Graphics Hardware (2005)
M. Meissner, B.- O. Schneider (Editors)

*i*PACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones

Jacob Ström[1] and Tomas Akenine-Möller[2]
[1]Ericsson Research, [2]Lund University

Abstract

We present a novel texture compression scheme, called i*PACKMAN, targeted for hardware implementation. In terms of image quality, it outperforms the previous de facto standard texture compression algorithms in the majority of all cases that we have tested. Our new algorithm is an extension of the PACKMAN texture compression system, and while it is a bit more complex than PACKMAN, it is still very low in terms of hardware complexity.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Texture

1. Introduction

For rasterization-based hardware architectures, the available bandwidth in the system is what usually limits performance [AMN03]. To that end, many different techniques have been proposed and implemented to reduce bandwidth requirements. One of these is hardware *texture compression* (TC), introduced by Knittel et al. [KSKS96] and Beers et al. [BAC96]. The core idea is simply to use (lossy) compression on the images, and store the compressed version of the texture. When accessing the texture during rendering, the compressed texture is transferred over the bus, and decompressed on-the-fly as needed, thus saving bandwidth.

To facilitate a hardware implementation, a texture compression/decompression system must have the following features. First, the cost in gates should be low, especially for mobile phones. To better exploit a texture cache [HG97, IEH99], textures can be stored in the cache in compressed form. If some type of texture filtering is used, then several units of texture decompression are needed, making the need for low complexity even higher. For example, for trilinear mipmapping, eight units are needed in order to deliver a filtered color per clock. Second, to make addressing simple and random access possible, a fixed compression rate (in terms of bits per pixel) is needed. Any fixed rate coder must be lossy if it is to compress, and hence all TC systems described in the literature are lossy fixed rate coders (to the best of our knowledge). Third, avoiding look-up tables (LUTs) that depend on the current texture is preferred, since that eliminates the need to update this LUT, and also avoids the level of indirection and the latency it introduces. Finally, the execution time for compressing a texture should be reasonably short, though this is not of extreme importance, since compression usually is done off-line as a preprocess.

Our new texture compression scheme was originally targeted for mobile phones, but is in no way limited to those platforms, and can thus be used for PC graphics cards and game consoles as well. It builds upon PACKMAN texture compression [SAM04], which has low complexity and reasonable image quality. In the present work, we have improved the image quality substantially over PACKMAN at only a slight increase in implementation complexity. In the majority of cases, this improved PACKMAN, or *i*PACKMAN for short, provides better image quality than the de facto standard, S3TC (called DXTC in DirectX) [MB98, INH99] and the recently proposed PVRTC [Fen03]. *i*PACKMAN compresses to a rate of 4 bits per pixel (bpp).

The basic idea of *i*PACKMAN is to use larger blocks, 4×4 pixels instead of 2×4 for PACKMAN. This is nothing new, but compared to PACKMAN, it gives greater opportunities to obtain better image quality (or compression rate) because spatial redundancy in a larger area can be exploited. Two new variants of the PACKMAN scheme are introduced in our paper, and the best of these is chosen for each 4×4 block. We show image quality comparisons on standard image benchmarks and provide hardware diagrams of our algorithm.

2. Previous Work

In this section, we present work that is related to texture compression with hardware implementation as target.

Delp and Mitchell [DM79] developed a simple scheme, called block truncation coding (BTC) for image compression. Even though their applications were not texture compression per se, several of the other schemes described in this section are based on their ideas. Their scheme compressed

© The Eurographics Association 2005.

gray scale images by considering a block of 4×4 pixels at a time. For such a block, two 8-bit gray scale values were stored, and each pixel in the block then used a single bit to index to one of these gray scales. This resulted in 2 bits per pixel (bpp).

A simple extension, called color cell compression (CCC), of BTC was presented by Campbell et al. [CDF*86]. Instead of using an 8-bit gray scale value, they use the 8-bit value as an index into a color palette. This allowed for compression of colored textures at 2 bpp. However, this does require a memory lookup in the palette, and the palette is restricted in size. Knittel et al. [KSKS96] suggested that CCC was implemented in hardware and used in a texturing system. In fact, they also used a texture cache (after decompression, however), but did not explore the many different parameters when designing a cache.

The S3TC texture compression method by Iourcha et al. [INH99] is probably the most popular scheme. It is used in DirectX [MB98] and there are extensions for it in OpenGL as well. Their work can be seen as a further extension of CCC. The block size for S3TC is 4×4 pixels that are compressed into 64 bits. Two base colors are stored in each 16 bits, and each pixel stores a two-bit index into a local color set that consists of the two base colors and two additional colors in-between the base colors. This means that all colors lie on a line in RGB space. S3TC's compression rate is 4 bpp. One disadvantage of S3TC is that only four colors can be used per block. Ivanov and Kuzmin attack this problem by using colors from neighboring blocks as well [IK00]. However, this increases the memory bandwidth used to decode a block, which is non-desirable.

Akenine-Möller and Ström present a variation of the S3TC scheme that compresses a 3×2 block into 32 bits [AMS03]. This scheme, called POOMA, is targeted for mobile phones as well. The major difference is that each base color uses fewer bits, and that only one in-between color is used. Also, note that the block width is three, which is awkward for hardware implementations.

