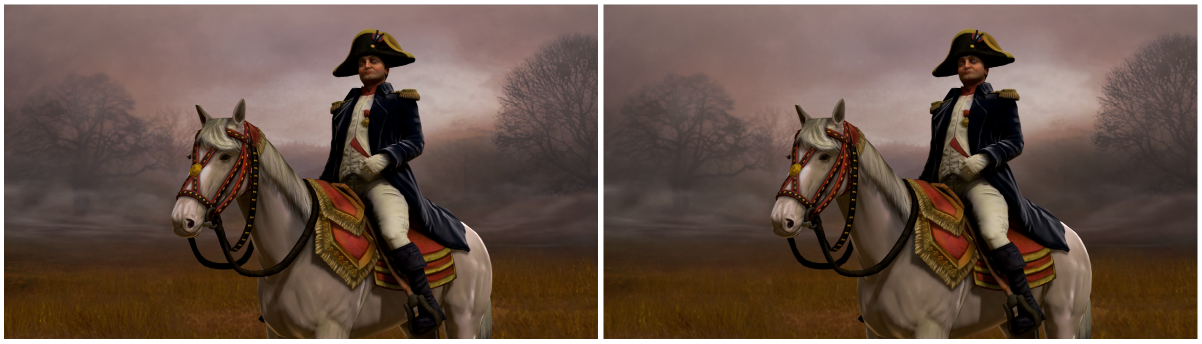
Eurographics Symposium on Rendering 2011
Ravi Ramamoorthi and Erik Reinhard
(Guest Editors)

Volume 30 (*2011*), *Number 4*

Variable Bit Rate GPU Texture Decompression

M. Olano[1,2], D. Baker[2], W. Griffin[1], and J. Barczak[2]

[1]UMBC
[2]Firaxis Games



(a) Raw textures (167.7 MB) (b) VBR textures (14.9 MB)

Figure 1: *Variable bit rate texture compression applied to the Napoleon Bonaparte scene from Sid Meier's Civilization® V using 27 textures. Image (b) using VBR compressed textures has a Mean SSIM error of 0.9935 (best=1) and SHAME-II color difference of 0.539 (best=0) compared to Image (a) using raw uncompressed textures.*

Abstract

Variable bit rate compression can achieve better quality and compression rates than fixed bit rate methods. None the less, GPU texturing uses lossy fixed bit rate methods like DXT to allow random access and on-the-fly decompression during rendering. Changes in games and GPUs since DXT was developed make its compression artifacts less acceptable, and texture bandwidth less of an issue, but texture size is a serious and growing problem. Games use a large total volume of texture data, but have a much smaller active set. We present a new paradigm that separates GPU decompression from rendering. Rendering is from uncompressed data, avoiding the need for random access decompression. We demonstrate this paradigm with a new variable bit rate lossy texture compression algorithm that is well suited to the GPU, including a new GPU-friendly formulation of range decoding, and a new texture compression scheme averaging 12.4:1 lossy compression ratio on 471 real game textures with a quality level similar to traditional DXT compression. The total game texture set are stored in the GPU in compressed form, and decompressed for use in a fraction of a second per scene.

Categories and Subject Descriptors (according to ACM CCS): I.4.2 [Image Processing and Computer Vision]: Compression—; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Texture

1. Introduction

GPUs use fixed bit-rate texture compression to save space and rendering bandwidth. Each block of texture is compressed to exactly the same size, so can be accessed and decompressed independently. For example, the DXT5 texture format compresses each 4x4 block of RGBA pixels to 128 bits, for a 4:1 compression ratio. Unfortunately, fixed bit rate compression inevitably has some blocks that are compressed too much, leading to artifacts, and some blocks that could be compressed more, leading to larger files.

GPUs have become more and more effective at hiding memory latency with threads, making the bandwidth savings of DXT less important for many applications. For example, for the game shaders in Sid Meier's Civilization® V, rendering with DXT vs. uncompressed textures did not make **any** noticeable performance difference, counter to the folk wisdom that DXT is necessary for game rendering bandwidth. In addition, DXT artifacts are becoming less acceptable as game quality expectations grow. Memory and disk savings are still a key reason to continue using texture compression. Most games use more texture than fits into memory. To swap working sets, they pause with "loading" screens, or initially load low resolution textures, so the player sees the textures "res in" as they are playing. In a game like Civilization V, the player can switch at any time between the game screen and any of the world leaders with no warning. In this context, "loading" screens are unacceptable.

Variable bit rate (VBR) compression adapts the compression rate to the data, but cannot be deterministically indexed. Much game data, including animation data and audio, is already variable-bit-rate compressed, but decoded on the CPU. Since memory capacity is a critical reason to use texture compression, we propose a higher compression rate VBR method with decompression into a working set of uncompressed textures for rendering. While decompression does not happen during rendering, it still must be fast enough to avoid game stalls to decompress a texture.

We present a VBR texture compression algorithm designed for fast GPU decompression. A high compression rate algorithm that can be decompressed on the GPU allows a vast increase in on-GPU texture storage. Even as future GPUs move to unified memory, memory will still limit texture capacity, and fast GPU decompression still increases total texture capacity. Unlike most image compression algorithms, which only reconstruct an image at a single resolution, our algorithm reconstructs the entire MIP chain [Wil83], avoiding the significant overhead of compressing each MIP level independently, while allowing independent selection of MIP filter or artistic tailoring of the MIP levels.

On RGB textures from the Kodak Image Suite [Fra10], our *lossless* compression averages a 3.1:1 compression ratio, half DXT's 6:1 rate while reproducing the exact raw pixel data. On a series of game textures from Civilization V, our *lossy* compression resulted in an average compression ratio of 12.4:1 (Figure 8). Figure 1 shows one of these scenes, consisting of 27 textures with sizes ranging from 128x128 to 2048x2048, including 9 RGBA diffuse textures, 7 RGBA textures encoding specular color and power, 7 normal maps, 3 additional single-channel opacity maps, and 1 single-channel skin blur map. For the DXT comparison in Figure 8, the diffuse and specular textures were encoded with the BC3 format, the opacity and skin blur maps with BC4, and the normal maps with BC5.

