Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

High Quality Normal Map Compression

Jacob Munkberg[1] Tomas Akenine-Möller[1] Jacob Ström[2]

[1]Lund University [2]Ericsson Research

Abstract

Normal mapping is a widely used technique in real-time graphics, but so far little research has focused on compressing normal maps. Therefore, we present several simple techniques that improve the quality of ATI's 3Dc normal map compression algorithm. We use varying point distributions, rotation, and differential encoding. On average, this improves the peak-signal-to-noise-ratio by 3 dB, which is clearly visible in rendered images. Our algorithm also allows us to better handle slowly varying normals, which often occurs in real-world normal maps. We also describe the decoding process in detail.

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

1. Introduction

Bump mapping [Bli78] is a widespread technique which adds the illusion of detail to geometrical objects in an inexpensive way. More specifically, a texture, called a *bump map* or *normal map*, is used at each pixel to perturb the surface normal. A common approach to generate normal maps is to start with a high polygon count model and create a low complexity model using some geometrical simplification algorithm (see, for example, Cohen et al's work [COM98]). The "difference" between these two models is then "baked" into a normal map. For real-time rendering, the normal map is applied to the low complexity model, giving it a more detailed appearance. These techniques are heavily used in recent games.

A possible disadvantage is that the savings in transform and rendering due to the lower vertex count is translated into an increase in bandwidth usage of textures (normal maps). A traditional technique to alleviate this problem is lossy *texture compression* (TC), which was introduced in 1996 [BAC96, KSKS96, TK96]. TC developed primarily for color can also be applied to normal maps [Gre04], but the quality can be higher if specialized algorithms are developed. One such technique, called 3Dc, has been proposed by ATI [ATI05].

However, little effort has been spent on developing new algorithms for *normal map compression*. One problem with 3Dc is that it cannot handle slowly varying normal maps well. This is illustrated in Figure 10. In this paper, we develop several variations and extensions of 3Dc that perform much better on average, and handle slowly varying data particularly well. We present visual proof showing that our normal mapping algorithms give higher quality renderings, and we also show that the peak-signal-to-noise ratio (PSNR) is improved.

2. Previous Work

The first example of normal compression in graphics that we know of is described in the context of geometry compression [Dee95], i.e., it was not targeted towards normal map compression. Deering presents a method for compressing surface normals, arguing that about 100,000 normals distributed over the unit sphere would give sufficient quality. These normals can be represented by a single 17-bit index, and by exploring symmetries on the sphere, only a 1/48 of the sphere needs to be represented. A regular grid in the angular space of one such patch is used as sample distribution. Nearby normals are encoded differentially. With these techniques he manages to compress a normal to about 12 bits. However, the decompression step includes a number of trigonometric operations and is quite costly compared to the schemes described below.

2.1. 3Dc Normal Compression

Next, we will review ATI's normal map compression scheme called *3Dc* [ATI05]. As far as we know, this is the only format dedicated to this purpose alone.

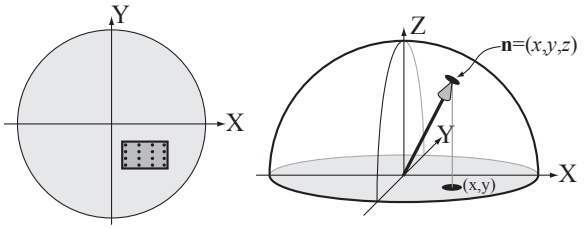

Figure 1: *3Dc selects a rectangle in the xy-plane (left), and places 8×8 points uniformly over this rectangle (in this figure, only 4×4 points were placed to make the illustration clearer). These points can be seen as a "palette" of xy pairs, and each texel in a 4×4 tile can select one of these pairs. To the right, one such (x,y)-point has been used to generate a normal, $\mathbf{n} = (x,y,z)$. This is done by requiring that we use unit normals.*

In the majority of cases today, bump mapping is performed in local tangent space, (X,Y,Z), of each rendering primitive (e.g. a triangle). Since the length of the normal is not of interest, 3Dc uses units normals, and hence it suffices to compress the x- and y-components. The third component is obtained through normalization:

$$z = \sqrt{1 - x^2 - y^2}, \tag{1}$$

and this computation can either be done in the pixel shader, or by special purpose hardware.

