Lossless Compression of Already Compressed Textures.

Conference Paper · August 2011

DOI: 10.1145/2018323.2018351 · Source: DBLP

Lossless Compression of Already Compressed Textures

Jacob Strom *
Ericsson Research

Per Wennersten†
Ericsson Research

Abstract

Texture compression helps rendering by reducing the footprint in graphics memory, thus allowing for more textures, and by lowering the number of memory accesses between the graphics processor and memory, increasing performance and lowering power consumption. Compared to image compression methods like JPEG however, textures codecs are typically much less efficient, which is a problem when downloading the texture over a network or reading it from disk. Therefore, in this paper we investigate lossless compression of already compressed textures. By predicting compression parameters in the image domain instead of in the parameter domain, a more efficient representation is obtained compared to using general compression such as ZIP or LZMA. This works well also for pixel indices that have previously proved hard to compress. A 4-bit-per-pixel format can thus be compressed to around 2.3 bits per pixel (bpp), or 9.6% of the original size, compared to around 3.0 bpp when using ZIP or 2.8 bpp using LZMA. Compressing the original images with JPEG to the same quality also gives 2.3 bpp, meaning that texture compression followed by our packing is on par with JPEG in terms of compression efficiency.

CR Categories: I.3.2 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, shadowing and texture E.4.1 [Data]: Coding and Information Theory—Data compaction and compression

Keywords: texture compression, data compression

1 Introduction

An important trend in real-time graphics rendering is that growth in computational power and RAM size outpace growth in memory bandwidth [Owens 2005]. This means that memory bandwidth is often a performance limiting factor for PC graphics cards, game consoles and other rendering devices today, and will likely be so increasingly often in the future. On mobile devices, memory accesses will also drain battery life. Texture compression, introduced by Knittel et al. [1996], Beers et al. [1996] and Torborg and Kajiya [1996] is one technique that saves bandwidth. It works by compressing the texture data, transferring it in compressed form over the bus, and finally decompressing it on-the-fly on the GPU. Compressed textures have the added bonus of also demanding less graphics RAM storage space, although the bandwidth saving effect is typically the more important, as mentioned by Inada and McCool [2006]. Finally compressed textures take shorter time to download over a network compared to uncompressed textures, and even compared to lossless (exact) compression methods such as PNG. However, compared to other lossy (non-exact) image compression methods such as JPEG, the difference is still huge: Compressing an image with JPEG to the same quality level as a 4 bpp texture codec may result in a file that is perhaps 60% of the size of the texture compressed file. In some cases the time it takes to download the textures of an application may be a major limiting factor on an application: You may be prepared to wait five minutes for your game to download and install, but perhaps not ten minutes. One solution can be to use a texture codec of lower bit rate, such as the 2 bpp PVRTC [Fenney 2003] used on the iPhone. However, the lower rate also brings a lower image quality which may not be acceptable in some cases. Furthermore, on many existing platforms the lowest bit rate codec available is a 4 bpp codec such as DXT1 [Iourcha et al. 1999] on desktops and Windows Phone 7 and ETC1 [Ström and Akenine-Möller 2005] on Android devices. Another solution may be to transmit the textures as JPEGs (or similar) over the network and recompress them on-the-fly to a texture compression format before rendering. This strategy is used by van Waveren to quickly stream in textures from disk [2006]. This is often a good solution, especially if low bit rates are of interest, but the resulting texture quality suffers for two reasons: First, the final texture will include image artifacts both from JPEG and from the texture codec. Second, to make recompression from JPEG to the texture codec quick enough, shortcuts may be necessary, especially on mobile devices with limited computational power. This lowers quality, especially when compared to slow, perhaps exhaustive compression. Another approach is to further compress the textures lossily [van Waveren 2006], which we will discuss in the next section.

The solution in this paper is instead to compress the original images with a texture codec and then try to further compress the resulting images using domain-specific lossless data compression. We will refer to this as *packing* in order to distinguish it from the compression in the texture codec. This way, slow off-line compression can be employed, and yet a reasonably-sized file can be transferred. On the client side, the data is unpacked after download and the file can be forwarded to the graphics hardware. Unlike general compression such as ZIP, we will apply domain-specific knowledge to increase compression efficiency.