A typical world leader scene in the game averages 98 MB of uncompressed texture. There are 18 leaders in the standard game, for a total of nearly 1.8 GB (downloadable content adds even more). It is not practical to keep all of the data in memory (host or GPU) due to its size and the substantial memory needs of unrelated game systems. In addition, it is *impossible* to predict texture demand, since the player may elect to visit any leader at any time. Dynamic loading from disk cannot be accomplished without unacceptable stalls or degrading image quality. Our VBR compression reduces the average size of a leader from 98 MB to around 7-8 MB. Given the small size, all of the leader textures can remain permanently resident on the GPU in compressed form and can be unpacked into a renderable form prior to entering a leader scene. The decompression requires under 10 ms per 2048x2048 MIP texture, or roughly 100 ms per leader. This avoids in-game loading delays, and enables use of larger textures, sized for HD resolution screens, rather than having to settle for lower resolution and quality textures to fit within memory and bandwidth limits.

In our case, we chose to render from RGBA textures because we deemed DXT unacceptable for our leader quality standards, and we saw no measurable performance difference. While we expect the performance difference to be nominal for many cases, applications which are bandwidth bound could elect to recompress the textures on the GPU [vWn07, Cas07]. We already use this technique for in-game dynamically generated terrain textures. In our experience, recompressing to DXT on DX11 requires minimal overhead, and is fast enough to be almost insignificant when compared to VBR decompression. While this would forfeit any image quality benefits from our technique, it would still yield all of the storage and streaming benefits.

2. Background and Related Work

Compression/Entropy Coding: Data compression exploits variations in the information entropy to store data more compactly while still allowing entirely lossless reconstruction. Since some patterns in the input stream are more likely than others, the more likely patterns can be encoded with fewer bits and less likely patterns with more bits. There are two main parts to the compression problem: modeling the probabilities for input symbols and using those probabilities to construct a compressed data stream.

The probability model can be either adaptive or nonadaptive [RL81]. Adaptive models refine an implicit initial estimate as more of the data is decompressed, avoiding data overhead, but increasing dependence and ordering constraints on the compressed data stream [SCE01]. Non-adaptive models analyze the input data probabilities. This can be a simple static statistics table [Wal92], or parameters to a more complex statistical model [BS99, LW99]. Despite the storage overhead, we choose to use a static statistics table since it requires the least decompression-time computation.

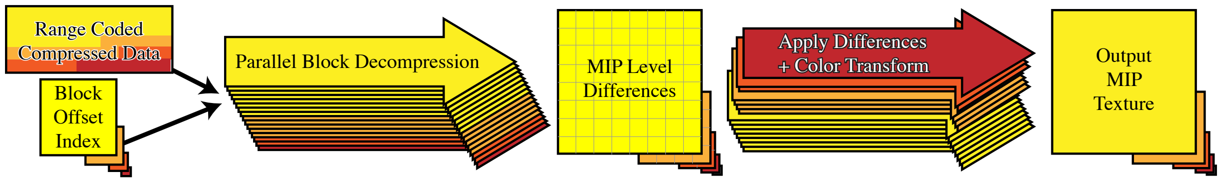


Figure 2: *Overview of GPU decompression algorithm. Each bundle of arrows is a parallel GPU compute kernel*

Huffman coding compresses each symbol independently [Huf52], but loses potential compression since each symbol uses an integer number of bits. Other approaches compactly encode small numbers [Eli75, MA05], or take advantage of repeating patterns in the input [ZL77].

Arithmetic and range coding can represent several common symbols with a single bit in the compressed stream, or an uncommon symbol with a long series of bits [HV94, Mar79]. We use a range coder because common range coder implementations operate on byte or larger units of data, which is more friendly for GPU implementation, while arithmetic coders operate a bit at a time. A range coder works directly from probability estimates for the symbol to code. The total range of possible values is represented as an integer, and each possible symbol uses a partition of this range.

Image Compression: Most image compression and decompression is designed to run on a CPU to compress a single image. Vector quantization can be applied at either the pixel or block level [Hec82, BAC96]. These approaches are fixed bit rate and inherently lossy, but amenable to random access within the compressed texture.

A better compression quality and/or rate can be achieved with variable bit rate compression. VBR image compression methods include one or more transforms before a final entropy encoding stage. Most transform to a luminance/chrominance space (YIQ, YCbCr, YCoCg, Luv, etc.) since humans are more sensitive to errors in luminance than errors in chrominance [HS00]. They then transform to a space with better probability characteristics for the variable bit rate encoding, often in the form of a difference from a prediction. Options include the DCT [Wal92] or the discrete wavelet transform [Sha93, SCE01]. Lossy compression is usually quantizes in this space. Quantized values are encoded losslessly to create the compressed data stream.

Wavelet Image Compression Zerotree encoding [Sha93] and the subsequent wavelet compression in JPEG 2000 [SCE01] are both designed for CPU decompression, and have interdependence between pixels or blocks that make them ill suited to GPU parallelization. We borrow a few key ideas from both in designing our compression algorithm. These approaches perform a wavelet transform on the image. Since each level of the wavelet pyramid approximates the level below, the detail coefficients are likely to be near zero. This probability differential is exploited in subsequent entropy coding, identifying subclasses of pixels that have differing and predictable probabilities.

GPU Texture Compression: GPU texture compression has primarily focused on fixed-bit rate lossy methods [INH99, SAM05, SP07]. These allow rendering directly from a compressed texture at a cost in the overall quality and/or compression rate. Specialized variations have been developed for normals [MAMS06] and high dynamic range data [MCH*06, RAI06].

Some hardware extensions have been proposed that would allow more flexible on the fly compression. Inada and McCool [IM06] proposed B-tree indexing hardware to support random access within a variable bit rate compressed texture, and Sun et al. [SLT*09] proposed a general configurable filtering unit. Our method decompresses variable bit rate textures using standard GPU computing.