The x- and y-components are compressed independently using a variant of S3TC/DXTC [INH99]. A block of 4×4 texels (a.k.a. a tile) is compressed into 128 bits, i.e., at eight bits per pixel (bpp). The x-coordinates are encoded in the following way. Two eight-bit values, x_{start} and x_{stop}, representing an interval enclosing the x-values in the tile, are found. Each texel can select from eight different x-values: $x_k = x_{start} + k(x_{stop} - x_{start})/7$, $k = 0 \ldots 7$, which are thus spread uniformly over the interval. This requires three bits per texel. To encode the x-values of a tile, we need 2×8 bits for x_{start} and x_{stop}, and 16×3 bits for the per-pixel indices. This sums up to 64 bits. The y-components are encoded in the same way, and the total cost per tile is 128 bits. An illustration of 3Dc is shown in Figure 1.

3. Improved Normal Compression

In the following three subsections, we present three simple general techniques for improving the quality of the 3Dc normal compression scheme. These are combined into a single compression format in Section 4, while keeping a bit budget of 8 bits per pixel (bpp). Compared to 3Dc, the extra cost is a more expensive decompression phase (Section 4.1).

First, however, we will explain how we can incorporate three new modes into 3Dc. It stems from the fact that swapping the values x_{start} and x_{stop} will produce exactly the same reconstruction levels $x_0 \ldots x_7$, albeit in the reversed order. Since these two representations are equivalent, it is possible to signal one extra bit, b: If $x_{start} < x_{stop}$, then $b \leftarrow 0$, else $b \leftarrow 1$. The same trick is used in DXT1 to signal whether a block is RGB or RGBA, and we call this trick the *ordering technique*. In 3Dc, the ordering technique can be used on both x and y, and hence two extra bits can be used.

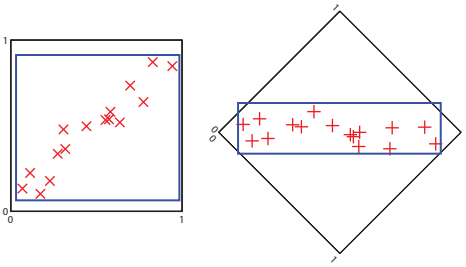

Figure 2: *By rotating the coordinate frame, we can often find a much tighter bounding box. This will improve the encoding precision.*

3.1. Rotation Compression

When the major axis of a minimal box around the (x,y) points of a tile do not coincide with either the x- or the y-axis, the quality of 3Dc decreases. By rotating the coordinate frame, a much tighter fit can be obtained, and the extra storage cost is only an angle per block. Figure 2 illustrates this scenario. For example, using a single extra bit, one can select to use an angle in the set $\{0, \pi/4\}$, and two bits increase the set to $\{0, \pi/8, \pi/4, 3\pi/8\}$. Note that the standard 3Dc case is included, thus, this technique can only achieve results equivalent to or better than 3Dc. As seen in Figure 3, the peak-signal-to-noise-ratio (PSNR) improves with more than a decibel on average, already with a set of three angles. Visual results are shown in Section 5.

3.2. Variable Point Distribution

Normally, the 3Dc technique places the sample points uniformly in a grid over the axis-aligned box defined by (x_{min}, y_{min}) and (x_{max}, y_{max}), where $x_{min} = \min(x_{start}, x_{stop})$, $x_{max} = \max(x_{start}, x_{stop})$, and ditto for y_{min} and y_{max}. However, other distributions may allow for better compression. A simple way of altering the sample distribution is to use different distributions depending on the aspect ratio of the box. For example, if the box is more than twice as wide as it is high, then it could be beneficial to use a 16×4-distribution rather than the standard 8×8-distribution. See Figure 4. No extra bits are needed to signal this, since the point distribution is automatically triggered by the aspect ratio, $a = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$, of the box. For 3Dc, the per-texel indices are encoded in six bits ($3+3$ bits for an (x,y) pair). However, if the aspect ratio triggers, say, the distribution 2×32, we

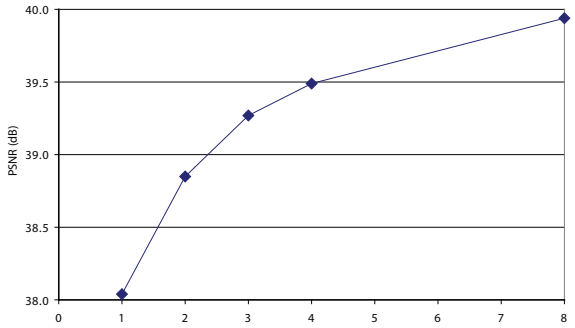

Figure 3: *The average PSNR for a set of 20 normal maps as a function of the number of angles in the compressor. Angle count 1 represent no rotation, 2 represent the two angles* $\{0, \frac{\pi}{2}\}$ *and generally, for an angle count a, the set of angles is* $\{0, \frac{\pi}{2a}, ..., \frac{\pi(a-1)}{2a}\}$.

