Graphics Hardware (2007)
Timo Aila and Mark Segal (Editors)

# Exact and Error-bounded Approximate Color Buffer Compression and Decompression

Jim Rasmusson[1,2] Jon Hasselgren[1] Tomas Akenine-Möller[1]

[1]Lund University [2]Ericsson Research

**Abstract**

*In this paper, we first present a survey of existing color buffer compression algorithms. After that, we introduce a new scheme based on an exactly reversible color transform, simple prediction, and Golomb-Rice encoding. In addition to this, we introduce an error control mechanism, which can be used for approximate (lossy) color buffer compression. In this way, the introduced error is kept under strict control. To the best of our knowledge, this has not been explored before in the literature. Our results indicate superior compression ratios compared to existing algorithms, and we believe that approximate compression can be important for mobile GPUs.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Image Generation; frame buffer operations

## 1. Introduction

The expected yearly performance increase in terms of bandwidth and latency of DRAM is about 25% and 5%, respectively. At the same time, the expected increase in computing capability of a processor is about 71% every year [Owe05]. Due to this, the gap between memory speeds and computational resources is steadily increasing. For desktop computer GPUs this is mitigated to some extent by wider and wider DRAM buses, a "luxury" that is basically not available for mobile devices. Hence, compression techniques aimed at saving memory bandwidth for GPUs are becoming increasingly important, especially for mobile GPUs. Examples include vertex compression, texture compression, depth buffer compression, and color buffer compression.

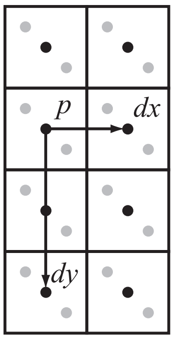
In this paper, we focus on *color buffer compression and decompression*. The purpose of our work is to provide the reader with a state-of-the-art report of existing algorithms, which are currently only available in the form of patents, and to introduce *new* algorithms.

In terms of new algorithms, we start by introducing a new *exact* algorithm, which first uses a reversible color transform, and then applies Golomb-Rice coding after using a simple predictor. Second, we experiment with *approximate* color buffer compression. The motivation here is that we can accept, for example, lossy video compression (e.g., MPEG), and approximate rendering using precomputed radiance transfer with spherical harmonics or wavelets. Even just after executing the pixel shader, conversion from floating point to 8-bit integers is done, and this is actually a type of lossy compression (truncation). In addition, most texture compression schemes are also lossy. Hence, one could ask whether and how this can be applied to color buffers as well.

This may sound dangerous, but we show that it is possible by developing error-bounded algorithms to keep the visual artifacts under precise control, and to avoid so called *tandem* compression artifacts, which may arise due to several passes of sequential lossy compression. We emphasize that approximate, i.e., lossy, color buffer compression is not always desired. For example, in GPGPU computations for fluid simulation, exact results is of uttermost importance, and in such cases, we suggest that the programmer can turn off this feature. However, for a GPU in a mobile phone, where it is important to reduce memory accesses over external buses [AMS03], it can be very convenient to enable approximate compression as this can increase the use time on a battery charge at a cost of slight image degradation. The major advantage of approximate compression is that higher compression can be obtained, which reduces memory bandwidth usage compared to lossless, i.e., exact, color buffer compression.

## 2. State-of-the-art Color Buffer Compression

In this section we summarize the color buffer compression algorithms we have found in patent databases. A summary of existing depth buffer compression schemes is already available [HAM06]. The reader is referred to Section 2 of this paper for an overview of the depth buffer architecture, which is almost identical to the color buffer architecture. This paper describes the general methodology for selecting and tracking what compressor to use for a specific tile, and how to handle tiles that cannot be compressed.



| | |
|---|---|
| Flag: | 1 bit (set to 0) |
| *p*: | 24 bits (R8G8B8) |
| *dx*: | 16 bits (R6G5B5) |
| *dy*: | 18 bits (R6G6B6) |
| Pixel Deltas: | 12 bits (R4G4B4) |
| Alpha: | 8 bit |

| | |
|---|---|
| Flag: | 1 bit (set to 1) |
| *p*: | 24 bits (R8G8B8) |
| *dx*: | 12 bits (R4G4B4) |
| *dy*: | 14 bits (R5G5B4) |
| Pixel Deltas: | 15 bits (R5G5B5) |
| Alpha: | 1 bit |

**Figure 1:** *Color plane compression. For this example, two samples (gray circles) per pixel are used, and these are collapsed (black circles). Each tile of* $2 \times 4$ *pixels are encoded together. A prediction plane is computed from the three reference pixels (indicated by p, dx, and dy), and the remaining pixels are stored as deltas between the prediction from the reference plane and the actual color of the pixel. The tables to the right show two suggested bit-allocations.*

### 2.1. Multi-Sampling Compression

In this section, we present an algorithm for compression of color buffers with multi-sampling. In the following, we assume that $n$ samples are used per pixel.