Beers et al. use a traditional approach called vector quantization [BAC96], and they could compress textures to as low as 1 bpp or 2 bpp. However, vector quantization as well as palettized textures, do require an additional memory access to determine which color to use. This is not feasible for a high-performance computer graphics pipeline.

A radically different approach is taken by Fenney [Fen03]. Two low-resolution images derived from the original texture are stored, and during decompression, a (local) bilinear magnification of those textures are created, and to create the final color of the texel, a linear blend is done between the two. Two modes are described that give 4 bpp and 2 bpp, respectively. In the 4 bpp version, two base colors are stored per 4×4 block, together with modulation data. To do the bilinear magnification, the neighboring 2×2 blocks are needed. Once these are in the texture cache, decompression should be fast.

Figure 1: *Here, the core idea of PACKMAN is illustrated. To the left, the base colors for each 2×4 block is shown. The image in the middle shows the per pixel luminance modulation. The rightmost image shows the decompressed image.*

Perebrin combines mipmapping and texture compression [Per99]. Each 4×4 block is compressed in YUV space, and it is assumed that box filtering is used for the mipmaps. Luminance is decomposed using the Haar wavelet basis, and the chrominance information is first subsampled before it is compressed. The bit rate is about 4.6 bpp.

3. Review of PACKMAN Texture Compression

In this section, we briefly describe the original PACKMAN-texture compression scheme [SAM04], since it is fundamental to our new algorithm.

The texture image is split into 2×4 blocks, where each block is represented by 32 bits. A single color, called a *base color*, is stored for each block in $4+4+4 = 12$ bits RGB (or RGB444 for short). 20 bits remains, and those modulate the luminance for each pixel in the block. An example is shown in Figure 1. More specifically, a constant, called a *modifier value*, is chosen from a small table of stored numbers, and that constant is added to each of the color components of the base color. A table consists of only four different numbers, and so each *pixel index* needs two bits for choosing which constant to use. Thus, these indices use $2 \times 4 \times 2 = 16$ bits. At this time, 28 bits have been used, and the remaining 4 bits are used as a *table codeword* to select one table out of 16 different tables, comprising a *codebook*.

The PACKMAN hardware decompression procedure for a single pixel is described in more detail below:

1. The 12-bit *base color* is expanded from 4 bits per color component to 8 bits. As an example, RGB=$(0,2,15)$ is converted to $(0,34,255)$.
2. The 4-bit *table codeword* is used to pick a specific table of four numbers from the codebook of tables. Using, for example, a table codeword of 1 means that the following table is selected: $\{-12,-4,4,12\}$ (see the codebook in Table 1).
3. The 2 *pixel index* bits associated with the pixel are used to choose a modifier value from the table, for instance -12.
4. The final step computes the final decompressed color by adding the modifier value to the expanded base color. Then the color is clamped to $[0,255]$. For example, if the

© The Eurographics Association 2005.

table codeword	0	1	2	3	4	5	6	7
	-8	-12	-31	-34	-50	-47	-80	-127
	-2	-4	-6	-12	-8	-19	-28	-42
	2	4	6	12	8	19	28	42
	8	12	31	34	50	47	80	127

Table 1: *First half of the codebook for PACKMAN.*

pixel index is 0, the modifier is -12, and the final color is $(0,34,255)+(-12,-12,-12)=(0,22,243)$, where values have been clamped to $[0,255]$.

The codebook consists of 16 different tables, each containing 4 different values. The tables associated with the table indices are shown Table 1. Tables 8–15 equal tables 0–7 scaled by a factor of two, and the first and second values of each table are the fourth and third values negated. Thus, only 16 numbers need to be stored, and these are constant for all textures.

4. *i*PACKMAN Texture Compression

In this section, our new *i*PACKMAN texture compression system will be presented. First, we describe the underlying design and discuss the motivation for the design choices. Then follows subsections for decompression and compression.

4.1. Basic Design and Motivation

When smooth blocks are encountered by PACKMAN, one of the leftmost tables with small modifier values can be used—see the codebook in Table 1. This means that the luminance of these blocks can be represented rather accurately—better that what the rather limited number of bits (12) of the base color would suggest. A PACKMAN-compressed image therefore has significantly less luminance banding than an image in which all pixels have been quantized to 12 bits. However, the chrominance has never more resolution than 12 bits. Therefore, in areas where the luminance is more or less constant, but where the chrominance shifts slowly over the blocks, chrominance banding can be visible, since even the smallest possible jump in chrominance is rather big with a 12-bit representation. Since only a single chrominance per block is used, the banding edges follow block boundaries, which makes this artifact worse. *i*PACKMAN attempts to overcome this problem as we will see.

One way to combat these chrominance banding artifacts is to improve the color representation for slowly changing areas. Instead of encoding the base color of each block independently with RGB444, it is possible to group two adjacent 2×4 blocks together to a 4×4 block, and encode the base colors differentially with respect to each other.

