Texture Compression: THUMB – Two Hues Using Modified Brightness

Martin Pettersson Jacob Ström

Ericsson Research

Abstract

We present a new texture compression system called THUMB, that can be used either as a stand-alone compression system or in combination with the *i*PACKMAN algorithm. We show how the combined system improves quality in the test images we have used, especially in the image blocks most problematic to *i*PACKMAN.

1 Introduction

Bandwidth is usually the factor limiting performance in rasterization-based rendering hardware [Aila et al. 2003]. Knittel et al. [1996] and Beers et al [1996] show how *texture compression* can be used to reduce bandwidth during rendering. By transferring the texels over the bus in compressed form, and decompressing needed texels on-the-fly, texture bandwidth can be reduced significantly. Previously introduced image compression techniques such as the CCC scheme [Campbell et al. 1986] can be used for the compression.

For mobile devices, which are powered by batteries, these bandwidth savings are also important from a power consumption perspective, since such off-chip memory accesses are often the most energy consuming operations in a computer system [Fromm et al. 1997]. In low-power processes, such as the ones used for mobile devices, off-chip memory accesses are more than an order of magnitude more energy consuming than accesses to a small on-chip SRAM memory. The bandwidth savings can therefore be translated into energy savings. The texture compression system presented here was originally intended for use on mobile devices, but could be used on PC systems and game consoles as well.

A texture compression system differs from a normal image compression system in a number of ways. First, it needs to allow random access to the texels, since rendering can start in any location in the texture, and be traversed in a non-scanline fashion. Most texture compression systems are therefore block-based fixed rate codecs, where each block of the image is given a fixed number of bits, which makes it simple to calculate the address of a particular block. Second, the decompression of a block should ideally be of low complexity. If some sort of filtering is used, many parallel decompression units are needed to process a single pixel. For instance, if trilinear filtering is used, eight parallel decompression units are needed to process one pixel per clock. By decompressing the texels right before filtering, it is possible to keep compressed texels in the texture cache, which means that the cache can be made several times smaller in terms of chip surface area. Third, it is advantagous to avoid texture dependent look-up tables (LUTs), such as color palettes, since the indirect addressing they introduce give rise to latencies that are hard and costly to hide.

Our new texture compression system is developed with *i*PACKMAN texture compression [Ström and Akenine-Möller 2005] in mind, and designed to be a complementing mode in that coder, taking care of the blocks that *i*PACKMAN has most difficulties with. However, it can also be used as a stand-alone codec, and we have presented results for both usages.

2 Previous Work

We will now go through previous work that is related to texture compression.

Delp and Michell [1979] present a scheme called block truncation coding (BTC) for gray scale images. The image is divided into 4×4 blocks, and two shades of gray are encoded in the block, together with a bit mask that decides for each pixel what shade to choose. The bit mask is thus 16 bits, and eight bits are used for each gray level, resulting in 32 bits per 4x4 block or 2 bits per pixel (bpp).

Campbell et al [1986] extend the BTC algorithm to color images in a system called CCC — Color Cell Compression. Each 4×4 block now includes two colors instead of two gray scales. By using a 256-wide color palette, the colors can be represented with eight bits each, yielding 2 bpp for color images. However, only two colors are possible per block, which limits image quality. Furthermore, having a color palette is a drawback in today's systems, where memory accesses are slow in relation to computation.

The de facto standard today is the S3TC texture compression method by Iourcha et al. [1999], and it can be seen as an extension of CCC. To increase quality compared to CCC, four colors can be chosen in each pixel, yielding two bits per pixel in the bit mask. To avoid a texture dependent LUT, no color palette is used. Instead two colors in RGB565 format are stored in the block, and two more colors are interpolated in-between these two colors. This means that colors in a block are restricted to lying on a line in RGB space. However, this is a rather good approximation of the color distribution in blocks from most natural images. An example can be seen in the left diagram in Figure 1, which shows a cross section of the RGB space, and where the colors of a block are plotted as points in RGB space. The point cloud is approximated by four equidistant reconstruction points along a line, as shown in the right diagram. The two end points (marked with squares) are the colors stored in the block, whereas the two middle points (marked with circles) are interpolated. With 64 bits per 4×4 block, the rate of S3TC is 4 bpp.

