Real-Time Normal Map DXT Compression

J.M.P. van Waveren **Ignacio Castaño**
Id Software, Inc. **NVIDIA Corp.**

February 7th 2008

Abstract

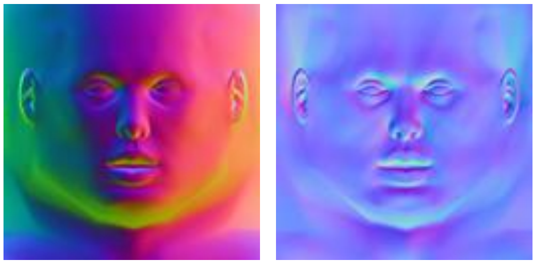
Using today's graphics hardware, normal maps can be stored in several compressed formats, that are decompressed on the fly in hardware during rendering. Several object-space and tangent-space normal map compression techniques using existing texture compression formats are evaluated. While decompression from these formats happens in real-time in hardware during rendering, compression to these formats may take a considerable amount of time using existing compressors. Two highly optimized tangent-space normal map compression algorithms are presented that can be used to achieve real-time performance on both the CPU and GPU.

1. Introduction

Bump mapping uses a texture to perturb the surface normal to give objects a more geometrically complex appearance without increasing the number of geometric primitives. Bump mapping, as originally described by Blinn [1], uses the gradient of a bump map heightfield to perturb the interpolated surface normal in the direction of the surface derivatives (tangent vectors), before calculating the illumination of the surface. By changing the surface normal, the surface is lit as if it has more detail, and as a result is also perceived to have more detail than the geometric primitives used to describe the surface.

Normal mapping is an application of bump mapping, and was introduced by Peercy et al. [2]. While bump mapping perturbs the existing surface normals of an object, normal mapping replaces the normals entirely. A normal map is a texture that stores normals. These normals are usually stored as unit-length vectors with three components: X, Y and Z. Normal mapping has significant performance benefits over bump mapping, in that far fewer operations are required to calculate the surface lighting.

Normal mapping is usually found in two varieties: object-space and tangent-space normal mapping. They differ in coordinate systems in which the normals are measured and stored. Object-space normal maps store normals relative to the position and orientation of a whole object. Tangent-space normals are stored relative to the interpolated tangent-space of the triangle vertices. While object-space normals can be anywhere on the unit-sphere, tangent-space normals are only on the unit-hemisphere at the front of the surface, because the normals always point out of the surface.



Example of an object-space normal map (left),
and the same normal map in tangent-space (right).

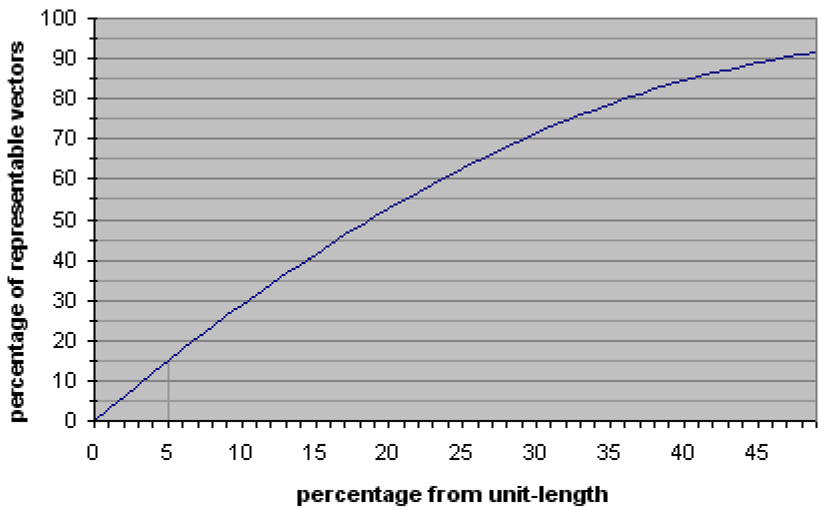
A normal does not necessarily have to be stored as a vector with the components X, Y and Z. However, rendering from other representations usually comes at a performance cost. A normal could, for instance, be stored as an angle pair (pitch, yaw). However, this representation has the problem that interpolation or filtering does not work properly, because there are orientations in which there may not exist a simple change to the angles to represent a local rotation. Before interpolating, filtering, or calculating the surface illumination for that matter, the angle pair has to be converted to a different representation like a vector, which requires expensive trigonometric functions.

Although a normal map can be stored as a floating-point texture, a normal map is typically stored as a signed or unsigned *integer* texture, because the components of normal vectors take values within a well defined range (usually [-1, +1]), and there is a benefit to having the same precision across the whole range without wasting any bits for a floating-point exponent. For instance, to store a normal map as an *unsigned* integer texture with 8 bits per component, the X, Y and Z components are rescaled from real values in the range [-1, +1] to integer values in the range [0, 255]. As such, the real-valued vector [0, 0, 1] is converted to the integer vector [128, 128, 255], which, when interpreted as a point in RGB space, is the purple/blue color that is predominant in tangent-space normal maps. To render a normal map stored as an *unsigned* integer texture, the vector components are first mapped from an integer value to the floating-point range [0, +1] in hardware. For instance, in the case of a texture with 8 bits per component, the integer range [0, 255] is mapped to the floating-point range [0, +1] by division with 255. Then the components are typically mapped from the [0, +1] range to the [-1, +1] range during rendering in a fragment program by subtracting 1 after multiplication with 2. When a *signed* integer texture is used, the mapping from an integer value to the floating-point range [-1, +1] is performed directly in hardware.

Whether using a signed or unsigned integer texture, a fundamental problem is that it is not possible to derive a *linear* mapping from binary integer numbers to the floating-point range [-1, +1], such that the values -1, 0, and +1 are represented exactly. The mapping in hardware of signed integer textures, used in earlier NVIDIA implementations, does not exactly represent +1. For an n-bit unsigned integer component, the integer 0 maps to -1, the integer 2^{n-1} maps to 0, and the maximum integer value 2^n-1 maps to 1 - 2^{1-n}. In other words, the values -1 and 0 are represented exactly, but the value +1 is not. The mapping used for DirectX 10 class hardware is non-linear. For an n-bit signed integer component, the integer -2^{n-1} maps to -1, the integer $-2^{n-1}+1$ also maps to -1, the integer 0 maps to 0, and the integer $2^{n-1}-1$ maps to +1. In other words, the values -1, 0 and +1 are all represented exactly, but the value -1 is represented twice.

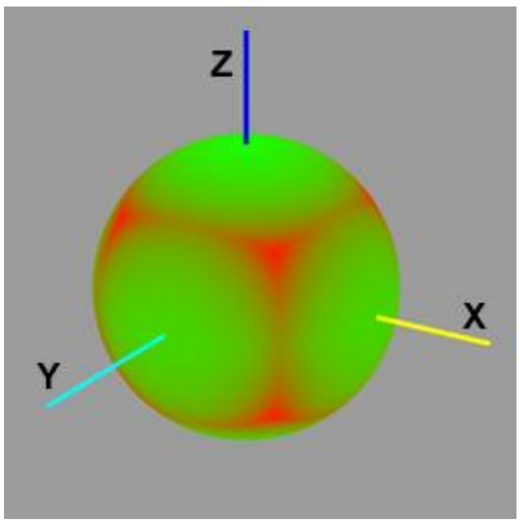
Signed textures are not supported on older hardware. Furthermore, the mapping from binary integers to the range [-1, +1] may be hardware specific. Some implementations may choose to not represent +1 exactly, whereas the conventional OpenGL mapping specifies that -1 and +1 can be represented exactly, but 0 can not. Other implementations may choose a non-linear mapping, or allow values outside the range [-1, +1], such that all three values -1, 0 and +1 can be represented exactly. To cover the widest range of hardware without any hardware specific dependencies, all normal maps used here are assumed to be stored as *unsigned* integer textures. The mapping from the range [0, +1] to [-1, +1] is performed in a fragment program by subtracting 1 after multiplication with 2. This may result in an additional fragment program instruction, which can be trivially removed when a *signed* texture is used. The mapping used here is the same as the conventional OpenGL mapping which results in an exact representation of the values -1 and +1, but not 0.