In order to see how much can be gained from such an approach, we selected twenty test images of various kinds,

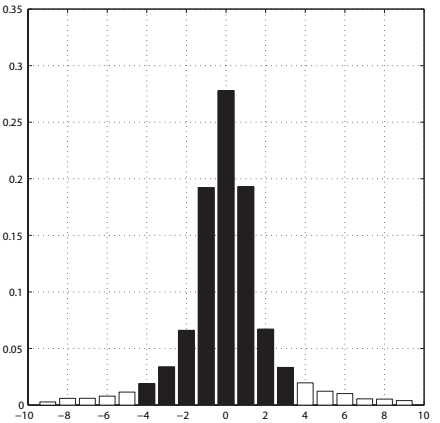

Figure 2: *A histogram over the difference between average colors of adjacent blocks quantized to 555. The color component deviating the most is used for each block. Note that the blocks where the largest deviation is between* -4 *and* 3 *(marked with black bars) account for* 88% *of the blocks.*

calculated the average color for adjacent pairs of 2×4 blocks, and quantized them to RGB555. The difference $(R_1-R_2,G_1-G_2,B_1-B_2)$ was then formed, and the difference from the color component deviating the most was registered. A histogram of these differences is shown in Figure 2. As can be seen in the figure, there is a strong peak around zero, which means that the average color of most blocks does not differ much from colors of adjacent blocks. In fact, if we count the number of blocks where the difference in all three components falls in the interval $[-4,3]$ (marked with black bars), we see that 88% of the blocks fall into this category. Thus, an overwhelming majority of the blocks can be coded differentially using three bits per color component.

Certainly, some blocks cannot be coded this way. Therefore, one bit must be preserved that determines whether we use differential coding in the 4×4 block or not. Preferably, this bit is taken from the table codeword, making it three bits (eight possible tables) instead of four bits (16 tables). This results in a drop in image quality, but a surprisingly small one, only about 0.2 dB averaged over our 20 test images. Alternatively, the bit it could be taken from, say, the blue component of the color code word, but that would defy the purpose of increasing the color accuracy. Taking the bit from the pixel index bits would be hard because two bits are needed per texel. Since we have two table codewords in the 4×4 block (one for each subblock), we end up with one spare bit. We use this bit to indicate whether the subblocks are vertically oriented (two 2×4 blocks side by side) or if they are horizontally oriented (two 4×2 blocks on top of each other).

The bit layout of a 4×4 block is shown to the left in Figure 3, and each block contains the following information:

© The Eurographics Association 2005.

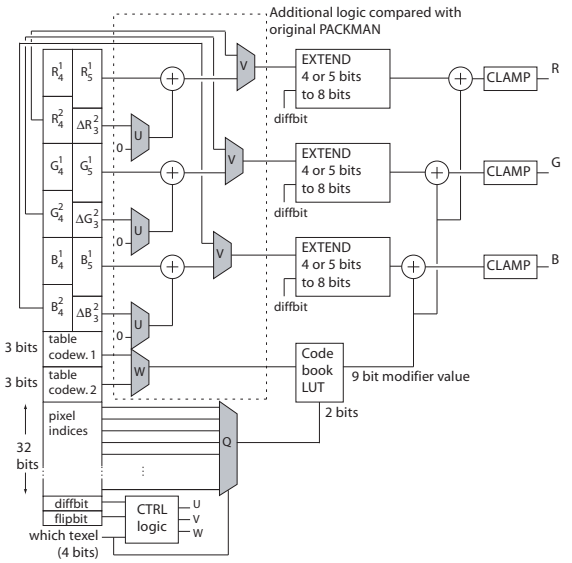

Figure 3: *This diagram shows a possible* iPACKMAN *decompressor. The bit layout can be seen to the left. Compared to the original* PACKMAN *decompressor, only the logic inside the dashed triangle has been added. As can be seen, the total system consists of very little, except for six adders (three six-bit adders and three nine-bit adders), a few MUXes (multiplexor units), and little logic.*

- a *diffbit*, which indicates whether differential or normal coding is used,
- a *flipbit*, indicating whether vertical (*flipbit*=0) or horizontal (*flipbit*=1) orientation is used,
- 16 2-bit *pixel indices* (one for each texel),
- two 3-bit *table codewords* (one for each subblock), indicating which table to use from the codebook, and
- two *color codewords*, which are used (independently or together) to encode the base color for the first subblock and the base color for the second subblock.

If the *diffbit* is set, the two color codewords are (R_5^1, G_5^1, B_5^1) and $(\Delta R_3^2, \Delta G_3^2, \Delta B_3^2)$. The first base color is obtained by expanding the first color code word (R_5^1, G_5^1, B_5^1) to 24 bits. The second base color is obtained by adding the two color code words $(R_5^1 + \Delta R_3^2, G_5^1 + \Delta G_3^2, B_5^1 + \Delta B_3^2)$ and thereafter expanding the result to 24 bits. If *diffbit* is not set, normal RGB444 encoding is used for both base colors, and the first and second base colors are obtained directly by expanding the color codewords (R_4^1, G_4^1, B_4^1) and (R_4^2, G_4^2, B_4^2) to 24 bits.

