is the target for our real-time texture compression algorithm.

2.1 GPU architecture

As previously mentioned, modern graphics cards have the ability to very rapidly draw triangles on the screen. An application provides triangles in the form of vertices to the graphics card, through APIs such as OpenGL or DirectX. The application may also supply additional data per vertex and transformation matrices. Figure 2.1 shows a schematic image of how the GPU of a modern graphics card transform vertices into colored triangles on screen. For a detailed description of a GPU, see Kilgariff and Fernando's article [5].

At the top, the vertex data stream from the application enters the vertex shader unit. This unit is responsible for transforming (rotating, translating, etc) and projecting vertices. On modern graphics cards this unit is programmable, meaning that an application can supply a vertex program that replaces the default functionality. By using such a program, it is possible to have the vertex shader unit move vertices, create animations or alter vertex data, such as vertex color or texture coordinates. Since the processing of a vertex is independent of other vertices, throughput can be significantly increased by adding several vertex shader units that operate in parallel.

The transformed vertex data stream is fed into the triangle setup and rasterization unit of the GPU. This non-programmable unit basically builds triangles from vertices and decide which pixels on screen that are to be colored for each triangle. Vertex data is interpolated over each triangle so that each of the triangle's pixels

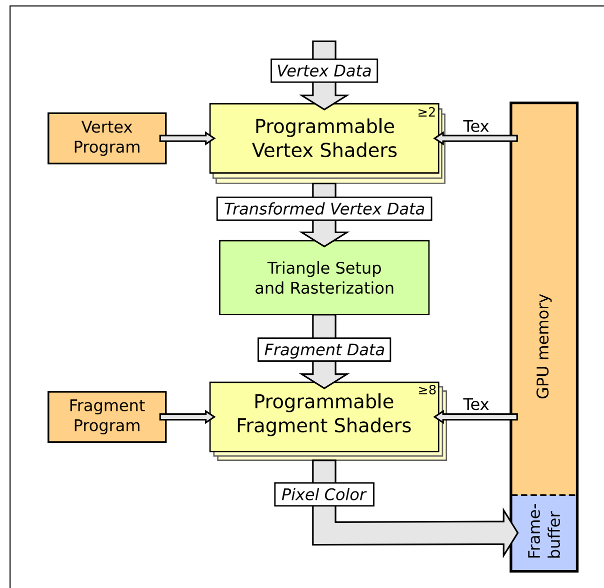


Figure 2.1: Simplified diagram of a typical GPU architecture

is assigned data corresponding to its position. The output from the triangle setup and rasterization unit is called fragment data. Fragment data is information about one pixel that is to be colored and interpolated vertex data for this pixel.

The fragment data stream is sent to the fragment shader unit. This unit is responsible for the final coloring of the pixels. Since fragment data can contain a lot of information (interpolated surface normals, texture coordinates, etc), advanced coloring such as texture mapping and lighting can be calculated in this unit. The fragment shader unit is, like the vertex shader unit, programmable, allowing an application to replace the simple default pixel coloring with a custom fragment program. Since the coloring of a single pixel is independent of the coloring of other pixels, the speed of this unit can be significantly improved by adding several parallel fragment shader units.

After the color of each pixel has been calculated, these values are stored in a memory area referred to as a framebuffer, which can be displayed on screen.

In both the vertex shader and fragment shader units, it is possible to do texture lookups. This, combined with the ability to use a texture as the framebuffer (render to texture) makes it possible to do advanced multi-pass rendering, where the output of one pass is used as input in later passes.

The application supplies vertex and fragment shader programs, usually written in a high level programming language such as GLSL, Cg [7] or HLSL. These languages resemble C but are highly customized for GPU programming. They supply a number of very useful features (vector and matrix data types and operations on these, etc), but they also have a number of limitations. One limitation that this thesis had to work around was the fact that GPUs only operate on floating point

numbers. Even though the high level programming languages supply integer data types, there is no way to do bit manipulations such as AND, OR, SHIFT, etc. It is also impossible to do random memory writes from within a fragment shader program, because the only possible output destinations are at fixed positions in the framebuffers (color, depth and stencil buffers).

Due to the fact that many parts of the GPU pipeline can be parallelized, it is fairly easy to construct faster GPUs by adding more parallel units. This has resulted in modern GPUs having incredible number crunching power.[1]

2.2 Variable vs fixed rate image compression

There are numerous popular compression formats for general image compression, such as JPEG and PNG, most of which use variable bit rates. This means that the average bit-depth per pixel vary over the image, being greater in complex areas. Variable bit rate provides for high image quality or even lossless compression, while keeping the compressed size down. There is, however, no way of directly knowing where in the compressed data the color of a given pixel is stored. The only way to find the color of a given pixel, is to decompress the image from the start and continue until the wanted pixel is reached. Thus, if variable bit rate compression was used for textures, the time and bandwidth needed for a single texture lookup would be proportional to the position of the sought after pixel. This would make texture lookups extremely costly, cancelling out the benefits of using compressed textures. What is needed is a format that directly allows the decompressor to know where in the compressed data stream the pixel value is stored. One solution is fixed rate compression.

2.3 S3 Texture Compression

For this thesis the S3 Texture Compression (S3TC) format (first presented in 1996 by Knittel et al [6]), specifically the DXT1 version [3], was chosen as the output format for the real-time texture compressor. S3TC is a fairly simple block-based image compression format originally designed to be used for texture compression. This format is fixed rate, and the logic needed to get a pixel color value from an S3TC compressed texture is very simple. Texture lookups from S3TC textures are supported by modern graphics cards, through both OpenGL and DirectX, making these textures easy to use in applications.

[1] The computational power of modern top-of-the-line desktop CPUs measure in tens of Gflops while the power of modern top-of-the-line desktop GPUs measure in hundreds of Gflops.

S3TC Block structure

An image compressed in the S3TC format is stored as blocks of 4×4 pixels, using 64 bits per block. In the first 32 bits of a block, two 16-bit base colors are stored using 5, 6 and 5 bits for the red, green and blue components respectively. During decompression, two additional colors are derived from these base colors, giving a local palette of 4 colors per block. The two last colors are derived differently, depending on the order in which the two base colors are stored:

- If the 16-bit unsigned integer value representing the first color is larger than that representing the second, the last two colors are derived by interpolating two colors evenly spaced between the base colors. In this paper, a block encoded this way will be referred to as a type-1 block.
- If the 16-bit unsigned integer value representing the first color is less than or equal to that representing the second, the last two colors are derived by interpolating one color midway between the base colors and setting the other as black. In this paper, a block encoded this way will be referred to as a type-2 block.

