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

Tight Frame Normal Map Compression

Jacob Munkberg,[1] Ola Olsson,[1] Jacob Ström[2] and Tomas Akenine-Möller[1]

[1]Lund University [2]Ericsson Research

Abstract

We present a new powerful and flexible fixed-rate normal map compression algorithm with higher quality than existing schemes on a test suite of normal maps. Our algorithm encodes a tight box with uniform normals inside the box, and in addition, a special mode is introduced for handling slowly varying normals. We also discuss several error measures needed to understand the qualities of different algorithms. We believe the high quality of our technique makes it a potential candidate for inclusion in OpenGL ES.

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

1. Introduction

Normal maps, also called bump maps [Bli78], allow for significant geometry savings, while preserving the illusion of geometric detail. Therefore, they are very popular in the latest generation of games. Texture bandwidth is a limiting factor, and to allow heavy use of normal maps in a real-time engine, there is a need of a compact representation of these textures. The focus of this article is twofold. First, we will discuss error measures for evaluating the quality of normal maps, and second, we will present a new compression algorithm. We argue that it is important to study not only the PSNR of the resulting maps, but also the *maximum pixel error*, and the *error distribution* over the images alongside with rendered results of the maps in use. Governed by our error measures, we present a new high quality compression algorithm, suitable for hardware implementation. Our technique supports very fast decompression, and robust behavior for a large range of input data.

2. Previous Work

A number of algorithms have been suggested for normal map compression. Most of these are fixed rate algorithms, which allows for fast random access without index tables, palettes or traversal trees.

Standard color texture compression techniques are not well suited for normal maps, which often contain many sharp features. To the best of our knowledge, Deering was the first to present compression of normals [Dee95]. By using symmetries on the sphere, and encoding the “sextants” of the octants, each normal could be stored in 12 bits. Note that this work was targeted for geometry compression. Fenney and Butler [FB04] also encode by the octants, but select one of four octant-pairs, each parameterized with 7+7 bits. Each normal uses 16 bits. The 3Dc format [ATI05] is a dedicated normal map compression technique, which compresses blocks of 4×4 pixels. The 16 (unit length) normals in a block are projected onto the unit circle, and the axis-aligned bounding box of the projected values is quantized into an 8×8 grid, giving 64 positions to choose from inside the box. Four values are encoded to determine the size of the box, and $3+3$ bits are encoded per normal in the block to determine which point in the grid to select. This results in a total of 128 bits per block of 16 pixels, or 8 bits per pixel. By exploiting unused encoding combinations, and using them as additional compression modes, an enhanced 3Dc (here abbreviated e3Dc) algorithm was defined [MAMS06]. This algorithm handles very slowly varying normal maps (e.g., car hoods), rotated frames and more uniform reconstruction point distributions. We have borrowed techniques for better point distributions and bit extraction from this work. Normal map encodings with adaptive bit rates [WB06, YP06] achieve better compression rates than fixed-rate approaches with comparable quality, but rely on complex addressing for decompression along with more memory accesses to index tables, which can make a hardware implementation significantly more complex. Vector quantization allows for more compact normal map compression and achieves impressive quality and compression rates [YA06]. However, the approach is limited to 8-bit normals, which is shown to be insufficient for slowly varying normal maps. An error analysis for normal maps based on unity condition [YHA05] discussed the impact of the popular elimination of the z-component while compressing normal maps. An interesting conclusion is that as long as the normals have small x, y and small errors in those components the z-error will be even smaller.

3. Error Analysis

As normal maps are not viewed directly, but rather used in shaders to define the local normal vector, standard image quality metrics are not directly applicable. It can be argued that the *mean square error* (MSE), is a good measure, as it gives an (averaged) error that indicates the quality of the normal map. However, it does not tell us whether there is a constant small error over all pixels or a small set of pixels with large errors. An excellent discussion of the limitations of the MSE is described in Wang et. al's paper about *structural similarity* [WBSS04], where different distortions are added to an image, all with equal MSE. A smooth normal map with a few isolated divergent normals will often look unacceptable as the divergent normals will give rise to cracks in the smooth surface. Therefore, we also use the *max error*, and histograms of the angle error (defined below) per image together with MSE values, to ensure that the algorithms behave robustly.