4.2. Decompression

Figure 3 illustrates a hardware diagram for an *i*PACKMAN decompressor. Below we describe in more detail how a single texel is decompressed using such hardware:

1. First, the base color needs to be obtained. In the differential mode (*diffbit* $= 1$), we should either use the five bit value R_5^1 directly, in which case MUX (Multiplexor unit) U chooses zero, or we should use the sum $R_5^1 + \Delta R_3^2$, in which case MUX U chooses ΔR_3^2. The sign of ΔR_3^2 is extended to six bits before the addition. For instance, if $(R_5^1, G_5^1, B_5^1) = (4, 15, 27)$ and $(\Delta R_3^2, \Delta G_3^2, \Delta B_3^2) = (-4, -2, 3)$, the resulting 5-bit color is $(0, 13, 30)$. No clamping is necessary since the encoder can make sure these values never overflow. In the non-differential mode, (*diffbit* $= 0$), we want either R_4^1 or R_4^2, both four bits. R_4^1 can be selected the same way as R_5^1, where the last bit will be treated as junk and removed in a subsequent step. R_4^2 can be selected by correctly switching MUX V, and is padded with a zero-bit to fit the 5-bit MUX V. The green and blue channels are selected the same way as the red channel.
2. The next step is to extend the 4- or 5-bit value coming out of MUX V to an 8-bit value. This can be done inexpensively by padding the missing lower order bits with the higher order bits. For instance, a four bit value 1011_{bin} will be converted to 10111011_{bin}, and our five bit example above $(0, 13, 30)$ will become $(0, 107, 247)$. The extender will need to get *diffbit* in order to know if it should extend from five or four of the incoming five bits. This is also where the junk bits in R_4^1 and R_4^2 are removed.
3. MUX W will choose which of the two table codewords to use. The 3-bit table codeword is fed to the codebook of tables, and thus a specific table of four values is chosen. Using, for example, a table codeword of 011_{bin} means that the following table is selected: $\{-42, -13, 13, 42\}$ (see the codebook below).
4. Four input bits are used to select which pixel to decompress using MUX Q. The resulting 2 pixel index bits are connected to the codebook, which thus selects a specific modifier value from the selected table of four numbers. For instance, if the pixel index is 11_{bin}, using the table selected above, the modifier value is 42.
5. The final step computes the final decompressed color by adding the modifier value to the expanded base color. Then the color is clamped to $[0, 255]$. For the example above, we will get $(0, 107, 247) + (42, 42, 42) = (42, 149, 255)$, where values have been clamped to $[0, 255]$.

The MUXes marked with U, V and W are operated by signals U through W from the control logic. The control logic takes the in-parameters *diffbit*, *flipbit*, and $\mathbf{w} = w_3 w_2 w_1 w_0$, where $\mathbf{w}$ are four bits describing which texel to decompress. The bits $w_3 w_2$ contain the y-coordinate in the block, and $w_1 w_0$ contain the x-coordinate.

U = *diffbit* AND W
V = *diffbit* OR $\neg$W
W = (*flipbit* AND w_1) OR ($\neg$*flipbit* AND w_3),

where $\neg$ is the *NOT*-operator. The codebook consists of eight different tables, each containing 4 different values. The codebook was generated by starting from random numbers

© The Eurographics Association 2005.

and then optimizing them by minimizing the error for a set of training images. The tables associated with the table codewords are shown below. Note that the first and second values of each table are the third and fourth values negated. Thus, only sixteen of the numbers need to be stored, which is the same number as for PACKMAN.

table index	0	1	2	3	4	5	6	7
	-8	-17	-29	-42	-60	-80	-106	-183
	-2	-5	-9	-13	-18	-24	-33	-47
	2	5	9	13	18	24	33	47
	8	17	29	42	60	80	106	183

4.3. Compression

In PACKMAN, the search space was small enough so that exhaustive search could be carried out by iterating over all possible colors (2^{12}), all possible tables (2^4), and all possible modifier values (2^2). However, for *i*PACKMAN, the colors for two subblocks are dependent, meaning that the color search space increases to (2^{24}), making exhaustive search harder. We have developed three schemes for *i*PACKMAN compression.

The fastest of them starts out by quantizing the average colors of the subblocks to 555 bits, and computes the difference between these. If the difference in each component is within the interval $[-4,3]$, it uses the differential mode. It then tries all tables and modifier values for each subblock, and uses the parameters that gives smallest error compared to the original image. Finally, the same procedure is carried out with the block flipped, and the best mode (flipped/not flipped) is chosen. If the colors cannot be differentially coded, they are encoded individually, quantizing the average color of the block to 444 bits. This is really fast: about 60 milliseconds for a 128×128 texture on a 1.2 GHz laptop computer.

However, due to the fact that the luminance is later modified, it is not certain that the 555 color closest to the base color is the best quantization. Therefore, in our second compression approach, all color pairs within ± 1 quantization steps are searched. For each color pair, all possible tables and modifier values are tried out. For the non-differential mode, exhaustive search can be used to find the 444 representation of the base color. This search method takes about 20 seconds for a 128×128 image, which is still much faster than the exhaustive mode for PACKMAN. The reason is that most blocks are differentially coded, and therefore the costly exhaustive search is mostly avoided.

If the two base colors are just out of reach of each other to be coded differentially, it can sometimes be better to move one of the colors closer so that differential coding becomes possible, than to use 444 encoding. A third scheme uses this fact, and tries differential encoding for all blocks whose base colors differ less than $[-9,8]$ from each other. Non-differential coding is also tried for all these blocks, and the best representation wins. This scheme is slow: about seven minutes per 128×128 texture.