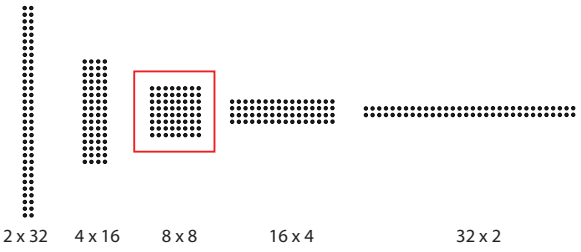

Figure 4: *Different point distributions are triggered automatically dependent on the aspect ratio,* $a = \frac{y_{max}-y_{min}}{x_{max}-x_{min}}$*, of the bounding box.*

aspect ratio ($a = \frac{y_{max}-y_{min}}{x_{max}-x_{min}}$)	distribution ($d_x \times d_y$)
$a < 1/8$	32×2
$1/8 \leq a < 1/2$	16×4
$1/2 \leq a \leq 2$	8×8
$2 < a \leq 8$	4×16
$a > 8$	2×32

Table 1: *The bounding box aspect ratio automatically selects a point distribution.*

simply move two bits, $3+3 \rightarrow 1+5$. It should be noted that this approach cannot guarantee higher quality in all cases. We have tested this technique on a set of 20 normal maps, with improved PSNR values on all maps. The bounds for selecting a distribution were chosen empirically and are presented in Table 1. The distributions 1×64 and 64×1 did not improve the quality, and are not used in our compressor.

3.3. Differential Encoding

One of the case where it is easy to detect compression artifacts is in areas that have a slight curvature, for example,

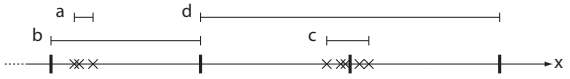

Figure 5: *The x-axis is shown with quantized values marked with bold vertical lines. Left:* **a** *is the desired interval, but the smallest interval representable in 3Dc is* **b**. *Right: With values on both sides of a quantized value, the smallest interval in 3Dc that covers the desired interval* **c** *is* **d**, *twice the size of the smallest representable interval* **b**.

on a car hood. The smoothness of the surface makes it easy for the viewer to predict what the image "should" look like, which is not as simple for a rough surface.

Compressing such slow varying normals with 3Dc poses two problems. First, the smallest representative interval is too wide. Since the quantized resolution is only eight bits, an interval of 1/255 of the range might be to coarse for representing nearly constant normals (see Figure 5a and b). Second, the smallest interval cannot be placed accurately enough, as the interval limits must coincide with the quantization steps. Thus, if values of a block are present on both sides of a quantized step (Figure 5c), the smallest interval covering all values will be at least twice the minimum interval (Figure 5d) . In this section, we will present a technique to make the precision higher in order to solve these to problems.

Our idea is to use the 32 bits that are normally used for storing x_{start}, x_{stop}, y_{start} and y_{stop} in a different way, with an encoding that is specialiced for representing small intervals accurately. However, we must be able to flag this mode of encoding, so some bits are irretrievably lost. Using a simple mapping technique described in the next paragraph, we can exploit 30 bits for a differential mode that handles slowly varying normals. In this mode, we use eleven bits each to encode x_{min} and y_{min} using 8.3 (eight bits for the integer part and three bits for the fractional part), and we spend four bits each on two delta values, Δ_x and Δ_y, using 2.2 bits. x_{max} is calculated as $x_{max} = x_{min} + \Delta_x$, and ditto for y_{max}. Due to the differential coding, we call this mode the *differential* mode, and it addresses both problems identified above: the smallest representable interval is now four times smaller, and since the precision of the location of the interval (3 fractional bits) is twice that of the smallest length (2 fractional bits), we can handle values on both sides of a border as in Figure 5c without doubling the interval.

In the following, we will present a general method useful when exploiting the ordering technique (see beginning of Section 3). Assume that we have detected a special mode signaled by $x_{start} \geq x_{stop}$. Unfortunately, we cannot set the bits of x_{start} and x_{stop} arbitrarily, since x_{stop} must be less than or equal to x_{start}. We thus want to solve the problem of exploiting a maximum number of the sixteen bits occupied by x_{start} and x_{stop}, while preserving $x_{start} \geq x_{stop}$. This

		x_{start}							
		0	1	2	3	4	5	6	7
x_{stop}	0	0	1	2	3	4	5	6	7
	1	■	9	10	11	12	13	14	15
	2	■	■	18	19	20	21	22	23
	3	■	■	■	27	28	29	30	31
	4	■	■	■	■	x	26	25	24
	5	■	■	■	■	■	x	17	16
	6	■	■	■	■	■	■	x	8
	7	■	■	■	■	■	■	■	x