Elder explains that due to multi-sampling, samples inside a pixel often share the exact same color, and this is an opportunity for compression [Eld06]. If all samples inside a pixel share the same color, then it suffices to flag this mode, and store only one color instead of $n$ colors. Another common case is when a triangle edge cuts through a pixel. In such a case, we can store two colors, and a one-bit index per sample to “point” at one of these colors. Elder also suggests that this compressed format is used inside the GPU as well. This has a number of benefits, such as using fewer operations when blending and during reconstruction of the final pixel color.

The RealityEngine [Ake00] used a similar coverage mask approach internally in their fragment pipelines. However, the depth and color-values were decompressed prior to frame buffer operations, and consequently some of the performance benefits were lost.

### 2.2. Color Plane Compression

Another example of exploiting multi-sampling color redundancy is the method described by Molnar et al. [MSM*04]. In a first step, they collapse pixels with identical sample colors, similarly to Elder’s work [Eld06]. When using four samples per pixel, this by itself gives sufficient compression to reach their predetermined bit-budget. However, in the case of two samples per pixel, they need to compress the data by an additional factor of two.

To this end, they introduce a plane compression mode. A predictor plane is computed from three collapsed reference pixels, as shown in Figure 1. This plane is stored with varying accuracy, and the remaining pixels are stored as differences between the actual pixel value, and the value predicted by the plane at that pixel. Bit allocations for the plane and delta values are detailed in Figure 1. The observant reader may note that these allocations only use 127 bits. The remaining bit is used to flag that a tile is in cleared state, which saves some bandwidth when clearing the color buffer.

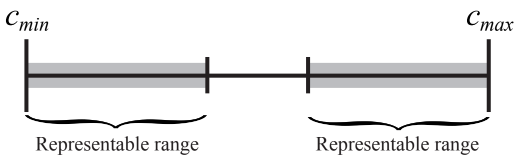


**Figure 2:** *Color offset compression, when using the min and max colors as references. Note that the width of the representable color intervals vary with the number of bits allocated for the per pixel offsets.*

### 2.3. Offset Compression

Some of the methods targeting depth buffer compression can also be used for color buffer compression. A good example of this is the offset compression method proposed by Morein and Natale [MN03].

The method compresses a tile by identifying a number of reference values. All pixels in the tile are then coded as an index to a reference value, and componentwise color offsets from that reference value. A typical implementation is to chose the minimum and maximum colors as reference values, similarly to depth offset compression. We can then represent the color range shown in Figure 2.

It should be noted that depth offset compression has one advantage over color offset compression, which is that the min and max depth values are already stored in on-chip memory for Zmax- and Zmin-culling, so we do not have to store the reference values explicitly. This makes offset compression slightly less efficient for color data than for depth data.

### 2.4. Entropy Coded Pixel Differences

Van Hook suggests compression schemes based on entropy coding of pixel differences [Hoo06]. First, he computes the componentwise pixel differences. Although the exact procedure is not specified, the patent indicates that different traversal orders may affect the magnitude of the pixel differences (and in the end the efficiency of the algorithm). This indicates that the pixel differences actually are the differences between the current pixel and the previously traversed pixel. The suggested implementation uses either horizontal or vertical scanline traversal of the tile, based on what gives the best compression.

It is well known that differences between adjacent pixels often have small magnitudes due to the continuous nature of images. Van Hook therefore proposes a variable bit length coding of the differences, which he refers to as *exponent encoding*. The general idea is to represent a value as $s(2^x - y)$, where $y \in [0, 2^{x-1} - 1]$, and $s$ is a sign bit. In order to compress this value, $x+1$ is stored using *unary encoding*, which simply amounts to storing $x+1$ bits set to one followed by a terminating zero-bit. For example, $x+1 = 4$ is encoded as $11110_b$. Normal binary encoding is used for $s$ and $y$. The reason for encoding $x+1$ instead of $x$ is that the encoding is not capable of representing a zero value. This special case is flagged when $x+1$ is set to zero.

To illustrate the exponent coding with an example, assume we want to encode the value $\pm 5 = \pm(2^3 - 3)$. The unary encoding of $x+1$ is again $11110_b$. The $y$-value will be in the range $[0, 2^2 - 1]$, so it can be represented using two bits with binary encoding, which gives us $11_b$. Finally, we need to store the sign bit $s$ in one bit. The final encoded value therefore becomes $11110s11_b$.

Exponent coding requires a very large amount of bits for values with large magnitudes. Van Hook therefore suggests using exponent coding only for difference values in the range $[-32, 32]$, remaining values are encoded using 16 bits, the first 8 bits must be set to $11111110_b$ to separate the exponent coded, and binary coded values. The full encoding is shown in the following table.

| Code | Representable value |
|---|---|
| $0_b$ | 0 |
| $10s_b$ | $\pm 1$ |
| $110s_b$ | $\pm 2$ |
| $1110sx_b$ | $\pm[3,4]$ |
| $11110sxx_b$ | $\pm[5,8]$ |
| $111110sxxx_b$ | $\pm[9,16]$ |
| $1111110sxxxx_b$ | $\pm[17,32]$ |
| $11111110xxxxxxxx_b$ | 8-bit absolute value |

A strong feature of this scheme is that it allows for adaptive bit rate inside a tile.