In the last 32 bits of a block, each of the 16 pixels is stored as a two bit index, referencing one of the four colors in the local palette. See Figure 2.2 for an illustration of the S3TC block structure.

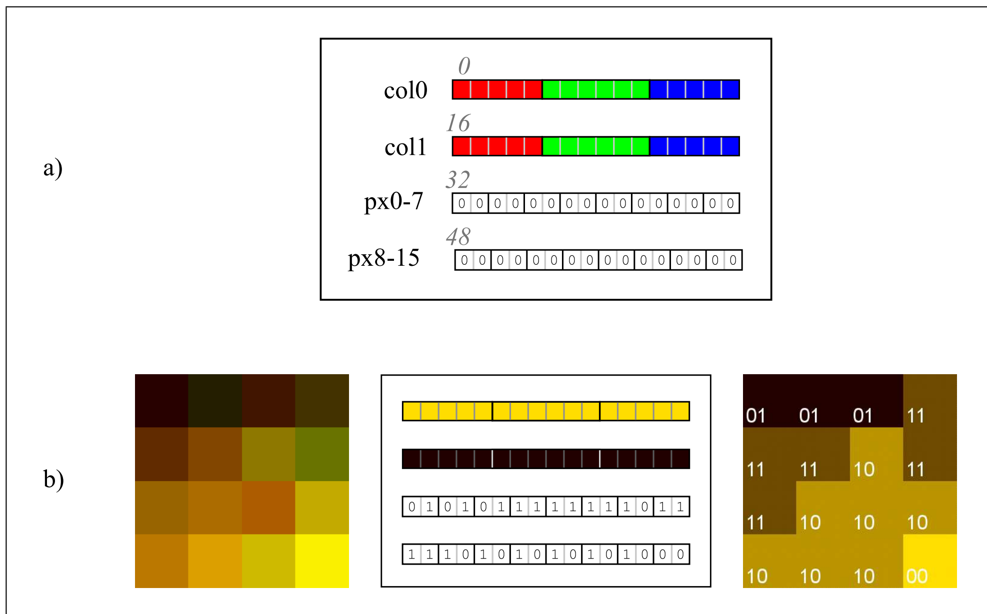


Figure 2.2: a) S3TC block structure. b) Example of how 4×4 pixels could be compressed and stored as an s3tc block, with the resulting decompressed block to the right.

To sum up, an uncompressed block of 384 bits (4×4 pixels, each holding a 24-bit color) is compressed to 64 bits (2 16-bit colors, and 16 2-bit indices) giving a

compression ratio of 6:1 or an effective bit rate of 4 bits per pixel. In short, an S3TC compression algorithm involves finding the two base colors that gives the best local palette and given this palette, choose the color that best represents each pixel.

Finding the base colors

When encoding an image to the S3TC format, there are a number of different ways of finding the two base colors to store in each block. Three methods are described below.

Brute-force A very naive brute-force method would be to encode the block using all possible pairs of 16-bit colors as base colors, and choose the combination that produces an S3TC block that most closely resembles the uncompressed block. This would produce an optimal solution, but there are 4.2 billion combinations of colors, which makes this method extremely slow.

Semi-brute-force A less naive, semi-brute-force method, would be to encode the block using all possible pairs of colors from the uncompressed block as base colors. This method produces good visual quality, but since there is no guarantee that the optimal base colors are present in the original block, the solution will generally not be optimal.

The semi-brute force method limits the number of combinations to try to 240, which is a significant improvement compared to the brute-force method.

Analytical method The analytical method described here produces significantly better results than the semi-brute force method, and executes in about the same time. It is used in available texture compression tools, and the real-time method developed for this thesis is based on it. The analytical method very quickly finds very good estimates of the base colors by analyzing the set of colors in the uncompressed block using a tool, borrowed from statistics, called principal components analysis (PCA). See Johnson and Wichern's book[4] for detailed information on PCA.

RGB colors can be seen as points in RGB color space (Figure 2.3), where the amount of red, green and blue is set off on the x-, y- and z-axis respectively. PCA is used to fit a line to the 16 points representing the original colors in a block and these points are then projected onto the fitted line (see Figure 3.1 for a 2D analogy). The estimated base colors are found as the outermost projected colors.

To increase the chance of finding optimal base colors, the block is encoded several times using colors in the vicinity of the estimate base colors, and the pair that

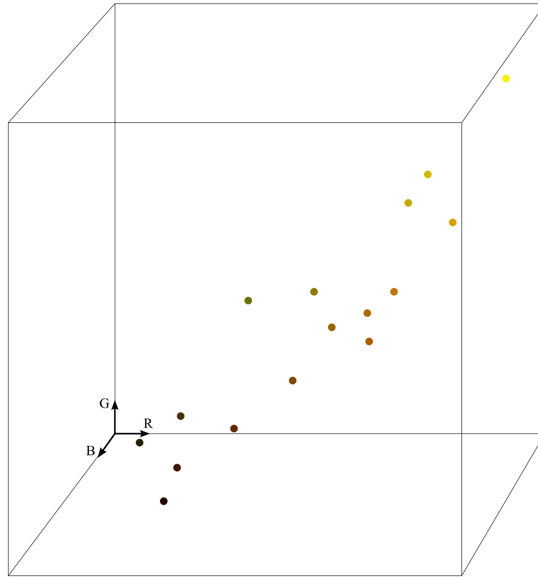


Figure 2.3: Colors represented as points in RGB space.

produces the best block is chosen. There is no guarantee that this method will find the optimal base colors, but the results are generally good. Since the analytical part of this method is relatively quick, the compression time largely depends on the number of combinations tried in this last stage.

Building the index table

Once the base colors have been found, it is a simple matter of iterating over the pixels in the block and, for each pixel, insert the index of the most similar palette color into the compressed block.

