Texture Compression of Light Maps using Smooth Profile Functions

Jim Rasmusson[1,2] and Jacob Ström[1] and Per Wennersten[1] and Michael Doggett[2] and Tomas Akenine-Möller[2]

[1] Ericsson Research [2] Lund University

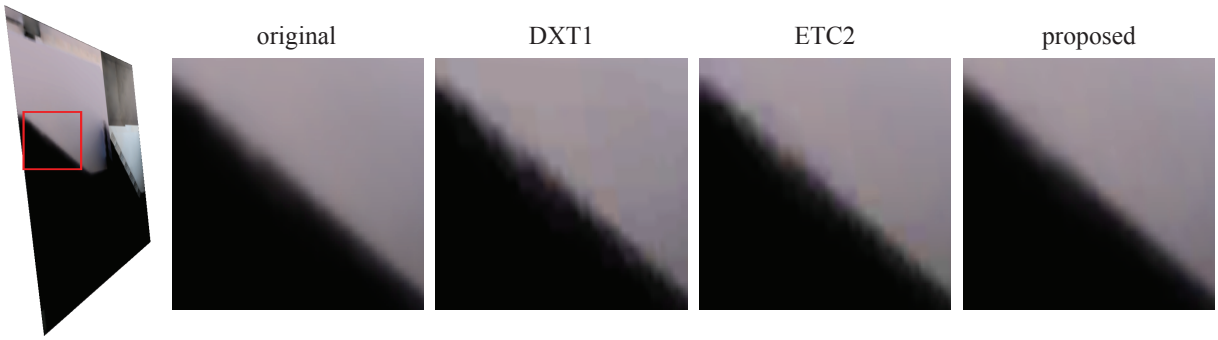

Figure 1: *From left to right: Image rendered with original lightmap, zoom-in on the rendering using original texture, using DXT1, using ETC2 and finally using the proposed method.*

Abstract

Light maps have long been a popular technique for visually rich real-time rendering in games. They typically contain smooth color gradients which current low bit rate texture compression techniques, such as DXT1 and ETC2, do not handle well. The application writer must therefore choose between doubling the bit rate by choosing a codec such as BC7, or accept the compression artifacts, neither of which is desirable. The situation is aggravated by the recent popularity of radiosity normal maps, where three light maps plus a normal map are used for each surface. We present a new texture compression algorithm targeting smoothly varying textures, such as the light maps used in radiosity normal mapping. On high-resolution light map data from real games, the proposed method shows quality improvements of 0.7 dB in PSNR over ETC2, and 2.8 dB over DXT1, for the same bit rate. As a side effect, our codec can also compress many standard images (not light maps) with better quality than DXT1/ETC2.

Categories and Subject Descriptors (according to ACM CCS): Data [E.4]: Coding and Information Theory—Data compaction and compression;

1. Introduction

Texture compression [BAC96, KSKS96, TK96] continues to be a very important technology in real-time rendering due to lower bandwidth consumption and less memory usage. At the same time, compute power increases at a much faster pace than DRAM latency and bandwidth [Owe05], making techniques like texture compression potentially even more important in the future.

Light maps have been used to increase realism of lighting in real-time rendering for a long time. Since lighting often changes quite slowly, a low-resolution light map can often be combined with a (repeated) high resolution texture in order to create a convincing effect. However, if sharp shadows are desired, a higher resolution must be used in the light map, and this will increase the need for texture compression. The situation is aggravated by higher quality rendering techniques, such as radiosity normal mapping (RNM) [McT04, MMG06, Gre07], since they in effect need three or more light maps per object. Examples include Valve's Source engine (e.g., Half-life 2) [McT04] and Mirror's Edge & Medal of Honor by DICE/EA [LH09]. Sometimes about 300 MB of (compressed) RNM textures are needed for a single level in a game, which puts pressure even on high-end graphics cards. In general, the RNM techniques precompute three light maps (each for a different normal direction), and at render time, a linear combination of these light maps is computed per pixel based on the direction on the normal.

The normal is typically accessed through a high-resolution, repeated normal map.

As can be seen in Figure 1, the industry standard, DXT1, fails to compress blocks with smoothly varying content with sufficient quality. Both DXT1 [INH99] and ETC1 [SAM05] work by using a small color palette that the pixels can choose color from. Since they are designed to handle quite arbitrary texture content, neighboring pixels are allowed to choose palette entries completely different from each other. This flexibility is expensive in terms of bits, and therefore the number of possible colors in the block must be restricted. For instance, DXT1 can only display four different colors within a 4×4 block, and ETC1 can display four different colors within a 4×2 block. Apparently, this is not enough for slowly varying textures such as light maps from radiosity normal maps. It should be noted that ETC2 [SP07] has a special mode for planar content in a block, but that too does not give enough flexibility for light maps.

This paper proposes a new texture compression algorithm to solve this problem. Instead of allowing neighboring pixels to obtain completely different colors, we exploit the spatial redundancy in the data by letting the position of the pixel in the block control the interpolation between two base colors. To allow for edges, a non-linear function is used to produce the interpolation value. This makes it possible to give unique colors to every pixel of a 4×4 block. For "non-smooth" blocks, we propose using a variant of ETC2 as a fallback.

2. Previous Work

In this section we review the most relevant previous work, which in our case leaves out all the work on alpha map compression, high dynamic range (HDR) texture compression, and lossless compression.

In 1996, the first three papers about texture compression were published [BAC96, KSKS96, TK96]. Beers et al. [BAC96] present a texture compression scheme based on vector quantization (VQ) which can compress down to two bits per pixel (bpp). They compress 2×2 RGB888 pixels (12 bytes) at a time using a code book of 256 entries, thus achieving a compression ratio of 12:1. The main issue with VQ based schemes is that they create memory indirection; the decompression hardware must fetch the index from memory before it knows from where in the code book to fetch data. This creates extra latency that can be hard to hide. To reach sufficient quality, one optimized code book is typically needed for each texture, which makes it harder to cache the code books.