GPU Compression/Decompression: Relatively little work has been done to date on direct compression or decompression on the GPU. van Waveren and Castaño [vWn07, Cas07] show DXT compression on the GPU. Lindstrom and Cohen [LC10] show GPU decompression of terrain. In their work, each block of terrain is encoded one vertex at a time, where three previously decoded vertices provide a planar approximation and encodes the difference from that plane using the RBUC residual coding algorithm [MA05].

3. VBR Algorithm

Our goals for a texture compression algorithm are:

1. Compress with equal or better quality and much more compactly than fixed-rate texture compression
2. Compress an entire MIP chain, without constraints on how the MIP levels are created
3. Decompress and load into GPU textures for rendering fast enough to prevent noticeable game stalls

In the sections that follow, we show that a variable bit rate (VBR) compression algorithm based on multi-resolution difference of MIP levels satisfies our first goal on size and quality and second goal for compressing an entire MIP chain. We achieve similar to DXT compression rates for lossless compression, and 3-5 times better than DXT for lossy compression. In the process, we also develop a novel GPU range decoder formulation that is more efficient on the GPU, but is 100% compatible with an existing CPU range coder [LI06].

The outline of our GPU decompression algorithm is shown in Figure 2. The texture is decompressed in 16x16 blocks, one GPU thread per block. Each thread uses an index to find its starting place in the compressed data and produces a block's worth of MIP level difference data. The difference data is converted into real pixel values and transformed from YCbCr or another color space to RGBA to produce an uncompressed texture with a MIP chain.

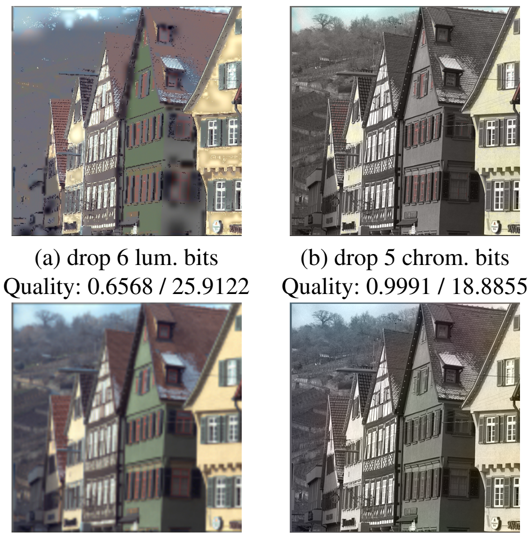
3.1. Difference of MIP levels

Most image compression operates on single images. Additional MIP levels must be compressed separately or recomputed after decompression. Wavelet compression has the nice property that an approximation pyramid is created by the decompression. Unfortunately, the approximation filter is completely determined by the wavelet basis. It will not work if we want a different MIP level filter for better filter quality, or artistic control over individual MIP levels.

Shapiro [Sha93] used a wavelet tree, but suggests that his zerotree approach could be applied to a Laplacian pyramid instead. Like wavelet detail coefficients, the difference of bilinearly interpolated MIP levels provides all of the information necessary to recreate the finer level from a coarser approximation. Unlike the wavelets used in prior work, the difference of MIP levels are not separable into horizontal and vertical filters, but since bilinear reconstruction is hardware accelerated, it is more efficient on the GPU than a separable wavelet filter with larger support. We do not make any assumptions on the method used to generate the MIP levels. Compression rate may suffer if the coarser MIP levels are not predictive of the finer ones, but we will always reconstruct the MIP levels given.

For lossy compression we drop bits from each level, or entire levels for a channel. Since the difference coefficients are applied to a bilinear reconstruction of the coarser level, overcompression looks like linear interpolation artifacts rather than blocking or ringing (Figure 3).

To avoid having loss in coarser MIP levels adversely impact the quality of the finer MIP levels, we compute the MIP differences **after** bit truncation. During compression, we compute the difference for a MIP level, truncate it, encode it, then reconstruct the MIP values using the truncated differences. The next level differences are computed relative to the previous level exactly as it will be reconstructed during decompression. Not only does this avoid propagating errors from one MIP level to the next, but allows finer MIP levels to be more accurate than the coarser level if desired.


(a) drop 6 lum. bits
Quality: 0.6568 / 25.9122

(b) drop 5 chrom. bits
Quality: 0.9991 / 18.8855

(c) drop 2 lum. levels
Quality: 0.6919 / 4.7223

(d) drop 6 chrom. levels
Quality: 0.9990 / 18.6832

Figure 3: *Over compression (Quality: MSSIM/SHAME-II).*

3.2. Entropy Coding

Our entropy coding of the MIP differences is inspired by the method used in JPEG 2000 [SCE01], but with critical modifications to be more efficient for GPU decoding. JPEG 2000 encodes one bit plane at a time, from MSB to LSB, allowing bit planes to be globally sorted by importance. The compressed stream can be truncated at any point to achieve continuous variation in quality vs. compression ratio. To determine entropy coding probabilities, each bit uses a 3x3 pixel neighborhood of previously decoded higher-order bits together with bits from coarser wavelet levels to choose one of three coding classes, likely to be 0, likely to be 1, or about even probability of either.

We determine the probability classes for each bit based solely on higher-order bits within a fixed 2x2 pixel neighborhood, and encode all bits for each 2x2 neighborhood before moving to the next. This has several advantages for GPU decoding. By not including other MIP levels in the class decision, we can decode the MIP differences of all levels simultaneously, increasing the number of GPU threads. Since we decode all bits of one 2x2 neighborhood before moving on, the partially decoded results can remain entirely in local registers, and be written to global memory just once, when the 2x2 neighborhood is complete. Coding loss is decided by dropping bits or levels at encoding time.

3.3. Coding Blocks

To achieve good GPU decoding speed, we divide the image into blocks, with GPU threads decoding the difference values for every block in every MIP level simultaneously. This introduces a tradeoff between compression rate and GPU occupancy. Smaller blocks increases the number of threads, increasing occupancy, but each block introduces overhead that reduces the compression rate.