4.3.1. Error Metric

When finding which of two representations is better, the two representations are decompressed, and an error metric is calculated over the block. The choice of error metric affects the selection of the luminance modifier. Disregarding clamping, finding the correct luminance modifier means finding a scalar k such that the base color $\mathbf{b}$ plus the modification $k(1,1,1)$ is as close as possible to the desired color $\mathbf{d}$:

$$\mathbf{b} + k(1,1,1) \approx \mathbf{d}.$$

For two colors $\mathbf{u} = (u_r, u_g, u_b)$ and $\mathbf{v} = (v_r, v_g, v_b)$, a simple error metric is described by:

$$e^2_{normal}(\mathbf{u},\mathbf{v}) = (u_r - v_r)^2 + (u_g - v_g)^2 + (u_b - v_b)^2.$$

The optimal k can be found by projecting the difference $\mathbf{d} - \mathbf{b}$ onto $(1,1,1)$. However, since the eye is more sensitive to green than to red and blue, it makes sense (from a perceptual point of view) to let green come closer to its desired value at the cost of a worse representation of blue and red. This can be done by changing to a more perceptually balanced error metric:

$$e^2_{percept}(\mathbf{u},\mathbf{v}) = w_r^2(u_r - v_r)^2 + w_g^2(u_g - v_g)^2 + w_b^2(u_b - v_b)^2,$$

where w_g can be larger than w_r and w_b and where $w_r^2 + w_g^2 + w_b^2 = 1$. $e_{percept}$ can be written in matrix form as

$$e^2_{percept}(\mathbf{u},\mathbf{v}) = (\mathbf{u} - \mathbf{v})^T W^T W (\mathbf{u} - \mathbf{v})$$

where $W = \mathrm{diag}(w_r, w_g, w_b)$. It turns out that the optimal k again can be found by projecting $\mathbf{a} = \mathbf{d} - \mathbf{b}$ onto $\mathbf{f} = (1,1,1)$, but now using the weighted scalar product $\langle \mathbf{a} | \mathbf{f} \rangle = \mathbf{a}^T W^T W \mathbf{f}$ instead of the unweighted $\langle \mathbf{a} | \mathbf{f} \rangle = \mathbf{a}^T \mathbf{f}$ as used before. We define luminance Y of a color $\mathbf{c}$ as $Y(\mathbf{c}) = 0.299c_r + 0.587c_g + 0.114c_b$, as is common in broadcast TV systems. If we choose the weights $(w_r, w_g, w_b) = (\sqrt{0.299}, \sqrt{0.587}, \sqrt{0.114})$, we find that the weighted scalar product results in a projection along a line of constant luminance, which means that the projected color has *exactly* the same luminance as the desired color, that is, $Y(\mathbf{b} + k(1,1,1)) = Y(\mathbf{d})$. This is a highly desirable effect. It means that edges come out much clearer since a monotonic ramp in luminance will be monotonic even after compression, something which is not guaranteed otherwise. In Figure 7 (color plate), we show two example images compressed with the normal and the perceptual error metric. Note that edges between different color areas are clearer with the perceptual error metric (this difference may be more pronounced on-screen than in print).

It should be noted that using YCrCb as done above is only a first-order approximation, and that other color spaces could

© The Eurographics Association 2005.

be used, such as, for example, CIE-Luv and Lab. However, these are non-linear spaces, which makes analysis more difficult.

5. Results

In this section, we present results showing the image quality of different texture compression schemes. Our new *i*PACKMAN system is compared to PACKMAN [SAM04], S3TC [INH99], and to the 4-bit version of PVR-TC [Fen03].

For PACKMAN, we have used exhaustive search in order to maximize quality, and for S3TC we have used ATI's The Compressonator v1.23.1049. The weights $(1,1,1)$ were used to maximize the quality metric. This may be the reason why S3TC performs better in our paper compared to Fenney's results. There is no publicly available codec for PVR-TC, so therefore we reverted to comparing exactly the same images used by Fenney [Fen03]. Most of these images are taken from an image test suite by Kodak. These are non-square, and therefore the top left 512×512 part of the images were used. None of these images were part of the training set used for optimizing the codebook.

Fenney reported his results in *root mean square error* (RMSE):

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

where w and h are the width and the height of the image, and ΔR_{xy}, ΔG_{xy} and ΔB_{xy} are the pixel differences in pixel (x,y) between the original and the decompressed image in the red, green and blue component respectively. We have chosen to present our results in *Peak Signal to Noise Ratio* (PSNR) instead:

$$PSNR = 10 \log_{10} \left(\frac{3 \times 255^2}{RMSE^2} \right), \qquad (1)$$

where the scale factor 3 in the numerator is due to the fact that 3×255^2 is the peak energy in a pixel.

Table 2 and Figure 4 show the result for our test suite of images. We use Equation 1 to convert the RMSE numbers from Fenney's paper to PSNR; his original RMSE values are preserved in brackets. The rightmost column in Table 2 shows the PSNR increase *i*PACKMAN gives averaged over all the images compared to each of the other texture compression schemes. *i*PACKMAN is thus 2.54 dB better than PACKMAN, 0.41 dB better than S3TC and 0.65 dB better than PVR-TC. To put this in perspective, a common rule of thumb used in the image compression community says that 0.25 dB makes for a visible difference. In Figure 4, the same results are showed in the form of a diagram. Considering individual images, we see that *i*PACKMAN outperforms PACKMAN for every image in the test, and it also beats S3TC and PVR-TC in five out of the seven images.