The Talisman architecture [TK96] uses a discrete cosine transform (DCT) codec, but this has seen little use. We speculate that this has to do with the steps following the DCT, which typically include run-length coding and Huffman coding. Both these steps are serial in nature, and cannot be decoded in a fixed number of steps. These are features that are seldomly desired in a hardware decompressor for graphics. Also, often the hardware decompressor is located *after* the texture cache, and hence, the decompressor only needs to decompress a single pixel, which also does not fit well with Huffman & run-length decoding; if the last pixel is requested, all previous pixels must anyway be decoded. Finally, Huffman coding produces variable length data, whereas most texture compression systems use a fixed rate in order to preserve random access. You could also imagine a DCT-based scheme where we avoid Huffman & run-length decoding in order to solve this problem, but that would reduce the compression ratio, and more importantly, each DCT coefficient would still affect every pixel to be decoded, making decompression times long. Hence, it seems that DCT-based codecs are not well-suited for hardware texture compression, but in other contexts, such as video compression, they work very well.

Delp et al. describe an gray scale image compression system called block truncation coding (BTC) [DM79], where the image is divided into 4×4 blocks. Two gray scale values are stored per block, and each pixel within the block has a bit indicating whether it should use the first or second color. By storing colors instead of gray scale values, Campbell et al. extend this system to color under the name color cell compression (CCC) [CDF*86]. Knittel et al. [KSKS96] develop a hardware decompressor for CCC and use it for texture compression (as opposed to image compression). The S3TC texture compression system by Iourcha et al. [INH99] can be seen as an extension of CCC; two base colors are still stored per block (in RGB565), but two additional colors per block are created by interpolation of the first two, creating a palette of four colors. Each pixel then uses two bits to choose from the four colors, with a considerable increase in quality as a result. In total 64 bits are used for a block, giving 4 bits per pixel (bpp). S3TC is now the de facto standard on computers and game consoles under the name DXT1.

Fenney [Fen03] also stores two colors per block, but uses the colors of neighboring blocks: The first color is used together with the first colors of the neighboring blocks to bilinearly interpolate the color over the area of the block. This way a smooth gradient can be obtained. The second color is treated similarly, creating two layers of color. Finally two bits per pixel are used to blend between these two layers, producing the final pixel color. Fenney present both a 4 bpp and a 2 bpp variant of the codec.

The ETC scheme [SAM05] also uses 4×4 blocks, and compress down to 4 bpp. Otherwise the approach is rather different. One color is chosen per 2×4 pixel block. The color of the second block is either encoded separately or differentially, where the latter allows for higher chrominance precision. Each block also encodes a table index, where each entry has four values of the form: $\{-b, -a, +a, +b\}$. Each pixel uses two bits to point into this table entry. The value from the table is added to all three color components, and

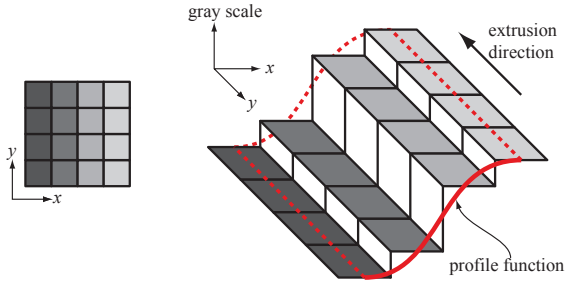

Figure 2: *Left: 4×4 pixels with a very regular pixel pattern. Right: illustration of how a profile function is used to describe the content of the pixel block to the left. Note how the profile function is merely a function of x, and then extruded in the y-direction.*

can hence be viewed as a luminance modification. ETC is standardized in OpenGL ES as an OES extension. By using invalid combinations in the ETC format, Ström and Pettersson [SP07] manage to squeeze in three more modes in ETC. This new format, called ETC2, is better at handling chrominance edges, and has a special mode for planar transitions.

Recently, new codecs with higher bit rates have been introduced, such as Microsoft's BC7 (called BPTC in OpenGL [BPT]) which operates at 8 bpp. However, faced with the prospect of doubling storage and bandwidth requirements, game developers may choose the imperfect quality of 4 bpp systems instead.

One technique in the field of image analysis and perception is to represent an image as a summation of a small number of oriented functions [OF96, DV03]. Our technique also uses oriented functions, but represents the blocks using only one function per block, not using a summation of functions.

3. Compression using Smooth Functions

In this paper, we focus on the compression of smooth light maps, and a general observation is that they often contain rather *little* information, while at the same time, each pixel in the block can have a *unique* color, even if the differences are not always that big. Another observation is that many blocks are *directional*, i.e., they contain more information in one direction (across an edge) than in the other direction (along an edge).

We start by describing the algorithm for gray scale light maps using a very simple example, illustrated in Figure 2. We have divided the image into blocks of 4×4 pixels, and are concentrating on one block. The x-coordinate of each pixel is used to evaluate a function, here called profile function, and the result is used as the gray shade for the pixel. Thus, a single profile function is sufficient to describe the gray scale content of the entire block. However, more flexibility is needed to be able to accurately represent a large

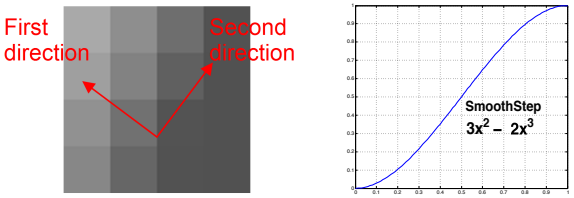

Figure 3: *Left: Two orthogonal directions are placed in a block of 4×4 pixels, so that the block varies maximally along the first direction, and minimally along the second. Right: A typical function that can be used to describe the variation along the first direction.*

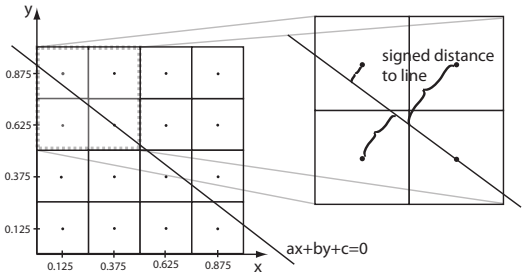

Figure 4: *Illustration of the decoding of the block. The edge line is positioned in the block and each pixel computes its signed distance to the edge. This in turn is fed into a smooth profile function in order to compute the value of the pixel.*

number of blocks. For example, a key parameter is to be able to *rotate* the profile function in the xy-plane.