Chapter 3

Real-time texture compression

In order to speed up S3TC compression, to the point where it is usable for real-time computer graphics, we simplified the analytical method from the previous chapter with speed in mind and adapted this simplified algorithm to take advantage of the computational power of modern GPUs. In this chapter, we describe the simplifications made, the resulting simplified algorithm is described in detail, and finally, we show how we adapted it for the GPU.

3.1 Simplifying S3TC for speed

Texture compression to the S3TC format, as it has been described in Section 2.3, has an execution time measured in seconds.[1] Obviously, the compression algorithm has to be altered somehow in order to do real-time on-the-fly texture compression. To speed up compression we have simplified the analytical method in a number of ways. These simplifications all have an impact on image quality, but if speed is preferred over quality, we have found this quality impact to be acceptable. See Chapter 4 for image quality comparisons. The following simplifications were made.

Use the base color estimates As mentioned in Section 2.3, the base colors computed using PCA are very good estimates of the optimal base colors. We avoid the time-consuming "searching part" of the analytical method and use the estimated base colors directly.

Encode using type-1 blocks only In order to find out whether to encode a block as a type-1 block or a type-2 block, the original block has to be compressed twice, using the two different types of blocks, which significantly increases the

[1] Using current CPUs. Texture sizes of 512×512 pixels and above.

compression time. When compressing our eight test images using ATI Compressonator, on average less than 1% of the blocks came out as type-2 blocks. Based on this and the fact that those blocks still *can* be encoded as type-1 blocks with reasonable quality, we decided not to use type-2 blocks at all.

Simplified error computation When deciding which of the four calculated palette colors a certain color most closely resembles, the squared distance in RGB space between this color and each palette color has to be calculated. On the GPU, which has support for vector operations in hardware, the squared distance, d^2, between points $\mathbf{a}$ and $\mathbf{b}$ is easily calculated as:

$$d^2 = (\mathbf{a} - \mathbf{b}) \cdot (\mathbf{a} - \mathbf{b}).$$

There are 16 pixel colors to compare with 4 palette colors in each block and calculating the squared distance on a CPU, which does not have vector operations, involves a lot of multiplications. To speed this up in our CPU implementation, we calculate Manhattan distance, which requires no multiplications, instead of true distance. The Manhattan distance d_{M} between two points $\mathbf{a}$ and $\mathbf{b}$ is calculated as:

$$d_{\mathrm{M}} = |a_x - b_x| + |a_y - b_y| + |a_z - b_z|.$$

Using a real world 2D analogy, it is the travel distance between two locations in an area with only perpendicular streets, hence the name.

3.2 Algorithm in detail

Our algorithm uses the analytical method, outlined in Section 2.3, with the above simplifications. This section explains our algorithm in detail.

The first step in the algorithm is to fit a line to the 16 points, $\mathbf{c}_i, i \in [1, 16]$, in RGB space representing the 16 colors of a block. This line will go through the mean, $\bar{\mathbf{c}}$, of the points, defined as:

$$\bar{\mathbf{c}} = \frac{1}{16} \sum_{i=1}^{16} \mathbf{c}_i,$$

and from the theory of PCA we know that if we split the colors into component sets $\mathbf{R}$, $\mathbf{G}$ and $\mathbf{B}$, then the direction of the best fit line can be found as the first eigenvector of the covariance matrix of these sets.

The covariance matrix is defined as

$$\mathbf{C} = \begin{bmatrix} cov(\mathbf{R}, \mathbf{R}) & cov(\mathbf{R}, \mathbf{G}) & cov(\mathbf{R}, \mathbf{B}) \\ cov(\mathbf{G}, \mathbf{R}) & cov(\mathbf{G}, \mathbf{G}) & cov(\mathbf{G}, \mathbf{B}) \\ cov(\mathbf{B}, \mathbf{R}) & cov(\mathbf{B}, \mathbf{G}) & cov(\mathbf{B}, \mathbf{B}) \end{bmatrix},$$

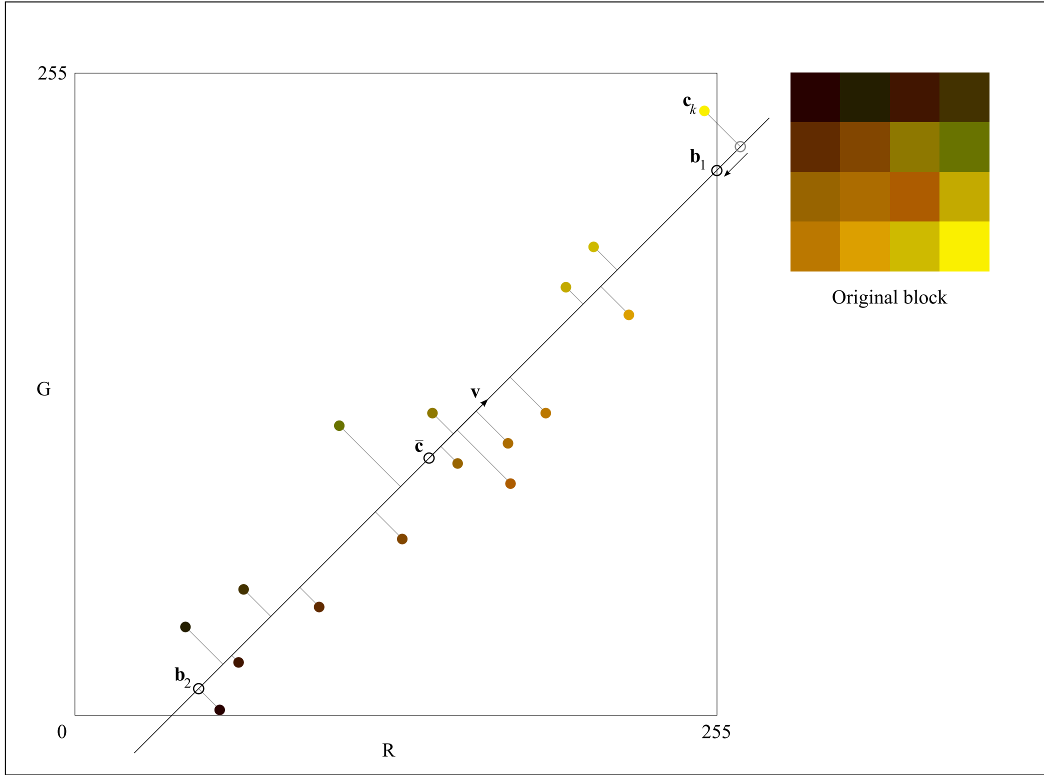


Figure 3.1: Colors, best fit line and projections for an example block (only red and green shown for simplicity). $\mathbf{b}_1$ *and* $\mathbf{b}_2$ *are the estimated base colors. As seen,* $\mathbf{c}_k$ *ended up outside valid RGB space, and instead, the closest valid color on the line was chosen as the base color.*

where (with 16 values in the sets)

$$cov\left(\mathbf{X},\mathbf{Y}\right)=\frac{1}{16}\sum_{i=1}^{16}\left(X_i-\bar{X}\right)\left(Y_i-\bar{Y}\right).$$