An integer normal map texture can typically be stored with 16 (5:6:5), 24 (8:8:8), 48 (16:16:16) or 96 (32:32:32) bits per normal vector. Most of today's normal maps, however, are stored with no more than 24 (8:8:8) bits per normal vector. It is important to realize there are relatively few 8:8:8 bit vectors that are actually close to unit-length. For instance, the integer vector [0, 0, 64], which is dark blue in RGB space, does not represent a unit-length normal vector (the length is 0.5 as opposed to 1.0). The following figure shows the percentage of representable 8:8:8 bit vectors that are less than a certain percentage off from being unit-length.



For instance, if it is not considered acceptable for normal vectors to be more than 5% from unit-length, then only about 15% of all representable 8:8:8 bit vectors can be used to represent normal vectors. Going to fewer bits of precision, like 5:6:5 bits, the number of representable vectors that are close to unit-length decreases quickly.

To significantly increase the number of vectors that can be used, each normal vector can be stored as a direction that is not necessarily unit-length. This direction then needs to be normalized in a fragment program. However, there is still some waste because only 83% of the all 8:8:8 bit vectors represent unique directions. For instance, the integer vectors [0, 0, 32], [0, 0, 64] and [0, 0, 96] all specify the exact same direction (they are multiples of each other). Furthermore, the unique normalized directions are not uniformly distributed over the unit-sphere. There are more representations for directions close to the four diagonals of the bounding box of the [-1, +1] x [-1, +1] x [-1, +1] vector space, than there are representations for directions close to the coordinate axes. For instance, there are three times more directions represented within a 15 degrees radius around the vector [1, 1, 1], than there are directions represented within a 15 degrees radius around the vector [0, 0, 1]. The figure below shows the distribution of all representable 8:8:8 bit vectors projected onto the unit-sphere. The areas with a low density of vectors are green, and the areas with a high density are red.



distribution of 8:8:8 bit vectors
projected on the unit-sphere

On today's graphics hardware, normal maps can also be stored in several compressed formats, that are decompressed in real-time during rendering. Compressed normal maps do not only require significantly less memory on the graphics card, but also generally render faster than uncompressed normal maps, due to reduced bandwidth requirements. Various different ways to exploit existing texture compression formats for normal map compression, have been suggested in literature [7, 8, 9]. Several of these normal map compression techniques, and extensions to them, are evaluated in section 2 and 3.

While decompression from these formats is done real-time in hardware, compression to these formats may take a considerable amount of time. Existing compressors are designed for high-quality off-line compression, not real-time compression [20, 21, 22]. However, real-time compression is quite useful for transcoding normal maps from a different format, compression of dynamically generated normal maps, and for compressed normal map render targets. In sections

4 and 5 two highly optimized tangent-space normal map compression algorithms are presented, that can be used to achieve real-time performance on both the CPU and GPU.

2. Object-Space Normal Maps

Object-space normal maps store normals relative to the position and orientation of a whole object. A normal in object-space can be anywhere on the full unit-sphere, and is typically stored as a vector with three components: X, Y and Z. Object-space normal maps can be stored using regular color texture compression techniques, but these techniques may not be as effective, because normal map textures do not have the same properties as color textures.

2.1 Object-Space DXT1

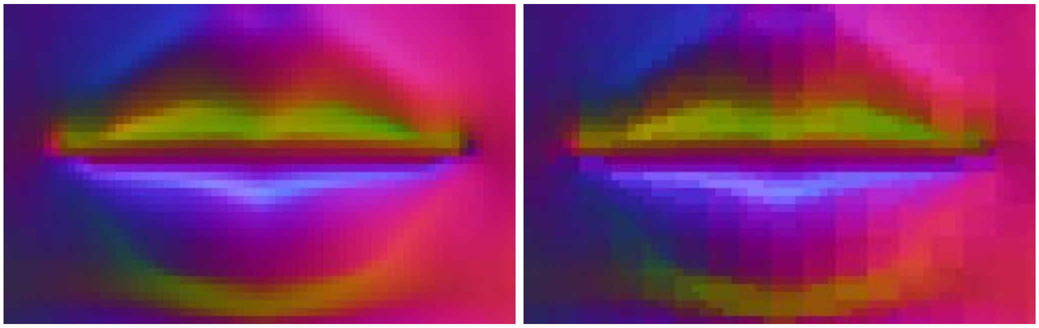
DXT1 [3, 4], also known as BC1 in DirectX 10 [5], is a lossy compression format for color textures, with a fixed compression ratio of 8:1. The DXT1 format is designed for real-time decompression in hardware on the graphics card during rendering. DXT1 compression is a form of Block Truncation Coding (BTC) [6] where an image is divided into non-overlapping blocks, and the pixels in each block are quantized to a limited number of values. The color values of pixels in a 4x4 pixel block are approximated with equidistant points on a line through RGB color space. This line is defined by two end-points, and for each pixel in the 4x4 block a 2-bit index is stored to one of the equidistant points on the line. The end-points of the line through color space are quantized to 16-bit 5:6:5 RGB format and either one or two intermediate points are generated through interpolation. The DXT1 format allows a 1-bit alpha channel to be encoded, by switching to a different mode based on the order of the end points, where only one intermediate point is generated and one additional color is specified, which is black and fully transparent.

Although the DXT1 format is designed for color textures this format can also be used to store normal maps. To compress a normal map to DXT1 format, the X, Y and Z components of the normal vectors are mapped to the RGB channels of a color texture. In particular for DXT1 compression each normal vector component is mapped from the range [-1, +1] to the integer range [0, 255]. The DXT1 format is decompressed in hardware during rasterization, and the integer range [0, 255] is mapped to the floating point range [0, 1] in hardware. In a fragment program the range [0, 1] will have to be mapped back to the range [-1, +1] to perform lighting calculations with the normal vectors. The following fragment program shows how this conversion can be implemented using a single instruction.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = normal.z ∈ [0, 1]
# input.w = 0

MAD normal, input, 2.0, -1.0
```

Compressing a normal map to DXT1 format generally results in rather poor quality. There are noticeable blocking and banding artifacts. Only four distinct normal vectors can be encoded per 4x4 block, which is typically not enough to accurately represent all original normal vectors in a block. Because the normals in each block are approximated with equidistance points on a line, it is also impossible to encode four distinct normal vectors per 4x4 block that are all unit-length. Only two normal vectors per 4x4 block can be close to unit-length at a time, and usually a compressor selects a line through vector space which minimizes some error metric, such that, none of the vectors are actually close to unit-length.



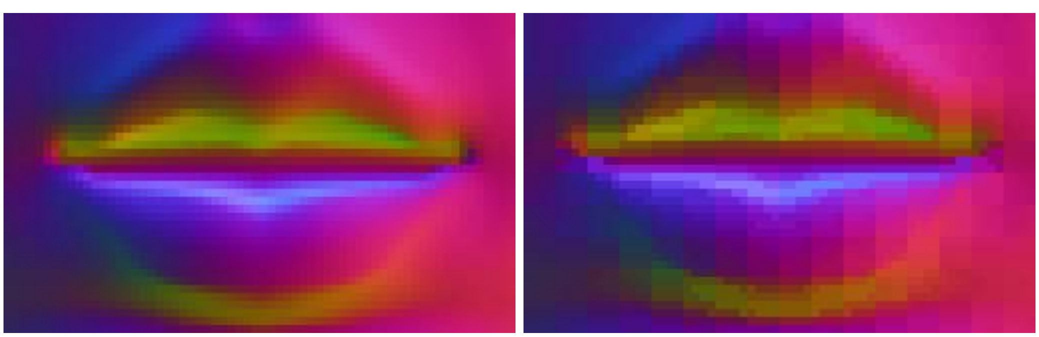
The DXT1 compressed normal map on the right shows noticeable blocking artifacts compared to the original normal map on the left.

To improve the quality, each normal vector can be encoded as a direction that is not necessarily unit-length. This direction then has to be re-normalized in a fragment program. The following fragment program shows how a normal vector can be re-normalized.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = normal.z ∈ [0, 1]
# input.w = 0

MAD normal, input, 2.0, -1.0
DP3 scale, normal, normal
RSQ scale.x, scale.x
MUL normal, normal, scale.x
```

Encoding directions gives the compressor more freedom, because the compressor does not have to worry about the magnitude of the vectors, and a much larger percentage of all representable vectors can be used for the end points of the line through normal space. However, this increased freedom makes compression a much harder problem.