The next section will describe previous work. Then our method will be presented, followed by a section on results. Finally, we will discuss limitations.

2 Previous Work

With the exception of van Waveren's work [2006], we have not found any previous work on the packing of already compressed textures. Therefore we will first describe general texture compression, followed by a description of van Waveren's work.

Delp and Mitchell [1979] propose a fixed rate image compression algorithm, where each pixel in a 4×4 block can choose from two gray levels using a bitmask. The two 8-bit gray values and the bit mask together occupy 32 bits, or 2 bits per pixel (bpp). Campbell et al. [1986] extend this to color by choosing two indexed colors instead of gray levels. The limitation to two colors per block gives rise to banding artifacts, so Iourcha et al. [1999] introduce two more colors in their DXTC/S3TC algorithm, which has become a de facto standard on desktops. The two extra col-

*e-mail: jacob.strom at ericsson.com
†e-mail:per.wennersten at ericsson.com

ors are linearly interpolated from the original two, and two bits per pixel are used to choose between the four colors, resulting in 64 bits per block or 4 bpp. Due to the interpolation the colors of a DXTC block lie along a line in RGB-space, which is a good approximation of most image blocks. Direct3D 11 contains two new texture compression formats; BC6 and BC7 [BPTC 2010]. They are based on DXTC but use two or three pairs of colors, generating two lines per block in RGB space. A bitmask to select between the two lines is needed, and this bitmask is vector quantized using up to 64 patterns. On handheld devices, several methods exist. PVRTC developed by Fenney [2003] is used on Apple's iPhone. It uses two low-resolution images A and B, both upscaled bilinearly twice. Each pixel can then choose its color from either image A, image B, or from two blend values between A and B. A 2 bpp version and a 4 bpp version exist. ETC1 [Ström and Akenine-Möller 2005] is another mobile texture compression format which is standardized—though not mandatory—on Android from version 2.2. It uses per-pixel luminance modification of a common base color, and obtains quality on par with S3TC. ETC1 has been extended in steps; ETC2 provides higher quality by adding in extra modes for certain corner cases [Ström and Pettersson 2007]; Rasmusson et al. describe a codec based on smooth functions that uses ETC2 as a fall-back [2010].

In the same paper where the JPEG-type streaming solution is presented, van Waveren also describes compression of already compressed DXTC data [2006]. The two colors in the block are compressed lossily; the two colors are placed into two RGB images of $\frac{1}{4}$ the size in both the x- and y- dimension. Then these two images are compressed using JPEG. The pixel indices are losslessly compressed using run length or LZ-based compression. The compression efficiency is improved by rotating and/or mirroring the indices to line them up. The compressed data is reported to be around 75% of the original index data. This is the most similar previous work, and has been the source of inspiration for our work.

Although not explicitly described anywhere, we also count regular ZIP-type compression of the entire compressed texture as previous work. Such compression typically gives a file size of of approximately 75% of the original size. Thus a 4 bpp format becomes around 3 bpp after zipping. LZMA gives higher compression, with file sizes of around 70% of the original.

3 The Proposed Algorithm

Our work is similar to that of van Waveren [2006] in that we take a compressed texture and attempt to further compress it for storage and transmission. Although we believe that the general principle of our work can be applied to any texture compression format, our way of packing has to be adapted to each texture codec specifically. We have chosen ETC1 as our demonstration codec, and will therefore go through that in some more detail.