## 3. A New Exact Color Buffer Compression Algorithm

In this section, we present a new exact, i.e., lossless, color buffer compression method. The algorithm operates on tiles, which are typically $8 \times 8$ pixels.

Note that the color buffer needs to be sent to the display in uncompressed form. Hence, there is a direct benefit from having color buffer decompression implemented in the display controller, or in any of the hardware processing blocks prior the display controller. For example, most mobile phones already have some type of display processing block which provides features like scaling, overlay, color depth transform, etc. A color buffer decompressor would fit there as well.

### 3.1. Reversible Color Transforms

Our new algorithms share the fact that they operate in a luminance-chrominance color space instead of the standard RGB color space. It is well-known in image and video compression that this typically enables more efficient compression due to the decorrelation of the RGB channels.

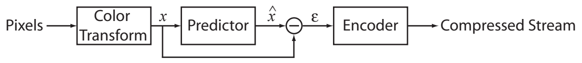


**Figure 3:** *Overview of our compression algorithm.*

In addition, it also enables the use of slightly different compression schemes for the luminance and the chrominance components [Per99]. This could potentially be useful since rendered gaming scenes often provide most details and dynamics in the luminance component.

Since we need lossless compression, the color space transform needs to be exactly reversible. We have chosen the reversible color transform *RGB* to $Yc_oc_g$ introduced by Malvar and Sullivan [MS03]. Transforming from *RGB* to $Yc_oc_g$ is done as shown below:

$$\begin{aligned} c_o &= R - B \\ t &= B + (c_o >> 1) \\ c_g &= G - t \\ Y &= t + (c_g >> 1). \end{aligned} \quad (1)$$

Transforming back is as simple:

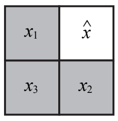
$$\begin{aligned} t &= Y - (c_g >> 1) \\ G &= c_g + t \\ B &= t - (c_o >> 1) \\ R &= B + c_o. \end{aligned} \quad (2)$$

Note that if the *RGB*-components are stored using $n$ bits each, the $Y$-component will require $n$ bits, and the chrominance components $n+1$ bits. So the price to pay for having a lossless reversible color transform is a small data expansion of two bits. Malvar and Sullivan also showed that this transform in certain video contexts can provide better compression ratios compared to *RGB* and $Yc_rc_b$. Note also that the commonly used standard $Yc_rc_b$ transform is not, in general, reversible without loss.

An alternative color transform to $Yc_oc_g$ would be the exactly reversible component transformation (RCT) from the JPEG2000 standardization [JPE00]. We have empirically concluded that, for our algorithm, these color transforms are roughly equal in terms of efficiency. Our experiments also showed that the use of these color transforms improved the compression rate of our algorithm by about 10% compared to an implementation in *RGB* space. We therefore think that using a color transform is well motivated.

### 3.2. The Algorithm

Our lossless compression algorithm is inspired by the LOCO-I algorithm [WSS96]. In our implementation, we work on $8 \times 8$ pixel tiles, but it should be straightforward to apply it to other tile sizes as well. The flow of our algorithm is illustrated in Figure 3. In a first step, we predict the color of each pixel based on neighbors which will be decompressed prior to the current pixel. The predicted colors are then subtracted from the actual colors to produce error residuals. Just like the differences used by Van Hook, these residuals are generally of small magnitude, and we entropy encode them using Golomb-Rice coding. Next, we describe the details of these steps.



We use the same predictor as Weinberger et. al [WSS96]. The color, $\hat{x}$, of a pixel is predicted as specified by Equation 3 below, and based on the colors of its three neighbors shown in the figure to the right. Note that the two first cases of the equation perform a very limited form of edge detection, in which case the color is predicted based on just one of the neighbors.

$$\hat{x} = \begin{cases} min(x_1, x_2), & x_3 \geq max(x_1, x_2) \\ max(x_1, x_2), & x_3 \leq min(x_1, x_2) \\ x_1 + x_2 - x_3, & \text{otherwise.} \end{cases} \quad (3)$$

For the pixels along the lower and left edge of a tile, we only have access to one of the neighbors. In that case, we simply use the color of that neighbor as the predicted color. In addition, we use the constant zero to predict the value of the lower left pixel in the tile. The effect is that the first error residual will be given the same value as the lower left pixel.