Table 2: *By mirroring the positions for number 8, 16, 17, 24, 25 and 26, it is possible to fit the numbers 0 through 31 without using positions where $x_{start} < x_{stop}$ (marked with black).*

can be solved by a simple mapping, illustrated in Table 2, where x_{start} and x_{stop} are 3-bit values instead of 8-bit values for simplicity. Here, we have entered the numbers 0 through 31 into the table, while avoiding the black boxes where $x_{start} < x_{stop}$. The numbers are entered row-by-row, except for the numbers which would have fallen in the forbidden positions, namely numbers 8, 16, 17, 24, 25 and 26. The positions for these numbers are therefore mirrored both in the vertical and horizontal direction relative to the center of the table. As can be seen, we have stored 32 numbers, and we can therefore extract five bits. This is the maximum number of bits we can obtain since roughly half the values are marked with black.

Decoding this 5-bit number is especially simple for the upper half (rows 0 through 3) using

$$value = (x_{stop} << 3) \text{ OR } x_{start},$$

where $<<$ represents a left shift and OR is the bit-wise logical OR operator. For the lower half (rows 4 through 7), we have to mirror x_{start} and x_{stop} first to $(7 - x_{start})$ and $(7 - x_{stop})$, which is the same as inverting their bits, and we can use

$$value = (\text{NOT}(x_{stop}) << 3) \text{ OR NOT}(x_{start}),$$

where NOT$(\cdot)$ denotes bit-wise inversion. For eight bit x-values, we shift with 8 instead of 3, and we can store 15 bits in *value*. Encoding is straightforward—we use the lower part of *value* for x_{start} and the upper part for x_{stop}, and invert both if $x_{stop} > x_{start}$ according to the pseudocode below:

```
xstart = value AND 0xff
xstop = (value » 8) AND 0x7f
if xstop > xstart
  xstart = NOT(xstart)
  xstop = NOT(xstop)
end
```

where NOT operates on all eight bits.

4. Proposed Scheme

In this section, we will combine the three techniques described above into a format that fits in an 8 bpp budget. The foundation for our combined mode is 3Dc, but we exploit redundancy in its encoding to allow for more modes. Next, we will describe how these two extra bits can be used to improve the quality of 3Dc substantially.

We allow two rotations and limit the differential mode to tiles where both the x- and the y-components can be encoded differentially. Altogether, we have four different modes: I) the standard 3Dc mode, II) a rotation with 30 degrees, III) a rotation with 60 degrees, and IV) a differential mode, encoded with $8.3 + 2.2$ bits. As seen in Figure 3, using three angles gives a significant improvement in quality. It would be possible to add yet another angle, but that mode is more wisely spent on the differential mode in terms of PSNR. The variable point distribution is applied to all modes except the differential one where it did not increase quality. Table 4 shows the quality contribution that each technique adds on a test series. The usage of each mode is further illustrated in Figure 6, showing how often the different modes are used for each test image. All modes are used quite frequently, which indicates a balanced algorithm.

Note that mode I differs slightly from 3Dc in that it uses variable point distribution. Alternatively, it is possible to avoid using variable point distribution in mode I. This would mean that existing 3Dc hardware designs could be reused to decode this mode. Maybe more important, it would allow existing 3Dc textures to be transcoded to our new format without loss, by swapping x_{start} and x_{stop} if $x_{start} > x_{stop}$ (and performing bit-wise NOT on the per-pixel indices to reflect the inverted ordering). However, this backward compatibility would come at a cost: On the test images of Section 5, the average PSNR for this alternative solution is about 1.3 dB lower than the proposed scheme.

4.1. Decoding

The decoding of a block is performed as follows:

mode	X	Y	bits	vpd
I: rot 0°	$x_{start} < x_{stop}$	$y_{start} < y_{stop}$	8+8	yes
II: rot 30°	$x_{start} \geq x_{stop}$	$y_{start} < y_{stop}$	8+8	yes
III: rot 60°	$x_{start} < x_{stop}$	$y_{start} \geq y_{stop}$	8+8	yes
IV: diff	$x_{start} \geq x_{stop}$	$y_{start} \geq y_{stop}$	8.3+2.2	no

Table 3: *The encoding modes for the combined normal compressor. vpd indicates "variable point distribution."*

mode	$\overline{PSNR}$ (dB)
3Dc	36.4
3Dc + Point Distr.	37.5
3Dc + Point Distr. + Rot	38.8
3Dc + Point Distr. + Rot + Diff	39.4