The DXT1 compressed normal map with re-normalization on the right compared to the original normal map on the left.

The above images show that, although the quality is a little bit better, the quality is generally still rather poor. Whether re-normalizing in a fragment program or not, the quality of DXT1 compressed object-space normal maps is generally not considered to be acceptable.

2.2 Object-Space DXT5

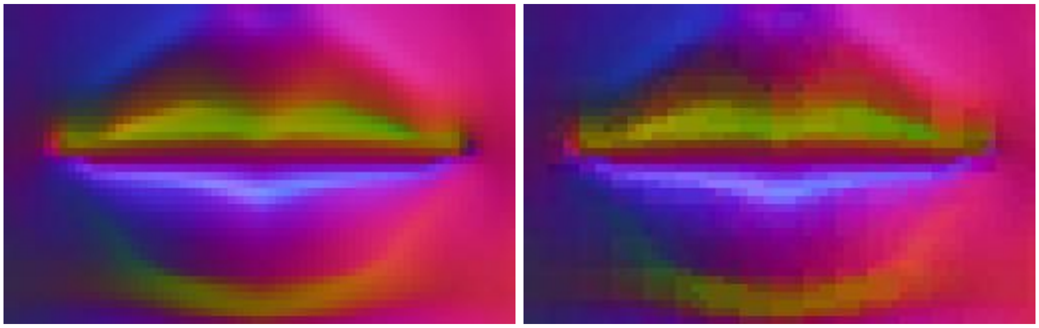
The DXT5 format [3, 4], also known as BC3 in DirectX 10 [5], stores three color channels the same way DXT1 does, but without 1-bit alpha channel. Instead of the 1-bit alpha channel, the DXT5 format stores a separate alpha channel which is compressed similarly to the DXT1 color channels. The alpha values in a 4x4 block are approximated with equidistant points on a line through alpha space. The end-points of the line through alpha space are stored as 8-bit values, and based on the order of the end-points either 4 or 6 intermediate points are generated through interpolation. For the case with 4 intermediate points, two additional points are generated, one for fully opaque and one for fully transparent. For each pixel in a 4x4 block a 3-bit index is stored to one of the equidistant points on the line through alpha space, or one of the two additional points for fully opaque or fully transparent. The same number of bits are used to encode the alpha channel as the three DXT1 color channels. As such, the alpha channel is stored with higher precision than each of the color channels, because the alpha space is one-dimensional, as opposed to the three-dimensional color space. Furthermore, there are a total of 8 samples to represent the alpha values in a 4x4 block, as opposed to 4 samples to represent the color values. Because of the additional alpha channel, the DXT5 format consumes twice the amount of memory of the DXT1 format.

The DXT5 format is designed for color textures with a smooth alpha channel. However, this format can also be used to store object-space normal maps. In particular, better quality normal map compression can be achieved by using the DXT5 format and moving one of the components to the alpha channel. By moving one of the components to the alpha channel this component is stored with more precision. Furthermore, by encoding only two components in the DXT1 block of the DXT5 format, the accuracy with which these components are stored typically improves as well. For object-space normal maps there is no clear benefit to moving any particular component to the alpha channel, because the normal vectors may point in any direction, and all values can occur with similar frequencies for all components. When an object-space normal map does have most vectors in a specific direction, then there is clearly a benefit to mapping the axis most orthogonal to that direction to the alpha channel. However, in general it is not practical to change the encoding on a per normal map basis, because a different fragment program is required for each encoding. The following fragment program assumes the Z component is moved to the alpha channel. The fragment program shows how the components are mapped from the range [0, 1] to the range [-1, +1], while the Z component is also moved back in place from the alpha channel.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = 0
# input.w = normal.z ∈ [0, 1]

MAD normal, input.xywz, 2.0, -1.0
```

Just like DXT1 without re-normalization, this format results in minimal overhead in a fragment programs. The quality is significantly better than DXT1 compression of object-space normal maps. However, there are still noticeable blocking and banding artifacts.



The DXT5 compressed normal map on the right
compared to the original normal map on the left.

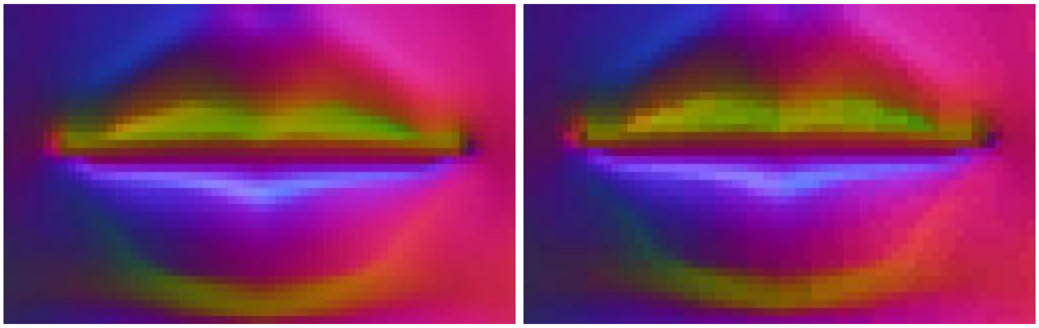
Using the third channel to store a scale factor like done for the YCoCg-DXT5 compression from [24] does not help much to improve the quality. The dynamic range of the individual components is typically too large, or the different components span different ranges that are far apart, while there is only one scale factor for the combined dynamic range.

Just like DXT1 compression of object-space normal maps, the quality can be improved by encoding a normal vector as a direction that is not necessarily unit-length. The following fragment program shows how to perform the swizzle and re-normalization.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = 0
# input.w = normal.z ∈ [0, 1]

MAD normal, input.xywz, 2.0, -1.0
DP3 scale, normal, normal
RSQ scale.x, scale.x
MUL normal, normal, scale.x
```

Encoding directions gives the compressor a lot more freedom, because the compressor can ignore the magnitude of the vectors, and a much larger percentage of all representable vectors can be used for the end points of the lines through normal space. The normal vectors are encoded using both the DXT1 block of the DXT5 format and the alpha channel, where the end points of the alpha channel are stored without quantization. As such, the potential search space for the end points of the lines can be very large, and high quality compression may take a considerable amount of time.



The DXT5 compressed normal map with re-normalization on
the right compared to the original normal map on the left.

On current hardware, the DXT5 format with re-normalization in a fragment program results in the best quality compression of object-space normal maps.

3. Tangent-Space Normal Maps

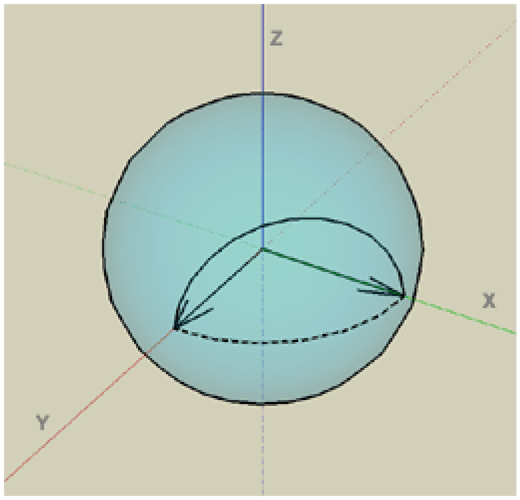
Tangent-space normal vectors are stored relative to the interpolated tangent-space of the triangle vertices. Compression of tangent-space normal maps generally works better than compression of object-space normal maps, because the dynamic range is lower. The vectors are only on the unit-hemisphere at the front of the surface (the normal vectors never point into the object). Furthermore, most normal vectors are close to the tip of the unit-hemisphere with Z close to 1.