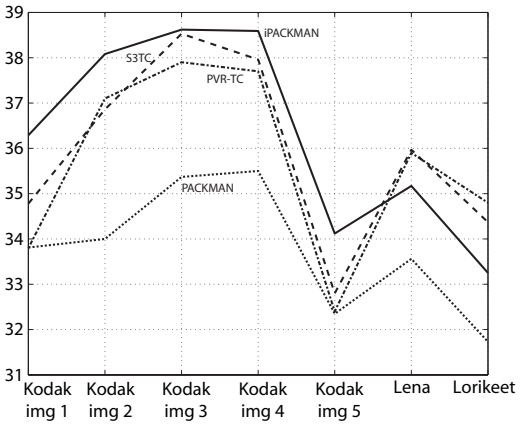

Figure 4: *Here the results from Table 2 are summarized as a graph. As can be seen, iPACKMAN is better than PACKMAN for all images, and iPACKMAN is better than S3TC and PVR-TC for 5 out of 7 images.*

Every compression system has its relative strengths and weaknesses. For instance, even though *i*PACKMAN on average outperforms S3TC for the images in our small test set, there are several blocks in these images where S3TC is better than *i*PACKMAN . In Figure 8 (color plate), the top row shows a part of an image with smooth chrominance transitions. Here S3TC is clearly superior to *i*PACKMAN, and Fenney's scheme based on a low frequency modulation would probably perform even better. The relative strength for *i*PACKMAN is in luminance detail—the bottom row of Figure 8 depicting a face shows how *i*PACKMAN preserves luminace detail better than S3TC, due to the possibility to have more than four colors in a 4×4 block. We have also included a game texture (middle row of Figure 8). Figure 9 (color plate) also shows how *i*PACKMAN performs on text; black or white text on colored blackground (or colored text on black or white background) looks significantly better than if both the text and the background are colored. The strengths of iPACKMAN in luminance can also be seen in Figure 5.

Without real hardware implementations of each decompressor, it is hard to compare the complexity of the different schemes. However, given the few number of components of *i*PACKMAN, as shown in Figure 3, it is quite clear that our system is of very low complexity.

6. Transparency

Most texture compression schemes can also handle transparency in some way. To be able to do that as well, more bits per block are needed. Here, we will describe two different solutions.

The first solution simply uses four bits of alpha for each pixel, and so $4 \times 4 \times 4 = 64$ extra bits are used. This makes it possible to have 16 different transparency values per pixel.

© The Eurographics Association 2005.

	Kodak img 1	Kodak img 2	Kodak img 3	Kodak img 4	Kodak img 5	Lena	Lorikeet	Avg gain
PACKMAN	33.81	34.00	35.37	35.50	32.35	33.56	31.73	+2.54 dB
S3TC	34.78	36.82	38.53	37.96	32.80	35.97	34.37	+0.41 dB
PVR-TC	33.8 [8.98]	37.1 [6.20]	37.9 [5.61]	37.7 [5.76]	32.4 [10.59]	35.9 [7.11]	34.8 [8.08]	+0.65 dB
*i*PACKMAN	36.29	38.08	38.62	38.59	34.12	35.17	33.25	—

Table 2: *The PSNR is reported from a test suite of images for PACKMAN, S3TC, PVR-TC, and* i*PACKMAN. The rightmost column shows the average gain when comparing* i*PACKMAN to the other schemes.*

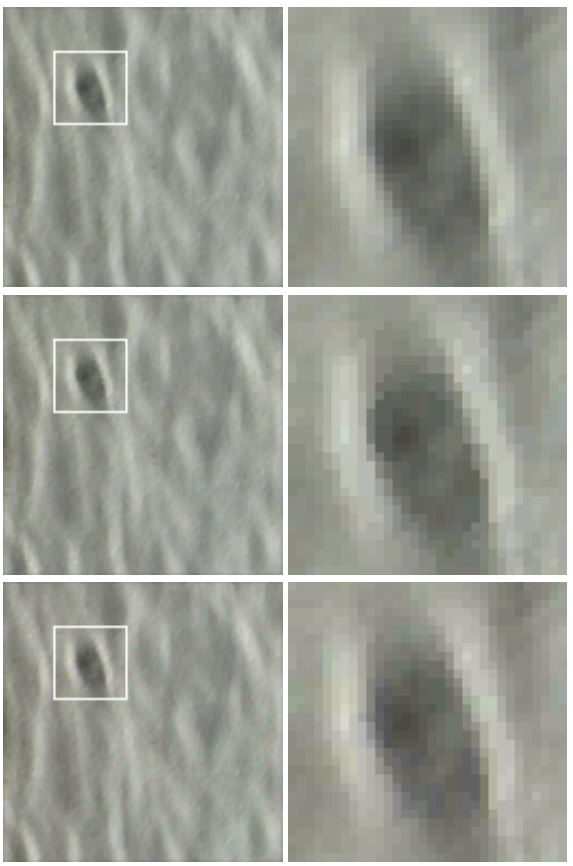

Figure 5: *Top: Original. Middle: S3TC. Bottom:* i*PACKMAN. Note how the ability to have more than four gray levels per* 4×4 *block is beneficial to* i*PACKMAN.*

Such a scheme would thus require $64 + 64 = 128$ bits, resulting in 8 bits per pixel (bpp). This is equivalent to the way that the alpha information is represented in the 8 bpp DXT2 and DXT3 formats — their first 64 bits contain color information, and the other 64 bits contain 4-bit-per-pixel alpha.