MSE is computed as a summation over all normals in the image:

$$MSE = \frac{1}{w \times h} \sum (\hat{x} - x)^2 + (\hat{y} - y)^2 + (\hat{z} - z)^2, \qquad (1)$$

where w and h are the width and the height of the image, $x \in [-1,1]$ is the x-component of the uncompressed normal and $\hat{x} \in [-1,1]$ is the corresponding compressed x-component, and similar for y and z. For normals, we use the *Peak Signal to Noise Ratio* (PSNR):

$$PSNR = 10 \log_{10}\left(\frac{1}{MSE}\right), \qquad (2)$$

where the nominator is one, since the *peak signal* for a normal of unit length will always be equal to one by construction.

There are mainly three components which will be affected by the precision of the normal in real-time graphics: diffuse shading, specular shading, and specular reflection. The errors in a rendered image due to the diffuse and specular shading are relatively small compared to that of the specular reflection. even a small angular error in a normal may result in a different texture access in the environment map. Therefore, it is important to look at the direct angle difference between the compressed and original normal, as well as studying bump mapped images with environment mapping.

We propose using the angular deviation [ANRS05], denoted E_{ad}, defined as:

$$E_{ad} = \arccos(\mathbf{n}_o \cdot \mathbf{n}_c), \qquad (3)$$

which measures the difference in angle between the uncompressed normal ($\mathbf{n}_o$) and the compressed one ($\mathbf{n}_c$).

In addition, we will show false color images of the errors in the normals maps, and also render images with environment mapped and bump mapped surfaces. For these, we will compute the structural similarity [WBSS04] quality measure.

4. New Algorithm

Let us start with a simple motivating example. Imagine we have a normal map, as in Figure 1, consisting mainly of parallel lines. If the lines are axis-aligned, 3Dc will handle this example pretty well, as a tight axis-aligned bounding box (AABB) would capture the details. If the lines are rotated, however, the projected values will be more spread out. Thus, the AABB will inevitably grow, resulting in less accurate encoding. The enhanced 3Dc (e3Dc) algorithm handles this by including a small set of angles, thus making the encoder less sensitive to directed features. However, we would like generalize this. The artist should not need to try out the "best" initial position before baking the texture for best compressed quality. We also note that texture atlases contain many small maps, which are packed into a single texture. This is often an automatic process, and can create arbitrarily oriented small texture pieces. This is another case where a *rotation-invariant* normal map compression scheme would be desired.

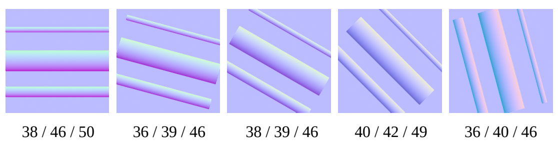

Figure 1: *An example with strong directed features. PSNR values are listed for 3Dc / e3Dc / Tight Frame respectively.*

4.1. Tight Frame Encoding

Here, we describe our rotation-invariant normal map compression algorithm. Instead of creating a bounded interval for our x- and y-values, we express a bounding box in a new coordinate frame using only two points, $\mathbf{p} = (p_x, p_y)$ & $\mathbf{q} = (q_x, q_y)$, and the aspect ratio, $a = \frac{height}{width}$, where $width$ is $||\mathbf{p} - \mathbf{q}||$, and *height* is the height of the rotated box. Figure 2 shows this setup. The two axes of this coordinate frame are simply $\hat{\mathbf{e}}_1 = \mathbf{q} - \mathbf{p}$, and $\hat{\mathbf{e}}_2 = (-\hat{e}_{1_y}, \hat{e}_{1_x})$. The lower left point in this frame is $\mathbf{s} = \mathbf{p} - 0.5a\hat{\mathbf{e}}_2$. It should be noted that a similar setup has been discussed in HDR texture compression [MCHAM06]. Once we have defined this oriented bounding box (OBB), we distribute points uniformly in the box, using the aspect ratio to select the number of divisions along the two axes. For example, in the case of a very wide OBB, it makes more sense to use more points along the widest axis. This *variable point distribution* (VPD) [MAMS06] becomes more powerful in our algorithm, as it is easier to find a compact OBB than a compact AABB (3Dc), or fix-rotation AABB (e3DC). See Figure 3 for an illustration of the benefits of VPD.