Using tangent-space normal maps in itself can be considered a form of compression compared to using object-space normal maps. A local transform is used to change the frequency domain of the vector components which reduces their storage requirements. The transform does require tangent vectors to be stored at the triangle vertices and, as such, comes at a cost. However, the storage requirements for the tangent vectors is relatively very small compared to the storage requirements for normal maps.

The compression of tangent-space normal maps can be improved by only storing the X and Y components of unit-length normal vectors, and deriving the Z components. The normal vectors are always pointing up out of the surface and the Z is always positive. Furthermore, the normal vectors are unit-length and, as such, the Z can be derived as follows.

```
Z = sqrt( 1 - X * X - Y * Y )
```

The problem with reconstructing Z from X and Y is that it is a non-linear operation, and breaks down under bilinear filtering. The problem is most noticeable when interpolating between two normals in the XY-plane. Ideally a normal map is scaled up using spherical interpolation of the normal vectors, where the interpolated samples follow the shortest great arc on the unit sphere at a constant speed. Bilinear filtering of a three component normal map, with re-normalization in the fragment program, does not result in spherical interpolation at a constant speed, but at least the interpolated samples follow the shortest great arc. With a two-component normal map, however, where the Z is derived from the X and Y, the interpolated samples no longer necessarily follow the shortest great arc on the unit sphere. For instance, interpolation between the two vectors in the figure below is expected to follow the dotted line. Instead, however, the interpolated samples are on the arc that goes up on the unit sphere.



Fortunately, real-world normal maps usually do not have many sharp normal boundaries with adjacent vectors close to the XY-plane, and most of the normals point straight up. As such, there are usually no noticeable artifacts when bilinearly or trilinearly filtering a two component normal map before deriving the Z components.

Only storing the X and Y components is in essence an orthographic projection of the normal vectors along the Z-axis onto the XY-plane. To reconstruct an original normal vector, a projection back onto the unit-hemisphere is used, by deriving the Z component from the X and Y. Instead of this orthographic projection, a stereographic projection can be used as well. For the stereographic projection the X and Y components are divided by one plus Z as follows, where (pX, pY) is the projection of the normal vector.

```
pX = X / ( 1 + Z )
pY = Y / ( 1 + Z )
```

The original normal vector is reconstructed by projecting the stereographically projected vector back onto the unit-hemisphere as follows.

```
denom = 2 / ( 1 + pX * pX + pY * pY )
X = pX * denom
Y = pY * denom
Z = denom - 1
```

The advantage of using the stereographic projection is that the interpolated normal vectors behave better under bilinear or trilinear filtering. The interpolated normal vectors are still not on the shortest great arc, but they are closer, and have less of a tendency to go up on the unit-hemisphere.

The stereographic projection also causes a more even distribution of the pX and pY components with the angle on the unit-hemisphere. Although this may seem desirable, it is actually not, because most tangent-space normal vectors are close to the tip of the unit-hemisphere. As such, there is actually an advantage to using the orthographic projection which results in more representations of vectors with Z close to 1. The compression techniques discussed below use the orthographic projection because for most normal maps it results in better quality compression.

Instead of the orthographic and stereographic projections it is also an option to use a perspective projection where the X and Y components are divided by the Z component. Normal maps that are transformed this way are also known as partial derivative normal maps.

```
pX = X / Z
pY = Y / Z
```

The original normal vector is reconstructed by normalizing the vector (pX, pY, 1) which projects the vector back onto the unit-hemisphere. This is particularly interesting because on some graphics hardware normalizing a vector in a fragment program is very efficient.

```
denom = 1 / sqrt( 1 + pX * pX + pY * pY )
X = pX * denom
Y = pY * denom
Z = denom
```

Obviously the projection fails if the Z component is zero. As a matter of fact only normal vectors that are 45 degrees or less from pointing straight up (Z > sqrt(1/3)) can be reconstructed correctly. The angle of this cone can be made wider or tighter by multiplying the Z component with a value larger than one or less than one respectively before dividing the X and Y components. In particular, the scale factor is the tangent of the desired angle where: tan(45°) = 1. The reciprocal scale factor will have to be used for the reconstruction of the components. Although the cone can be made infinitely small it is not possible to flatten the cone to a plane such that all normals on the complete hemisphere can be properly reconstructed (tan(90°) = infinity).

Despite these drawbacks this projection can result is surprisingly good quality compression of normal maps if most or all normals are within a known cone centered about the up vector in tangent space. The compression techniques discussed below, however, use the orthographic projection because this allows for proper compression of normal maps with normals that cover the complete hemisphere.

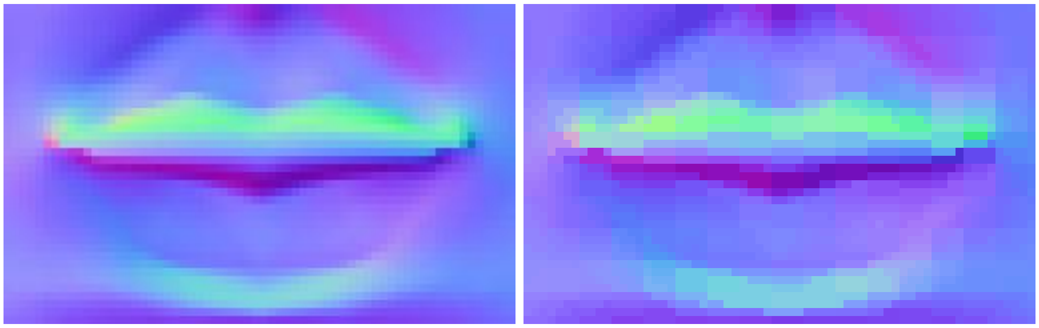
3.1 Tangent-Space DXT1

Using tangent-space normal maps only the X and Y components have to be stored in the DXT1 format, and the Z component can be derived in a fragment program. The following fragment program shows how the Z can be derived from the X and Y.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = 0
# input.w = 0

MAD normal, input, 2.0, -1.0
DP4_SAT normal.z, normal, normal;
MAD normal, normal, { 1, 1, -1, 0 }, { 0, 0, 1, 0 };
RSQ temp, normal.z;
MUL normal.z, temp;
```

The following images show a XY_ DXT1 compressed normal map on the right, next to the original normal map on the left. The DXT1 compressed normal map shows noticeable blocking and banding artifacts.



XY_ DXT1 compressed normal map on the right
compared to the original normal map on the left.

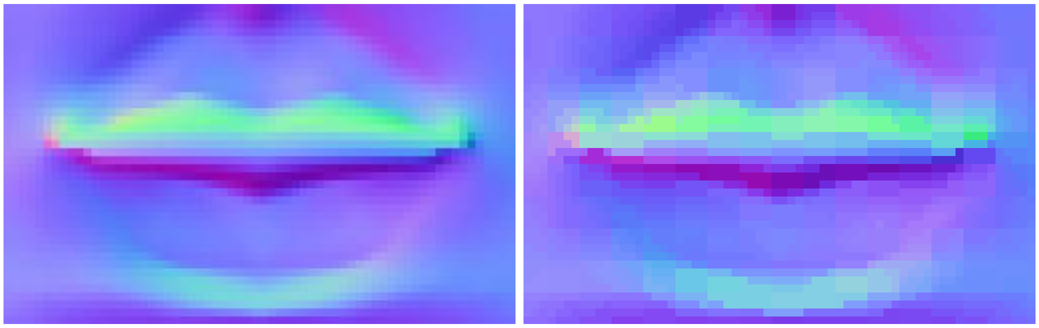
Although at first it may seem this kind of compression should produce superior quality, better quality compression can generally be achieved by storing all three components and re-normalizing in a fragment program, just like for object-space normal maps. When only the X and Y components are stored in the DXT1 format, the reconstructed normal vectors are automatically normalized by deriving the Z component. When the X and Y components are distorted due to the DXT1 compression, where all points are placed on a straight line through XY-space, the error in the derived Z can be quite large.

The fragment program shown below for re-normalizing the DXT1 compressed normals, is the same as the one used for DXT1 compressed object-space normal maps with re-normalization.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = normal.z ∈ [0, 1]
# input.w = 0

MAD normal, input, 2.0, -1.0
DP3 scale, normal, normal
RSQ scale.x, scale.x
MUL normal, normal, scale.x
```

The following images show a DXT1 compressed normal map with re-normalization on the right, next to the original normal map on the left.