In ETC1, the image is divided into 4×4 blocks which is are compressed to 64 bits each. The block is further divided into two half-blocks of size 4×2 or 2×4, depending on the value of a bit called the *flip bit*. Each half-block is assigned a *base color*, thus two colors per 4×4 block are required. These are encoded either as two RGB444-colors, or as one RGB555 color and a differential color with three bits per color component. Selection between the individual and the differential encoding is signalled via a *diff bit*. For each pixel, a 2-bit *pixel index* is also stored. This is used to modify the luminance of each pixel by adding to the base color a modifier from a table of four numbers. If the table is $\{-8, -2, 2, 8\}$, the pixel index can specify to use, say, -8 as the modifier. Assume the base color for the half block is $(119, 51, 255)$ after extension to eight bits. The resulting color for that pixel is then $(119, 51, 255) + (-8, -8, -8) = (111, 43, 247)$. Each half-block also uses a *table value* to specify which of eight tables to use. For smooth blocks, the table above can be used, whereas a block containing a sharp edge may select a table such as $\{-106, -33, 33, 106\}$. The method of luminance modulation is illustrated in Figure 1. The bits stored for each 4x4 block are *diff bit* (1 bit), *flip bit* (1 bit), two *base colors* (24 bits), *table value* (3·2 = 6 bits) and *pixel indices* (32 bits), all in all 64 bits.

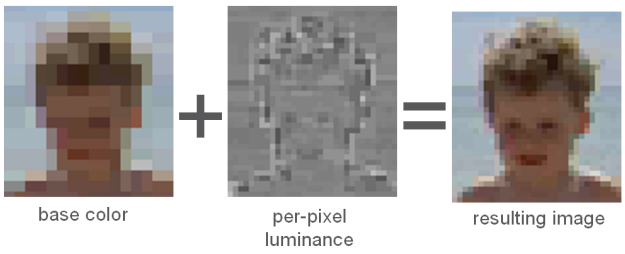

Figure 1: *In ETC1 each 8-pixel half-block (left) is modulated by luminance (middle) to produce the final output (right).*

3.1 Compressing the Pixel Indices

As van Waveren points out [2006], the pixel indices are hard to compress. The reason can be seen in Figure 2; even an area that is flat in the decompressed image (middle) can be associated with pixel indices that have lots of variance (right). This has led us to

Figure 2: *Left: Decoded ETC1 image. Middle: A zoomin of a smooth area. Right: Pixel indices of that area. It seems resonable to believe that it is simpler to predict the colors (middle) than the pixel indices directly (right).*

the idea that it may be easier to predict the color of the pixel than to predict the pixel index. Once the color has been predicted, all four possible pixel indices from the table are tried. The pixel index that produces a color closest to our predicted color will be selected as our prediction of the pixel index. The method for pixel index prediction can be summarized in the following steps.

1. Transmit all non-index bits, e.g., *diff*-, *flip*- *color*- and *table-value* bits for the half-block in question.
2. Find the colors of the neighboring pixels that have already been transmitted
3. Use the color of these neighboring pixels to predict the color of the current pixel
4. Find the pixel index that produces the color closest to the predicted color. This is now our predicted pixel index
5. Encode the pixel index with help of the predicted pixel index

The decoder will have to do similarly. In particular, as soon as a pixel index has been received, the pixel color will be decoded so that it can be used for prediction of neighboring pixel indices:

1. Decode all non-index bits, e.g., *diff-*, *flip- color-* and *table value* bits for the half-block in question.
2. Access the neighboring pixels that have already been decompressed
3. Use the color of these neighboring pixels to predict the color of the current pixel
4. Find the pixel index that produces the color closest to the predicted color. This is now our predicted pixel index
5. Decode the pixel index with help of the predicted pixel index
6. Decompress the pixel to obtain the pixel color

Using the surrounding pixel colors to thus predict the pixel index gives a better guess than just using the surrounding pixel indices. This is not that surprising given that the prediction is based on more information, namely the other parameters of the block. Assume for instance that we have predicted the color (249, 150, 25). Assume also that the base color in the current half block is (240, 130, 0), and that the current table is $\{-60, -18, 18, 60\}$.

The four possible pixel indices would then produce the following colors:

pixel index	modifier	color	$\|\|\text{color} - \text{prediction}\|\|^2$
3	-60	(180, 70, 0)	11786
2	-18	(222, 112, 0)	2798
0	18	(255, 148, 18)	89
1	60	(255, 190, 60)	2861

Directly we can see that only pixel index 0 gives a color that is anywhere near the prediction. This is due to the fact that the table contained such big values, information that is lost if prediction is only done from previous pixel indices.