Given these predicted values, we compute error residuals and wish to encode them using as few bits as possible. The residuals are generally of small magnitude, mixed with relatively unfrequent large values. These latter values are typically found for discontinuity edges, or where the behavior of the predictor does not match the structure of the image. We encode the residuals using a Golomb-Rice [Ric79] coder, which is a variable bit-rate coding method similar to the exponent coding described in Section 2.4.

In Golomb-Rice encoding, we encode a residual value, $\varepsilon = x - \hat{x}$, by dividing it with a constant $2^k$. The result is a quotient $q$ and a remainder $r$. The quotient $q$ is stored using unary coding, and the remainder $r$ is stored using normal, binary coding using $k$ bits. To illustrate with an example, let us assume that we want to encode the values 3,0,9,1 and assume we have selected the constant $k = 1$. After the division we get the following $(q, r)$-pairs: $(1,1), (0,0), (4,1), (0,1)$. As mentioned in Section 2.4, unary coding represents a value by as many ones as the magnitude of the value followed by a terminating zero. The encoded values therefore becomes $(10_b, 1_b), (0_b, 0_b), (11110_b, 1_b), (0_b, 1_b)$ which is 13 bits in total.

In our compression algorithm, we compute the optimal Golomb-Rice parameter $k$ for each $2 \times 2$ pixel sub-tile using an exhaustive search. We also detect the special case, when the quotients of all values in the sub-tile is zero. This gives us the opportunity of removing the terminating zero-bit, which would otherwise be introduced by the unary coding.

We empirically examined the frequencies of different values of $k$, and when the special case was used. Our results indicate that $k$ is relatively evenly distributed in the range [0,6] while the special mode was almost only used in the case $k = 0$, which is equivalent to that the whole sub-tile consists only of zero values. With this in mind, we encode each $2 \times 2$ sub-tile as a 3-bit header in which we store the value of $k$. If $k = 7$ the whole sub-tile is zero and we store no more data, and in the other cases the header is followed by the Golomb-Rice coded componentwise residuals.

We present the results of our lossless compression algorithm in Section 5.

**Discussion** Using exhaustive search to find the best Golomb parameter may seem too expensive for a real-time compression algorithm. However, we want to point out that the search is limited to 8 unique cases that can be evaluated in parallel. Furthermore, it is very inexpensive to evaluate the size of a value after it has been Golomb encoded. This requires just one shift and one addition.

One might also argue that the cost of a variable bit rate compressor is too high for practical use, but we believe it is realizable. Trying to encode a full 2048 bit vector in a single cycle is too expensive, but if we limit ourselves to compressing one sub-tile per cycle we get a more manageable 0-128 bits to write. A tile would then take a total of 16 clock cycles to compress, a delay that could most likely be hidden using pre-fetching [IEP98]. To put this figure in perspective, the expected memory latency reported in the CUDA programming guide [NVI] is 200-300 cycles.

## 4. Error-bounded Approximate Compression

The obvious reason to use lossy (approximate) compression algorithms is that you are allowed to throw away information in the compressed signal, and this can make for substantially higher compression ratios. If done well, the visual impact can be marginal. Since a rather big amount of power is required to drive the capacitances of the buses to off-chip memory, battery-driven mobile devices, in particular, will benefit from lossy buffer compression. It should be noted that for both mobile devices and for desktop GPUs, we may also get higher performance due to better utilization of the memory bandwidth resources.

As argued in the introduction, lossy techniques are used in many different algorithms for graphics, video, and imaging. The prime example is probably digital TV, where we put up with pretty poor approximations in the encoded video stream. It is therefore a bit surprising that there has been no documented attempts to use approximate compression for the color buffer.

The reason for this might be that it is possible to get *un-*

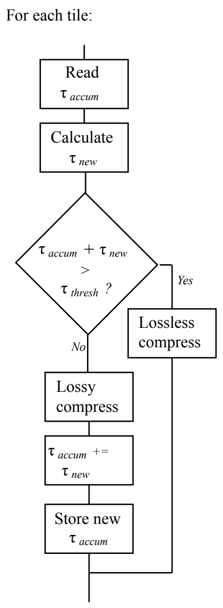
*bounded*[†] errors. This can occur when lossy compression (LC) is applied several times, e.g., once per triangle written to a tile. See Figure 4, where the concept of *tandem compression* is illustrated. To counteract this, we need an error-bounded algorithm with precise control of the accumulated error. This is the topic of the next subsection.

Note that buffer compression & decompression must be symmetric, i.e., execute in about the same amount of time, since these procedures run in real time inside the GPU. This means that the majority of all (lossy) texture compression schemes immediately disqualify, since compression often takes several seconds or even minutes.

## 4.1. The Error Control Mechanisms

To guarantee that the introduced error stays within bounds, we need to gauge and track the accumulated error in the image being rendered.

Our approach is to calculate and update an accumulated error measure, $\tau_{accum}$, per tile, as illustrated to the right. As an example, we could use the accumulated mean square error. This measure is stored together with the compressed tile parameters. For even more precise control, more than one error measure can be tracked and stored. For example, it may make sense to track and store a maximum error level, which is normally a more conservative error metric than the mean square error metric. When the accumulated tile error measure has reached a configurable upper limit (threshold), $\tau_{thresh}$, the compression stage reverts to lossless compression only in the following compression steps.