DXT1 compressed normal map with re-normalization on the right compared to the original normal map on the left.

Either way, whether only storing two components in the DXT1 and deriving the Z, or storing all three components in the DXT1 format with re-normalization in the fragment program, the quality is rather poor.

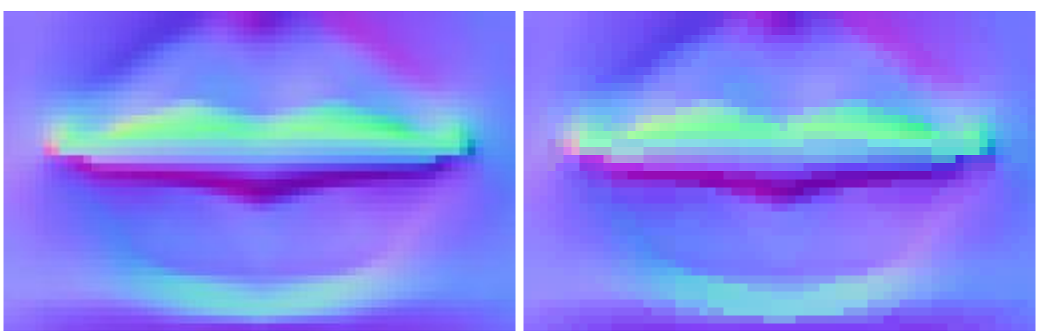
3.2 Tangent-Space DXT5

Just like for object-space normal maps, all three components can be stored in the DXT5 format. The best results are usually achieved when storing _YZX data. In other words the X component is moved to the alpha channel. This technique is also known as RxGB compression, and was employed in the computer game DOOM III. By moving the X component to the alpha channel, the X and Y components are encoded separately. This improves the quality because the X and Y components are most independent with the largest dynamic range. The Z is always positive and typically close to 1 and, as such, storing the Z component with the Y component in the DXT1 part of the DXT5 format causes little distortion of the Y component. Storing all three components results in minimal overhead in a fragment program as shown below.

```
# input.x = 0
# input.y = normal.y ∈ [0, 1]
# input.z = normal.z ∈ [0, 1]
# input.w = normal.x ∈ [0, 1]

MAD normal, input.wyzx, 2.0, -1.0
```

The following images show that, although the quality is better than DXT1 compression, there are still noticeable banding artifacts.



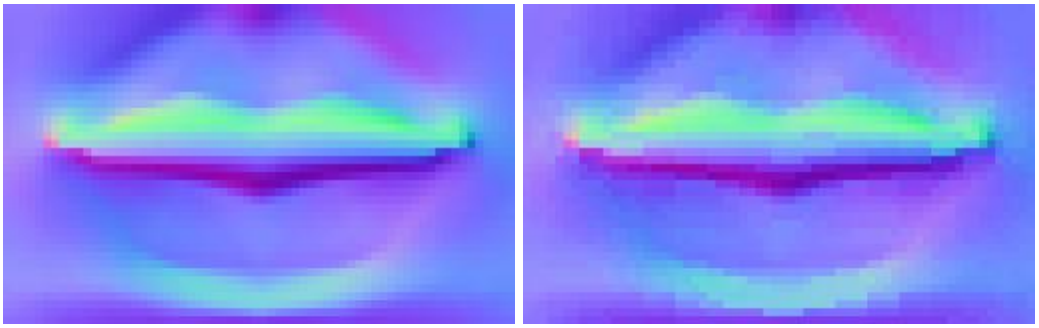
DXT5 compressed normal map on the right compared
to the original normal map on the left.

Just like for object-space normal maps the quality can be improved by storing directions that are not necessarily unit-length. The best quality is typically achieved by also moving the X component to the DXT5 alpha channel. The following fragment program shows how the directions are re-normalized after moving the X component back in place from the alpha channel.

```
# input.x = normal.x ∈ [0, 1]
# input.y = normal.y ∈ [0, 1]
# input.z = 0
# input.w = normal.z ∈ [0, 1]

MAD normal, input.wyzx, 2.0, -1.0
DP3 scale, normal, normal
RSQ scale.x, scale.x
MUL normal, normal, scale.x
```

The following images show that encoding directions with re-normalization in a fragment program reduces the banding artifacts, but they are still quite noticeable.



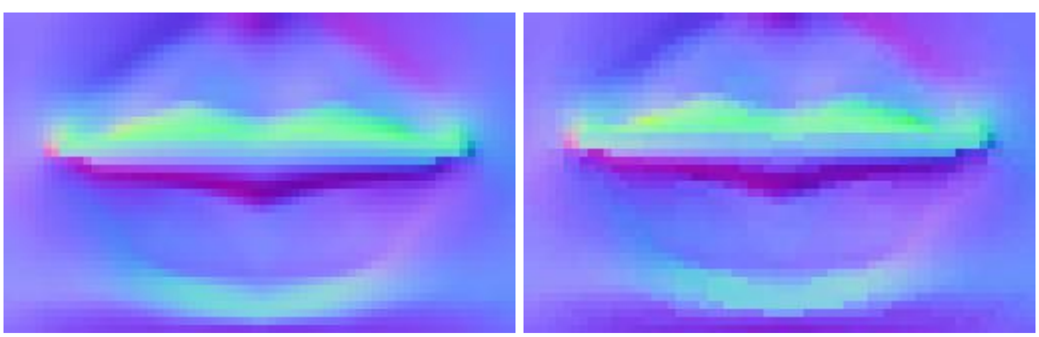
The DXT5 compressed normal map with re-normalization on the right compared to the original normal map on the left.

For most tangent-space normal maps better quality compression can be achieved by only storing the X and Y components in the DXT5 format and deriving the Z. This is also known as DXT5nm compression, and is most popular in today's computer games. The following fragment program shows how the Z is derived from the X and Y components.

```
# input.x = 0
# input.y = normal.y ∈ [0, 1]
# input.z = 0
# input.w = normal.x ∈ [0, 1]

MAD normal, input.wyzx, 2.0, -1.0
DP4_SAT normal.z, normal, normal;
MAD normal, normal, { 1, 1, -1, 0 }, { 0, 0, 1, 0 };
RSQ temp, normal.z;
MUL normal.z, temp;
```

The following images show that only storing the X and Y and deriving the Z, further reduces the banding artifacts.



DXT5 compressed normal map storing only X and Y on the right compared to the original normal map on the left.

When using XY_ DXT1, _YZX DXT5 or _Y_X DXT5 compression for tangent-space normal maps, there is at least one spare channel that can be used to store a scale factor, which can be used to counter quantization errors similar to what the YCoCg-DXT5 compressor from [24] does. However, trying to upscale the components to counter quantization errors does not improve the quality much (typically a PSNR improvement of less than 0.1 dB). The components can only be scaled up when they have a low dynamic range. Although most normals point straight up, and the magnitude of most X-Y vectors is relatively small, the dynamic range of the X-Y components is actually still quite large. Even if all normals never deviate more than 45 degrees from straight up, then each X or Y component may still map to the range [-cos(45°), +cos(45°)], where cos(45°) ≅ 0.707. In other words even with a deviation of less than 45 degrees from straight up, which is 50% of the angular range, each component may still cover more than 70% of the maximum dynamic range. On one hand, this is a good thing, because for the components of tangent-space normal vectors this means the largest part of the dynamic range covers the most frequently occurring values. On the other hand this means it is hard to upscale the components because of a relatively large dynamic range.