The flexibility of the OBB combined with the redistribution of sample points (VPD) makes for a simple, yet powerful algorithm which gives high quality compression when

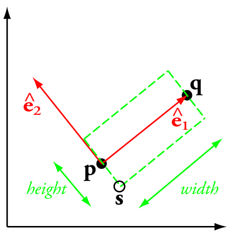

Figure 2: *The coordinate system of our tight frame (TF) coding algorithm.*

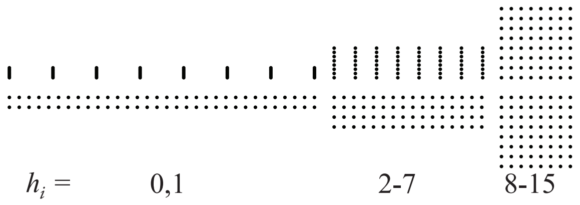

Figure 3: *Without (top) and with (bottom) variable point distribution (VPD). By adapting the point distribution to the aspect ratio of the bounding box, the area is more evenly sampled. h_i is a four bit number, as described below.*

there is correlation to exploit between the x- and y-channels. Hereafter, this technique is called *tight frame* (TF) coding. The target of our algorithm is 8 bits per pixel (bpp), i.e., 128 bits for 4×4 pixels. Similar to 3Dc and e3Dc, we use six bpp for indices. This leaves 32 bits for encoding the bounding box. The information needed to reconstruct the bounding box comprises the two points **p** & **q** and the aspect ratio a. To stay in the bit budget of 32 bits, **p** and **q** are quantized to $7+7$ bits per point, leaving four bits to a. Note that the points **p** and **q** can always be oriented so that a is a number between zero and one. Being able to encode $a = 1.0$ means that there are two ways of expressing the same bounding box (rotate the first box 90 degrees). In order to avoid this redundancy, we use a maximum value of a which is somewhat smaller than 1.0. In addition, $a = 0.0$ is not particularly useful. For these reasons, we use the following reconstruction levels: $a = \frac{1}{32} + h_i \frac{1}{16}$, where h_i is the 4-bit number stored. Since a increases in steps of $\frac{1}{16}$, the height can be inexpensively calculated from the width using shifts, additions, and integer multiplication with h_i only.

4.2. Differential Mode

Similarly to e3Dc, we include a special mode for handling slowly varying normals inside a block. This is mode is triggered when $p_x \geq q_x$ and $p_y \geq q_y$ [MAMS06], and the same trick is used to recover the payload bits for this mode. However, our encoding is slightly different. To increase the accuracy of the bounding box positions (**p** and **q**) of this mode, we encode normals inside a (non-rotated) *square*. We encode the lower-left corner of the square using 2×11 bits, and the length of the square side is coded using 8 bits. Inside the square, we use 8×8 uniformly distributed points, which costs $3+3$ index bits per pixel. All in all, this mode costs $22+8+16 \times 6 = 126$ bits per block. Since we target slowly varying normals with this mode, we limit the square's side length for added precision. As an example, we can use a maximum length of $1/4$. This implies that the minimum side of the square is $\frac{1}{4 \times 2^8} = \frac{1}{1024}$. If we select a smaller maximum size, say $1/32$, we get square sizes in $[\frac{1}{32768}, \frac{1}{32}]$. For the test series used in this paper, a max length of $1/4$ worked well. For comparison, e3Dc uses a differential mode with 2×11 bits for positions and 2×4 bits for a differential vector. This implies a length of the differential vector in the smaller interval $[\frac{1}{512}, \frac{1}{32}]$, but the mode is not restricted to squares, making it a bit more flexible, where applicable.

4.3. Decompression

A proposal for a hardware decompressor is illustrated in Figure 8. The two vectors spanning the bounding box, $\hat{v} = a\hat{e}_2$ and $\hat{e}_1$, as well as the lower left point s, are calculated by the green part. The red part calculates the same values for the differential version of the coder. The blue part assigns the right bits for the variable point distribution.