An eigenvector of $\mathbf{C}$ is a vector $\mathbf{v}=\left[\begin{array}{ccc} a & b & c \end{array}\right]^{\mathrm{T}}$ such that

$$\mathbf{C}\mathbf{v}=\lambda\mathbf{v},$$

where $\lambda\in\mathbb{R}$ is the eigenvalue of $\mathbf{C}$ corresponding to $\mathbf{v}$.

To calculate the sought after eigenvector $\mathbf{v}\neq 0$ we first find the corresponding eigenvalue:

$$\mathbf{C}\mathbf{v}=\lambda\mathbf{v}\quad\Rightarrow\quad\mathbf{C}\mathbf{v}-\lambda\mathbf{v}=\mathbf{0}\;\Rightarrow$$

$$\mathbf{C}\mathbf{v}-\lambda\mathbf{I}\mathbf{v}=\mathbf{0}\quad\Rightarrow\quad(\mathbf{C}-\lambda\mathbf{I})\mathbf{v}=\mathbf{0}$$

If $(\mathbf{C}-\lambda\mathbf{I})\mathbf{v}=\mathbf{0}$ for some $\mathbf{v}\neq\mathbf{0}$ then $\mathbf{C}-\lambda\mathbf{I}$ cannot be invertible which in turn means that $\det(\mathbf{C}-\lambda\mathbf{I})=0$. Expanding this results in a cubic equation

which can be solved to get the eigenvalues, λ. If we insert the largest eigenvalue in $(\mathbf{C} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$ we get a system of equations with three unknown variables $(a, b, c) = \mathbf{v}$ and three equations. Solving this system gives us the eigenvector $\mathbf{v}$, which is direction of the best-fit line.

We now have a line, defined by the point, $\bar{\mathbf{c}}$, and the direction vector, $\mathbf{v}$. We project the original colors $\mathbf{c}_i$ onto this line (see Figure 3.1 for a 2D analogy), and the estimated base colors are found as the outermost projected colors. If a color, when projected, ends up outside the valid RGB space (as shown in the figure), the color at the closest intersection between the line and the RGB space boundary is used instead.

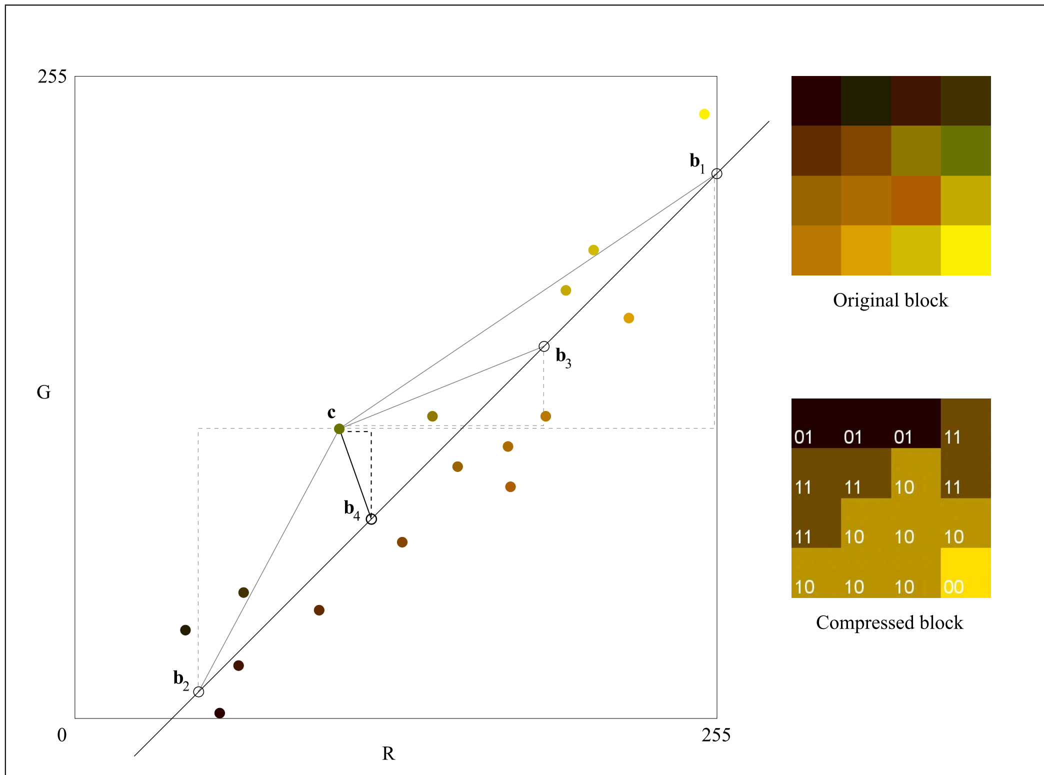


Figure 3.2: Two additional palette colors $\mathbf{b}_3$ *and* $\mathbf{b}_4$ *have been calculated through interpolation. The Manhattan distance (dashed) and true distance (solid) from a pixel color* $\mathbf{c}$ *to each palette color is shown. In this case,* $\mathbf{b}_4$ *would be selected using either distance for comparison, and the value 3 (indices start at 0) would be inserted into the S3TC block as the palette index for that pixel.*

After the two base colors have been found, two additional colors are linearly interpolated between these to obtain the four palette colors (Figure 3.2). The distance between each of the 16 original pixel colors and the four palette colors is calculated, and the closest palette color is chosen for each pixel. Depending on whether the algorithm is implemented on the GPU or on the CPU, either squared distance or Manhattan distance is used.