Hence, the core idea of our research is to encode the orientation of an edge in each pixel block, and then specify a *profile* across the edge using a function with a small number of parameters.

To enable orientation and translation, we first establish two orthogonal directions in the block, and an origin. Here, we would like to arrange the directions so that the gray scale varies maximally along the first direction, and minimally along the other direction. The "origin" is placed in the lower left corner of the block. An example is shown to the left in Figure 3. Mathematically, this can be done by fitting a line, $a_1x + b_1y + c_1 = 0$, so that the "normal," (a_1, b_1), of the line coincides with the first direction. Note that the subscript is used to distinguish it from other lines. By normalizing the equation so that $a_1^2 + b_1^2 = 1$, we can compute the distance, d_1, from any point, (p_x, p_y), to the line by simply inserting the point into the line equation: $d_1 = d_1(p_x, p_y) = a_1p_x + b_1p_y + c_1$. This distance is signed meaning that it is negative on one side of the line and positive on the other. See Figure 4 for an illustration of the edge line and the signed distances from the pixel centers to this line.

Next, we use a scalar-valued *profile* function, $f(d_1)$, to represent the variation inside the block. Since d_1 will be

constant for points on a line parallel with the second direction, $f(d_1)$ and hence the gray scale value of the block will also be constant in that direction. Points along the other direction will give rise to a varying d_1, and hence different gray scale values, $f(d_1)$. Using the line definition above, we obtain the gray scale value in a pixel, (p_x, p_y), as $f(d_1) = f(a_1 p_x + b_1 p_y + c_1)$.

The profile function, $f(d_1)$, can be a function going from dark to bright, such as the one depicted to the right in Figure 3. This works well for blocks where there is an edge in the block. It could also be a function going from dark, bright and then dark again. Such a function would be better suited for blocks depicting a white line on a black background.

3.1. Extension to Color

A naïve way of extending our method to color would be to duplicate the above-mentioned procedure three times, i.e., once for each color component. However, that would cost too many bits, and would not exploit the fact that the color channels often are well correlated. Instead of directly calculating the grayscale value using the function $f(d_1)$, we use it to calculate an interpolation factor, $i = f(d_1)$, where $i \in [0,1]$. We then use this interpolation factor to interpolate between two different colors, $\mathbf{c}_A$ and $\mathbf{c}_B$, which are representative for the block. The interpolation is then done as:

$$\mathbf{c}(p_x, p_y) = (1-i)\mathbf{c}_A + i\mathbf{c}_B, \tag{1}$$

where $i = f(d_1) = f(a_1 p_x + b_1 p_y + c_1)$. As can be seen, the same interpolation factor, i, is used for all three color channels. So far, we only need to store the colors $\mathbf{c}_A$ and $\mathbf{c}_B$, the line equation (for instance using the constants a_1, b_1, c_1), and indicate which function, f, was used for calculating the interpolation factor in order to decompress a block.

3.2. Second Direction Tilt

Up to this point, we have assumed that all the variation is *across* the edge, i.e., in the first direction. The variation *along* the edge, i.e., in the second direction (see Figure 3), is often non-negligible in practice and needs to be represented in our model as well. Most of the bits will be allocated for the placement of the edges and for the variation across the edge, and hence, we need to limit the number of bits describing the variation in the second direction. We have found that a simple linear variation, i.e., a slope, along the edge does a decent job. This slope is described with a single parameter, γ, which is the same for all three color components, i.e., a linear luminance variation. An additional term, γd_2, is added to each component of the color:

$$\mathbf{c}(p_x, p_y) = (1-i)\mathbf{c}_A + i\mathbf{c}_B + \gamma d_2(1,1,1), \tag{2}$$

where $d_2 = a_2 p_x + b_2 p_y + c_2$ is the signed distance from the pixel, (p_x, p_y), to a line orthogonal to the first line, and $(1,1,1)$ is the maximum color. The line equation for this second line is $a_2 x + b_2 y + c_2 = 0$, where:

$$(a_2, b_2, c_2) = (-b_1, a_1, -0.5(a_1 + b_1)). \tag{3}$$

This means that the direction of the first line is rotated by 90 degrees, and that the translation, c_2, is computed so that the line goes through the origin $(0.5, 0.5)$ of the pixel block. This new term gives us the opportunity to add a luminance tilt along the edge, as illustrated in Figure 6d.

3.3. Profile Functions

We currently use four different profile functions, $f_i(d_1)$, $i \in \{1,2,3,4\}$, as shown in Figure 5, and we pick the profile function that best suits the content of each pixel block. All four functions are based on the simple *smoothstep* function, $s(d)$, shown below in pseudo-code:

```
x = d/w          //scale by width, w
x += 0.5         //center function around origin
x = clamp(x,0,1) //clamp between 0 to 1
return 3x^2 - 2x^3 //evaluate smoothstep
```

Note that a width parameter, w, is used to control how rapidly the smoothstep function should increase. A small value will give rise to a sharp edge, whereas a large value will create a smooth transition. The value 0.5 is added so that the function is centered around $d = 0.0$

Figure 5a shows a function we call the *symmetric single*, which is the basic smoothstep function, $f_1 = s(d)$ with a *width* parameter to control the shape. Figure 5b shows the *asymmetric single* function, f_2, where the smoothstep function also is used. However, two different width values are used here; one to the left of the y-axis, and one to the right. Figure 5c shows the *asymmetric double* function, f_3, which uses two "concatenated" smoothstep functions (hence the name 'double'), again using different widths on either side of the y-axis. In contrast to the two previous profile functions, this function uses three colors for the interpolation calculation. A third color is used at $d_1 = 0$, and to save storage this color is not a separate color but instead generated as the average of the other two and scaled with a scale factor.