Without implemeting 3Dc, e3Dc and TF in VHDL, it is hard to estimate relative gate counts for the different algorithms. However, comparing Figure 8 with the diagram of e3Dc [MAMS06] et al., we see that TF has twice the number of "multiply and divide" units compared to e3Dc, plus two extra smaller multipliers in the green area. Thus a fair guess would be that TF is up to twice as complex as e3DC, which in turn is slightly more complex than 3Dc.

5. Results

To evaluate the visual quality of our compressor, we have used 20 representative normal maps, which are the same ones used previously in normal map compression research [MAMS06].

In Figure 4, we present both individual PSNR and maximum angle deviation for the test suite. As can be seen, our algorithm has slightly better scores than e3Dc for the majority of the normal maps, and significantly better scores than 3Dc for all maps. For the “bumpy”-map, e3Dc is better due to that our algorithm uses 7+7 bits for the endpoints, while e3Dc uses 8+8. Further, as all normals in that image are essentially along a horizontal line, there is no gain from being able to rotate the boxes. In the table to the right, we present PSNR values obtained by first averaging the MSE values for all the normal maps. PSNR is then computed on this accumulated MSE using Equation 2. Note that it is not correct to simply average the PSNR scores of the individual images, since this is not a linear operator. In the extreme — if one image would get zero error, it would get infinite PSNR and the aggregate PSNR figure would also be infinite, irrespectively of the errors in the other images. Averaging the MSE and then calculating the PSNR avoids this pitfall. As can be seen, our algorithm has better scores than both 3Dc and e3Dc.

3Dc	e3Dc	TF
30.87	32.74	33.50

The maximum angle error (bottom part of Figure 4) indicates that our algorithm is more robust than the other algorithms in all but one image. In Figure 5, we show the histograms over the angular error. Intuitively, it is better to have less area to the right, and more area to the left. As can be seen, our TF algorithm consistently performs a bit better in this respect.

To further illustrate the improvement of our algorithm, we show false color images of the compressed normal maps in Figure 6, and zoomed-in renderings in Figure 7.

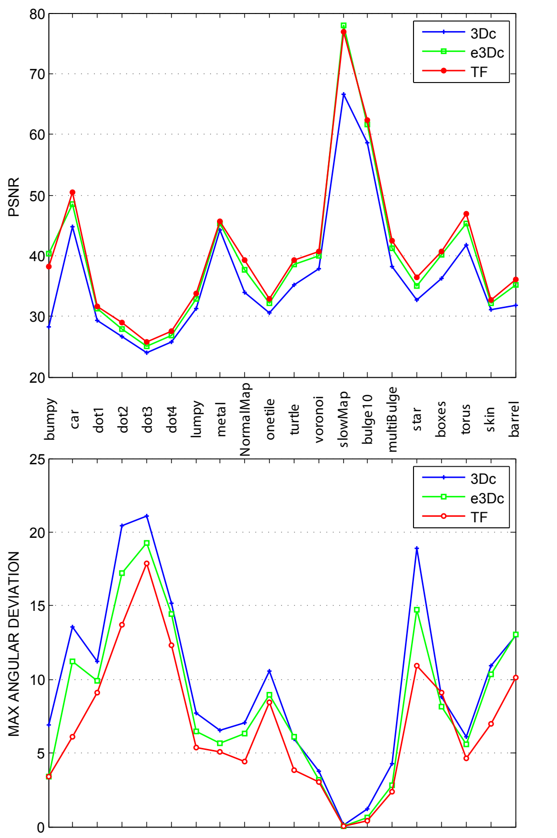

Figure 4: *The PSNR (top) and the maximal angular error (bottom) of all images in the test. We can clearly see a more robust behavior for our tight frame (TF) algorithm in both error measures. Please note that all encoders are optimized for MSE.*

6. Conclusion

In a sense, our work here is quite incremental, since we have basically put together building blocks from other texture & normal map compression research. However, we have shown that this novel combination gives a powerful normal map compression algorithm with high quality under a wide set of error/quality measures. Furthermore, for mobile devices, compression algorithms are very important, and we hope that our technique can be considered for inclusion in OpenGL ES.

Acknowledgments

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks for the NVIDIA fellowship. Thanks to Jon Hasselgren for proofreading.