This can be done by having a conditional lossy compression stage, meaning that each time an error is about to be introduced, e.g., due to when sub-sampling or quantization, we test if the updated accumulated error measure exceeds a configurable threshold $\tau_{thresh}$. If it is still less than the threshold, we use the approximation. Otherwise, we revert to lossless compression (and do not update $\tau_{accum}$).

Note also that if we have reverted to lossless compression, we can go back to lossy compression if all pixels are written to inside a tile.

[†] Here, we used the term "unbounded" to indicate a maximum error in a value. Assuming eight bits, this happens when an original value of 255 is compressed into 0, for example.

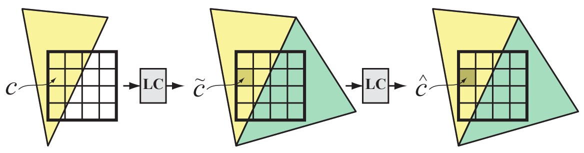


**Figure 4:** *Illustration of tandem compression. Left to right: first a triangle is written to a tile. For one pixel, we track its original color, c. After lossy compression (LC), we obtain an approximation, $\tilde{c}$. However, when a second triangle is written to the tile, $\tilde{c}$ may be compressed again, with another loss of information, so we get yet another color, $\hat{c}$.*

Our approach is conservative, in that the error (in the error metric used) never grows larger than the thresholds. Hence, the introduced errors are bounded, which effectively reduces the visual quality impact (given the configured error thresholds are low enough).

## 4.2. A Lossy Algorithm

We have chosen to track and store the accumulated root mean square error (RMSE) error per tile, $\tau^{rmse}_{accum}$. This measure is quantized to 16 levels and stored together with the compressed parameters as 4 bits per tile. Note that the choice of this error metric also bounds the maximum error in a tile. Assume the threshold is $\tau^{rmse}_{thresh}$, and that we have $n$ pixels in a tile. Some simple calculations gives:

$$\tau^{max}_{thresh} = \sqrt{n} \times \tau^{rmse}_{thresh}. \quad (4)$$

As an example, if $\tau^{rmse}_{thresh} = 2$ with $8 \times 8$ pixel tiles, we have $\tau^{max}_{thresh} = 8 \times 2 = 16$. In a practical implementation it may make more sense to use MSE instead of RMSE, since this avoids the expensive square root. However, that also doubles the number of bits for the accumulated error. Another useful error metric is the sum of absolute differences (SAD).

When it comes to the actual approximation, we have taken a gentle approach and use only (conditional) $2 \times 2$ subsampling of the chrominance components. Higher compression rates can of course be obtained with more "brutal" subsampling, quantization, and other lossy methods.

Since the human visual system (HVS) is more susceptible to errors introduced in the luminance than the chrominance components, we use lossless compression for the luminance and lossy compression for the chrominance components respectively. This results in a good compromise between high visual quality and high compression ratios.

It should be noted here the benefits of utilizing an exactly reversible color transform. This enables the possibility to mix lossless and lossy compression freely in the same compression block. For example, it enables us to have lossless compression of the luminance components and simultaneously have lossy compression of the chrominance components[‡]. When the error threshold is reached, we can revert to lossless-only chrominance compression to effectively stop further error build-up. Furthermore, if a non-exact color transform is used, that would introduce further errors, and the error accumulation mechanism would have to deal with that as well. With our approach, that can be avoided altogether.

Decompression is done in the opposite direction. The sub-sampled chrominance components are up-sampled by simply copying the sub-sampled component to the corresponding components in the $2 \times 2$ quad.

## 5. Results

We have evaluated our compression algorithms using a software based simulation framework, which implements a tile-based triangle rasterizer with a modern color buffer architecture. It also simulates tile caching, using a 1 kB fully associative cache, and implements the color compression algorithms described in this paper. In addition, we used a logging OpenGL driver to record the rendering calls from actual games. This means that our results include the full, incremental, rasterization process of the games. They are not just compressed screenshots.

To benchmark the color compression algorithms, we used the four test scenes shown in Figure 7. The first scene is designed to stress high contrast colors, and the following two scenes are relatively colorful scenes taken from Quake 3 maps.[§] The final scene features complex particle effects with blending, and is taken from the game Quake 4. It should be noted that this scene use blending based on the alpha value stored in the color buffer. Therefore, we compress the full RGBA components for this scene, while we only compress the RGB components for the remaining scenes. This shows that all algorithms are suitable for compressing alpha data as well.

Note that we will refer to each compression algorithm by the names we used in Section 2. See the titles of each subsection.