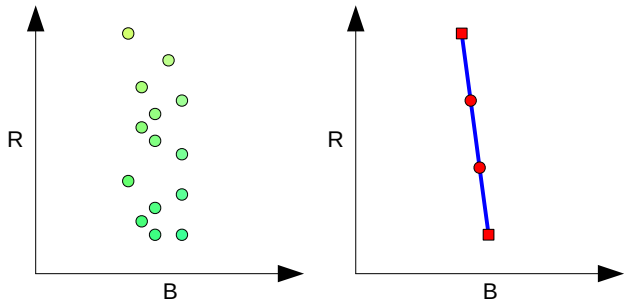

Figure 1: Left: Possible distribution of the colors of a 4×4 block in RGB-space. Right: In S3TC the colors are approximated by four equidistant points along a line.

Akenine-Möller and Ström [2003] present a variant of S3TC called POOMA, where the biggest difference is that only one in-between color is used, and blocks are 2×3 pixels. Here the main

Figure 2: Here, the core idea of PACKMAN is illustrated. To the left, the base color 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.

target is to reach 32 bits per block to match bus-sizes of mobile phones on a rendering system without a texture cache. However, this reduces quality and increases the rate as compared to S3TC, and the block size of 2×3 pixels is awkward for hardware implementation.

Beers et al. [1996] use vector quantization for texture compression to reach compression rates of 1-2 bpp. However, this requires a big LUT, which introduces indirect addressing resulting in latencies that can be hard to hide.

Fenney [2003] uses a different approach, exploiting the fact that an upscaled low-resolution version of an image is often similar to the image itself. Fenney uses two such low-resolution images A and B, both upscaled bilinearly two times, yielding only one color sample each per 4×4 block. Each pixel can then choose its color from either image A, image B, or from two blend values between A and B. Using 16 bits for each color and 2 bits per pixels for choosing the blend value results in 64 bits per block or 4 bpp. A 2 bpp mode is also present.

Since our work is built on the PACKMAN [Ström and Akenine-Möller 2004] and *i*PACKMAN algorithms, we will go through them in more detail in the next section.

3 PACKMAN and iPACKMAN

The PACKMAN algorithm exploits the fact that the human visual system is more sensitive to changes in luminance than in chrominance. It takes the rather radical approach of only having a single chrominance per 2×4 pixel block, represented as a RGB444 color (12 bits). Each pixel can then modify the luminance of this base color additively, as shown in Figure 2. More specifically, a *modifier value* is added to all three components (R, G and B) of the base color. The modifier value is taken from a small table of four entries, and hence two bits, called *pixel indices* are needed to select the value for each pixel. Finally, four bits are spent on a *table codeword*, to select the small table from a list of 16 prefixed tables. Altogether, 32 bits are used for 2×4 pixels, giving a rate of 4 bpp.

3.1 iPACKMAN

This algorithm has been improved under the name *i*PACKMAN (also called Ericsson Texture Compression, ETC) in two ways [Ström and Akenine-Möller 2005]. First and most important, a differential mode is introduced, allowing two neighboring 2×4 blocks to be coded together. The base color of the left block can then be encoded using RGB555, i.e., with higher precision, and the right base color also in RGB555 format, but coded using a differential dRdGdB333, where dR, dG and dB can assume values between -4 and $+3$. Thus, for pairs of blocks with similar base colors, the chrominance resolution effectively goes up from RGB444 to RGB555 in both blocks. Blocks that cannot be encoded well using the differential mode will be coded as before, i.e., with two individually coded RGB444 colors. This mode is called the individual mode.

Figure 3: Left: Original. Right: Image compressed using *i*PACKMAN. Note the blocky artifacts coming from that only one hue is allowed per subblock (not visible in b/w reproduction).

The second improvement is that blocks can be flipped so that a 4×4 block consists of either two 2×4 block next to each other, or two 4×2 blocks on top of each other. Two mode bits are needed, one to choose between individual and differential mode, and one to indicate the flip status. Space for these two bits are created by shrinking the number of possible tables from 16 to eight, thus reducing the number of table bits in each sub-block from four to three. Figure 8 shows the bit layout in the differential (top) and the individual (bottom) modes.