Table 4: *The average PSNR for the normal maps presented in Figure 8.*

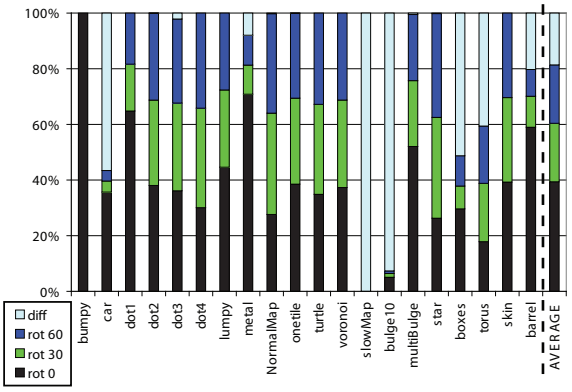

Figure 6: *The frequencies of the different algorithms for the images used in the test.*

1. First, x_{start}, x_{stop}, y_{start} and y_{stop} are tested to see which mode the block belongs to, according to Table 3. For instance, if $x_{start} < x_{stop}$ and $y_{start} \geq y_{stop}$, then mode III is selected.
2. The next step is to calculate x_{min} and x_{max}. For modes I through III, this is simply done using $x_{min} = \min(x_{start}, x_{stop})$ and $x_{max} = \max(x_{start}, x_{stop})$, and likewise for y_{min} and y_{max}. All resulting numbers will be between 0 and 255. For mode IV, the 15-bit *value* is first calculated from x_{start} and x_{stop} as described in Section 3.3. Then, the first eleven bits of *value* are used to decode x_{min} in format 8.3, i.e., with eight bits for the integer part and three for the fractional part, resulting in a number between 0 and 255.875. The last four bits of *value* are decoded as an offset, Δ_x, in fixed-point format 2.2, resulting in a number between 0 and 2.75. x_{max} is finally calculated as $x_{min} + \Delta_x$. Similar computations are performed for y_{min} and y_{max}.
3. The aspect ratio $a = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$ is computed, and a point distribution is selected according to Table 1. Denote the distribution $d_x \times d_y$. For mode IV, the distribution is always 8×8.
4. The reconstruction levels are calculated using $x_k = x_{min} + \frac{k}{d_x - 1}(x_{max} - x_{min})$, $k = 0, \ldots, d_x - 1$, and likewise for y_k.
5. The pixel indices are now used to determine which reconstruction level to use. For instance, a value of 010_{bin} selects reconstruction level x_2 for x. The y-value is obtained analogously.
6. For modes II and III, we will also rotate the coordinates using $\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{M} \begin{pmatrix} x \\ y \end{pmatrix}$, where $\mathbf{M} = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$ is a rotation matrix and ϕ is $-\pi/6$ or $-\pi/3$. See Section 4.2 for an efficient implementation. For modes I and IV, we just use $x' = x$ and $y' = y$.
7. Division by 255, and remapping to $[-1, 1]$ follows: $x'' = 2x'/255 - 1$ and $y'' = 2y'/255 - 1$. In the differential mode, clamping the values to the interval $[-1, 1]$ can also be necessary.
8. Finally, the z coordinate is calculated as $z'' = \sqrt{1 - x''^2 - y''^2}$. The decompressed normal for the pixel is (x'', y'', z'').

The last two steps can be performed in the pixel shader.

4.2. Efficient Rotation

In this section, we suggest a hardware-friendly rotation. For modes II and III of our algorithm, the decompressor needs to rotate a two-dimensional point by -30 and -60 degrees. In the following, we develop an inexpensive, approximate rotation for -30°. The case with -60° uses the same constants, but at different locations in the matrices, so this is omitted from our description. The matrix for rotating -30 degrees is:

$$\mathbf{M} = \begin{pmatrix} \cos(-\pi/6) & -\sin(-\pi/6) \\ \sin(-\pi/6) & \cos(-\pi/6) \end{pmatrix} = \begin{pmatrix} 0.86602... & 0.5 \\ -0.5 & 0.86602... \end{pmatrix}. \quad (2)$$