First, each GPU thread needs to know where its block starts, with one 4-byte index per block. In addition, each independent block has a compressed static probability table, plus a 4-byte overhead to flush the entropy encoder at the end of the block. The block start position is stored in a MIP index texture, reduced from the original MIP texture by the block size. The index offsets could be compressed using the average expected compression rate as a prediction, but every thread would need to decompress the index to find its starting location, adding significantly to the total GPU decompression time. Instead, we leave the index uncompressed as a fixed overhead. For example, 16x16 blocks give an index 1/256th the size of the original texture, so a 2048x2048 texture with 12 MIP levels has a 128x128 index with 8 MIP levels.

The block size must be a multiple of the 2x2 neighborhood size. Since our textures are all power of two dimensions, we also would like the block size be a power of two to avoid having to deal with partial blocks (though partially filled blocks could be accommodated if non-power-of-two textures were needed). Figure 10 shows that the best compression rate was for 32x32 blocks. Larger than this, and the static statistics don't predict well enough. Smaller than this, and the per-block overhead starts to increase the compressed size. None the less, we ultimately ended up using 16x16 blocks. The smaller blocks result in slightly worse compression rate, but allow four times as many threads, which gives a noticeable boost to the GPU decoding performance.

The number of threads is given by the MIP expansion equation from the number of blocks in the base MIP level:

$$\begin{aligned} base &= image_x * image_y / block^2 \\ threads &= (4 * base - 1)/3. \end{aligned}$$

The thread IDs are assigned first to each block in the base level, then the next smallest, etc. Level L starts at

$$start(L) = (base - base * 2^{-2L}) * 4/3.$$

Given this, each thread, t, can determine its level and position within the level:

$$\begin{aligned} L &= \left\lfloor \frac{1}{2} \log_2 \left(\frac{4\,base}{4\,base - 3\,t} \right) \right\rfloor \\ offset &= t - start(L) \end{aligned}$$

3.4. Color Space Transformation

A data dependent transform prior to encoding can help overall compression rate and quality. We have implemented two, though others are possible. Color data is transformed to a luminance/chrominance space. Chrominance channels can generally be encoded with fewer bits and fewer MIP levels without perceptible difference [HS00, Sha93, Wal92, SCE01]. We use the YCbCr color space using an invertible transform to allow for lossless compression [SCE01].

For masks or images with an alpha channel, if we are dropping d bits, we can use the transform

$$alpha' = \frac{alpha * 255 - (2^d - 1)}{255 - 2 * (2^d - 1)}$$

This guarantees that even with the maximum error, a raw $alpha$ of 0 will give a compressed $alpha'$ of 0, and a raw $alpha$ of 1 will give a compressed $alpha'$ of 1.

```
Store any MIP levels smaller than 2x2 as raw colors
Convert to appropriate color space
For each MIP level from coarsest to finest
  compute difference with previous level
  truncate bits
Compute probabilities for all levels < block size
Compress levels below the block size together
Flush compression stream
For each MIP level
  For each block
    Write starting position to index texture
    Compute static probabilities
    Encode range of bits and probability table
    For each 2x2 neighborhood
      For each bit
        Compute classes and encode bits
    Flush compression stream
```

Figure 4: *Pseudo-code for full VBR compression algorithm*

```
Decode MIP levels smaller than 2x2 as raw colors
Read stats for levels below block size and decompress
For each block in index in parallel
  Decode bit range and probability table
  For each 2x2 neighborhood
    For each bit
      Compute classes and decode bits
    Write 2x2 neighborhood to difference texture
For each MIP level from coarsest to finest
  For each pixel in parallel
    Apply MIP differences and inverse color transform
```

Figure 5: *Pseudo-code for VBR decompression algorithm*

3.5. VBR Algorithm Summary

The final VBR encoding and decoding algorithms are shown in Figures 4 and 5. Note that the smallest levels that cannot contain a 2x2 neighborhood (1x1 for a square texture; 1x2, 1x4, etc. for rectangular textures) are encoded as raw pixels. All levels from this to the block size (2x2, 4x4 and 8x8 for a square texture) are encoded with a single probability table and decoded as a single serial step. Only levels from the block size up are decoded in parallel. For a 2048x2048 texture, there are 21,845 16x16 blocks in the MIP pyramid. The MIP differences are turned back into color in a process similar to GPU MIP generation, with a kernel per level, each with one GPU thread per pixel.

4. Range Decoder

In addition to tailoring the compression algorithm to the GPU's computational and memory architecture, our fast GPU decompression also relies on a GPU friendly range decoder. This decoder is an alternate formulation of the CPU range decoder by Lindstrom and Isenburg [Sub99,LI06] and is 100% compatible with it. Figure 6 shows decoding code for this coder. The entire compressed file is viewed as one huge integer. The coder works with a 32-bit window on this integer (code), and tracks the bottom (low) and size (range) of a subrange of this 32-bit window.

Figure 6 has several aspects that hurt the GPU efficiency. The compressed stream is read multiple times within the update function, including once within a loop. Further, those accesses are a byte at a time, when GPU memory accesses are naturally 32-bit. We make three observations that allow significant improvements (Figure 7).

First, the loop runs a maximum of three iterations:

$$low \wedge (low + range) \quad (1)$$

has its highest-order bit where the top and bottom of the current subrange differ. Each time through the loop, low and range are both shifted left one byte, so the condition is also shifted left one byte. Since range must be non-zero, equation (1) has 1-3 zero bytes before the loop. Therefore, the loop run a maximum of three times, and we can tell how many times by the number of high-order zero bytes in equation (1) and directly add that many bytes from the stream.

The second observation is that the window adjustment is effectively doing the same operation, but with a slightly different derivation for the new range. Adjusting for the expected change to range in the loop, we can determine ahead of any access to the code stream whether and how many additional bytes this might add. Together, these allow us to consolidate the compressed stream accesses to a single instance which may fetch between one and three new bytes.