Getting a good pixel index prediction thus depends on getting a good prediction of the color for the current pixel. We use the neighboring pixels that have already been decoded; the pixel to the left, the pixel above, called *up*, and the pixel one step up and one step left, which we call *diag*. Four predictions are used, and to choose which prediction to use, we calculate three measures: The first one measures how much difference there is between the upper and left pixel in terms of deviation from the diagonal: $m_A = ||\text{diag}_\text{g} - \text{upper}_\text{g}| - |\text{diag}_\text{g} - \text{left}_\text{g}||$, where $|\cdot|$ denotes absolute value. We only use the green component of the colors in these calculations. The second one measures the difference between the upper and the diagonal pixel: $m_B = |\text{diag}_\text{g} - \text{upper}_\text{g}|$, and the third between the left and the diagonal: $m_C = |\text{diag}_\text{g} - \text{left}_\text{g}|$.

The prediction is then selected by the following pseudo code:

```
if m_A < 4 AND m_B < 4
   use left+up-diag
else if m_A < 10
   use (left+up)/2
else if m_A < 64
   if m_B < m_C
      use (3*left+up)/4
   else
      use (3*up+left)/4
   end
else
   if m_B < m_C
      use left
   else
      use up
   end
end
```

The reasoning is that if all three pixels are similar, it is reasonable to believe that we are in a rather smooth area, and we can use a planar prediction $\text{pred} = \text{left} + \text{up} - \text{diag}$. However, for pixels which are less similar, we use $\text{pred} = (\text{left} + \text{up})/2$, which is more robust. If *left* and *up* are dissimilar, we assume that there is a line going through the four pixels. Hence if the upper and diagonal pixels are similar to each other but the left one is not, we assume a common value in the left and the current pixel, and the prediction becomes $\text{pred} = \text{left}$. We also have an "in between" type of prediction which is $(3 * \text{left} + \text{up})/4$. Given the prediction of the color, we can now get the prediction of the pixel index. We have also tried the predictor from LOCO-I/JPEG-LS [Weinberger et al. 1996]:

$$pred = \begin{cases} min(up, left), & \text{if diag} \geq max(up, left) \\ max(up, left), & \text{if diag} \leq min(up, left) \\ up + left - diag, & \text{otherwise.} \end{cases} \quad (1)$$

This performs only slightly worse, yielding about 1% more bits on the index data, and given its lower computational complexity it may be an interesting alternative.

Our task is now reduced to compress the actual pixel index with the help of the predicted pixel index. We use an adaptive arithmetic coder [Witten et al. 1987] implemented by Wheeler [1996]. For reference we have used the source code of Wheeler without changes. We have chosen an arithmetic coder mostly due to the convenience of encoding single instances of random variables (with skewed probability distributions) with less than one bit. In short, an arithmetic coder can compress a variable given its probability distribution. An *adaptive* arithmetic coder continuously estimates this probability distribution during encoding/decoding. Wheeler's implementation can handle alphabets of any size; hence there is no need to binarize the data before handing it over to the arithmetic coder. Arithmetic coding is quite complex — if a hardware or GPU implementation is desired, a more hardware friendly entropy coder may be preferrable.

One possibility would now be to encode the error between the predicted pixel index and the actual pixel index, $\text{error} = \text{actual} - \text{pred}$. However, for our four possible pixel indices, this error value ranges from -3 through 3, although only four values are actually possible for each pixel. By using the same probability distribution for all four predictions, the arithmetic coder would estimate non-zero probabilities even for the impossible values, resulting in an inefficient encoding. Therefore our solution is to encode the pixel value as is, but use four different probability distributions, one for each prediction value. For instance, if the prediction is 0, the following probability distribution may be used after convergence: $\{0.65, 0.15, 0.12, 0.08\}$. If the prediction is 2, we will use another probability function that, after convergence, may look like $\{0.06, 0.12, 0.69, 0.13\}$. Another note is that we use different probability distributions for the different predictions. For instance, the planar prediction is usually more accurate than, say, the prediction $(\text{left} + \text{up})/2$. Using the same probability distributions for both would lower the efficiency of both encodings.