References

[ANRS05] Abate A. F., Nappi M., Ricciardi S., Sabatino G.: Fast 3D Face Recognition Based On Normal Map. In *Proceedings of ICIP* (2005), vol. 2, pp. 946–949.

[ATI05] ATI: *Radeon X800: 3Dc White Paper*. Tech. rep., 2005.

[Bli78] Blinn J.: Simulation of Wrinkled Surfaces. In *Proceedings of SIGGRAPH* (1978), pp. 286–292.

[Dee95] Deering M.: Geometry Compression. In *Proceedings of SIGGRAPH* (1995), ACM Press, pp. 13–20.

[FB04] Fenney S., Butler M.: Method and Apparatus for Compressed 3D Unit Vector Storage and Retrieval. Patent WO 2004/008394 A1, 2004.

[MAMS06] Munkberg J., Akenine-Möller T., Ström J.: High Quality Normal Map Compression. In *Graphics Hardware* (2006), pp. 95–101.

[MCHAM06] Munkberg J., Clarberg P., Hasselgren J., Akenine-Möller T.: High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics, 25*, 3 (2006), 698–706.

[WB06] Wong A., Bishop W.: Adaptive Normal Map Compression for 3D Video Games. In *Proceedings of Future Play* (2006).

[WBSS04] Wang Z., Bovik A. C., Sheikh H. R., Simoncelli E. P.: Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing, 13*, 4 (2004), 600–612.

[YA06] Yamasaki T., Aizawa K.: Fast and Efficient Normal Map Compression Based on Vector Quantization. In *Proceedings of ICASSP* (2006), vol. 2, pp. 2–12.

[YHA05] Yamasaki T., Hayase K., Aizawa K.: Mathematical Error Analysis of Normal Map Compression Based on Unity Condition. In *Proceedings of ICIP* (2005), vol. 2, pp. 253–269.

[YP06] Yang B., Pan Z.: A Hybrid Adaptive Normal Map Texture Compression Algorithm. In *International Conference on Artificial Reality and Telexistence* (2006), IEEE Computer Society, pp. 349–354.

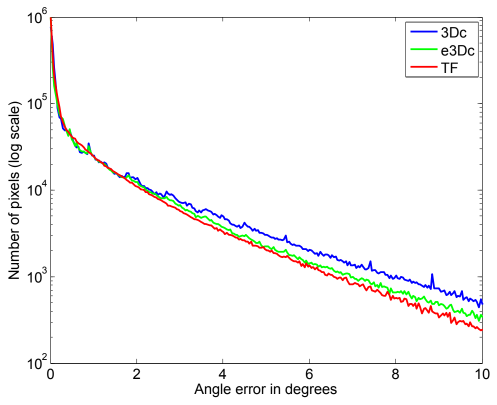

Figure 5: *The error distribution of the algorithms.*

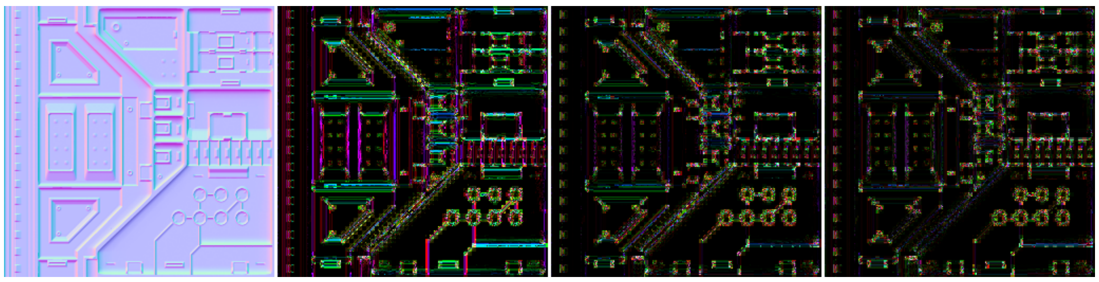

Figure 6: *False color images of the pixel errors: From left to right: Original map, 3Dc, e3Dc and TF. We can clearly see improved performance of the TF algorithm over the two 3Dc compressor variants.*