### 5.1. Exact Compression

The effective compression ratios of the different exact algorithms are presented in Figure 7. We used $8 \times 8$ pixel tiles and variable bit-rate encoding for offset compression, entropy coding, and our algorithm. Variable bit-rate coding comes very natural to our algorithm and entropy coding. For offset compression, we implemented variable bit-rate in the sense that we use no fixed bit allocations for the offsets. We simply use the least amount of bits that is capable of representing the largest offset in the tile.

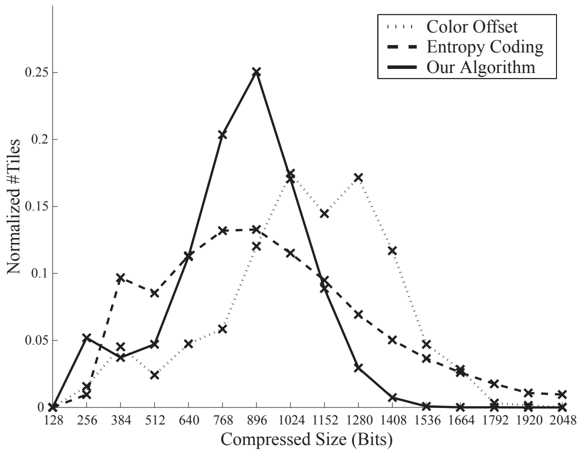


**Figure 5:** *A normalized histogram of the number of tiles that are compressed to a given size (in multiples of 128 bits), using different algorithms. We use $8 \times 8$ pixel tiles, which means that 2048 bits indicate uncompressed tiles. The histogram is based on an average over all our test scenes. Note that our algorithm has a distinct peak, which makes it efficient when only a few compressed sizes can be used. It is of course also essential that the peak is located at good compression ratios, i.e., to the left in this diagram.*

The plane compression algorithm is special in that we used the specified $2 \times 4$ tile size and only the two modes specified in the patent. We would like to emphasize that this algorithm is disfavored in this evaluation since it is so specialized. A generalized version of plane compression may generate better results, but this is left for future work.

We would also like to point out that our measurements (Figure 7) do not take hardware limitations, such as the width of the memory bus, into account. Furthermore, in most hardware implementations, we can only afford a few different compressed sizes, since the compressed size of a tile is typically stored in on-chip memory so that we know beforehand how many memory words to read. In order to measure the algorithms performance with respect to these limitations, we computed the compressed size histograms shown in Figure 5. Note that we have left out plane compression since it is so specialized, and only allows for 128 or 256 bits per tile.

Next, we show how to interpret this diagram with an example. Assume that we allow two fixed sizes for tiles: 1024 bits for compressed and 2048 bits for uncompressed. In this case, the number of tiles compressed to 1024 bits will be the integral from 0 to 1024 over the histogram, while the uncompressed tile will be the integral from 1024 to 2048. Using the histogram, we can easily find the best $n$ sizes for each algorithm using an exhaustive search. In Table 1, we present the best compressed sizes for $n = 1, 2, 3$. Note that an extra uncompressed size always needs to be available as a fallback.

‡ It should be noted that the chrominance errors can spread into the luminance channels due to tandem compression.

§ The maps have been taken from the Quake 3 add-on "Urban Terror".

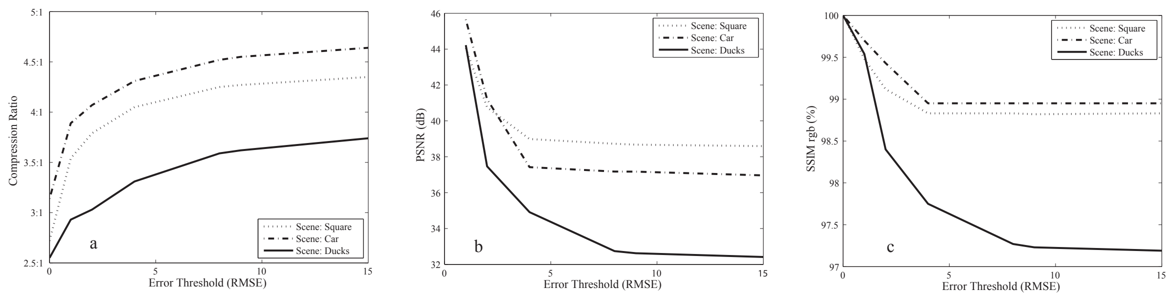


**Figure 6:** *Approximate (lossy) compression results for three of the test scenes (average of three rendering resolutions* $320 \times 240$*,* $640 \times 480$ *and* $1280 \times 1024$ *pixels). From the left: a) compression ratio vs.* $\tau^{rmse}_{thresh}$*, b) PSNR vs.* $\tau^{rmse}_{thresh}$*, and c)* $SSIM_{rgb}$ *vs.* $\tau^{rmse}_{thresh}$*.*