In the case of the _Y_X DXT5 compression of tangent-space normal maps there are two unused channels, and one of these channels can be used to also store a bias to center the dynamic range. This significantly increases the number of 4x4 blocks for which the values can be scaled up (such that typically more than 75% of all 4x4 blocks use a scale factor of at least 2). However, even using a bias to increase the number of scaled 4x4 blocks does not help much to improve the quality. The real problem is that the four sample points of the DXT1 block are simply not enough to accurately represent all the Y components of the normals in a 4x4 block. Introducing more sample points would significantly improve the quality but this is obviously not possible within the DXT5 format.

Instead of storing a bias and scale, one of the spare channels can also be used to store a rotation of the normal vectors in a 4x4 block about the Z-axis, as suggested in [11, 12]. Such a rotation can be used to find a much tighter bounding box of the X-Y vectors. In particular using _Y_X DXT5 compression such a rotation can be used to make sure that the axis with the largest dynamic range maps to the alpha channel, which, as such, is compressed with more precision. To be able to map the axis with the largest dynamic range to the alpha channel, a rotation of up to 180 degrees may be required. This rotation can be stored as a constant value over the whole 4x4 block in one of the 5-bit channels. Instead of storing the angle of rotation, the cosine of the angle can be stored, such that the cosine does not have to be calculated in a fragment program where the vectors need to be rotated back to their original positions. The sine for a rotation in the range [0, 180] degrees is always positive and can, as such, trivially be derived from the cosine in a fragment program as follows.

```
sine = sqrt( 1 - cosine * cosine )
```

The PSNR improvement from rotating the normals in a 4x4 block is significant and typically in the range 2 to 3 dB. Unfortunately adjacent 4x4 blocks may need vastly different rotations, and under bilinear or trilinear filtering noticeable artifacts may appear for filtered texel samples at borders between two 4x4 blocks with different rotations. The X, Y and rotation are filtered separately before the rotation is applied to the X and Y components. As such, a filtered rotation is applied to filtered X and Y components, which is not the same as filtering X and Y components that are first rotated back to their original position. In other words, unless the normal map is only point sampled, using a rotation is also not an option to improve the quality of DXT1 or DXT5 normal map compression.

Of course a denormalization value can still be stored in one of the spare channels as described in [8]. The denormalization value is used to scale down the normal vectors for lower mip levels, such that specular highlights fade with distance to alleviate aliasing artifacts.

3.3 Tangent-Space 3Dc

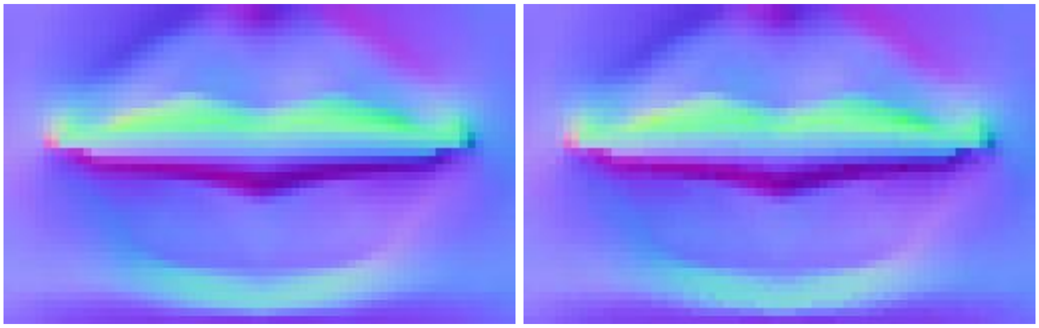
The 3Dc format [10] is specifically designed for tangent-space normal map compression and produces much better quality than DXT1 or DXT5 normal map compression. The 3Dc format stores only two channels and, as such, cannot be used for object-space normal maps. The format basically consists of two DXT5 alpha blocks for each 4x4 block of normals. In other words for each 4x4 block there are 8 samples for the X components and also 8 independent samples for the Y components. The Z components have to be derived in a fragment program.

The 3Dc format is also known as BC5 in DirectX 10 [5]. The same format can be loaded in OpenGL as LATC or RGTC. Using the LATC format the luminance is replicated in all three RGB channels. This can be particularly convenient, because this way the same swizzle (and fragment program code) can be used for both LATC and _Y_X DXT5 (DXT5nm) compressed normal maps. In other words the same fragment program can be used on hardware that does, and does not support 3Dc. The following fragment program shows how the Z is derived from the X and Y components when the normal map is stored in RGTC format.

```
# normal.x = x ∈ [0, 1]
# normal.y = y ∈ [0, 1]
# normal.z = 0
# normal.w = 0

MAD normal, normal, 2.0, -1.0
DP4 normal.z, normal, normal;
MAD normal, normal, { 1.0, 1.0, -1, 0 }, { 0, 0, 1, 0 };
RSQ temp, normal.z;
MUL normal.z, temp;
```

The following images show how 3Dc compression of normal maps, results in significantly less banding compared to _Y_X DXT5 (DXT5nm).



3Dc compressed normal map on the right compared to the original normal map on the left.

Several extensions to 3Dc are proposed in [11] and a new format specifically designed for improved normal map compression is presented in [12]. However, these formats are not available in current graphics hardware. On all DirectX 10 compatible hardware the 3Dc (or BC5) format results in the best quality tangent-space normal map compression. On older hardware which does not implement 3Dc the best quality is generally achieved using _Y_X DXT5 (DXT5nm).

4. Real-Time Compression on the CPU

While decompression from the formats described in the previous sections is done real-time in hardware, compression to these formats may take a considerable amount of time. Existing compressors are designed for high-quality off-line compression, not real-time compression [20, 21, 22]. However, real-time compression is quite useful to compress normal maps that are stored on disk in a different (more space efficient) format, and to compress dynamically generated normal maps.

In today's rendering engines, tangent-space normal maps are far more popular than object-space normal maps. On current hardware there are no compression formats available for object-space normal maps that work really well. The object-space normal map compression techniques described in section 2 all result in noticeable artifacts, or the compression is exceedingly expensive.

An object-space normal map can also not be used on an animated object. While the object surface animates the object-space normal vectors stay pointing in the same object-space direction. Tangent-space normal maps on the other hand, store normals relative to the tangent-space at the triangle vertices. When the surface of an object animates and the tangent vectors (stored at the triangle vertices) are transformed with the surface, the tangent-space normal vectors that are stored relative to these tangent vectors will also animate with the surface. As such the focus here is on real-time compression of tangent-space normal maps.

On hardware where the 3Dc (BC5 or LATC) format is not available, the _Y_X DXT5 (DXT5nm) format generally results in the best quality tangent-space normal map compression. The real-time _Y_X DXT5 compressor is very similar to the real-time DXT5 compressor from [23].

First the bounding box of X-Y normal space is calculated. The two lines that are used to approximate the X and Y-values go from the minimums to the maximums of this bounding box. To improve the Mean Square Error (MSE), the bounding box is inset on either end with a quarter the distance between the sample points on the lines. The Y components are stored in the "green" channel and there are 4 sample points on the line through "color" space. As such, the minimum and maximum Y values are inset with 1/16th of the range. The X components are stored in the "alpha" channel and there are 8 sample points on the line through "alpha" space. As such, the minimum and maximum X values are inset with 1/32nd of the range. The inset is implemented such that the minimum and maximum values are rounded outwards just like the YCoCg-DXT5 compressor from [24] does.