The final observation is that these extra byte fetches may not be aligned with the GPU word boundaries, so we end up with fairly complex alignment code to fetch between zero and two words and align the new bytes from them. Yet the code word is just an unaligned 32-bit window on the compressed stream. No operations are ever performed on it except to move the window within the stream. Given that, the update process really only needs to keep track of the current byte position in the stream. If we make that change, then the use of code in the decode() function becomes an unaligned fetch of one word of data from the stream. In our current implementation, we build that unaligned fetch out of two aligned fetches from a GPU buffer, but future implementations could attempt to improve the memory bandwidth by pulling in a larger window with a coalesced read then grabbing unaligned 32-bit blocks from that local copy of the data.

5. Results

Unlike mean square error or peak signal-to-noise ratio, recent objective image comparison metrics weight differences in local image structure over visually difficult to detect global changes. The Structural Similarity Index Metric

```
// return value for the next symbol, within a total
// integer range for all possible next symbols
decode(TotalRange) {
  return (code−low) / (range/=TotalRange);
}

// update state once we know that the start value
// and integer subrange for the symbol
update(SymbolLow, SymbolRange) {
  // adjust bounds to current subrange
  low += SymbolLow∗range;
  range ∗= SymbolRange;

  // shift window if done with top byte
  while((low ^ (low+range))>>24 == 0) {
    code = code<<8 | ∗streamPtr++;
    low <<= 8;
    range <<= 8;
  }

  // adjust window if range is getting too small
  if (range >> 16 == 0) {
    code = code << 8 | ∗streamPtr++;
    code = code << 8 | ∗streamPtr++;
    low <<= 16;
    range = −low;
  }
}
```

Figure 6: *Lindstrom and Isenberg range decoder*

```
// This is the only place words are read from the stream
decode(TotalRange) {
  return (unalignedWord(pos) − low)
          / (range/=TotalRange);
}

update(SymbolLow, SymbolRange) {
  low += SymbolLow∗range;
  range ∗= SymbolRange;

  // figure out haw many bits to shift
  uint bitTest = low ^ (low+range);
  uint bitShift = 24 − (firstbithigh(bitTest) & ∼0x7);
  uint rangeShift = 16 ∗ (range <= (0xffff>>bitShift));
  bitShift += rangeShift;

  // update stream state for new bytes
  pos += bitShift >> 3;
  low <<= bitShift;
  range = rangeShift ? (∼low+1) : (range<<bitShift);
}
```

Figure 7: *GPU range decoder*

(SSIM) [WBSS04] is a good luminance comparison metric, but is not suited for use on RGB images, since it is designed to use luminance and contrast to extract structure from a luminance-only image. SHAME-II [PH09] is a recent color image quality metric with reasonable correlation to existing human subject user studies. It combines color differencing with contrast sensitive filtering and hue-angle weighting. Both are full-reference comparison metrics which compare a distorted image to a reference image. For luminance and structure comparisons, we prefer SSIM, but for color differences, we supplement SSIM with SHAME-II. For a survey and evaluation of other metrics, see Sheikh et al. [SSB06].

Most of our game color textures are RGBA. SSIM and SHAME-II cannot measure differences in alpha, while mean square error or PSNR can measure those differences, but fail to adequately capture the impact of changes in alpha on the final image. The same is true for other non-color data like normal maps, tangent maps, opacity or blur maps, etc. The most reliable method of comparing compressed texture quality is a perceptual metric like SSIM or SHAME-II applied to rendered images that use the compressed texture data. We present two forms of comparison to evaluate our result. For comparison against other (largely RGB, single-image) compression methods, we use the Kodak Image Suite [Fra10] standard image set. These comparisons are somewhat biased against our method since they do not include non-color data, nor encoding of the full MIP chain. None the less, they do provide a common baseline comparable to other work. The second comparison is for rendered leader scenes within Civilization V. This provides the true measure of in-game image quality and scene load times.

Standard Image Suite Results: Figure 9 plots comparison metric vs. compression ratio for a set of fixed bit-rate hardware compression algorithms, two quality settings for JPEG2000, and our VBR algorithm with compression settings in Figure 11. SSIM is 1 for an exact luminance match, with decreasing values indicating worse structural quality. For SHAME-II, a value of 0 is an exact color match with increasing values for worse color quality.

The left two plots of Figure 9 show comparison metrics vs. compression ratio on all of the Kodak Image Suite images [Fra10]. The VBR v1 variant achieves as good or better compression ratios than BC7, but with better SSIM quality (greater than 0.9985). It trades a little in color quality, but achieves compression ratios as high as 6.62:1. All of the VBR variants maintain SSIM quality greater than 0.99 and color quality no worse than the fixed-rate algorithms while achieving compression ratios as high as 17.07:1.

The right two plots of Figure 9 show the comparison metrics vs. compression ratio on the ‘Parrots’ image. Lossless VBR achieves a compression ratio of 3.6:1 while reproducing the exact raw texture. VBR v2 achieves a compression ratio of 10.75:1, over 1.79 times that of DXT1 and 2.5 times BC7, while still having better SSIM quality. It does trade some color quality compared to BC7 for this higher compression ratio, but the color quality is still the second best and is only about 0.536 worse than BC7.

Figure 12 shows cropped reference and closeup compressed images. Notice the distinct blocking artifacts in DXT1 and BC7 as compared to VBR v3. Also, the yellow and black area has a loss of crispness and color quality in the DXT1 and BC7 compared to the VBR images.

Figure 13 is a study of quality and compression rate for a range of lossy compression settings, dropping 0-3 bits and 0-1 levels of luminance, and 2-4 bits from the chrominance channels. Each horizontal band in the top plot has differing loss chrominance for a particular luminance setting. These bands show that loss in luminance affects SSIM quality while loss in chrominance has no effect on SSIM, but does increase compression ratio. The bands are labeled according to the combination of luminance bits and levels dropped – Lum-BxLy drops x bits and y levels.

The bottom plot of Figure 13 shows that SHAME-II quality is affected by loss in both luminance and chrominance. The brown, red, and green points all drop three luminance bits, and two, three, and four chrominance bits respectively. The blue, magenta, and cyan points all drop four luminance bits, and two, three, and four chrominance bits respectively. The '$+$' data points drop no luminance levels, while the '$\circ$' data points drop one luminance level.