The 0.5-terms above are not expensive to implement, but multiplication by $\sqrt{3}/2 \approx 0.86602$ is. To that end, we suggest that the hardware-friendly matrix $\tilde{\mathbf{M}}$ is used instead:

$$\mathbf{M} \approx \tilde{\mathbf{M}} = \begin{pmatrix} 1 - \frac{1}{8} & 0.5 \\ -0.5 & 1 - \frac{1}{8} \end{pmatrix} = \begin{pmatrix} 0.875 & 0.5 \\ -0.5 & 0.875 \end{pmatrix}, \quad (3)$$

where multiplication by 0.875 can be implemented as a shift by three and a subtraction. Note that $\tilde{\mathbf{M}}$ is *not* an orthogonal matrix, i.e., $\tilde{\mathbf{M}}\tilde{\mathbf{M}}^{\mathrm{T}} \neq \mathbf{I}$. Therefore, we emphasize that we cannot use $\mathbf{M}^{\mathrm{T}}$ during compression, because it also holds that $\tilde{\mathbf{M}}\mathbf{M}^{\mathrm{T}} \neq \mathbf{I}$. Instead, we must use the inverse of $\tilde{\mathbf{M}}$ during compression:

$$\tilde{\mathbf{M}}^{-1} = \frac{64}{65} \begin{pmatrix} 0.875 & -0.5 \\ 0.5 & 0.875 \end{pmatrix} \approx \begin{pmatrix} 0.8615... & -0.4923... \\ 0.4923... & 0.8615... \end{pmatrix}. \quad (4)$$

If $\tilde{\mathbf{M}}^{-1}$ is used to transform a rectangle, the result will be different from the rectangle obtained by using $\mathbf{M}^{-1} = \mathbf{M}^{\mathrm{T}}$. In fact, when using $\tilde{\mathbf{M}}^{-1}$, the rectangle will get a slight skew due to the fact that the transform is not orthogonal. However, the average PSNR for all our test images was only reduced by 0.03 dB on average, which is not significant.

See Figure 7 for a possible hardware implementation.

5. Results

To evaluate the visual quality of our compressor, we have tested several normal maps, taken from the set in Figure 8, in a real-time shader development application, in order to mimic a typical user scenario. We have also rendered images using a high-end renderer, with anisotropic mipmap filtering, HDR environment mapping and screen space anti-aliasing. When compressing with 3Dc, we perform exhaustive search for the base values in the x- and y-direction separately, to ensure that our 3Dc compressor is near-optimal. A full exhaustive search over x and y simultaneously was too costly.

In Figure 10, we show visual results obtained using a normal map with slowly varying normals. The pixel shader implemented simple environment mapping in order to better

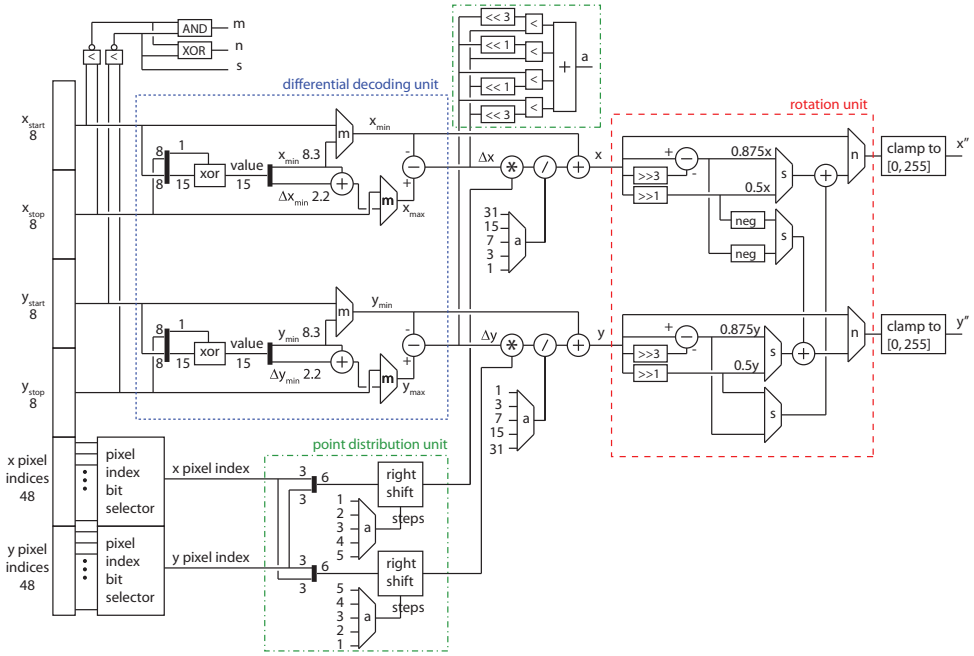

Figure 7: *A hardware decompressor unit for our normal map compression algorithm. To the left, 128 bits of data are shown, and these are used to decode one of the 16 normals in a* 4×4 *tile. As can be seen, our three techniques have been clearly marked. The remaining parts is basically 3Dc (except that 3Dc only divides by 7).*

show the quality. As can be seen, our technique provides superior results compared to ATI's 3Dc technique. For this particular map, we have observed an increase of 10 dB in PSNR compared to 3Dc.