3.2 Compression of the Other Parameters

The other parameters are also encoded using the arithmetic encoder. The flip bit is compressed as is, without prediction, using its own probability distribution. Similarly, the diff bit is compressed without prediction using its own distribution.

The table values are also sent as is. A table value of zero is the most common, except in the cases when surrounding blocks have nonzero table values. Therefore we will use different probability distributions in these two cases. If the block is flipped (two 4×2 blocks) it makes sense to look for non-zero values in the above

block when encoding the top table value — we use the second table value of the above block in these circumstances. This is illustrated in Figure 3. In (a) and (b), where the block to predict (blue) is

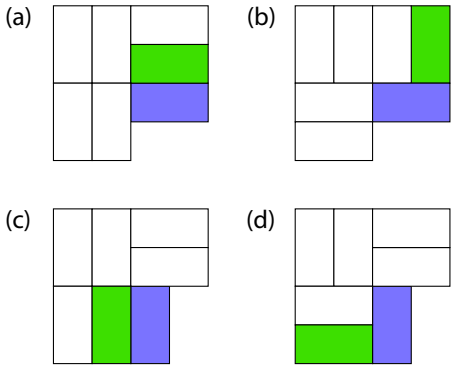

Figure 3: *Table prediction. (a) and (b): If the first sub-block (blue) is flipped, the second sub-block (green) in the above block will be used for prediction. If the green block has a non-zero table value, one probability distribution will be used. If the green block has a table value of zero, the other probability distribution will be used. (c) and (d): If the first sub-block is non-flipped, the second sub-block (green) in the left block will be used for prediction.*

flipped, the second sub-block of the above block is used (marked with green). If the block is non-flipped the second sub-block of the left block will be used, as shown in (c) and (d). The second table value (corresponding to the second sub-block) is encoded the same way, but will use the first table value in the current block to select probability distribution.

The two base colors in the block make up a large share of the data, so any gain in compression efficiency will influence the end result quite a bit. Therefore it is important to try to predict them well.

Just as for the pixel indices, the prediction of the base colors rely not on previous base colors, but on the surrounding pixels. The colors are compressed component by component, starting with red. This is done as shown in Figure 4. In the example in the figure we have

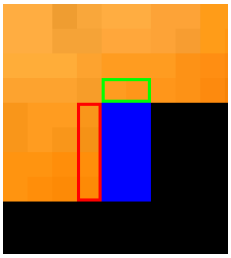

Figure 4: *Color prediction. The base color of the half-block marked with blue is predicted from the surrounding pixel colors that have already been encoded/decoded.*

flip bit = 0, meaning that we have a standing 2×4 half-block. The half-block is marked with blue, and pixels that have not yet been encoded/decoded are black. For such a standing block, the prediction is simply the average of the red component of the surrounding pixels; the two above, marked with green, and the four to the left, marked with with red. The average must also be converted to the appropriate number of bits: if the first color is supposed to be stored with five bits we multiply our average color by (31/255) to get the predicted red component. The prediction error in the red component, $\mathrm{error}_{\mathrm{red}} = \mathrm{actual}_{\mathrm{red}} - \mathrm{prediction}_{\mathrm{red}}$, is now compressed with the arithmetic coder. Just as in the case with the pixel indices, only some of the values are possible. For instance, if the predicted value is 21, only values -21 through 10 are possible, but since we use the same probability distribution irrespectively of the prediction, all values will have nonzero probabilities. However, values far from zero have very small probabilities, so the inefficiency is not as great as for the pixel indices, and a single probability distribution can be used.

The green component is compressed in a similar fashion, but with a twist. Since the color components are correlated, if the predicted red component was much lower than the actual red component, it is likely that the green prediction is also too low. Therefore it is beneficial to compensate the prediction of the green component with the error in red: $\mathrm{prediction}_{\mathrm{green}} = \mathrm{average}_{\mathrm{green}} + \mathrm{error}_{\mathrm{red}}$. This is possible since the red component has already been transmitted and decoded. Likewise, the blue component is compressed using the green error as a correction.