Game Results: We have compressed the textures for world leaders in Sid Meier's Civilization® V. Since the game can be played at HD resolutions, texture sizes range up to 2048x2048. The leaders include textures for diffuse color, specular color and exponent, normals, tangents, transparency and masking, and skin blur factors. For low-end hardware, textures are decompressed on the CPU at a reduced resolution and paged into the GPU after the leader scene is already playing. For DX11-class hardware, all leader textures are kept resident on the GPU in compressed form and decompressed when needed. Statistics on the leaders are shown in Figure 8.

Compression rates vary by leader from about 10:1 to almost 20:1. Most textures were compressed with the v2 variant of Figure 11, with some individual textures hand-tweaked to use less lossy compression. Load times for both VBR and DXT are shorter than would be expected for a typical user due to the high RPM disk on our test system. The total set of textures is almost 2 GB. Even in DXT form, the full set of textures is almost half a gigabyte. In VBR compressed form, these 18 leaders only take 142 MB, so we can keep all of them resident on the GPU. Decompressing the texture set for any leader takes from about 50 to 150 ms. If we include the additional expansion leaders (22 in all), the total is 2.25 GB of uncompressed texture, but only 180 MB with VBR compression.

6. Conclusions

We have presented a VBR image compression algorithm capable of lossless compression with average compression rate for RGBA textures approaching that of current widely used fixed bit-rate lossy compression algorithms, and lossy compression rate averaging better than 12:1. This compression algorithm is based on a difference of MIP levels, and naturally reproduces the MIP levels given. This decouples the choice of MIP filter from the compression scheme, and seamlessly allows artist tweaked MIP levels. Using our algorithm, a 2048x2048 MIP texture can be decompressed on the GPU in under 10 ms, and the average decompression rate over a range of real game textures is under 3.2 ms.

The compression method in this paper was developed for a AAA game targeting a multi-core PC with a DX11 GPU at HD resolution. Like most games in this class, we have gigabytes of total texture, but use no more that 5-10% of it in any one scene. For Civilization V, we have the added constraint that we need to be able to switch to any arbitrary scene at any time, making typical dynamic loading solutions untenable. With VBR compression, we can store the working set for one scene, plus the compressed form of the rest of the leader textures locally. Our method will be most useful for games targeting high-end platforms, with high quality standards, where the uncompressed current scene and compressed global texture set can live on the GPU. It could also be useful in for a game with an even larger total working set but less random scene access, with a multi-level texture caching scheme of disk or memory storage, local GPU compressed storage, and GPU decompressed storage. As capabilities from high-end PCs move to lower-end PCs and consoles, we expect game quality standards to increase, and even more games to need advanced compression methods to manage their texture resources.

We have described in detail the design decisions behind our specific VBR algorithm, but a key contribution of this paper is the insight that there is value to fast GPU decompression that is not part of the rendering process. Existing GPU texture compression is handicapped by having to support decompression during random texel access while rendering. By decoupling the decompression from the rendering, the GPU is capable of significantly better compression rates and quality. Even though the working set of textures is full size, this approach vastly increases the total amount of texture that can be stored on the GPU in compressed form to be quickly decompressed into a working texture when needed.

In addition, though the combined load and decompress time for VBR is about the same as the load time for DXT, the load time is only about a quarter of the total. As GPU computational power increases, VBR textures may become a useful tool to accelerate streaming of textures during game play.

We have many ideas on how to improve the compression quality, compression rate, and performance in future work. SSIM, SHAME-II or similar color metric could drive the loss decisions per block rather than relying on the eye of a programmer or artist to choose a good setting. Additional color transforms could improve the compression rate and quality for non-color textures. A smarter probability model for the statistics table could reduce the per-block overhead. We might be able to rearrange the compression order to allow more efficient consolidated GPU writes. Finally, knowing the expected probabilities for each of the probability classes, it would be worth further investigating dynamic probability estimation, since a dynamic estimator would do away with the need for a static statistics table if one can be found that is fast enough and sufficiently accurate.

References

[BAC96] BEERS A., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 373–378. 3

[BS99] BUCCIGROSSI R., SIMONCELLI E.: Image compression via joint statistical characterization in the wavelet domain. *IEEE Transactions on Image Processing 8*, 12 (Dec. 1999), 1688 –1701. 2

[Cas07] CASTAÑO I.: *High Quality DXT Compression Using CUDA*. Tech. rep., NVIDIA, February 2007. 2, 3

[Eli75] ELIAS P.: Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21*, 2 (Mar. 1975), 194 – 203. 3

[Fra10] FRANZEN R.: Kodak lossless true color image suite, 2010. http://r0k.us/graphics/kodak/. 2, 7

[Hec82] HECKBERT P.: Color image quantization for frame buffer display. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1982), ACM Press, pp. 297–307. 3

[HS00] HAO P., SHI Q.: Comparative study of color transforms for image coding and derivation of integer reversible color transform. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on* (2000), vol. 3, pp. 224 –227 vol.3. 3, 5

[Huf52] HUFFMAN D.: A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers 40*, 9 (1952), 1098–1101. 3

[HV94] HOWARD P., VITTER J.: Arithmetic coding for data compression. *Proceedings of the IEEE 82*, 6 (June 1994), 857 –865. 3

[IM06] INADA T., MCCOOL M.: Compressed lossless texture representation and caching. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Vienna, Austria, 2006), ACM, pp. 111–120. 3

[INH99] IORCHA K., NAYAK K., HONG Z.: *System and Method for Fixed-rate Block-based Image Compression with Inferred Pixel Values*. Tech. Rep. 5956431, US Patent, 1999. 3

[LC10] LINDSTROM P., COHEN J.: On-the-fly decompression and rendering of multiresolution terrain. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), I3D '10, ACM, pp. 65–73. 3

[LI06] LINDSTROM P., ISENBURG M.: Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on 12*, 5 (2006), 1245 –1250. 4, 6

[LW99] LEKATSAS H., WOLF W.: Random access decompression using binary arithmetic coding. In *Data Compression Conference, 1999. Proceedings. DCC '99* (Mar. 1999), pp. 306 –315. 2