These two small differences have a substantial effect on image quality, which jumps 2.5 dB in terms of Peak Signal to Noise Ratio (PSNR), suddenly putting *i*PACKMAN on par with S3TC. Visually, *i*PACKMAN lacks the disturbing banding artifacts that are a result of the low chrominance resolution in PACKMAN. However, *i*PACKMAN does not change the fact that only one chrominance can be used for a block of eight pixels.

4 THUMB Texture Compression

In this section, we present our new THUMB texture compression scheme. First we describe the design and motivate the different design choices. Then follows descriptions for decompression and compression of the stand-alone version of THUMB. The last subsection describes how THUMB can be combined with *i*PACKMAN to get a better solution.

4.1 Basic Design and Motivation

Almost all fixed-rate block compression techniques have particular blocks that are coded worse than others. This is also the case for *i*PACKMAN. The overall performance of *i*PACKMAN is very good, but subblocks with more than one distinct hue are sometimes coded with poor result as the pixel value can only be modified in the direction $(1, 1, 1)$ in RGB space. An image with typical problem blocks is shown in Figure 3. The image is cropped from a larger image showing a road with a yellow line. Since the human visual system is good at picking up block artifacts such as these, the main goal of this paper is to be able to better handle such blocks. However, we do not want to compromise quality in other blocks in order to reach that goal, so our secondary goal is that overall quality should stay the same or increase.

Our new scheme is called *Two Hues Using Modified Brightness*, or THUMB for short. Just like *i*PACKMAN, it is based on 4×4 blocks. In the stand-alone version, each block is coded with 64 bits. As the name suggests, THUMB can handle up to two different hues

in a block. To allow this, two independent base colors are used to code each block. This is also the case for *i*PACKMAN where each base color is restricted to a 2×4 subblock. In THUMB however, each base color can be used for any pixel in a block. The colors are encoded in RGB554 format.

Modifying the brightness for each pixel has proved to be a good solution for *i*PACKMAN. Therefore this approach has been used in THUMB as well. The technique however is somewhat different. 32 bits are used as pixel indices, giving us two bits to code each one of the 16 pixels in a block. Hence, we can have four different paint colors.

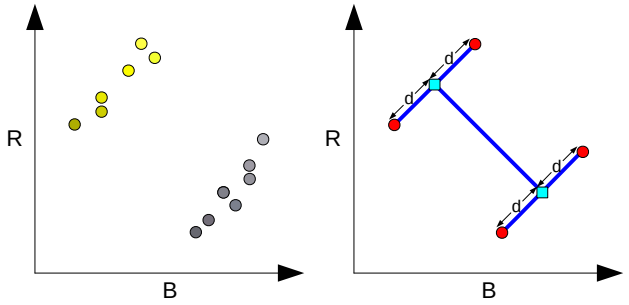

Figure 4: Left: The colors of the original block can be located in two different hues in RGB space. Right: Two base colors (marked with squares) are selected. Four paint colors (marked with circles) are derived by adding a distance d in direction $(1,1,1)$ to form an H-pattern.

These paint colors can be chosen in several ways since we are using two independent base colors per block. THUMB defines two different patterns of how to retrieve the four paint colors. In the first pattern, these paint colors are derived using a distance d added in direction $(1,1,1)$ from the base colors. This pattern is called the H-pattern, since the paint colors and the base colors can be placed as an H in RGB space. The H-pattern is illustrated in Figure 4, where a cross section of the RGB space is shown. Note that, just as in S3TC, the colors are approximated with line segments. Whereas S3TC can choose any orientation of the line segment (see Figure 1), the line segments in THUMB must be oriented parallel with the intensity direction $(1,1,1)$. On the other hand, THUMB can use two line segments whereas S3TC can use only one.

Sometimes the colors of the original block are clustered more around one base color than the other. A different pattern might then be a better match. In the second pattern, the first two paint colors are the base colors themselves. To get the other two paint colors, the distance d is added to the first base color in direction $(1,1,1)$. In this way, three paint colors with the same hue can be represented in a block. The fourth paint color is then chosen independently from the first three. This pattern is called the T-pattern, since the paint colors can be placed to form a T in RGB space as illustrated in Figure 5. A *pattern bit* is used to resolve which one of the two patterns to use when generating the paint colors. The patterns are approximately equally common. The distance d is coded using three bits in the stand-alone version. As a total we will have eight possible distance values to choose from for each block. The distances are taken from a hardware lookup table. This table was created using a combination of different optimizing techniques for a test suite of twenty images. The optimized table is shown in Table 1. We have tried approximating the tables using shifts as well, and even though it only gives a small performance loss, it is not clear that we actually would gain much in terms of HW complexity.