The fourth function is different since it adds another smoothstep in an arbitrary direction, $d_3 = a_3 p_x + b_3 p_y + c_3$, and multiplies two smoothstep functions, $s(\cdot)$, together:

$$f_4(d_1, d_3) = s(d_1)s(d_3). \tag{4}$$

Hence, each smoothstep uses its own line equation, but they also use their own widths. The scalar value, $f_4(d_1, d_3) \in [0,1]$, is used as an interpolation factor in exactly the same way as for the other functions. See Figure 5d for an example. As can be seen, this gives us the ability to represent various smooth corners in a block. Since the fourth function is already two-dimensional, no second direction tilt is needed.

Our choice of profile functions is the result of a combination of reasoning and trial and error, therefore it may seem

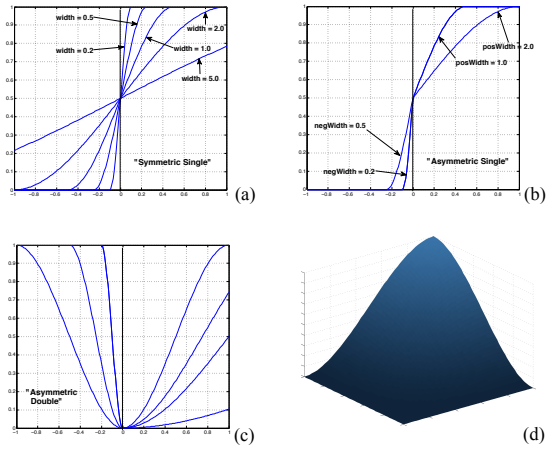

Figure 5: *The four functions used to fit the color variation. a) symmetric single, b) asymmetric single, c) asymmetric double, and d) corner, which is the multiplication of two smoothstep functions along two different directions.*

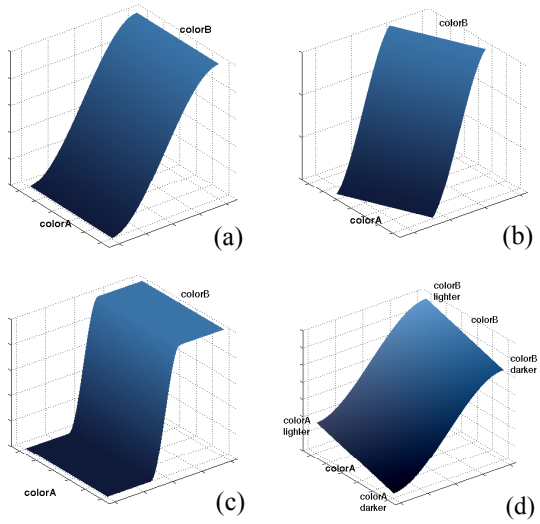

Figure 6: *Illustration of the function parameters of the symmetric single smoothstep function: a) two end-point colors, b) rotation and translation, c) width, and d) slope.*

a bit arbitrary. We started out with the assumption that the single symmetric function should capture smooth and sharp edges well, which turned out to be true. We then tried over 20 different functions, and picked the ones that gave the best results. In retrospect, it is clear that the asymmetric single captures blocks containing discontinuous edges, and that the asymmetric double can capture a smooth line inside a block, as mentioned earlier. The corner function captures corners, which often occurs in light maps.

3.4. Fallback method

While many blocks will compress well using the above approach, there will be others that do not have any directional structure, or any structure at all. For such blocks, we use a variant of ETC2 as a fall-back coder [SP07]. One bit per block will therefore be used to indicate whether profile functions should be used, or the ETC2 variant should be used. To preserve a bit rate of 64 bits per block, we need to steal one bit from the ETC2 codec. This is done by disabling the individual mode in ETC2, which frees up the diff-bit. Thus, 63 bits are left to compress the block using profile functions.

3.5. Function Parameters, Quantization, and Encoding

In this section, we will describe, in detail, all the parameters of our codec, how they are quantized and encoded. Our codec can use either of the four profile functions in Figure 5, and hence, two bits are needed to select profile function. This leaves $63-2=61$ bits to encode the parameters for the selected profile function.

In Figure 6, a visualization of the key parameters for symmetric single are shown. There are the two end-point colors, ColorA ($\mathbf{c}_A$) & ColorB ($\mathbf{c}_B$), and line placement parameters, consisting of orientation, θ_1, and translation, c_1. The line equation then becomes: $a_1x+b_1y+c_1=0$, where $a_1=\cos\theta_1$ and $b_1=\sin\theta_1$. As we have seen earlier, this line is used to place the profile function in its best possible location. Furthermore, Figure 6c shows the function *width*, which determines how rapidly the function rises (see the width parameter, w, in Section 3.3). The effect of the width can be seen in Figure 5a. Finally, Figure 6d shows the second direction tilt parameter, γ, described in Section 3.2. This improves image quality in that it often reduces block artifacts.

The bit distribution varies depending on the choice of profile function. We chose a starting distribution for each function and then iteratively optimized them over a small training image. The final bit distributions for our parameters in the four smoothstep-based profiles are listed in Table 1. The symmetric single uses $3\cdot 6$ bits per color, five bits for its width, w, seven bits for its orientation angle, θ_1, seven bits for its translation, c_1, and six bits for its tilt parameter, γ.