The base color for the second half of the block is compressed in the same manner. However, in this case some of the surrounding pixels will be located in the first half-block. To be able to recover these pixels so we can predict from them, we have to send all parameters for the first half-block before the parameters of the second half-block are predicted. Hence the transmission order must be: *flip bit*, *diff bit*, *table values*, *base color* for first half-block, *pixel indices* for first half-block, *base color* for second half-block, *pixel indices* for second half-block. The prediction of the second base color is similar to that of the first half-block. The main difference is that the second color is sometimes differentially encoded. Once we have obtained our average red value (and multiplied by 31/255 to convert to 5 bits), we must therefore subtract the red component of the first base color. The result is then clamped to a value in $[-4, 3]$, and this is then used as prediction for the red differential. Just as in the case with pixel indices each prediction value (such as -4) gets its own probability distribution. The green and blue components of the second colors are dealt with the same way.

One may assume that just using the base colors in the surrounding sub-blocks for prediction would give equal performance. We tried this, but it did not work as well as basing the prediction on the closest pixels as done in Figure 4. One explanation for this is that while the base color is often very close to the average of the the pixels in the sub-block, this is not always the case. Especially when using exhaustive compression, the encoder may find that using a base color that is much darker than the pixels in the sub-block works better than using the average color. Predicting the base color in the adjacent sub-block form this outlier will not result in good performance, but predicting from the resulting pixels of the same sub-block will work fine.

3.3 Run Length Coding Addition

Some images contain very simple and repetitive data, such as a horizontal line or a constant background color. In order to compress these better we have added a run length system to our scheme. After having compressed a block in the above-mentioned way, a number between 0 and 7 is also sent. The number represents how many identical copies of the block follows. This saves a lot of bits for simple repetitive data. One would think that it would be a great burden for images with non-repetitive content, but in this case the adaptive arithmetic encoder quickly learns that the probability for a nonzero number is virtually zero, and very few extra bits are sent. Highly structured blocks are unlikely to be repeated, and therefore we only send the run-length symbol after blocks where all pixels of a row have the same color.

4 Results

We have implemented the system and compressed a set of 64 images. The test set includes typical game textures, but also natural images and synthetic images such as text.

The images were compressed to ETC1 using exhaustive compression. The resulting .pkm files were then packed with the proposed method. They were also packed with ZIP and LZMA as references. Two different ZIP tools were tried, the one built into Windows Vista, and the ZIP coder in IZArc version 3.81 build 1550. Both give similar results so we reported the best. For LZMA we used 7-Zip version 9.20 using the "ultra" compression level and the default setting for all other parameters. Gzip was also tried but gave results similar to ZIP so we have not reported those.

We also compressed the original (non-compressed) images to JPEG using ImageMagick version 6.5.8-3. The quality level was selected for each image so that the quality in terms of PSNR matched that of the ETC1 decompressed image as closely as possible.

Some of the images in the test set depicted objects in front of a white background. Since the white parts of these images are very simple to encode both for our proposed scheme (due to the run-length coding) and for ZIP, we have reported a second set of results excluding these white-background images.

The bit rate is reported as the number of bits per pixel averaged over the entire test set.

	ETC1 +ZIP	ETC1 +LZMA	ETC1 +proposed	JPEG of equal quality
all images (bpp)	2.85	2.60	2.21	2.25
nonwhite (bpp)	3.01	2.76	2.31	2.35

If no packing is used, ETC1 has a bit rate of 4.0 bits per pixel (bpp). As can be seen in the table, ZIP lowers this to around 3 bpp on average for the nonwhite images, and LZMA gives around 2.8. However, the proposed method substantially reduces this to around 2.3 bpp, a reduction of 16%. It is also interesting to see that ETC1 + our proposed scheme gives a better result than JPEG when measured at the same quality level. This is quite remarkable when you consider that the first step is a fixed rate coder giving an equal number of bits to every block. This will mean that bits in simple-to-code blocks in ETC1 will be spent on coding noise, which will contribute little to the PSNR value and which is typically hard to compress. Yet caution should be used when interpreting these figures; they depend on which JPEG compressor you choose. Furthermore, JPEG is no longer the state-of-the-art encoder for still images: JPEG2000 [JPEG 2000], HD-photo [Microsoft 2006] and even the intra part of the H.264 video standard [Süehring 2009] are typically all more efficient. Still, we think that JPEG is relevant since it is much used, and it is interesting to note that the proposed solution can at least compete with transform based solutions.