[MA05] MOFFAT A., ANH V.: Binary codes for non-uniform sources. In *Proceedings of the Data Compression Conference* (Washington, DC, USA, 2005), DCC '05, IEEE Computer Society, pp. 133–142. 3

[MAMS06] MUNKBERG J., AKENINE-MÖLLER T., STRÖM J.: High quality normal map compression. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware* (New York, NY, USA, 2006), ACM, pp. 95–102. 3

		Size (MB)			Load time (ms)			Decompress	Compression	Quality	
Leader	Textures	Raw	DXT	VBR	Raw	DXT	VBR	on GPU (ms)	Ratio	SSIM	SHAME-II
Al-Rashid	33	110.6	30.4	9.1	615.9	176.5	64.7	107.8	12.1:1	0.9954	0.783
Alexander	39	161.7	40.7	12.5	838.5	200.7	49.5	117.7	12.9:1	0.9911	0.652
Askia*	21	93.1	24.4	9.9	462.0	96.5	112.8	76.3	9.4:1	0.9849	2.163
Augustus*	20	65.7	15.2	4.7	374.9	44.4	8.8	59.1	14.0:1	0.9908	1.453
Bismarck	24	96.4	24.5	7.9	511.5	108.4	17.5	68.3	12.3:1	0.9968	0.599
Catherine	31	163.7	41.1	8.9	925.8	198.9	26.5	116.8	18.5:1	0.9958	0.478
Darius	16	37.9	9.5	2.8	110.3	20.9	6.1	44.0	13.6:1	0.9950	0.913
Elizabeth	22	91.3	23.6	8.4	419.1	85.8	16.1	78.1	10.9:1	0.9921	1.043
Gandhi	23	91.5	24.3	6.1	377.8	107.2	12.8	69.9	15.1:1	0.9901	0.679
Hiawatha	29	110.8	27.6	8.9	465.1	98.4	18.6	88.3	12.5:1	0.9962	1.148
Montezuma*	32	109.2	27.7	11.1	431.8	91.8	23.2	106.7	9.9:1	0.9870	1.822
Napoleon	27	167.7	42.0	14.9	888.1	188.5	44.4	104.3	11.3:1	0.9935	0.539
Oda	27	58.5	15.2	4.8	160.9	39.5	9.3	75.5	12.2:1	0.9940	0.616
Ramesses	30	44.7	12.2	5.1	184.0	32.1	9.1	83.9	8.8:1	0.9953	2.499
Ramkhamhaeng	17	105.7	27.9	8.6	469.3	100.6	15.6	70.4	12.4:1	0.9943	0.464
Sulieman	39	98.4	25.2	8.5	387.0	89.7	18.9	109.2	11.6:1	0.9935	1.203
Washington*	22	80.5	21.6	5.0	298.2	70.4	12.4	68.9	16.0:1	0.9904	1.158
Wu	19	78.9	20.5	5.0	302.7	55.2	9.7	57.3	15.8:1	0.9889	1.690
Totals	471	1766.5	453.7	142.0	8223.0	1805.6	475.9	1502.4	12.4:1		

Figure 8: *Texture statistics for the original 18 Civilization® V leaders (CPU: Intel Xeon X5460 Quad Core 3.16 GHz; GPU: NVIDIA GTX 480; Disk: 15,000 RPM Hitachi UltraStar 15K300). Leaders marked * have particle systems disabled for image comparison. Textures is the total number for that leader. Raw textures are uncompressed, using only the appropriate number of channels for their data. DXT textures use BC3, BC4 or BC5 as appropriate for the texture. The VBR size includes both compressed data and index. Load times are the time to load textures or compressed data from the disk to the GPU. Decompress is the time to decompress the textures on the GPU from compressed data already resident there. Compression ratio is VBR relative to RAW. Best Mean SSIM=1 and best SHAME-II=0.*

[Mar79] Martin G.: Range encoding: An algorithm for removing redundancy from a digitised message. In *Proceedings of the Video and Data Recording Conference (Southampton, UK, July 24-27, 1979)* (1979). 3

[MCH*06] Munkberg J., Clarberg P., Hasselgren J., , Akenine-Möller T.: High dynamic range texture compression for graphics hardware. *ACM Trans. Graph. 25* (July 2006), 698–706. 3

[PH09] Pedersen M., Hardeberg J.: A new spatial hue angle metric for perceptual image difference. In *Computational Color Imaging* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 81–90. 6

[RAI06] Roimela K., Aarnio T., Itäranta J.: High dynamic range texture compression. *ACM Trans. Graph. 25* (July 2006), 707–712. 3

[RL81] Rissanen J., Langdon, Jr. G.: Universal modeling and coding. *IEEE Transactions on Information Theory 27*, 1 (Jan. 1981), 12 – 23. 2

[SAM05] Ström J., Akenine-Möller T.: iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *GH '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (New York, NY, USA, 2005), ACM, pp. 63–70. 3

[SCE01] Skodras A., Christopoulos C., Ebrahimi T.: The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine 18*, 5 (Sept. 2001), 36–58. 2, 3, 4, 5

[Sha93] Shapiro J.: Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing 41*, 12 (Dec. 1993), 3445 –3462. 3, 4, 5

[SLT*09] Sun C., Lok K., Tsao Y., Chang C., Chien S.: CFU: multi-purpose configurable filtering unit for mobile multimedia applications on graphics hardware. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 29–36. 3

[SP07] Ström J., Pettersson M.: ETC2: texture compression using invalid combinations. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 49–54. 3

[SSB06] Sheikh H., Sabir M., Bovik A.: A statistical evaluation of recent full reference image quality assessment algorithms. *IEEE Transactions on Image Processing 15*, 11 (2006), 3440 – 3451. 6

[Sub99] Subbotin D.: Carryless rangecoder, 1999. http://search.cpan.org/src/SALVA/Compress-PPMd-0.10/Coder.hpp. 6

[vWn07] van Waveren J., no I. C.: *Real-Time YCoCg-DXT Compression*. Tech. rep., id Software, September 2007. 2, 3