Compared to the symmetric single, the only new parameter for the asymmetric single is that it has two widths, w_1 and w_2, instead of only one. The asymmetric double has an additional parameter, called the scale factor, m. As mentioned briefly in Section 3.3, this factor is used to compute the color in the "middle" of the asymmetric double function. The color is computed as: $\mathbf{c}_{mid}=m(\mathbf{c}_A+\mathbf{c}_B)/2$. Finally, the corner codec uses two line equations, also described in Section 3.3, and hence encodes two orientation, θ_1 and θ_3, and two translations, c_1 and c_3.

All of the parameters used are viewed as floating point values, with individual ranges as shown in Table 2. The

Number of bits	SSingle	ASingle	ADouble	Corner
ColorA ($\mathbf{c}_A$)	666	565	555	555
ColorB ($\mathbf{c}_B$)	666	565	555	555
Function widths ($w_1 w_2$)	5-	44	55	44
Rotations ($\theta_1 \theta_3$)	7-	8-	6-	66
Translations ($c_1 c_3$)	7-	9-	6-	56
Scale factor (m)	-	-	5	-
Second direction tilt (γ)	6	4	4	-
Total	**61**	**61**	**61**	**61**

Table 1: *Bit distribution for the four modes; symmetric single (SSingle), asymmetric single (ASingle), asymmetric double (ADouble), and corner smoothstep mode.*

Parameter	*min float value*	*max float value*
Color components	0	255
Function widths (w_1&w_2)	0.005	5
Rotations (θ_1&θ_3)	0°	180°
Translations (c_1&c_3)	-2	+2
Scale factor (m)	0	2
Second direction tilt (γ)	-16	+16

Table 2: *This table shows the valid range for each parameter before quantization. The minimum values are the same after quantization, but the maximum values are slightly smaller depending on the number of bits used.*

stored bits are treated as fixed point representations, with the decimal point placed to achieve a similar range as in the floating point domain. For example, in the 6-bit second direction tilt used in the symmetric single mode, the decimal point is before the least significant bit, giving a range of -16 to +15.5. In the other modes which use 4 bits for the second direction tilt, a zero is inserted after the last of the four bits to get a 5-bit integer value, which will be in the range of -16 to +14. There are two exceptions to this rule: first, the bits representing the color components are repeated which ensures we are always able to represent both extreme points 0 and 255. Second, the width parameters use a non-linear quantization, which in the 5-bit case is:

$$w = 0.005 \times (1000^{\frac{1}{31}})^q \tag{5}$$

where q is the stored value. In the four-bit case, the equation is slightly altered to get a similar range of values:

$$w = 0.005 \times (1000^{\frac{1}{31}})^{2q} \tag{6}$$

4. Compression algorithm

Compression is done in two iteration stages, the first with the parameters in the floating point domain and the second, after parameter quantization, in the fixed point domain. We use cyclic coordinate search [BSS93] to minimize the error function, i.e., for each coordinate (i.e., parameter, such as the *width*) we approximate the gradient with respect to that parameter, and go a step in the opposite direction.

However, if the error function changes very rapidly, a small step size is necessary in order to be guaranteed a lower error in the new point. This small step may be smaller than what it is possible to step in the quantized domain. To solve that problem, we do the first round of optimization using floating point arithmetic. After a global minimum has been found, and cyclic coordinate search is no longer fruitful, we quantize the values and do a second round of optimization in the quantized domain. This is necessary since we will not be able to reach the floating point position exactly when quantizing it, so we must try a number of quantized positions around that point.

For each 4×4 pixel block the following is performed:

1. Find the direction of the maximum variation. This is represented as the rotation angle, θ.
2. Use this rotation for initial orientation of the function.
3. For initial ColorA ($\mathbf{c}_A$) and ColorB ($\mathbf{c}_B$), we use the "min" and "max"- colors along the maximum variation line.
4. Iteratively search over the entire floating-point parameter space (ColorA, ColorB, rotation, offset, width and slope) using cyclic coordinate search to minimize the image error.
5. Quantize all parameters.
6. Perform a second iterative search of the quantized parameters.
7. Choose the mode/function with the smallest error.
8. Pack into 64 bits.

To ensure a "somewhat" exhaustive search of the possible parameter space, we ran 200 iterations of the floating point parameter search and 6 iterations of the fixed point parameter search. This is of course a trade-off between compression time and performance. However, performance did not improve much above 200 float / 6 fixed iterations. Our compression algorithm compresses an image with 192×192 pixels in about 50 seconds. This is done with a multi-threaded implementation on a MacPro with dual Intel quad core CPUs at 2.26 GHz. Early on, we implemented the encoder in OpenCL on an NVIDIA GPU and managed to get a speed up factor of 42× compared to a single-threaded CPU version. However, we decided to focus our programming efforts on a multi-threaded CPU implementation, since the OpenCL tools for debugging and profiling were rudimentary at best (Mac OSX 10.6.1/2).

5. Decompression algorithm

The decompressor is much simpler and faster. It is designed to have low complexity in order to ensure inexpensive hardware implementation. To decompress a single pixel in a 4×4 block, the following is done:

1. The rotation angle, θ, and offset parameter, c, are used

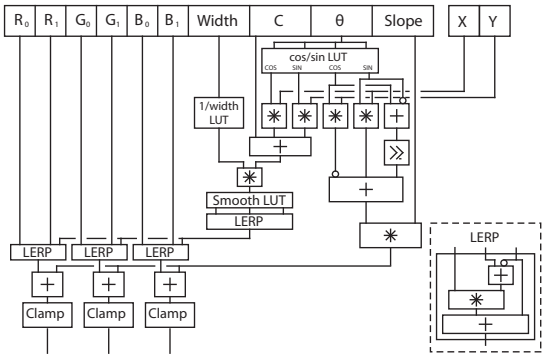

Figure 7: *Hardware design for a* symmetric single *decompression unit. Linear interpolation is replaced with a LERP unit as shown in the lower right of the diagram. Three lookup tables are used for sin/cos,* $\frac{1}{width}$ *and smoothfunc evaluation.*

to reconstruct the edge line, $ax+by+c=0$, where $a=cos(\theta)$ and $b=sin(\theta)$.