If JPEG is used as the transport format, the textures will need to be compressed on-the-fly to a texture compression format such as ETC1 after download. To find out how much this transcoding will lower the quality, we compressed the JPEG images to ETC1 and measured the average mean square error. The result was an increased error equivalent to a PSNR drop of 2.02 dB than if just ETC1 encoding were used. Thus even if the JPEGs have equal quality to the proposed scheme, after the transcoding there will be a significant quality penalty. The transcoding used slow exhaustive compression—fast transcoding will give an even higher penalty.

For the proposed method, the bits are spent according to the following percentages:

	RL	flip	diff	table	color	index
all images	0.33%	2.43%	1.21%	10.02%	31.05%	54.97%
nonwhite	0.26%	2.45%	1.18%	9.85%	30.82%	55.44%

We need both the run-length data (marked with RL in the table) and the index data itself to decode the indices. Thus the indices are encoded using $0.33\% + 54.97\% = 55.3\%$ of the 2.2 bpp data for the "all images"-case, or about 61% of the original 2 bits per pixel spent on index data. For comparison we compressed only the index data using LZMA. The result was 70% of the original data; hence the proposed metod gives a significant reduction for index data. The corresponding figures for the nonwhite images are 64% of the original data for the proposed method, and 74% for LZMA.

Our data base of images is not publicly available, so we also give numbers for the top left 512×512 part of the ten first images in the Kodak suite available at http://r0k.us/graphics/kodak/. The rate in terms of bits per pixel with ETC1 plus the proposed method is 2.68, 2.28, 2.01, 2.38, 2.75, 2.55, 1.95, 2.73, 1.97 and 2.08 respectively, averaging at 2.3 bpp. Zipping the compressed textures will give an average of 3.1 bpp, and 2.8 bpp for LZMA. On this smaller database JPEG fared a bit better at 2.2 bpp. In general, large smooth areas such as a sky favors JPEG whereas high-contrast details such as text favours our system.

Since the proposed scheme reaches a bit rate of 2.2-2.3 bpp, it is tempting to compare it to 2 bpp codecs such as PVRTC. However, such a comparison is not straightforward: First, PVRTC textures can be zipped, bringing them down to 1.6 bpp for the nonwhite images, and 1.5 bpp for all. Second, the proposed scheme could of course be tried also for PVRTC. Third, a 2 bpp codec would give benefits in terms of memory footprint and memory bandwidth consumption compared to a 4 bpp solution, which is not true for the proposed solution. Finally, the quality difference between a 4 bpp codec and a 2 bpp codec is quite big, as can be seen in Figure 5. Therefore we see our proposal as a complement to and not as a replacement for 2 bpp codecs.

Figure 5: *Left: original, 24 bpp. Middle: ETC1, 4 bpp. Right: PVRTC, 2 bpp.*

5 Conclusion and Limitations

We have presented a novel way of further packing compressed textures. The main idea is to predict the parameters by first predicting the color values, and then see what parameters best fit these predicted colors. The improved prediction is especially noticeable for pixel index data, which are known to be hard to compress [van Waveren 2006]. We have applied the idea to the ETC1 texture codec but expect that it will work fine with other codecs such as DXT-type codecs and PVRTC. However, there are some limitations. The proposed system only solves the problem of slow transmission time over the network. Once it is unpacked, it does not improve the memory footprint in the graphics memory, nor the number of GPU-memory accesses over that of regular 4 bpp texture compression. If this is the bottleneck, a lower-bit rate codec should be used instead. Also, the unpacking of the data will take some time. The