Figure 11 illustrates a test with a typical game normal map [Gre04] with sharp edges. Our algorithm handles many difficult tiles better due to the flexibility offered by the extra rotation and variable point distribution. We rendered the images in Figure 10 and 11 using an NVIDIA GeForce FX 6800 graphics card. In the tests, we use RGB*fp16* textures, which are supported by the GPU.

Another visual test is shown in Figure 12, which was rendered using a high-quality offline renderer.

In addition to obtaining visual results, we also used the *mean square error* (MSE), which is computed as a summation over all normals in the image:

$$MSE = \frac{1}{w \times h} \sum (\hat{x} - x)^2 + (\hat{y} - y)^2 + (\hat{z} - z)^2, \quad (5)$$

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 normal values, we use the *Peak Signal to Noise Ratio* (PSNR):

$$PSNR = 10 \log_{10} \left(\frac{1}{MSE} \right), \quad (6)$$

where the nominator is one, since the *peak signal* for a normal of unit length will always be equal to one, by construction. PSNR values for all images tested, for 3Dc and our combined algorithm are presented in Figure 9, with improved values on all maps. The average improvement is about 3 dB. We see large differences on slowly varying maps and maps with sharp egdes.

6. Conclusions

We have designed three new techniques which can be used in conjunction to the 3Dc normal compression format. As shown in our paper, the combination of these handles many of 3Dc's weaknesses much better. Our techniques are combined into a scheme that still fits into a bit budget of 8 bpp and requires only small additions to a hardware decompressor. The new format is more flexible, with 3Dc as a subset, and we have obtained better results on all normal maps tested, both visually and in the PSNR error measure. For a series of 20 normal maps, the average PSNR increased with 3 dB.

Acknowledgments

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet.

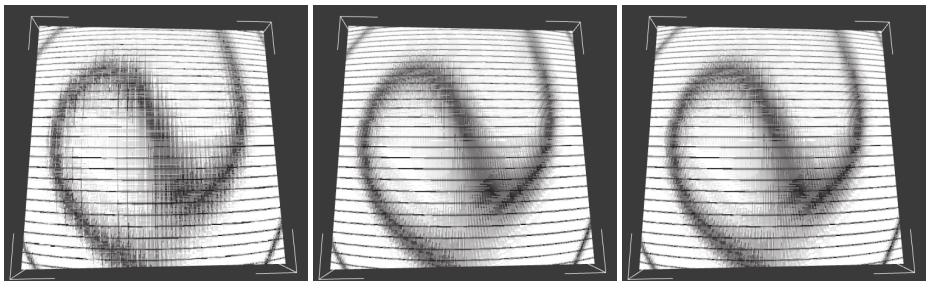

Figure 10: *A grid cube-map environment is used for these images. The normal map is a very slowly varying map (**m**) from Figure 8. Left: normal map compressed with ATI's 3Dc technique. Middle: rendered using original normal map. Right: normal map compressed with our algorithm.*

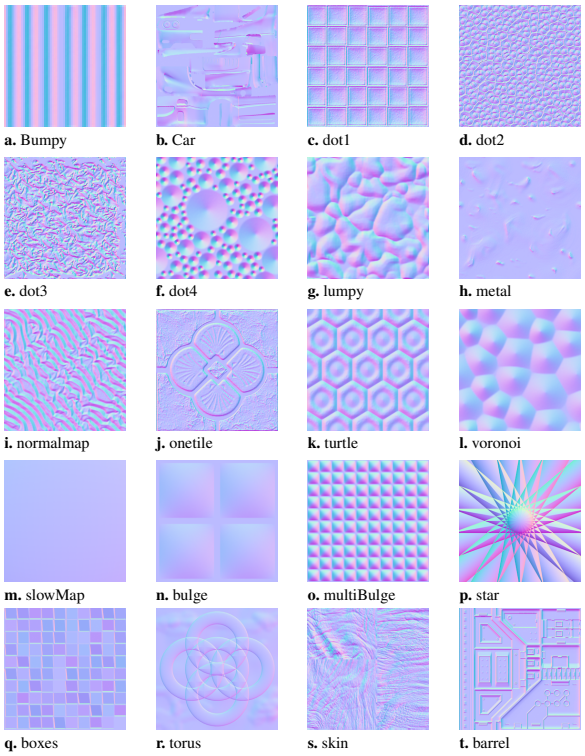

Figure 8: *The set of normal maps used for evaluating our compression algorithm. **m**, **n**, **o**, **p**, **q**, and **r** are 32 bit/channel maps, all other maps are 8 bit/channel.*