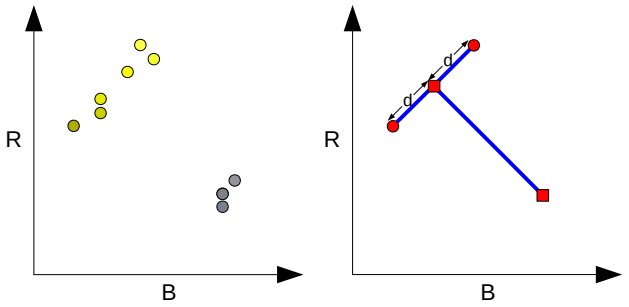

Figure 5: Left: An uneven distribution of the original block colors. Right: The T-pattern. Both base colors are used as paint colors. The distance d is added in direction $(1,1,1)$ to get the other two.

table index	0	1	2	3	4	5	6	7
distance	3	6	11	16	23	32	41	64

Table 1: Hardware lookup table with optimized distances.

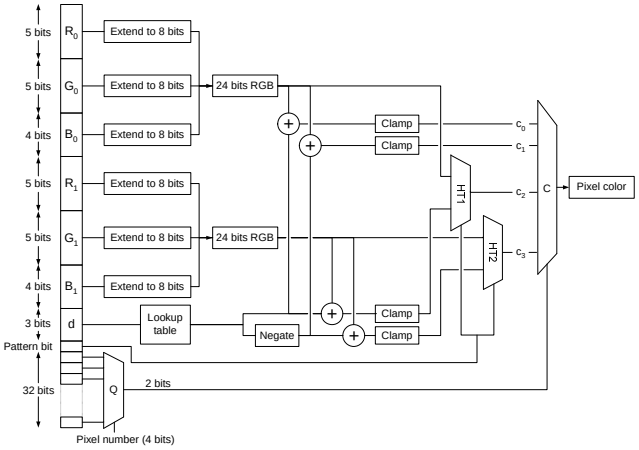

Figure 6: Hardware diagram for a possible THUMB decoder. The bit layout can be seen to the left. The expanded color components have been combined to RGB-colors to make the diagram more readable.

4.2 Decompression

Figure 6 shows a possible implementation of how to decode a pixel in THUMB. At most, the values of twenty bits are needed to retrieve a pixel. The two bits from the pixel indices are used to determine which one of the paint colors to decode. Below is a description of how each paint color is derived:

1. To retrieve the first paint color, the first base color must be read. Each component is expanded to eight bits to form a 24-bit RGB-color. The distance is read from the lookup table, and added to each component of the base color. The color components are then clamped to the interval $[0,255]$, resulting in the first paint color.

2. The second paint color is decoded in a similar way as the first one. The only difference is that the distance is negated before it is added to the components of the base color.

3. Decoding the third paint color is a bit trickier than the first two ones. Depending on the value of the *pattern bit*, two different

paint colors can be acquired. If the *pattern bit* is zero, the first base color is expanded and used as the paint color without adding a distance. If the *pattern bit* is set, the second base color is expanded and the distance is added to get the third paint color.

4. The *pattern bit* is also needed to determine the fourth paint color. If the *pattern bit* is zero, the second base color is expanded and used as paint color. If the *pattern bit* is set, the negated distance is added to all components of the expanded second base color to get the fourth paint color.

4.3 Compression

The problem of compression is to find the best possible pair of base colors. Exhaustive search is not yet feasible due to the number of combinations that must be tested. Iterating over all possible base colors (2^{28}), patterns (2), distances (2^3) and paint colors (4) means that up to 2^{34} different combinations would have to be tried for each pixel. Therefore, three non-exhaustive compression methods have been developed.

LBG Compression. The LBG vector quantization algorithm [Linde et al. 1980] is used to find the two base colors. Since we only have two reconstruction values (the base colors), the algorithm converges quite fast. Starting with two random base colors, only ten iterations are needed to get a satisfying result. After the base colors are found, all possible patterns, distances and paint colors are tried. The parameter combination giving the lowest Mean Square Error (MSE) is chosen for the block. Encoding a 512×512 texture takes less than five seconds using a 800 MHz PC with 256 MB of RAM.