2. For the pixel (x,y), calculate a signed distance, $d=ax+by+c$. See Figure 4.
3. An interpolation value, i, is calculated using the selected base function, $i=f(d), i\in[0,1]$.
4. i is then used to interpolate between ColorA and ColorB.
5. Finally, the slope parameter is used to add a luminance ramp (along the edge line) to the output color.

For example, in the symmetric single case, Step 3 uses the smoothstep function, $s(d)$, as shown in Section 3.3. For Step 5 the distance from the orthogonal line must be first calculated using an orthogonal line equation. This orthogonal distance is multiplied by a scaling factor to vary the color along the edge line.

5.1. Fixed function decompression algorithm

We have implemented a fixed function version of the decompressor algorithm. This decompressor uses the functions shown above with only fixed point addition and multiplication with 7 bits precision up until the final lerp, addition and clamp, and several lookup tables for the more complex functions. A possible hardware design is shown in Figure 7.

The scaling of the width of the smoothstep function requires a division. We replace this division with a lookup of $\frac{1}{width}$ and a multiplication. The width value is represented with 3 bits of integer and 7 bits of fractional precision.

The *sin* and *cos* used in computing the coefficients a and b from the rotation angle parameter θ in Step 1 are stored in a lookup table with 128 entries and 7 bits per entry for values ranging from 0 to 180 degrees.

We approximate the smoothstep equation, $3x^2-2x^3$, with a piecewise linear function stored in a lookup table. This table only stores the region between 0.5 and 1 since the function is symmetrical. We then split the 0.5 to 1 region in half recursively to create 8 segments and store the end point values in a table with 7 bits of precision. When decoding we find the segment by counting the number of leading zeros and then interpolate the endpoints from the lookup table.

Figure 8: *PVRTC handles smooth transitions well due to its bilinear interpolation. Instead, compression artifacts manifest themselves in other parts of the textures, as shown above.*

We have performed a rough hardware complexity estimate, using estimates such as 2.2 gates per bit of a MUX and 4.4 gates per bit of an ADD, where one gate equals a 2-input NAND. The result was a gate count somewhere between 4000 and 5000 for our decoder, including the ETC2 fallback. This is roughly 4 times the size of the ETC2 decoder and 5 times the size of a DXT1 decoder estimated in a similar manner.

6. Results

We have tested our algorithms on radiosity normal maps (RNMs) from two real games, namely *Mirror's Edge* and *Medal of Honor* from DICE/EA. While our main target is smooth light maps, we also wanted to test the hypothesis that our codec can also be used as a more generic texture compression method. Therefore, we also used the 64 regular textures which were used for evaluating ETC2 [SP07]. We have compared our results against DXT1 and ETC2 for all of the test sets. For DXT1 we used The Compressonator version 1.50.1731, and for ETC2 exhaustive compression was used. In both cases, error weights of (1,1,1) were used to maximize PSNR. We contemplated also comparing to the 4 bpp version of PVRTC [Fen03], especially since its use of bilinear interpolation handles smooth transitions well. Unfortunately, compression artifacts creep up in other places instead as shown in Figure 8, and on the Mirror's Edge test set, it was 1.9 dB lower than DXT1 when compressed using PVRTexTool Version 3.7. For these reasons, we have excluded PVRTC from our results.

In the following subsections, we first present the results for Mirror's Edge, which is followed by Medal of Honor's results. In Section 6.3, we present results for the 64 regular (non-light map) textures used to evaluate ETC2 and also a test done with a set of 24 regular photos from Kodak. Finally, we show zoomed-in crops of some textures in Figure 16.

Figure 9: *Texture compression results using a test set of 24 radiosity normal maps from the game Mirror's Edge. Our method is 0.7dB better than ETC2 and 2.8 dB better than DXT1 when all 24 results are combined together.*

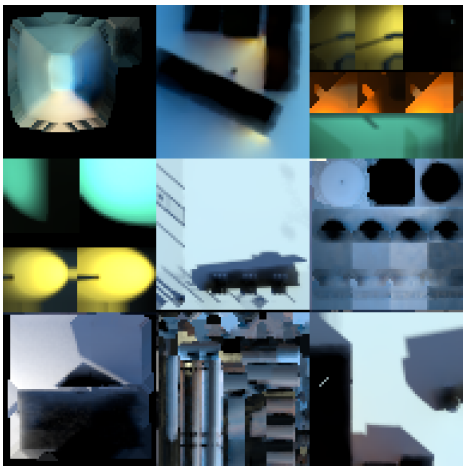

Figure 10: *Original light maps from Mirror's Edge, used for training our algorithm. Images courtesy of Electronic Arts.*

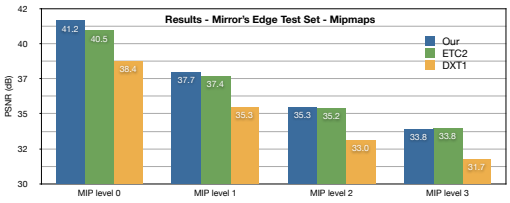

Figure 11: *Results for the mipmapped versions of the 24 image test set from Mirror's Edge. Note that mipmap level 0 is the full resolution.*

6.1. Mirror's Edge

During the development of our algorithm (including bit distribution, profile function selection, quantization, and more), we used nine representative crops from the game Mirror's Edge as a training set. These are shown in Figure 10, and for all our results, these images were excluded in order to avoid biasing the result. The remaining RNM images from Mirror's Edge were used to form a larger test set of 24 RNMs.