The base colors truncated to RGB565 format, and the palette index for each color is then stored in the output block.

3.3 Achieving real-time performance

The algorithm described in the previous section, when implemented on a CPU, executes reasonably fast (Section 4.1). Our goal was to be able to compress dynamically generated textures in real time, something an implementation running on current CPUs cannot do. Another problem with compressing textures on the CPU is that dynamically generated textures are, in many cases, already in the GPU's memory. Compressing such a texture would involve downloading it to the CPU, compressing it and then uploading it to the GPU again, consuming a lot of memory bandwidth. To solve these issues, we adapted our simplified algorithm to run directly on the GPU.

Our GPU implementation does most of the work in the fragment shader unit. We first create an RGBA8 texture that is $\frac{1}{6}$ the size of the texture to be compressed, and through render-to-texture methods, this texture is set as the output framebuffer. We then draw a quad that completely covers the output buffer so that each pixel will be filled once. By supplying a fragment program we can decide what data will be written to each pixel in the output buffer.

Basically, we have created a fragment program that does 16 texture lookups from the uncompressed texture, executes the algorithm from Section 3.2, and writes an S3TC block to the output texture. Ideally, a single execution of the fragment program would produce a full S3TC block from 16 input texture pixels, but since a full block is 64 bits, and a fragment program can only output 32 bits[1] to a single texture, this is not possible. Instead, the algorithm has two branches:

- For odd output pixels we compute and write the base colors.
- For even output pixels we compute the base colors then choose and write the indices.

Figure 3.3 illustrates how the GPU version operates. The image shows four fragment shader units running in parallel, meaning that two S3TC blocks are created simultaneously. Top-of-the-line GPUs contain up to 24 fragment shader units, allowing up to 12 S3TC blocks to be created in parallel.

GPUs do not provide any integer data types in hardware and no bitwise operations. To solve this, we represent each of the 8 bit output color components as a float in the fragment program, and simulate bitwise operations (AND, OR, SHIFT) with combinations of standard arithmetics (+, -, *).

[1]Actually 64 bits of output is possible but only if the output buffer is in FP16 format (a 16-bit floating point value per color component), which is unusable for our purposes since we need an integer output format in order to control which bits are set.

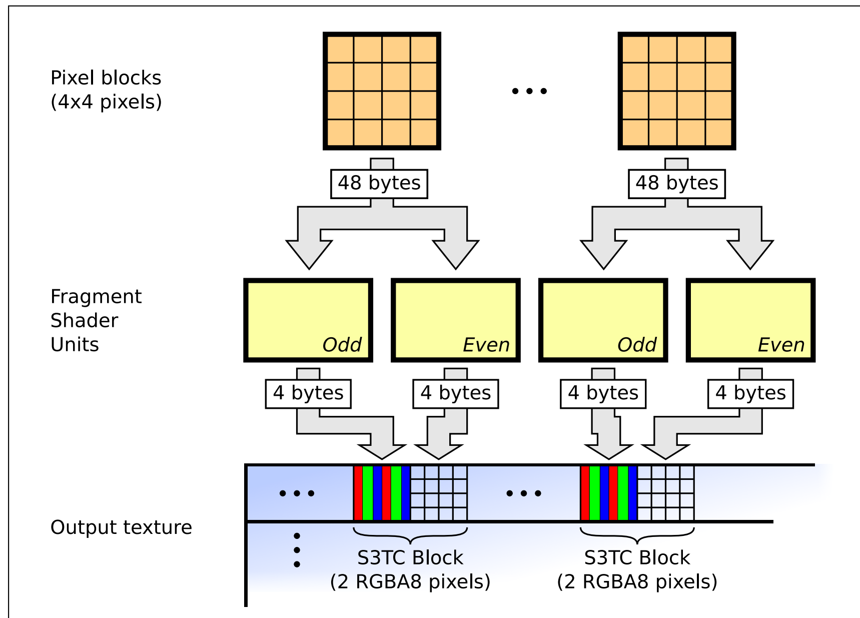


Figure 3.3: Creation of S3TC blocks using a fragment shader program. The same 4×4 pixels are used as input for two consecutive output pixels. The code path for odd pixels outputs the first 4 bytes of an S3TC block (base color part), and the even pixels code path outputs the last 4 bytes (palette color indices).

When we output colors from the fragment program, color values have to be in the range 0.0-1.0. These values are then automatically mapped, by the hardware, to 8-bit integers in the range 0-255 which are then stored in the output framebuffer. If we for example need to store the value 127 in one of the output bytes, we output the value 127.0/255.0 from the fragment program.

Once we have rendered the quad, our fragment program has written valid S3TC blocks to the output RGBA8 texture. Somehow, we now need to tell the hardware that "from now on this RGBA8 texture is an S3TC texture", a procedure we call *texture casting*. Unfortunately, there is currently no way, in either DirectX or OpenGL to do this cast, though supplying it should be a simple matter of updating the API and graphics driver. The only way we can make the texture available for future rendering in S3TC format is to download the texture to the CPU as an RGBA8 texture and then upload the same data to the GPU as an S3TC texture. This is the last step performed by our GPU implementation. Since this download/upload step is an issue that could be solved by a simple API and driver update, we omit the bandwidth implications of it when we calculate possible bandwidth savings in Section 4.3 below.

With our method, we calculate the base colors twice for each block. One might argue that if the ultimate goal is performance, then a two pass method (calculate base colors in pass one and use this as input for pass two where we write the

indices) might be faster. We have decided not to explore this method further since it would involve more memory reads and writes (due to texture caching, with our method, the 16 pixels will only be read from memory once), which would increase bandwidth usage and probably cancel out the benefits of calculating the base colors once. A two pass method would also require two fragment programs and involve OpenGL state changes, which would further slow down execution.

Chapter 4

Results

For this thesis we have developed a fragment shader, written in the OpenGL shading language (GLSL), and supporting C++ classes making it possible to rapidly compress textures to the S3TC format for immediate use. We have also created a number of applications that use and test various aspects of our compressor. The results are presented below.