[Wal92] Wallace G.: The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics 38*, 1 (Feb. 1992), xviii–xxxiv. 2, 3, 5

[WBSS04] Wang Z., Bovik A., Sheikh H., Simoncelli E.: Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600 –612. 6

[Wil83] Williams L.: Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), vol. 17, ACM Press, pp. 1–11. 2

[ZL77] Ziv J., Lempel A.: A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (May 1977), 337 – 343. 3

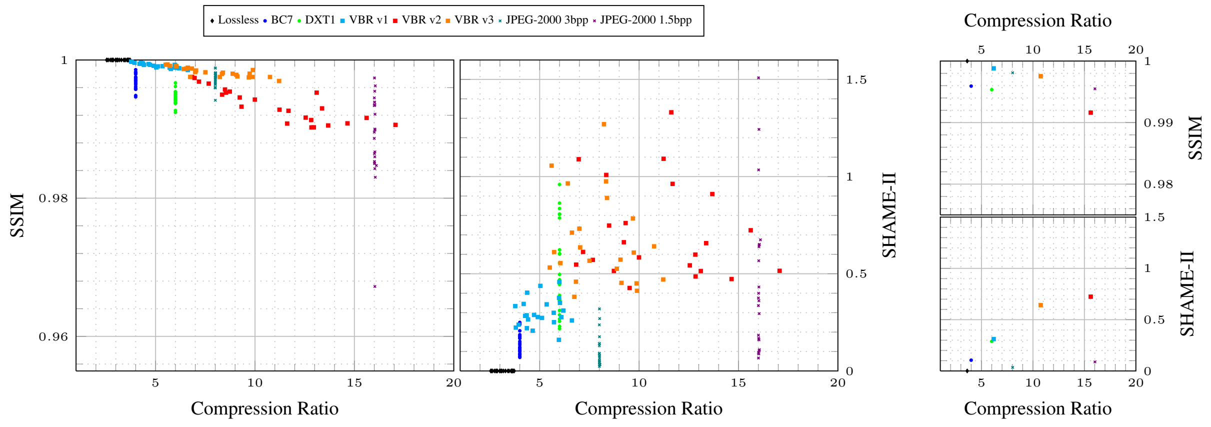


Figure 9: *Quality versus Compression Rate. The left two plots show all Kodak Image Suite images. The right two plots just show the 'Parrots' image. For the SSIM metric, an exact pixel match has a value of 1, with decreasing values indicating worse structural quality. For the SHAME-II metric, a value of 0 indicates an exact color match (no color difference) with increasing values indicating worse color quality.*

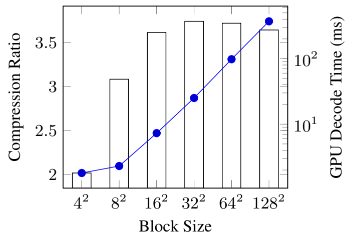


Figure 10: *Lossless compression ratio (black bars) and GPU decode speed (blue line) of varying block sizes with a 512x512 texture ('Parrots') on an NVIDIA GTX480.*

	Bits Dropped			Levels Dropped		
Name	**Y**	**Cb**	**Cr**	**Y**	**Cb**	**Cr**
Lossless	0	0	0	0	0	0
VBR v1	0	2	2	0	0	0
VBR v2	2	2	2	0	1	1
VBR v3	1	2	2	0	1	1

Figure 11: *Compression parameters for tested VBR variants.*

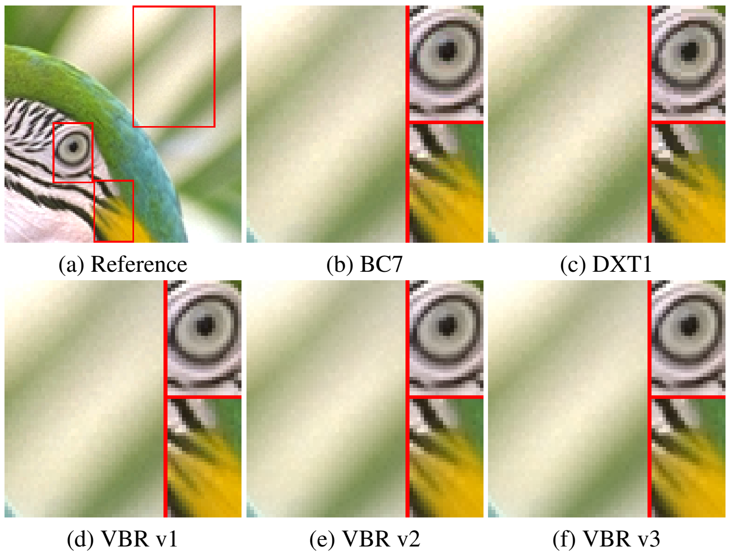


Figure 12: *Crops of the "Parrots" image.*

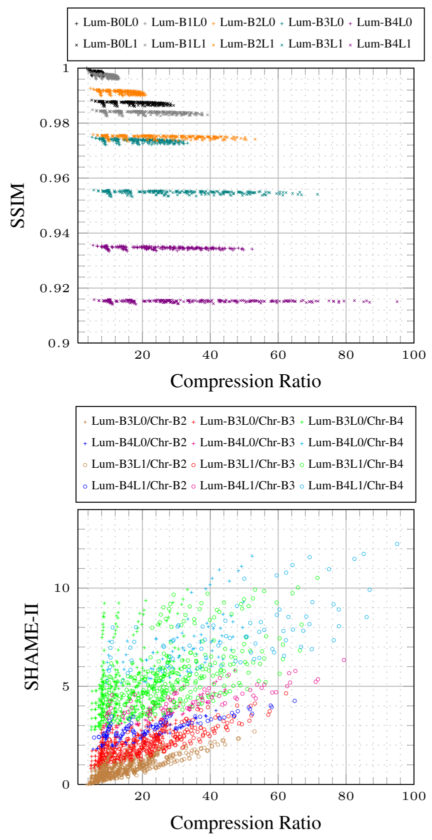


Figure 13: *Quality versus Compression Rate for 2,250 different compression settings.*