The results are shown in Figure 9. The combined PSNR results for this set are 41.2 dB, 40.5 dB and 38.4 dB for our method, ETC2, and DXT1, respectively. Our method consistently performs better than ETC2 and DXT1 for this entire test set. A few resulting images (with zoomed-in crops) from this test set are shown in the top two rows of Figure 16. It is interesting to note that for the images with lowest PSNR, the improvement from our algorithm is almost zero compared to ETC2. The reason for this is that when ETC2 and DXT1 perform poorly, the image content is usually rather noisy, in which case our codec has little chance of improving quality.

We also generated mipmapped versions of these 24 RNMs to investigate how our method behaves with low-pass filtered and lower resolution versions of the RNMs. The results are shown in Figure 11. With decreasing resolution, ETC2 gradually approaches our method and in the lowest mipmap level, ETC2 is slightly ahead. This is expected since the smooth areas get smaller the lower the resolution, and the texture becomes gradually more noisy.

6.2. Medal of Honor

We compressed a set of 36 RNMs from the recent game *Medal of Honor* from EA. The combined PSNR results were 37.06 dB, 37.01 dB, and 34.15 dB for our method, ETC2, and DXT1, respectively. As can be seen, our method was only insignificantly better than ETC2 for this test set. In fact, in 8 out of 36 images, the performance of our method was lower than ETC2. Those images contain large numbers of very small light maps baked into a large texture, as shown in Figure 12. If we remove all light maps containing any subtextures smaller than or equal to 16×16 pixels, the relative PSNR result changes: our method now has a 0.53 dB advantage over ETC2 and 3.05 dB over DXT1. Thus, the Medal of Honor test set clearly shows that our method is only improving quality for relatively large textures, i.e., bigger than 16×16 pixels. This is not surprising given the nature of our method, which depends on exploiting smooth structures.

Even for the textures where the proposed coder is worse than ETC2, it is hard to spot flaws visually. This may be because the textures are very small and quite noisy, and difficult for a human eye to understand what is the "correct" structure. To give a feeling for the worst case, we automatically found the 16×16 block which gave the worst performance relative ETC2, and it is depicted in Figure 13. Note

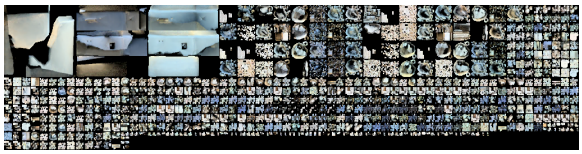

Figure 12: *An example where our method performs rather poorly from the* Medal of Honor *test set. This light map was used for radiosity normal mapping in the game.*

Figure 13: *This shows the* 16×16 *block where our method performed the worst (PSNR wise) compared to ETC2. This is part of the texture in Figure 12 from Medal of Honor.*

that it is only in blocks where the individual mode is chosen that ETC2 can be at an advantage over the proposed codec, since the latter includes a subset of ETC2.

6.3. Regular Textures and Photos

We also tested our method using the same set of regular texture as used in the evaluation of ETC2 [SP07]. This test set is a broad mixture of photos, game textures, and some computer generated images. While our algorithm was not designed with such diverse textures in mind, it was still 0.34 dB better than ETC2, and 1.25 dB better than DXT1 for the full resolution. The performance for the mipmapped versions was similar to our previous mipmap results, but at the highest level (smallest texture), ETC2 performed slightly better. The combined results for the full resolution and the mipmapped versions are illustrated in Figure 14. A few zoomed in examples from this test set are shown in Figure 16.

To compare against publicly available data, we tried our method on the Kodak data set http://r0k.us/graphics/kodak/. The results are shown in Figure 15.

7. Conclusion

We have presented a new codec for texture compression. It is based on parameterized smooth profile functions, with a subset of ETC2 as a fallback for noisy blocks. Our method often generates images with much higher quality than competing algorithms for light maps, and as a positive side effect, our codec also increases the image quality on regular images.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

Many thanks to Henrik Halén at EA/DICE for providing us with light map textures from recent games.

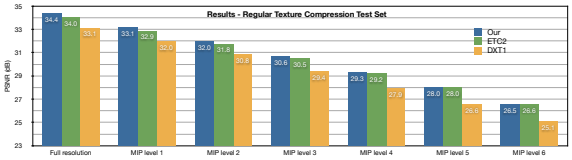

Figure 14: *Results from the regular test set of 64 texture images. Our method is 0.34 dB better than ETC2 and 1.25 dB better than DXT1 for the full resolution. The results for the mipmapped versions are in here as well.*

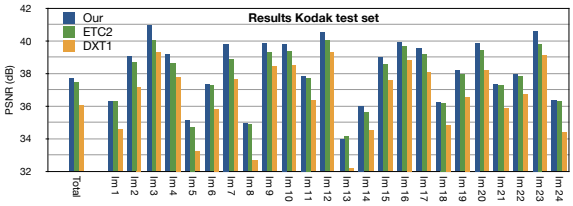

Figure 15: *Results from a public test set of 24 photos from Kodak. Combining the results from all 24 photos, our method is 0.24 dB better than ETC2 and 1.65 dB better than DXT1.*

References

[BAC96] BEERS A., AGRAWALA M., CHADDA N.: Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96* (1996), pp. 373–378. 1, 2

[BPT] `http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt`. 3

[BSS93] BAZARAA M. S., SHERALI H. D., SHETTY C.: *Nonlinear Programming — Theory and Algorithms*. Wiley, 1993. 6

[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 *Computer Graphics (Proceedings of ACM SIGGRAPH 86)* (1986), pp. 215–223. 2

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

[DV03] DO M. N., VETTERLI M.: The Finite Ridgelet Transform for Image Representation. *IEEE Transactions on Image Processing 12*, 1 (2003), 16–28. 3

[Fen03] FENNEY S.: Texture Compression using Low-Frequency Signal Modulation. In *Graphics Hardware* (2003), pp. 84–91. 2, 7

