Compressing Dynamically Generated Textures on the GPU

Oskar Alexandersson, Lund University / TAT
Christoffer Gurell, Lund University / TAT
Tomas Akenine-Möller, Lund University

1 Introduction

To reduce bandwidth usage to textures in a graphics hardware architecture, it is common to use texture compression (see Fenney's paper [2003] for an overview of previous work). The idea is to compress textures to a fixed rate. When the rasterizer requests texels, they are sent in compressed form over the bus, and the GPU decompresses them as needed. For texture compression, the focus is often on image quality. Therefore compression is done offline, allowing for compression times of several seconds or even minutes. Hence, traditionally, texture compression isn't used for compression of dynamically generated textures. Examples include dynamically generated cube maps and user interfaces. In this sketch, we show that by allowing a slight decrease in image quality, textures can be compressed about 100 times faster using the GPU, and that compressing dynamically generated textures can significantly reduce texture bandwidth usage.

2 Algorithm

We have chosen to use the S3 texture compression scheme (S3TC) [Brown 2001] since it is available in both OpenGL and DirectX. However, we could have used any TC scheme to prove our concept. In S3TC, two base colors are quantized to RGB565 for each 4×4 tile, and during decompression, two additional colors are interpolated in between these base colors. Thus, each tile obtains a local palette of four colors, and each texel in the tile can select either of these using a two-bit index. In total, 64 bits of storage is needed for a 4×4 tile. To compress a texture, we fit a line in RGB space that minimizes the mean square error (MSE). This line will pass through the average of the colors and the direction is found using principal component analysis (PCA). By projecting the colors onto this line, an interval encompassing all tile colors is found, and this interval gives us the two base colors. For each texel we then choose the two-bit index that represents the closest color. A basic "GPU port" of this algorithm is fairly simple with the following exceptions:

Lack of integers. The compression algorithm involves a lot of integer math and bitwise operators. This becomes problematic, since current GPUs lack these features. Instead, we have to rely on floats. We represent each 32-bit integer by four floats, each holding a value between 0 and 255. Bitwise operations have to be done using standard arithmetics. Before we write to the output texture, each of the four floats are divided by 255 in order to obtain values between 0.0 and 1.0. When writing to an RGBA8 texture, these values are automatically mapped to 8-bit integers.

Four bytes of output. In a single pass, we can never write more than four bytes to a single render target. Since we need to write 8 bytes (64 bits) per tile, the shader needs different code paths for odd/even pixels. We can use either a two-pass or a one-pass shader program. Even though the latter alternative involves computing the base colors twice, we use this approach because it is more straightforward and it benefits more from texture caching.

No "texture casting". In order to have bit-level control over the constructed S3TC texture, we have to output to an RGBA8 texture. There is currently no way to immediately make the graphics hardware use this RGBA8 texture as an S3TC texture. Instead, we currently read it back to RAM and upload it as an S3TC texture. Ideally, we would need a way to inform the graphics driver that "this RGBA8 texture is now an S3TC texture". We call this procedure *texture casting*.

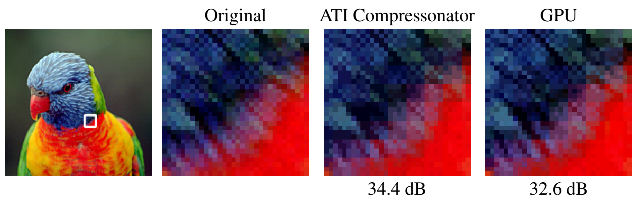

Figure 1: Quality comparison. PSNR numbers are shown below the images.

3 Results

Compressing a texture involves reading the original texture once, and writing the compressed texture once. Assuming the original texture uses x bytes, the compression costs[1]: $x + \frac{x}{6} = \frac{7x}{6}$ bytes. If all texels in the texture are accessed every frame, then rendering n frames costs: $\frac{7x}{6} + \frac{nx}{6}$, while the cost using non-compressed textures is: nx. This implies that bandwidth savings occur when $n > 1.4$, i.e., already on the second frame. Rendering using compressed textures converges towards $\frac{x}{6}$ bytes per frame. Although this is a special case, the same applies to a real GPU architecture where tiles of texels are being read rather than individual texels.

Our implementation compresses a 512×512 texture in 25 ms using the GPU[2], which is about $100\times$ faster than on the CPU. This speed advantage has two causes: 1) the compression algorithm is well suited for a streaming architecture, so a GPU implementation provides generous speedups, and 2) a CPU implementation usually searches for a better line fit in a small region around the line endpoints, and this is not done on the GPU. As a result, our image quality is on average about 2.5 dB lower in peak-signal-to-noise-ratio (PSNR) using eight test images, but we find the image quality useful for real-time purposes.

The speed advantage of our algorithm opens up new possibilities in terms of applications for texture compression. For example, dynamically generated textures in user interfaces (e.g., for mobile phones) can be compressed on the fly using the GPU and used in subsequent frames. This can provide for substantial bandwidth savings as can be seen in our accompanying video. Our user interface renders two images, which are compressed on the GPU, and then a cross fade between these is generated. Since the compressed textures are used during several frames in the cross fade, the bandwidth savings become significant.

The major point of our sketch is to spur an interest in texture compression on the GPU, and to show that it can provide for great bandwidth savings. We also hope that graphics hardware manufacturers will be inspired to implement *texture casting* in their drivers, and perhaps even include hardware texture compression and casting as part of APIs, such as OpenGL and DirectX. We also want to emphasize that the bandwidth savings are critical in order to reach high performance, but also for mobile graphics, minimizing memory accesses is of uttermost importance since it helps preserving battery power.

References

BROWN, P. 2001. GL_EXT_texture_compression_s3tc. In *OpenGL Extension Registry*, `http://oss.sgi.com/projects/ogl-sample/registry/`.

FENNEY, S. 2003. Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware*, 84–91.

[1]All costs are in terms of memory bandwidth.

[2]NVIDIA Quadro FX 1400 Go (Geforce 6800 Go equivalent)