4.1 Compression speed

As previously mentioned, compression when using available off-line tools, such as ATI's Compressonator or NVIDIA DDS Utilities, typically takes several seconds. One of the original ideas behind this thesis was to see whether real-time texture compression could be achieved using the computing power of modern GPUs.

To test the speed of our simplified algorithm and our GPU implementation, we created two test programs. They both repeatedly update a texture of 512×512 pixels, compresses it, and shows it on screen. The first program downloads the uncompressed texture from the graphics card, compresses it on the CPU, using our simplified algorithm, and then uploads it again. The other program does the compression directly on the GPU using our fragment shader. To see how fast compression is, the number of compressions per second is computed. On an AMD Athlon64 3200+ (running at 2.0GHz), we achieve a rate of 20 compressions per second, giving a compression time of 50ms. The GPU version running on a NVIDIA GeForce 7900GTX graphics card achieves a rate of 240 compressions per second. This gives a compression time of 4.2ms, 12 times faster than our CPU version and several hundred times faster than traditional off-line methods.

4.2 Compressed image quality

A common method for testing the image quality of a compressed image is to calculate the peak signal to noise ratio (PSNR). The formula for calculating PSNR is most easily defined by first defining the root mean square error (RMSE):

$$RMSE = \sqrt{\frac{1}{w \times h} \sum_{y=0}^{h-1} \sum_{x=0}^{w-1} ((\Delta R_{xy})^2 + (\Delta G_{xy})^2 + (\Delta B_{xy})^2)},$$

where R_{xy}, G_{xy} and B_{xy} are the red, green and blue values of pixel (x, y), and w and h is the width and height of the image.

From this, PSNR is defined:

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^2}{RMSE^2} \right).$$

PSNR is measured in dB and a higher value means better quality. We calculated the PSNR for eight test images compressed with our GPU compressor, ATI Compressonator[1] and NVIDIA DDS Utilities[2]. The results are presented in Table 4.1 and Figure 4.1.

Compressor / **Image**	lena	mandrill	lorikeet	kodim01	kodim02	kodim03	kodim04	kodim05	*Average*
GPU S3TC	33.6	27.3	32.6	31.7	34.3	35.5	34.9	30.2	*32.5*
ATI Compressonator	36.0	28.7	34.4	34.8	36.9	38.5	38.0	32.8	*35.0*
NVIDIA DDS Utilities	35.2	27.8	33.1	34.6	36.5	38.0	37.6	32.5	*34.4*

Table 4.1: PSNR values of various images compressed using different compression tools. All values are in dB.

The PSNR values of our compression is on average 2.5dB lower when compared with the values of ATI's Compressonator, which gets the best overall results. This is due to the simplifications we did to the compression algorithm in order to speed up compression. 2.5dB is a significant difference (0.5 dB is normally visible), but when visually comparing the actual images (See Appendix A for high resolution images), the quality appears to be better than these numbers suggest, and definitely sufficient for most real-time applications.

[1] Version 1.27.1066, RGB weights (1.0, 1.0, 1.0).
[2] Version 7.82, RGB weights (1.0, 1.0, 1.0).

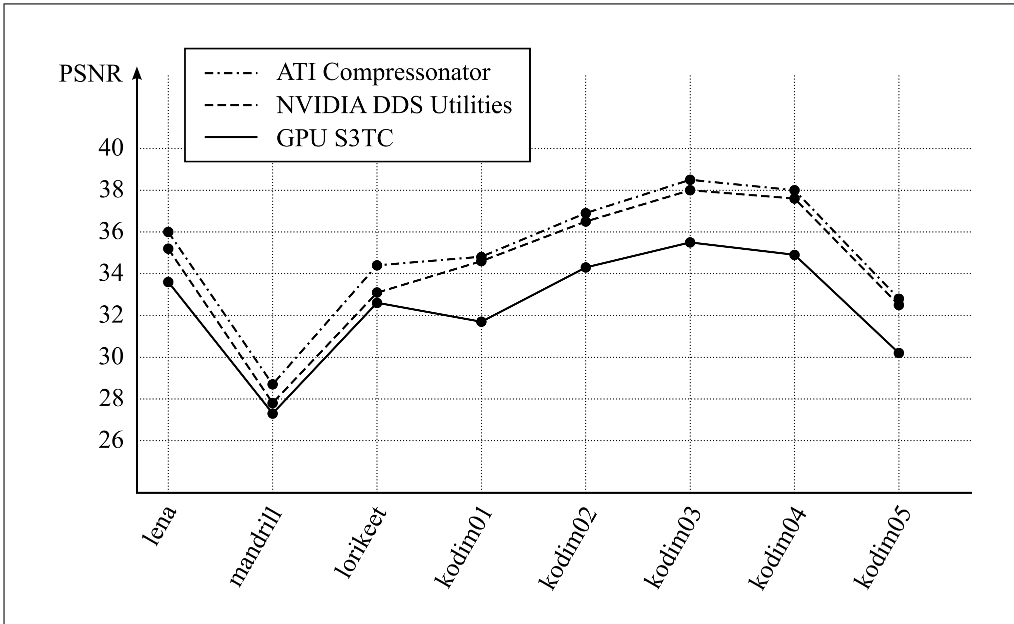


Figure 4.1: Graph of PSNR values of various images compressed using different tools. All values are in dB.

4.3 Bandwidth savings

Using compressed textures reduces the amount of memory that has to be transferred from the graphics card's memory to the shader units. Our compression algorithm has to read the entire uncompressed image once and write the compressed image once before the compressed texture can be used in rendering. The compressed texture is one sixth the size of the original uncompressed image. This means that if the uncompressed image is x bytes in size then compressing the texture involves transferring: $x + \frac{x}{6} = \frac{7x}{6}$ bytes to or from memory. If, when rendering scenes using the texture, every pixel from the texture is used in every frame, then rendering n frames requires $\frac{7x}{6} + \frac{nx}{6}$ bytes to be transfered over the memory bus. If we used the original uncompressed texture the corresponding number of bytes would be nx. From this we can derive how many frames that have to be rendered in order to save bandwidth using our compressor:

$$\frac{7x}{6} + \frac{nx}{6} < nx \Rightarrow \frac{7}{6} < \frac{5}{6}n \Rightarrow \frac{7}{5} < n \Rightarrow n > 1.4.$$

So, when rendering more than 1.4 frames, compressing the texture before using it, will reduce bandwidth usage.