Radius Compression. This method is much slower than LBG compression, but will also give a better result. Initially, two base colors are found using the LBG-algorithm as above. Then for each quantized base color, all possible colors within a $(2k+1) \times (2k+1) \times (2k+1)$ cube centered around the base color are tried. Loosely speaking, k can be called the radius of the cube, hence the name. The encoding time increases very quickly with respect to k, while the gain in image quality is decreasing: Since there are two colors per block, and they cannot be tested independently, radius compression is $(2k+1)^6$ times slower than LBG compression. For instance, radius compression with $k = 1$ is 729 times slower than LBG compression, $k = 2$ is 15625 times slower and so on. In practice, the extra encoding time for a radius level over two will not justify the small gain in quality. The gain in image quality using the first level of radius is on average around 1 dB in terms of Peak Signal to Noise Ratio (PSNR), compared to LBG compression.

Selective Compression. This solution exploits the fact that all surrounding colors in radius compression are not equally probable. Empirical studies show that for almost all blocks where the first radius level is used, it is sufficient to try only the colors shown in figure 7. This means that only nine different colors need to be tested for each base color instead of 27. Thus, the encoding time is decreased a factor nine, compared to radius compression. The loss in image quality however, is only about 0.05 dB.

4.4 Combining THUMB and iPACKMAN

Blocks containing two distinct hues are coded very well with THUMB. However, the overall performance is in general slightly worse than for *i*PACKMAN. This is mainly because *i*PACKMAN

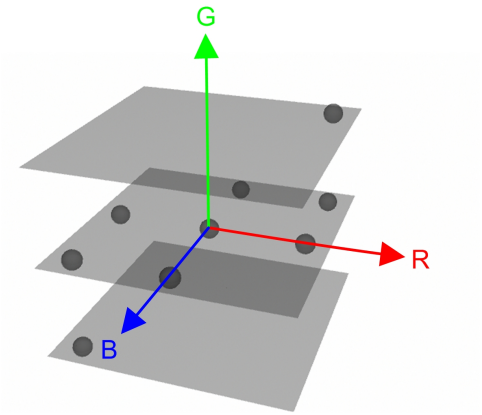

Figure 7: All colors tried in radius search are not equally common. In selective compression, only the most frequent colors from radius compression are tested. The figure shows the constellation of these colors for the first level of radius.

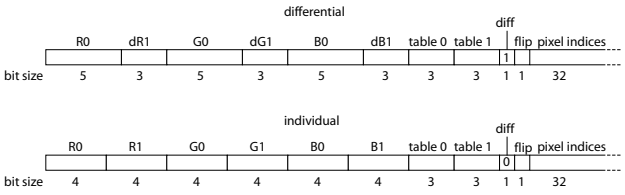

Figure 8: Modes in original *i*PACKMAN. Top: Differential mode. Bottom: Individual mode.

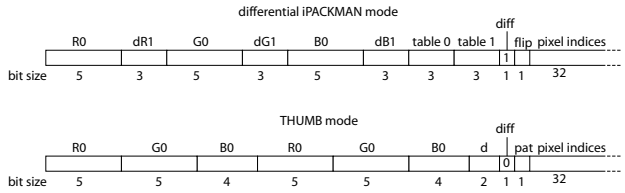

Figure 9: Modes in the combined *i*PACKMAN and THUMB coder. Note how the individual mode in *i*PACKMAN has been exchanged with a 63-bit THUMB mode.

has the possibility to have eight different paint colors per 4×4 block, as the blocks are divided into two subblocks.

Since *i*PACKMAN and THUMB are good at different types of blocks, it makes sense to combine the two. Each block is encoded using both *i*PACKMAN and THUMB separately. The one giving the lowest MSE will be used to represent the block. In order to fit both *i*PACKMAN and THUMB into 64 bits, both algorithms need to be represented using fewer bits.