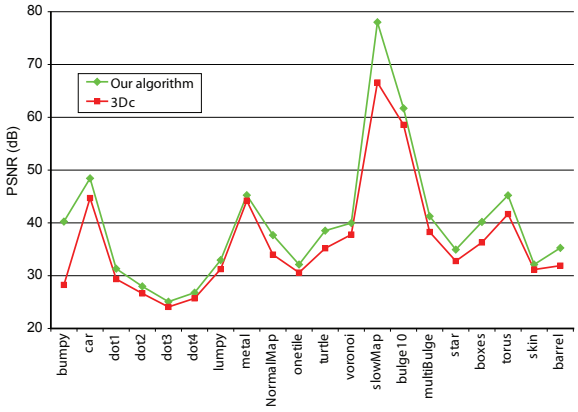

Figure 9: *This chart shows the PSNR values for the images in Figure 8 for 3Dc and our algorithm. Our algorithm is the combined algorithm, using a standard 3Dc mode, rotations (30 and 60 degrees), a differential mode and variable point distribution.*

References

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

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

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

[COM98] COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplification. In *Proceedings of SIGGRAPH* (1998), ACM Press, pp. 115–122.

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

[Gre04] GREEN S.: *Bump Map Compression*. Tech. rep., NVIDIA, 2004.

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

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

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

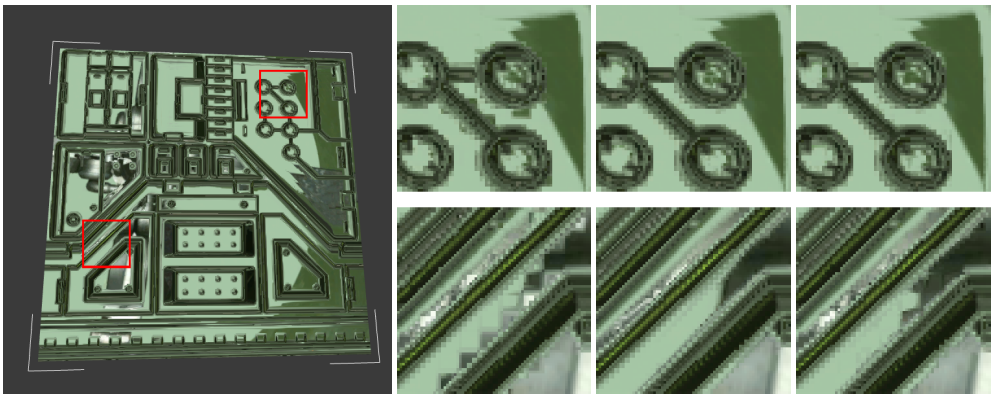

Figure 11: *A typical game normal map (**t**), rendered in a real-time shader development application, with a cube reflection map. Left: normal map compressed with ATI's 3Dc technique. Middle: rendered using original normal map. Right: normal map compressed with our technique.*

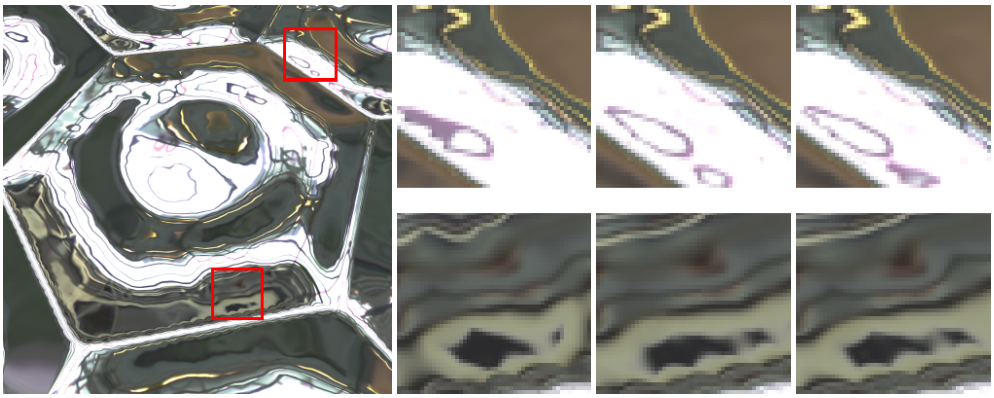

Figure 12: *The normal map (**k**), rendered in a high-end off-line renderer, with HDR environment mapping, texture filtering and advanced anti-aliasing. Left: 3Dc. Middle: uncompressed map. Right: our algorithm. As can be seen in the images, 3Dc shows more "wobbling" artifacts, and some features even disappear. Our new algorithm shows higher quality, even though some artifacts remains.*