Chapter 5

Discussion

As shown in Chapter 4, the texture compressor developed for this thesis produces high quality compressed images, rapidly enough for use in real-time graphics. There are, however, some cases of 4×4 pixels that cannot be compressed as an S3TC block without a significant quality loss. This is mainly due to the fact that an S3TC block can only contain four colors, and these colors are all on a line in RGB space. If the 16 colors of the uncompressed block are evenly distributed in RGB space or if some of the colors are far from the best fit line, then S3TC is not the ideal format. This is not a problem that stems from our compressor, but applies to all S3TC compressors.

There are cases where the use of type-2 blocks would help, i.e. when there are a few very dark pixels standing out from the rest of the pixels in the block. These cases are rare in natural images, such as photos, but more common in generated images, such as screenshots of GUIs. Since this is mainly an issue regarding the S3TC format, and not specific to our algorithm, exploring the use of other texture compression formats could prove interesting. The problem with compression algorithms for many other available formats is that they are based on compressing the block many times to find the best alternative. Such algorithms are not as easy to simplify as the S3TC algorithm is. Since most, if not all, compressed texture formats are designed with image quality in mind, developing a format with speed in mind, perhaps allowing a few more bytes per block, might be worthwhile as future work.

The potential bandwidth savings presented in this thesis are calculated under the assumption that texture casting can be done.[1] Since there is currently no such functionality in graphics APIs and drivers, we would like to propose an OpenGL function that could do this:

[1] Note, though, that even when "casting" is done by downloading and uploading the compressed texture, compression pays off already after using the texture for a few frames.

```
void glTexCastEXT(GLint internalformat)

Casts the currently bound texture to the format specified by
internalformat (currently only GL_COMPRESSED_RGB_S3TC_DXT1_EXT
is supported).
```

The entire process of compressing a texture on the fly could also easily be integrated into OpenGL. In keeping with the OpenGL way of doing things, this could be done by introducing converter objects. Using these might look something like:

```
GLuint textures[2];
GLuint converter;

glGentextures(2,&textures)

/* create uncompressed texture */
glBindTexture(GL_TEXTURE_2D, textures[0] );
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 512, 512, 0, GL_RGBA, NULL);

/* create compressed texture */
glBindTexture(GL_TEXTURE_2D, textures[1] );
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB_S3TC_DXT1_EXT, 512, 512, 0, 0, 0);

/* create converter object */
glGenConvertersEXT(1,&converter);

/* bind output texture to converter object */
glBindConverterEXT(converter);
glConverterTexture2DEXT(textures[1]);

/* render to uncompressed texture somehow */
rendertotexture(textures[0]);

/* bind input texture */
glBindTexture(GL_TEXTURE_2D, textures[0] );

/* convert currently bound texture to
converter object's output texture */
glConvertEXT();

/* render using compressed texture */
glBindTexture(GL_TEXTURE_2D, textures[1] );
glBegin(...)

...
```

The potential areas where compression of dynamic textures could be useful are many, and range a broad spectrum of applications within real-time computer graphics. On one side we have computer games, where compression could be used to compress cube maps either created directly after loading a level or created continuously as objects move around the game world. On the other side we have composite window managers where textures could be compressed, in order to save bandwidth usage, prior to doing crossfades or other transition effects. Another possible use could be compressing a video feed at a low frame rate and using it as a texture in a high frame rate rendering.

The advantages of compressed textures are especially appealing in mobile devices, where memory is limited and bandwidth usage costs in terms of battery power. The results presented in this thesis are based on our algorithm running on state-of-the-art desktop graphics cards and cannot currently be achieved on mobile devices.

This may, however, change in the near future. More and more mobile devices feature dedicated 3D hardware and the OpenGL ES 2.0 specification (OpenGL for mobile platforms) includes fragment shader programs.

Bibliography

[1] Tomas Akenine-Möller and Jacob Ström, *Graphics for the masses: a hardware rasterization architecture for mobile phones*, ACM Trans. Graph. **22** (2003), no. 3, 801–808.

[2] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha, *Rendering from compressed textures*, Computer Graphics **30** (1996), no. Annual Conference Series, 373–378.

[3] Pat Brown, *GL_EXT_texture_compression_s3tc*, OpenGL Extension Registry, `http://oss.sgi.com/projects/ogl-sample/registry/`, 2001.

[4] R. A. Johnson and D. W. Wichern (eds.), *Applied multivariate statistical analysis*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[5] Emmett Kilgariff and Randima Fernando, *The geforce 6 series gpu architecture*, GPU Gems 2 (Matt Pharr, ed.), Addison-Wesley, March 2005, pp. 471–491.

[6] Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer, *Hardware for superior texture performance*, Computers & Graphics **20** (1996), no. 4, 475–481.

[7] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard, *Cg: a system for programming graphics hardware in a c-like language*, ACM Trans. Graph. **22** (2003), no. 3, 896–907.

[8] Jay Torborg and James T. Kajiya, *Talisman: commodity realtime 3d graphics for the pc*, SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM Press, 1996, pp. 353–363.

[9] Lance Williams, *Pyramidal parametrics*, SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM Press, 1983, pp. 1–11.

Appendix A

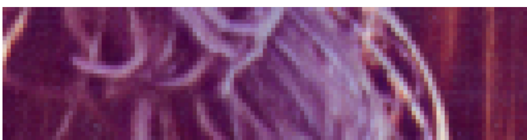
Visual image comparison

In this appendix, three of our test images are shown in high resolution. For each image the original uncompressed image is shown along with the image compressed using our GPU compressor, ATI Compressonator and NVIDIA DDS Utilities.

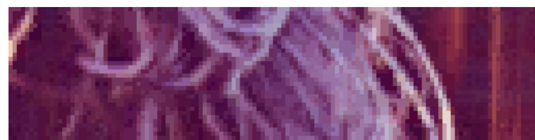
Due to copyright issues we cannot publish the other five images referred to in Section 4.2.