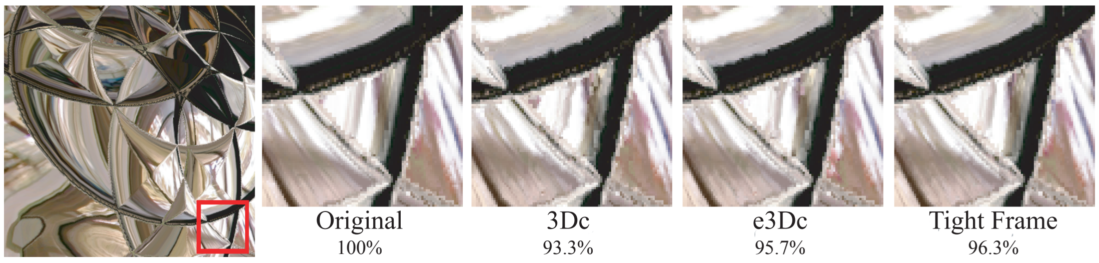

Figure 7: *Rendered quality in a real-time engine. Note that the figures below the zoomed images are Structural Similarity values for the entire screenshot.*

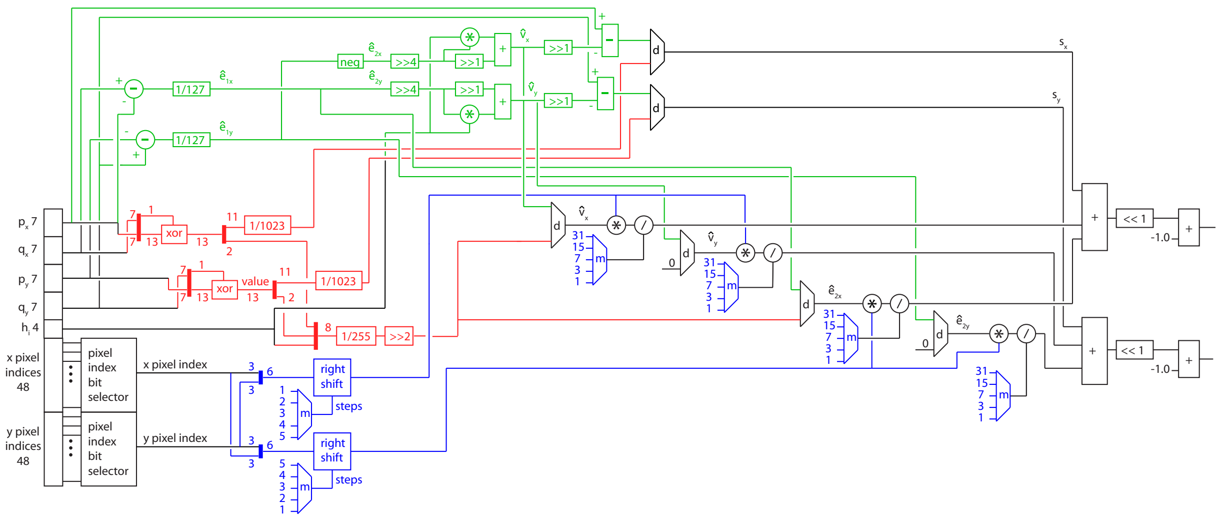

Figure 8: *Diagram of a proposal for a hardware implementation of the decoder.* ***Green:*** *calculates* $\hat{\mathbf{v}} = a\hat{\mathbf{e}}_2$ *and* $\hat{\mathbf{e}}_1$ *which are the two vectors spanning the bounding box. It also calculates* $\mathbf{s}$, *which is the lower left point of the bounding box.* ***Red:*** *calculates,* $\hat{\mathbf{v}}$, $\hat{\mathbf{e}}_1$ *and* $\mathbf{s}$ *for the differential version of the decoder. Note that, if* $p_x \geq q_x$ *and* $p_y \geq q_y$, *differential data is stored in* p_x, p_y, q_x *and* q_y. ***Blue:*** *assigns the the right bits for the variable bit distribution using the multiplexors marked with m. Likewise, the multiplexors marked with d choose between regular and differential input. Note that multiplication with* $1/127$ *can be approximated efficiently as 1/128 + 1/16384 which is implementable with shifts and additions (not shown). Likewise for 1/255 and 1/1023.*