Only a single channel of the "color" channels is used to store the Y components of the normal vectors. Using this knowledge, the real-time DXT5 compressor from [23] can be optimized further specifically for _Y_X DXT5 compression. The best matching points on the line through Y-space can be found in a similar way the best matching points on the line through "alpha" space are found in the DXT5 compressor from [23]. First a set of cross-over points are calculated where a Y value goes from being closest to one sample point to another.

```
byte mid = ( max - min ) / ( 2 * 3 );

byte gb1 = max - mid;
byte gb2 = ( 2 * max + 1 * min ) / 3 - mid;
byte gb3 = ( 1 * max + 2 * min ) / 3 - mid;
```

A Y value can then be tested for being greater-equal to each of the cross-over points, and the results of these comparisons (0 for false and 1 for true) can be added together to calculate an index. This results in the following order where index 0 through 3 go from the minimum to the maximum.

index:	0	1	2	3
value:	min	(max + 2 * min) / 3	(2 * max + min) / 3	max

However, the "color" sample points are ordered differently in the DXT5 format as follows.

index:	0	1	2	3
value:	max	min	(2 * max + min) / 3	(max + 2 * min) / 3

Subtracting the results of the comparisons from four, and wrapping the result with a bitwise logical AND with 3, results in the following order.

index:	0	1	2	3
value:	min	max	(2 * max + min) / 3	(max + 2 * min) / 3

The order is close to correct, but the min and max are still swapped. The following code shows how the Y values are compared to the cross-over points, and how the indices are calculated from the results of the comparisons, where index 0 and 1 are swapped at the end by XOR-ing with the result of the comparison (2 > index).

```
unsigned int result = 0;
for ( int i = 15; i >= 0; i-- ) {
    result <<= 2;
    byte g = block[i*4];
    int b1 = ( g >= gb1 );
    int b2 = ( g >= gb2 );
    int b3 = ( g >= gb3 );
    int index = ( 4 - b1 - b2 - b3 ) & 3;
    index ^= ( 2 > index );
    result |= index;
}
```

Using SIMD instructions each byte comparison results in a byte with either all zero bits (when the expression is false), or all one bits (when the expression is true). When interpreted as a signed (two's-complements) integer, the result of a byte comparison is equal to either the number 0 (for false) or the number -1 (for true). Instead of explicitly subtracting a 1 for a comparison that results in true, the actual result of the comparison can simply be added to the value four as a signed integer.

The calculation of the indices for the "alpha" channel is very similar to the calculation used in the real-time DXT5 compressor from [23]. However, the calculation can be optimized further by also selecting the best matching sample points with subtraction as opposed to addition. First a set of cross-over points are calculated where an X value goes from being closest to one sample point to another.

```
byte mid = ( max - min ) / ( 2 * 7 );

byte ab1 = max - mid;
byte ab2 = ( 6 * max + 1 * min ) / 7 - mid;
byte ab3 = ( 5 * max + 2 * min ) / 7 - mid;
byte ab4 = ( 4 * max + 3 * min ) / 7 - mid;
byte ab5 = ( 3 * max + 4 * min ) / 7 - mid;
byte ab6 = ( 2 * max + 5 * min ) / 7 - mid;
byte ab7 = ( 1 * max + 6 * min ) / 7 - mid;
```

An X value can then be tested for being greater-equal to each of the cross-over points, and the results of these comparisons (0 for false and 1 for true) can be subtracted from 8 and wrapped using a bitwise logical AND with 7 to calculate the index. The first two indices are also swapped by xoring with the result of the comparison (2 > index) as shown in the following code.

```
byte indices[16];
for ( int i = 0; i < 16; i++ ) {
    byte a = block[i*4];
    int b1 = ( a >= ab1 );
    int b2 = ( a >= ab2 );
    int b3 = ( a >= ab3 );
    int b4 = ( a >= ab4 );
    int b5 = ( a >= ab5 );
    int b6 = ( a >= ab6 );
    int b7 = ( a >= ab7 );
    int index = ( 8 - b1 - b2 - b3 - b4 - b5 - b6 - b7 ) & 7;
    indices[i] = index ^ ( 2 > index );
}
```

The full implementation of the real-time _Y_X DXT5 compressor can be found in appendix A. MMX and SSE2 implementations of this real-time compressor can be found in appendix B and C respectively.

Where available, the 3Dc (BC5 or LATC) format results in the best quality tangent-space normal map compression. The real-time 3Dc compressor first calculates the bounding box of X-Y normal space just like the _Y_X DXT5 compressor does. The two lines that are used to approximate the X and Y-values go from the minimums to the maximums of this bounding box. To improve the Mean Square Error (MSE), the bounding box is inset on either end with a quarter the distance between the sample points on the lines. The 3Dc format basically stores two DXT5 alpha channels both with the same encoding and 8 sample points. As such, on both axes the bounding box is inset on either end with 1/32th of the range. The same code as used for the _Y_X DXT5 compression, is used here as well to calculate the "alpha" channel indices, except that it is used twice. The full implementation of the real-time 3Dc compressor can be found in appendix A. MMX and SSE2 implementations of this real-time compressor can be found in appendix B and C respectively.

5. Real-Time Compression on the GPU

Real-time compression of tangent-space normal maps can also be performed on the GPU. This is possible thanks to new features available on DX10-class graphics hardware that enable rendering to integer textures and the use of bitwise and arithmetic integer operations..

To compress a normal map, a fragment program is used for each block of 4x4 texels by rendering a quad over the entire destination surface. The result of this fragment program is a compressed DXT block that is written to the texels of an integer texture. Both, DXT5 and 3Dc blocks are 128 bits, which is equal to one RGBA texel with 32 bits per component. As such, an unsigned integer RGBA texture is used as the render target when compressing a normal map to either format. The contents of this render target are then copied to the corresponding DXT texture by using Pixel Buffer Objects. This process is very similar to the one used for YCoCg-DXT5 compression that is described in more detail in [24].

3Dc compressed textures are exposed in OpenGL through two different extensions: GL_EXT_texture_compression_latc [25], and GL_EXT_texture_compression_rgtc [26]. The former maps the X and Y components to the luminance and alpha channels, while the latter maps the X and Y components to red and green respectively, where the remaining channels are set to 0.

In the implementation described here the LATC format is used. This is slightly more convenient, because it allows sharing the same shader code used for the normal reconstruction:

```
N.xy = 2 * tex2D(image, texcoord).wy - 1;
N.z = sqrt(saturate(1 - N.x * N.x - N.y * N.y));
```

When using LATC the luminance is replicated in the RGB channels, so the W-Y swizzle maps the luminance and alpha components to X and Y. Similarly, when using _Y_X DXT5, the W-Y swizzle maps the green and alpha components to X and Y.

The same code as used in [24] to encode the alpha channel for YCoCg-DXT5 compression, can also be used to encode the X and Y components for 3Dc compression, and the X component for _Y_X DXT5 compression. As shown in Section 4, the _Y_X DXT5 compressor can also be optimized to compute the DXT1 block by fitting only the Y component. However, as noted in [23], the alpha space is a one-dimensional space and the points on the line through alpha space are equidistant, which allows the closest point for each original alpha value to be calculated through division. On the CPU this requires a rather slow scalar integer division, because there are no MMX or SSE2 instructions available for integer division. The division can be implemented as an integer multiplication with a shift. However, the divisor is not a constant which means a lookup table is required to get the multiplier. Multiplication also increases the dynamic range which limits the amount of parallelism that can be exploited through a SIMD instruction set. On the CPU there is a clear benefit to exploiting maximum parallelism by using simple operations on the smallest possible elements (bytes) without increasing the dynamic range. However, on the GPU, scalar floating point math is used, and a division and/or multiplication is relatively cheap. As such, the X and Y components can be mapped to the respective indices by applying only a scale and a bias. The CG code for the index calculation of the Y component for the _Y_X DXT5 format is as follows:

```
const int GREEN_RANGE = 3;

float bias = maxGreen + (maxGreen - minGreen) / (2.0 * GREEN_RANGE);
float scale = 1.0f / (maxGreen - minGreen);

// Compute indices
uint indices = 0;
for (int i = 0; i < 16; i++)
{
    uint index = saturate((bias - block[i].y) * scale) * GREEN_RANGE;
    indices |= index << (i * 2);
}

uint i0 = (indices & 0x55555555);
uint i1 = (indices & 0xAAAAAAAA) >> 1;
indices = ((i0 ^ i1) << 1) | i1;
```