The second solution uses the codebook technique for transparency as well. However, instead of having a three-component base color per 2×4 (or 4×2) subblock, only a scalar value is required. Thus, two *base alpha values* are required per 4×4 block, and these can be encoded in 8 bits (either $5 + 3$ or $4 + 4$, analogously to the color compression case). A *flipbit* and a *diffbit* for the intensity are also used (2 bits), as well as two 3-bit table codewords, one for each sub-block (6 bits). However, these select tables that are eight values long instead of four values. Hence a 3-bit pixel index is needed per pixel (48 bits). Again, this sums up to 128 bits per 4×4 block, or 8 bpp. However, this second method can take advantage of the spatial redundancy of alpha images to produce better images than the first method. This way of encoding is more akin to the 8 bpp formats DXT4 and DXT5 techniques, where the coding of the alpha information is similar to the coding of the color information (but different from the techniques proposed in this paper).

It should be pointed out that this is work in progress. Yet another solution could be to attempt to get transparency into a variant of the 4 bpp *i*PACKMAN scheme.

7. Conclusion and Future Work

For mobile devices, such as portable game consoles and mobile phones, it is of uttermost importance to preserve bandwidth usage as much as possible as this significantly reduces power consumption. Furthermore, for these devices, the implementation complexity must be kept small due to size constraints. We argue that our presented *i*PACKMAN texture compression system fulfils these demands as it provides compression at 4 bits per pixel with very small hardware complexity. Furthermore, averaging the PSNR over our test suite of images, *i*PACKMAN has proven to provide better image quality than both S3TC, PACKMAN, and PVR-TC. It should noted, however, that *i*PACKMAN is not limited to usage on mobile devices.

There are several possible improvements to the *i*PACKMAN scheme that are candidates for future work. In our scheme we have used the same codebook for the differentially coded blocks (*diffbit*= 1) as for the individually coded blocks (*diffbit*= 0). It is not hard to imagine that the differentially coded blocks usually are smoother and that they therefore should use a codebook with smaller values. It would also be interesting to attempt to adapt our technique to normal maps. We have done preliminary work by treating a XYZ-normal map in tangent space as an RGB texture and compressed it with *i*PACKMAN. The result, which can be seen in Figure 6, indicates that *i*PACKMAN could be used at least for some normal maps. Future work should include a comparison with state-of-the-art normal compression schemes, and investigate what artifacts *i*PACKMAN gives.

© The Eurographics Association 2005.

Acknowledgments

Thanks to Eric Fausett for pointing out that exhaustive search is indeed feasible (by independently implementing it). Thanks also to `www.gametutorials.com` for the "game texture," and to Kevin Harris for the normal map texture. Finally, thanks to the Swedish Foundation for Strategic Research for funding part of this project.

References

[AMN03] AILA T., MIETTINEN V., NORDLUND P.: Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3 (2003), 792–800.

[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.

[BAC96] BEERS A., AGRAWALA M., CHADDA N.: Rendering from Compressed Textures. In *Proceedings of SIGGRAPH* (1996), pp. 373–378.

[CDF*86] CAMPBELL G., DEFANTI T. A., FREDERIKSEN J., JOYCE S. A., LESKE L. A., LINDBERG J. A., SANDIN D. J.: Two Bit/Pixel Full Color Encoding. In *Proceedings of SIGGRAPH* (1986), vol. 22, pp. 215–223.

[DM79] DELP E., MITCHELL O.: Image Compression using Block Truncation Coding. *IEEE Transactions on Communications 2*, 9 (1979), 1335–1342.

[Fen03] FENNEY S.: Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware* (2003), ACM Press, pp. 84–91.

[HG97] HAKURA Z. S., GUPTA A.: The Design and Analysis of a Cache Architecture for Texture Mapping. In *24th International Symposium of Computer Architecture* (June 1997), ACM/IEEE, pp. 108–120.

[IEH99] IGEHY H., ELDRIDGE M., HANRAHAN P.: Parallel Texture Caching. In *Graphics Hardware* (1999), ACM Press, pp. 95–106.

[IK00] IVANOV D., KUZMIN Y.: Color Distribution – A New Approach to Texture Compression. In *Proceedings of Eurographics* (2000), vol. 19, pp. C283–C289.

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and Method for Fixed-Rate Block-based Image Compression with Inferred Pixels Values. In *US Patent 5,956,431* (1999).

[KSKS96] KNITTEL G., SCHILLING A., KUGLER A., STRASSER W.: Hardware for Superior Texture Performance. *Computers & Graphics 20*, 4 (July 1996), 475–481.

[MB98] MCCABE D., BROTHERS J.: DirectX 6 Texture Map Compression. *Game Developer Magazine 5*, 8 (August 1998), 42–46.

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

[SAM04] STRÖM J., AKENINE-MÖLLER T.: PACKMAN: Texture Compression for Mobile Phones. In *Sketches program at SIGGRAPH* (2004).