**Color Offset**

| #Sizes | Best sizes (Bits) | Effective Compression |
|---|---|---|
| 1 | 1280 | 1.43:1 |
| 2 | 1024,1408 | 1.58:1 |
| 3 | 896,1152,1408 | 1.61:1 |
| ∞ | | 2.04:1 |

**Entropy Coding**

| #Sizes | Best sizes (Bits) | Effective Compression |
|---|---|---|
| 1 | 1024 | 1.52:1 |
| 2 | 768,1280 | 1.75:1 |
| 3 | 640,1024,1408 | 1.88:1 |
| ∞ | | 2.45 : 1 |

**Our Algorithm**

| #Sizes | Best sizes (Bits) | Effective Compression |
|---|---|---|
| 1 | 1024 | 1.78:1 |
| 2 | 896,1152 | 2.04:1 |
| 3 | 640,896,1152 | 2.17:1 |
| ∞ | | 2.88:1 |

**Table 1:** *The tables show how the algorithms perform when given a number of allowed compressed sizes, as well as what selection of sizes that worked best for our test suite. Note that our algorithm performs very well even with very few compressed sizes.*

## 5.2. Approximate Compression

In Figure 6, we show the results from our experiments with approximate compression. The Quake 4 scene is excluded since alpha handling is currently not implemented in the lossy part. As can be seen, the additional compression gains can be quite substantial. We can gain an additional 25–60% compression by allowing approximate compression. The visual impact is normally small as can be seen in Figure 8 and in the $SSIM_{rgb}$ plot (Figure 6c). See Section 5.3 for more information on $SSIM$. However, the "ducks" scene clearly shows artifacts (Figure 9) already for small levels of $\tau^{rmse}_{thresh}$. This is due to that we use chrominance sub-sampling, which makes chrominance leak out to surrounding pixels. For a case like this, a more conservative error metric could be used, e.g., a maximum error threshold. This would decrease the effect of these artifacts. Our most important contribution for lossy buffer compression is the error control mechanism, and we believe our results shows that it works well, and that it can keep high image quality. However, more research is clearly needed on lossy compression algorithms.

## 5.3. Structural Similarity Index - SSIM

In addition to the common error metric, PSNR, we also use the structural similarity index, SSIM, as suggested by Wang et. al [WBSP04]. This is a visual quality metric which attempts to mimic the human visual perception. The SSIM index is a number between 0% and 100%, where 100% is perfect similarity. Note that the SSIM index is normally calculated using the luminance alone. In order to get the errors in all three color channels, R, G and B respectively, we have chosen to calculate the SSIM index for the R,G and B channels independently and combining them into a single number, $SSIM_{rgb}$, according to:

$$SSIM_{rgb} = 0.2126 * SSIM_R + \\ 0.7152 * SSIM_G + 0.0722 * SSIM_B, \qquad (5)$$

where the weights comes from ITU-BT.709 [IR02].

## 6. Conclusions and Future Work

Color buffer compression is available in almost all (if not all) GPUs, but up until now, this type of algorithms have not been described in the literature. By providing an overview of existing algorithms, we now have an important stepping stone in place, which is needed to invent new algorithms.

In addition, we have presented new algorithms based on a decorrelated color transform, which is also exactly reversible. Our results show that this can improve the compression ratio compared to other algorithms. Since it is well-known in the image and video compression community that the human visual system is more sensitive to luminance than chrominance, we have also done some initial results on approximate color buffer compression with this reversible transform.

We note that it is very important to keep the accumulated error under strict control, and we presented a simple mechanism to do this. We realize that approximate compression is a feature that must be turned off for some applications, but for, e.g., gaming on mobile devices, it can be very valuable with a longer use time on the battery with a only slight degradation in image quality.

We have only experimented with simple compression algorithms for approximate color buffers, and for future work, there is much to learn and transfer from the image and video processing field. We have started to investigate more sophisticated and fine-grained sub-sampling and quantization schemes. There is definitely room for inventing new algorithms. High dynamic range (HDR) color buffer compression is also an interesting topic for further studies.

In our paper, we have not handled multi-sampling, but several of the techniques [Eld06, MSM*04] for this can be merged relatively quickly into our work. Finally, we note that lossy depth buffer compression might not be feasible, due to the artifacts that can arise when surfaces intersect. However, this could be worth further investigation.

## Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research, Vetenskapsrådet, Ericsson, and NVIDIA's Fellowship program. Thanks to the anonymous reviewers for their helpful comments.

## References

[Ake00] AKELEY K.: RealityEngine Graphics. *Readings in computer architecture* (2000), 507–514.

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[Eld06] ELDER G. M.: Method and Apparatus for Anti-Aliasing using Floating Point Subpixel Color Values and Compression of Same. In *US Patent Application 2006/0188161 A1* (2006).

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Graphics Hardware* (2006), pp. 103–110.

[Hoo06] HOOK T. J. V.: Method and Apparatus for Compression and Decompression of Color Data. In *US Patent Application 7039241 B1* (2006).