decoding takes around 300 ms for a 512×512 texture with completely unoptimized code on a 1.33 GHz laptop, but if a lot of data is transmitted even small decoding times will add up. ZIP is considerably faster at around 60 ms for the same data. Also, arithmetic coding is sequential by nature, although different textures can of course be decoded in parallel. Hopefully, part or all of the decoding time will be hidden by the time it takes to download the texture: If downloading the texture takes more time than decompression, the decoding can happen during download with minimum penalty. Another limitation is that it is hard to reach very low bit rates for transmission. JPEG can typically compress images down to 10% of the original file size without noticeable artifacts. In our case we have gone from 4 bpp (which equals 16% of the original 24 bpp) down to 2.3 bpp or 9.6% of the original size, thus reaching the same size as high-quality JPEG-compression. However, JPEG can also provide reasonable quality images at 5% of the original data, whereas our proposal does not give the possibility to trade quality for bit rate. Thus, if really low bit rates are imperative, the texture streaming approach of van Waveren is most likely a better solution. However, if the highest possible quality is desired (no recompression and slow texture encoding), our solution has an advantage. If one wants to further lower the bit rate while sticking to the proposed solution, it is possible to change the parameters to values that are cheap to encode. For instance, if pixel index = 2 will give the same error against the original image as pixel index = 1, it makes sense to select the cheaper one. A rate-distortion optimization of the image would be possible for finding which parameter values lower the bit rate the most while losing the least quality. This could be an avenue for future work. Another possibility would be to make the method more parallelizable/GPU friendly. Almost instant decompression might be more valuable in real applications than maximum compression efficiency.

Acknowledgements: Thanks to Andrey Norkin for interesting discussions on alternative binarizations, and to the reviewers for valuable input.

References

BEERS, A., AGRAWALA, M., AND CHADDA, N. 1996. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, 373–378.

BPTC, 2010. ARB_texture_compression_bptc. Available online: www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt.

CAMPBELL, G., DEFANTI, T. A., FREDERIKSEN, J., JOYCE, S. A., LESKE, L. A., LINDBERG, J. A., AND SANDIN, D. J. 1986. Two Bit/Pixel Full Color Encoding. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 215–223.

DELP, E. J., AND MITCHELL, O. R. 1979. Image Compression using Block Truncation Coding. *IEEE Transactions on Communications 2*, 9, 1335–1342.

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

INADA, T., AND MCCOOL, M. 2006. Compressed Lossless Texture Representation and Caching. In *Graphics Hardware*, 111–120.

IOURCHA, K., NAYAK, K., AND HONG, Z., 1999. System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431.

JPEG, 2000. JPEG 2—ISO/IEC 15444-1:2005. Available online:http://www.jpeg.org/jpeg2000/.

KNITTEL, G., SCHILLING, A. G., KUGLER, A., AND STRASSER, W. 1996. Hardware for Superior Texture Performance. *Computers & Graphics, 20*, 4, 475–481.

MICROSOFT, 2006. HD Photo. Available online: http:/www.microsoft.com/windows/windowsmedia/forpros/wmphoto/default.aspx.

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

RASMUSSON, J., STRÖM, J., WENNERSTEN, P., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Texture Compression of Light Maps using Smooth Profile Functions. In *High-Performance Graphics*, 143–152.

STRÖM, J., AND AKENINE-MÖLLER, T. 2005. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware*, 63–70.

STRÖM, J., AND PETTERSSON, M. 2007. ETC2: Texture Compression using Invalid Combinations. In *Graphics Hardware*, 49–54.

SÜEHRING, K. 2009. *JM Software H.264/AVC*. http://iphome.hhi.de/suehring/tml/.

TORBORG, J., AND KAJIYA, J. 1996. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, 353–364.

VAN WAVEREN, J., 2006. Real-Time Texture Streaming and Decompression. Id Software Technical Report, available at http://software.intel.com/file/17248/.

WEINBERGER, M., SEROUSSI, G., AND SAPIRO, G. 1996. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Proc. IEEE Data Compression Conference, Snowbird, Utah, March-April 1996*.

WHEELER, F., 1996. Adaptive Arithmetic Coding Source Code. available at http://www.cipr.rpi.edu/ wheeler/ac/.

WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. 1987. Arithmetic Coding for Data Compression. *Communications of the ACM 30*, 6.