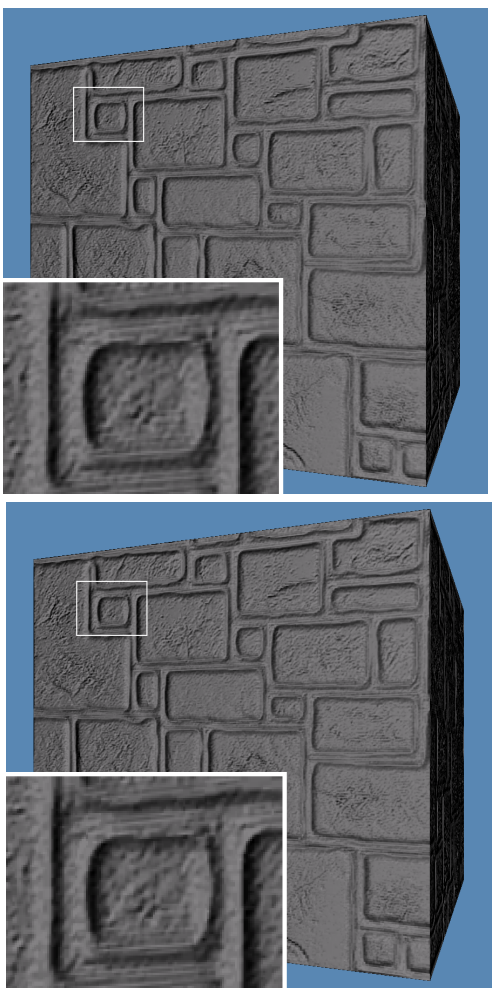

Figure 6: *Rendered images from a surface with uniform gray color and normal mapping. Top: Original. Bottom: Normal map compressed with iPACKMAN. Texture courtesy of Kevin Harris.*

© The Eurographics Association 2005.

Figure 7: *Left: Original. Middle: Images compressed using normal error measure. Right: Images compressed using perceptual error measure. Notice the yellow/white edge (top) and the purple/green edge (bottom). (May look nicer on screen than in print.)*

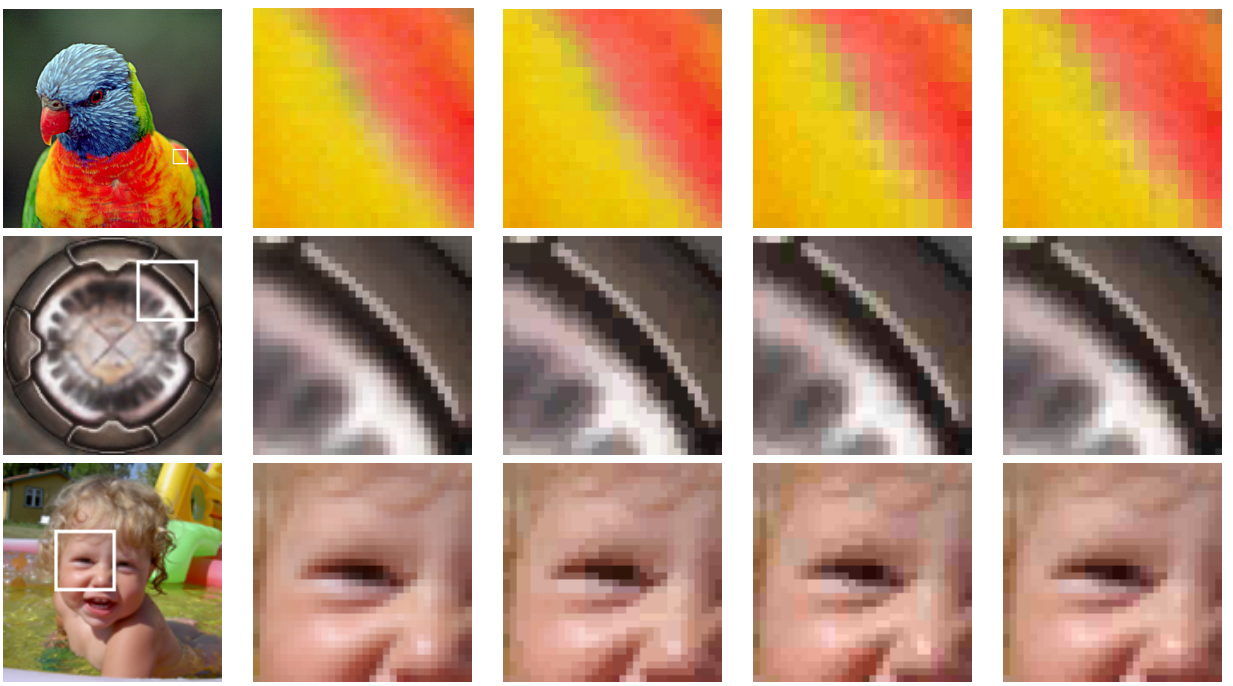

Figure 8: *Top row: Here we show a weakness of* i*PACKMAN. In general,* i*PACKMAN performs worse than S3TC in regions with smooth chrominance transitions. Middle row: A game textue (courtesy of* `www.gamedevelopers.com`*). Bottom row: Note how the block artifacts in S3TC in the eyes region dissapear with* i*PACKMAN.*

Figure 9: *Text compression: Note how black or white text on a colored background works decently for* i*PACKMAN, whereas our coder has more difficulties with colored text on colored background.*

© The Eurographics Association 2005.