[IEP98] IGEHY H., ELDRIDGE M., PROUDFOOT K.: Prefetching in a Texture Cache Architecture. In *Graphics Hardware* (1998), pp. 133–142.

[IR02] ITU-R S.: ITU, Recommendation BT.709 : Parameter values for the HDTV standards for production and international programme exchange. In *ITU-R, BT.709* (2002).

[JPE00] JPEG2000: ISO/IEC 15444-1:2000, JPEG 2000 Image Coding System, Annex G2, Reversible Component Transformation. In *ISO/IEC 15444-1:2000* (2000).

[MN03] MOREIN S. L., NATALE M. A.: System, Method, and Apparatus for Compression of Video Data using Offset Values. In *US Patent Application 2003/0038803 A1* (2003).

[MS03] MALVAR H., SULLIVAN G.: YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. In *JVT-I014r3* (2003).

[MSM*04] MOLNAR S. E., SCHNEIDER B.-O., MONTRYM J., DYKE J. M. V., LEW S. D.: System and Method for Real-Time Compression of Pixel Colors. In *US Patent Application 6825847 B1* (2004).

[NVI] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide. `http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_Programming_Guide_0.8.pdf`.

[Owe05] OWENS J. D.: Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 457–470.

[Per99] PEREBERIN A.: Hierarchical Approach for Texture Compression. In *Proceedings of GraphiCon '99* (1999), pp. 195–199.

[Ric79] RICE R. F.: *Some Practical Universal Noiseless Coding Techniques*. Tech. Rep. 22, Jet Propulsion Lab, 1979.

[WBSP04] WANG Z., BOVIK A. C., SHEIKH H. R., P.SIMONELLI E.: Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing, 13*, 4 (2004), 600–612.

[WSS96] WEINBERGER M. J., SEROUSSI G., SAPIRO G.: LOCO-I: A low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conference* (1996), pp. 140–149.

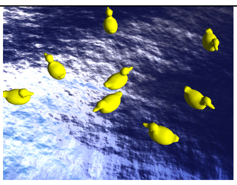
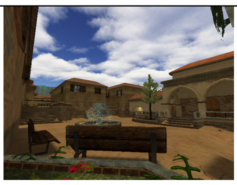
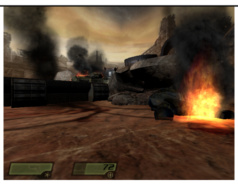
| |  |  | |  |
|---|---|---|---|---|
| Plane | 1.13:1 | 1.43:1 | 1.40:1 | 1.47:1 |
| Color Offset | 1.90:1 | 2.06:1 | 2.15:1 | 2.30:1 |
| Entropy Coding | 2.09:1 | 2.60:1 | 2.66:1 | 2.93:1 |
| Our | 2.64:1 | 2.87:1 | 3.36:1 | 3.05:1 |

**Figure 7:** *Evaluation of the compression algorithms: the table shows compression ratios for our test scenes (from the left: "Ducks", "Square", "Car", and "Quake4") using exact compression algorithms. We computed the compression ratios as the average compression ratio for rendering resolutions* $320 \times 240$, $640 \times 480$ *and* $1280 \times 1024$ *pixels. The algorithms scaled similarly with varying resolutions.*

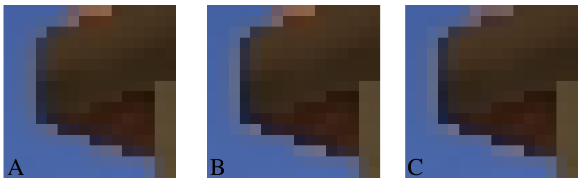


**Figure 8:** *Crops from the "Square" scene. A: original, B:* $\tau^{rmse}_{thresh} = 4$, *PSNR* $= 39.0$ *dB,* $SSIM_{rgb} = 98.8\%$, *compression ratio = 4.1:1, C:* $\tau^{rmse}_{thresh} = 15$, *PSNR* $= 35.6$ *dB,* $SSIM_{rgb} = 98.8\%$, *compression ratio = 4.3:1.*

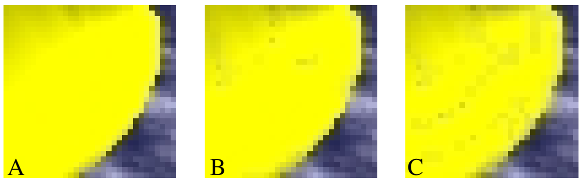


**Figure 9:** *Crops from the "Ducks" scene. A: original, B:* $\tau^{rmse}_{thresh} = 4$, *PSNR* $= 34.9$ *dB,* $SSIM_{rgb} = 97.8\%$, *compression ratio = 3.3:1, C:* $\tau^{rmse}_{thresh} = 15$, *PSNR* $= 32.4$ *dB,* $SSIM_{rgb} = 97.2\%$, *compression ratio = 3.8:1.*