A.1 Lena

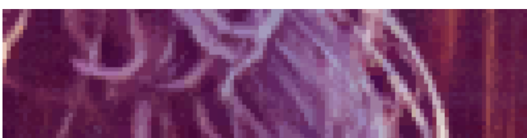
Uncompressed image



GPU S3TC



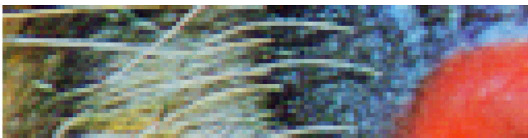
ATI Compressonator



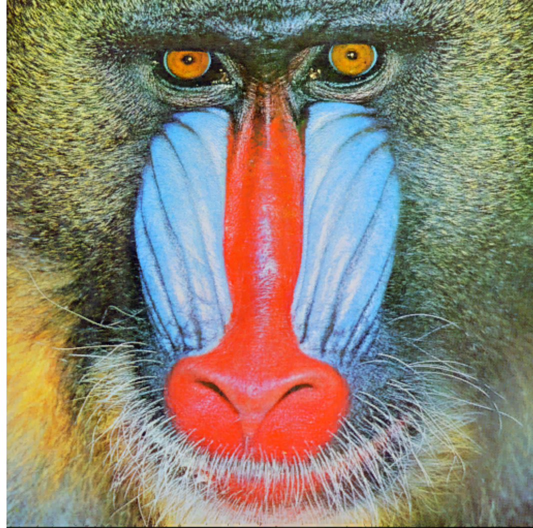
NVIDIA DDS Utilities

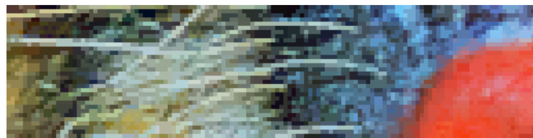
A.2 Mandrill

Uncompressed image

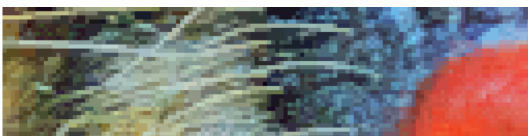


GPU S3TC

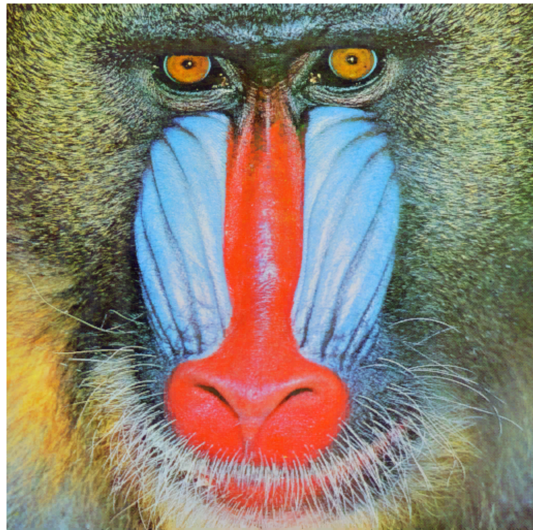




ATI Compressonator

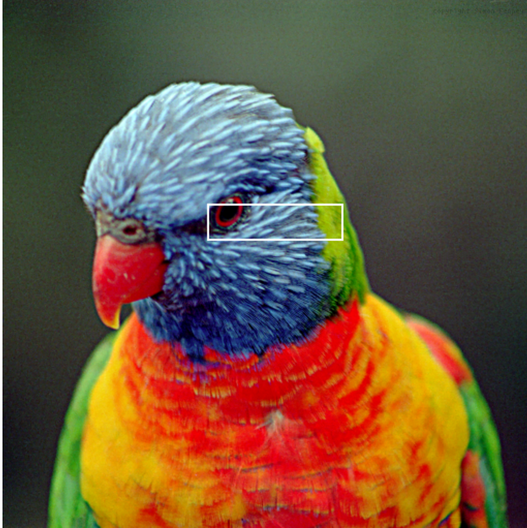


NVIDIA DDS Utilities

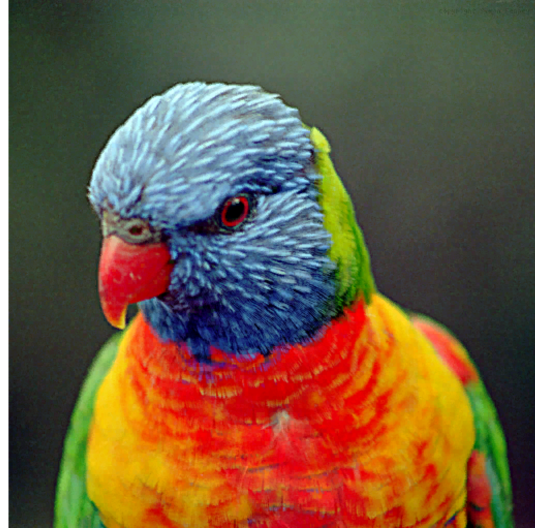


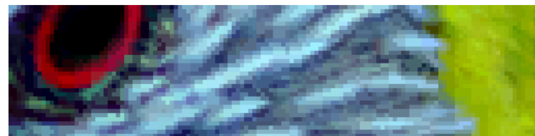
A.3 Lorikeet

Uncompressed image

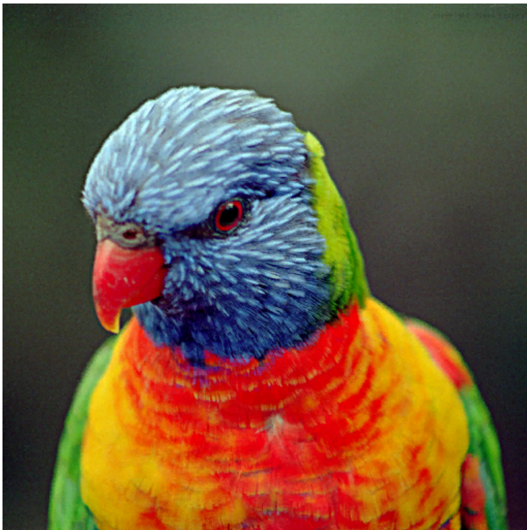


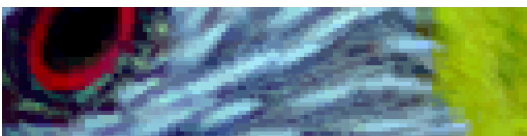
GPU S3TC





ATI Compressonator





NVIDIA DDS Utilities