[Gre07] GREEN C.: Efficient Self-Shadowed Radiosity Normal Mapping. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)* (2007). 1

[INH99] IOURCHA K., NAYAK K., HONG Z.: System and Method for Fixed-Rate Block-Based Image Compression with Inferred Pixel Values. US Patent 5,956,431, 1999. 2

[KSKS96] KNITTEL G., SCHILLING A. G., KUGLER A., STRASSER W.: Hardware for Superior Texture Performance. *Computers & Graphics, 20*, 4 (1996), 475–481. 1, 2

[LH09] LARSSON D., HALÉN H.: The Unique Lighting of Mirrors Edge. In *Game Developers Conference* (2009). 1

[McT04] MCTAGGART G.: Half-Life 2 / Valve Source Shading. In *Game Developers Conference* (2004). 1

[MMG06] MITCHELL J., MCTAGGART G., GREEN C.: Shading

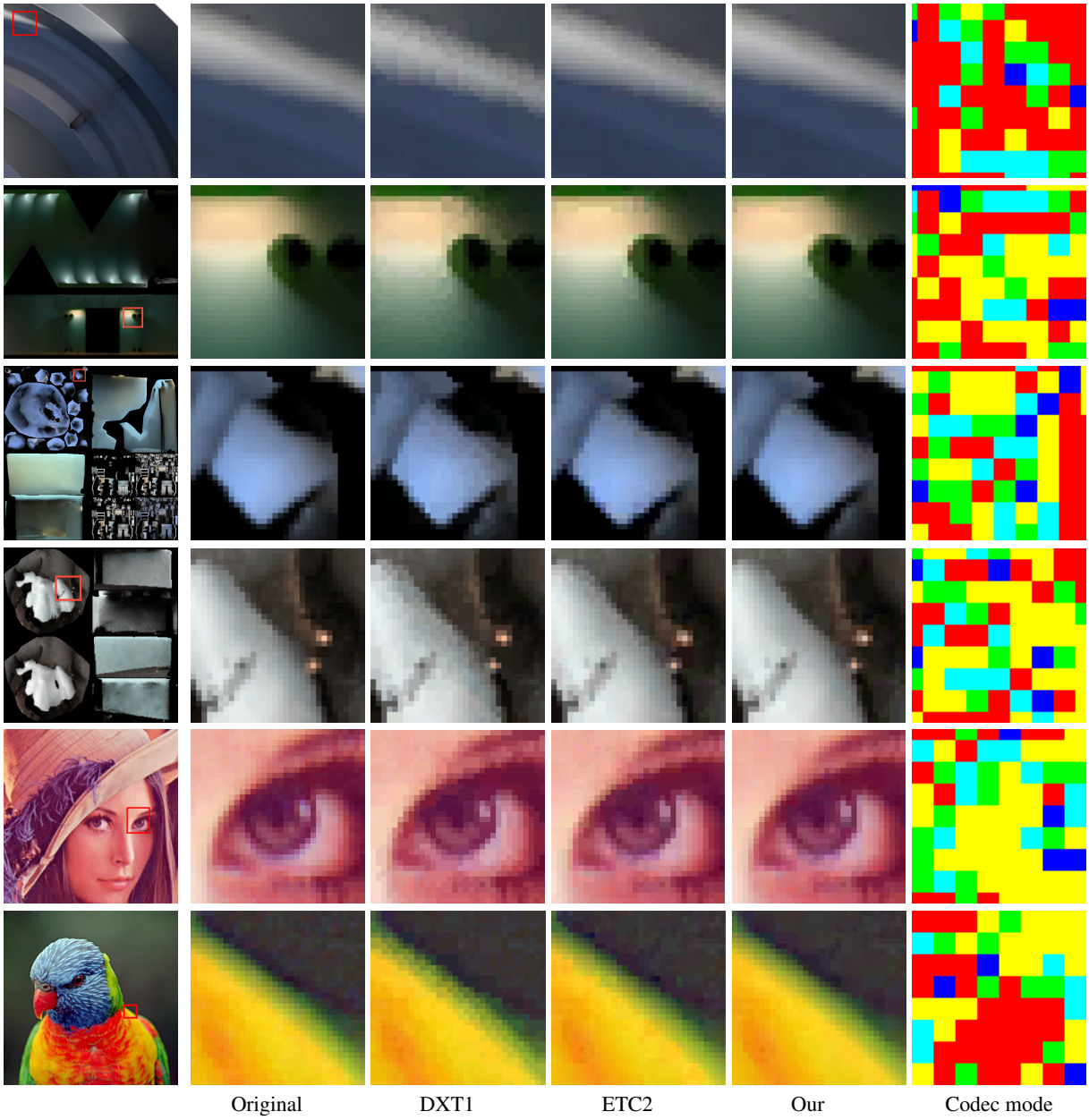

Figure 16: *The top two rows show light maps from Mirror's Edge. The mid two rows show light maps from Medal of Honor, and the last two rows are examples of regular textures. The rightmost column shows which of our 5 codec modes that was selected: Red=SymmetricSingle, Green=AsymmetricSingle, Blue=Double, LightBlue=Corner, Yellow=modified ETC2 (fallback mode)*

in Valve's Source Engine. In *Advanced Real-Time Rendering in 3D Graphics and Games (SIGGRAPH course)* (2006). 1

[OF96] OLSHAUSEN B. A., FIELD D. J.: Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images. *Nature 381* (1996), 607–609. 3

[Owe05] OWENS J. D.: Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 457–470. 1

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware* (2005), pp. 63–70. 2

[SP07] STRÖM J., PETTERSSON M.: ETC2: Texture Compression using Invalid Combinations. In *Graphics Hardware* (2007), pp. 49–54. 2, 3, 5, 7, 9

[TK96] TORBORG J., KAJIYA J.: Talisman: Commodity Real-time 3D Graphics for the PC. In *Proceedings of SIGGRAPH* (1996), pp. 353–364. 1, 2