How to do this is not obvious—the design space is truly huge—and we have only been able to cover a small number of the possible constellations. However, the solution that has given the best result among the ones we have tried is also the one that we think is the most straight-forward: The individual mode in *i*PACKMAN is replaced with a 63-bit THUMB mode, as can be seen in Figures 8 and 9. This has a number of advantages: Firstly, all decoding hardware for the individual mode in *i*PACKMAN can be removed and replaced by the THUMB mode. Also, when encoding, only two modes (differential, THUMB) need to be tested, just as in *i*PACKMAN (differential, individual). It is also likely that blocks

	Kodak img 1	Kodak img 2	Kodak img 3	Kodak img 4	Kodak img 5	Lena	Lorikeet	Avg gain
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]	+0.85 dB
S3TC	34.78	36.86	38.53	37.96	32.80	35.97	34.37	+0.61 dB
*i*PACKMAN	36.29	38.09	38.62	38.59	34.12	35.17	33.25	+0.21 dB
Stand-alone THUMB	34.80	36.86	37.88	37.34	32.97	35.31	33.69	+0.96 dB
Combined Solution	36.26	38.12	38.92	38.66	34.08	35.66	33.88	—

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

that are coded using the individual mode in *i*PACKMAN will be well represented with THUMB, since two very different base colors are easily representable in THUMB.

Combining *i*PACKMAN with THUMB in this way means that THUMB must operate at 63 bits, since the diff bit occupies one bit. This is solved by using two instead of three bits for the distance table. The new optimized table contains the distances { 6 13 24 43 }.

5 Results

In this section we compare the image quality of different texture compression schemes. Our new schemes THUMB and the combined solution are compared to *i*PACKMAN, S3TC and PVR-TC.

To maximize image quality, the slowest compression mode of *i*PACKMAN has been used.

THUMB is encoded with radius compression using the second level of radius. This is also the case for the THUMB-mode in the combined solution.

S3TC is encoded using the Compressonator software package from ATI. The DirectX mode was used in this comparison, setting the weights to $(1,1,1)$ for the lowest error score.

There is no publicly available codec for PVR-TC, so the results are taken from Fenneys publication.

The results of PVR-TC was presented in *root mean squared error* (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.

To be able to compare the results with PVR-TC, the same seven images used for the testing of PVR-TC have been used here. Just as in Fenney's study, these have been cropped to 512×512 pixels.

The results of the comparison between the different schemes are found in Table 2. It can be seen that the combined solution outperforms both S3TC and PVR-TC with over 0.5 dB. Using THUMB separately is almost one dB worse than using the combined solution. The same number for *i*PACKMAN is only 0.21 dB. Generally in image coding, a difference below 0.25 dB is hard to notice. However, the main goal of this paper was not to increase overall quality, but to handle the particular blocks that are most problematic for *i*PACKMAN, and this objective has been met. Examples can be seen in Figure 10. The secondary goal, that overall quality on average should stay the same or increase, has also been reached.

It must be said that seven images are not enough to make a good statistical analysis of the results. A larger test suite with images intended for texture mapping would be desirable. What can be said however, is that THUMB solves some of the worst problem blocks for *i*PACKMAN.

6 Conclusion

We have presented a new texture compression algorithm, THUMB, that can be used as a stand-alone system or in combination with *i*PACKMAN. If combined, THUMB can improve some of the worst blocks in *i*PACKMAN, and at the same time raise overall quality some. Still, some blocks, such as gradients between two colors illustrated by the explosion in Figure 10, are better handled by S3TC than with the proposed combination. This opens up for future work, and perhaps new modes.

References

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay Streams for Graphics Hardware. *ACM Transactions on Graphics, 22*, 3, 792–800.

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

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

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 *Proceedings of SIGGRAPH*, vol. 22, 215–223.

DELP, E., AND MITCHELL, O. 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.

FROMM, R., PERISSAKIS, S., CARDWELL, N., KOZYRAKIS, C., MCCAUGHY, B., PATTERSON, D., ANDERSON, T., AND YELICK, K. 1997. The Energy Efficiency of IRAM Architectures. In *24th Annual International Symposium on Computer Arhchitecture*, ACM/IEEE, 327–337.

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

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

LINDE, Y., BUZO, A., AND GRAY., R. M. 1980. An Algorithm for Vector Quantizer Design. *IEEE Transactions on Communication 28*, 1, 84–95.

STRÖM, J., AND AKENINE-MÖLLER, T. 2004. PACKMAN: Texture Compression for Mobile Phones. In *Sketches program at SIGGRAPH*.

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

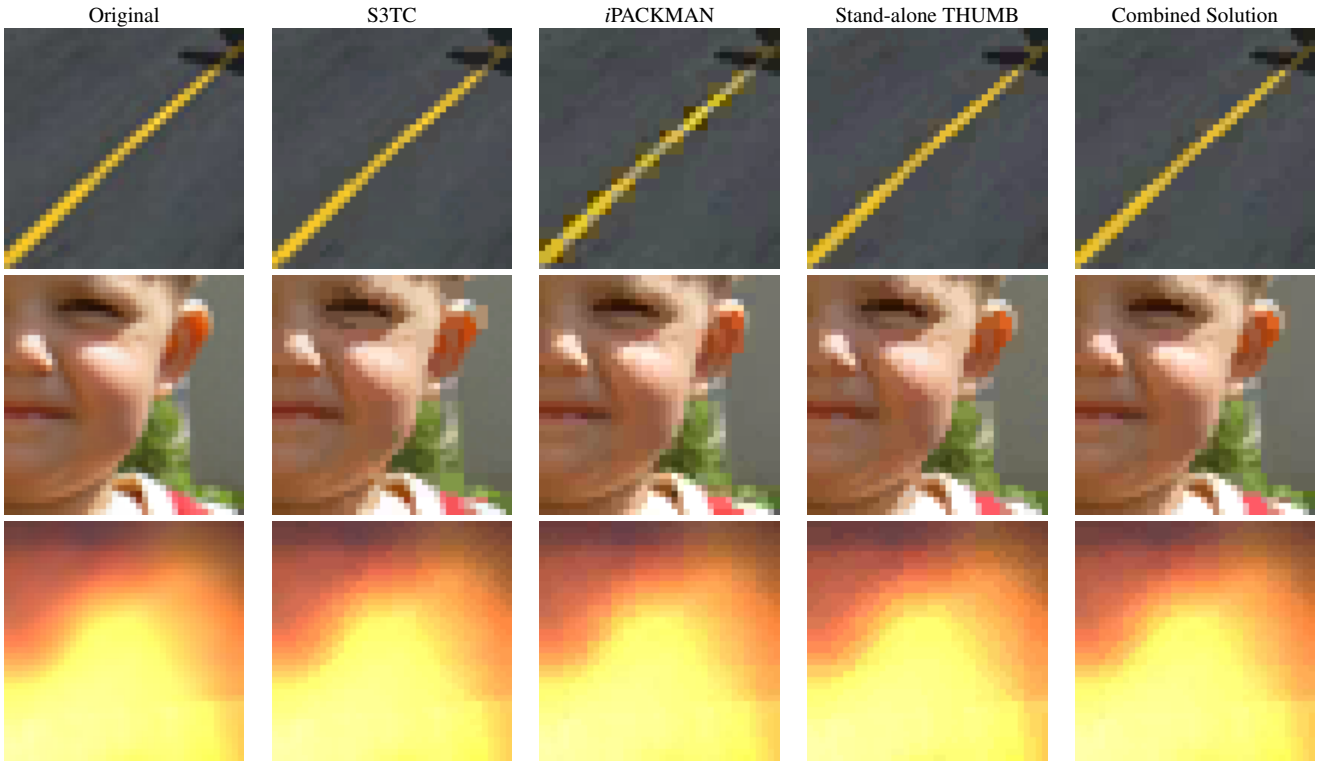

Figure 10: In this figure three examples of typical problem blocks are illustrated. Top: In this example, which shows a road, the main problem of *i*PACKMAN can be seen. Subblocks containing two distinct hues are coded poorly. Middle: Here is an example of the strength of *i*PACKMAN. Small transitions in luminance are coded well as can be seen around the eye and on the cheek. THUMB and S3TC have a more blocky appearance. A large image artifact can also be seen in the ear for S3TC. Since the combined solution inherits the strength of *i*PACKMAN, these blocks are coded well. Bottom: This last example is a cut-out of an explosion. Here S3TC performs the best thanks to its linear interpolation between the base colors. As the blocks contain more than one hue, the result tends to be blocky for *i*PACKMAN. THUMB encodes the image a little bit better even though some edges can be seen.