The same can be done for the X component of the _Y_X DXT5 format, and for both the X and Y component of the 3Dc format:

```
const int ALPHA_RANGE = 7;

float bias = maxAlpha + (maxAlpha - minAlpha) / (2.0 * ALPHA_RANGE);
float scale = 1.0f / (maxAlpha - minAlpha);

uint2 indices = 0;

for (int i = 0; i < 6; i++)
{
    uint index = saturate((bias - block[i].x) * scale) * ALPHA_RANGE;
    indices.x |= index << (3 * i);
}

for (int i = 6; i < 16; i++)
{
    uint index = saturate((bias - block[i].x) * scale) * ALPHA_RANGE;
    indices.y |= index << (3 * i - 18);
}

uint2 i0 = (indices >> 0) & 0x09249249;
uint2 i1 = (indices >> 1) & 0x09249249;
uint2 i2 = (indices >> 2) & 0x09249249;

i2 ^= i0 & i1;
i1 ^= i0;
i0 ^= (i1 | i2);

indices.x = (i2.x << 2) | (i1.x << 1) | i0.x;
indices.y = (((i2.y << 2) | (i1.y << 1) | i0.y) << 2) | (indices.x >>
16);
indices.x <<= 16;
```

The full Cg 2.0 implementations of the real-time _Y_X DXT5 (DXT5nm) normal map compressor, and the real-time 3Dc (BC5 or LATC) normal map compressor, can be found in appendix D.

6. Compression on the CPU vs. GPU

As shown in the previous sections high performance normal map compression can be implemented on both the CPU and the GPU. Whether the compression is best implemented on the CPU or the GPU is application dependent.

Real-time compression on the CPU is useful for normal maps that are dynamically created on the CPU. Compression on the CPU is also particularly useful for transcoding normal maps that are streamed from disk in a format that cannot be used for rendering. For example, a normal map or a height map may be stored in JPEG format on disk and, as such, cannot be used directly for rendering. Only some parts of the JPEG decompression algorithm can currently be implemented efficiently on the GPU. Memory can be saved on the graphics card, and rendering performance can be improved, by decompressing the original data and re-compressing it to DXT format. The advantage of re-compressing the texture data on the CPU is that the amount of data uploaded to the graphics card is minimal. Furthermore, when the compression is performed on the CPU, the full GPU can be used for rendering work as it does not need to perform any compression. With a definite trend to a growing number of cores on today's CPUs, there are typically free cores laying around that can easily be used for texture compression.

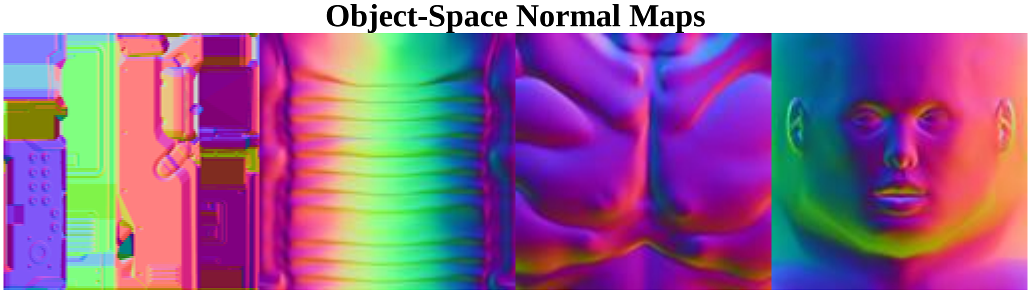
Real-time compression on the GPU may be less useful for transcoding, because of increased bandwidth requirements for uploading uncompressed texture data and because the GPU may already be tasked with expensive rendering work. However, real-time compression on the GPU is very useful for compressed render targets. The compression on the GPU can be used to save memory when rendering to a texture. Furthermore, such compressed render targets can improve the performance if the data from the render target is used for further rendering. The render target is compressed once, while the resulting data may be accessed many times during rendering. The compressed data results in reduced bandwidth requirements during rasterization and can, as such, significantly improve performance.

7. Results

7.1 Object-Space

The object-space normal map compression techniques have been tested with the object-space normal maps shown below.

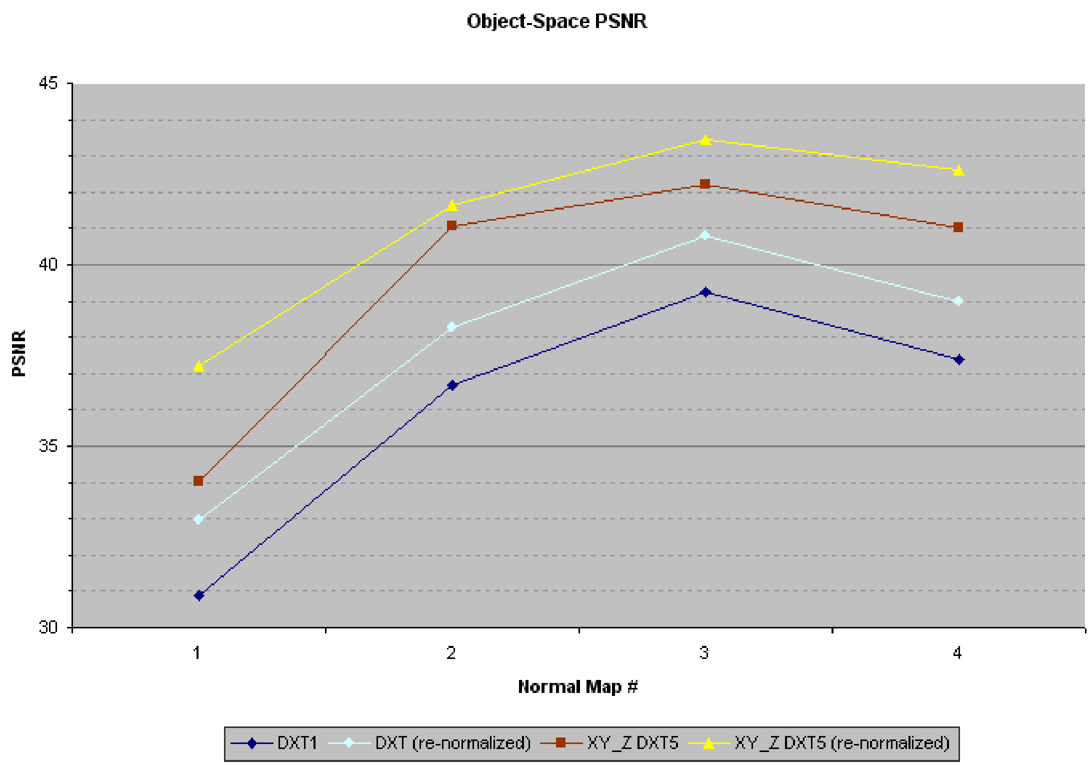
Object-Space Normal Maps



1. arcade | 2. tentacle | 3. chest | 4. face

The Peak Signal to Noise Ratio (PSNR) has been calculated over the unweighted X, Y and Z values, stored as 8-bit unsigned integers.

PSNR				
image	**XYZ DXT1**	**re-normalized XYZ DXT1**	**XY_Z DXT5**	**re-normalized XY_Z DXT5**
01_arcade	30.90	32.95	34.02	37.23
02_tentacle	36.68	38.29	41.04	41.62
03_chest	39.24	40.79	42.22	43.47
04_face	37.38	38.99	41.03	42.60

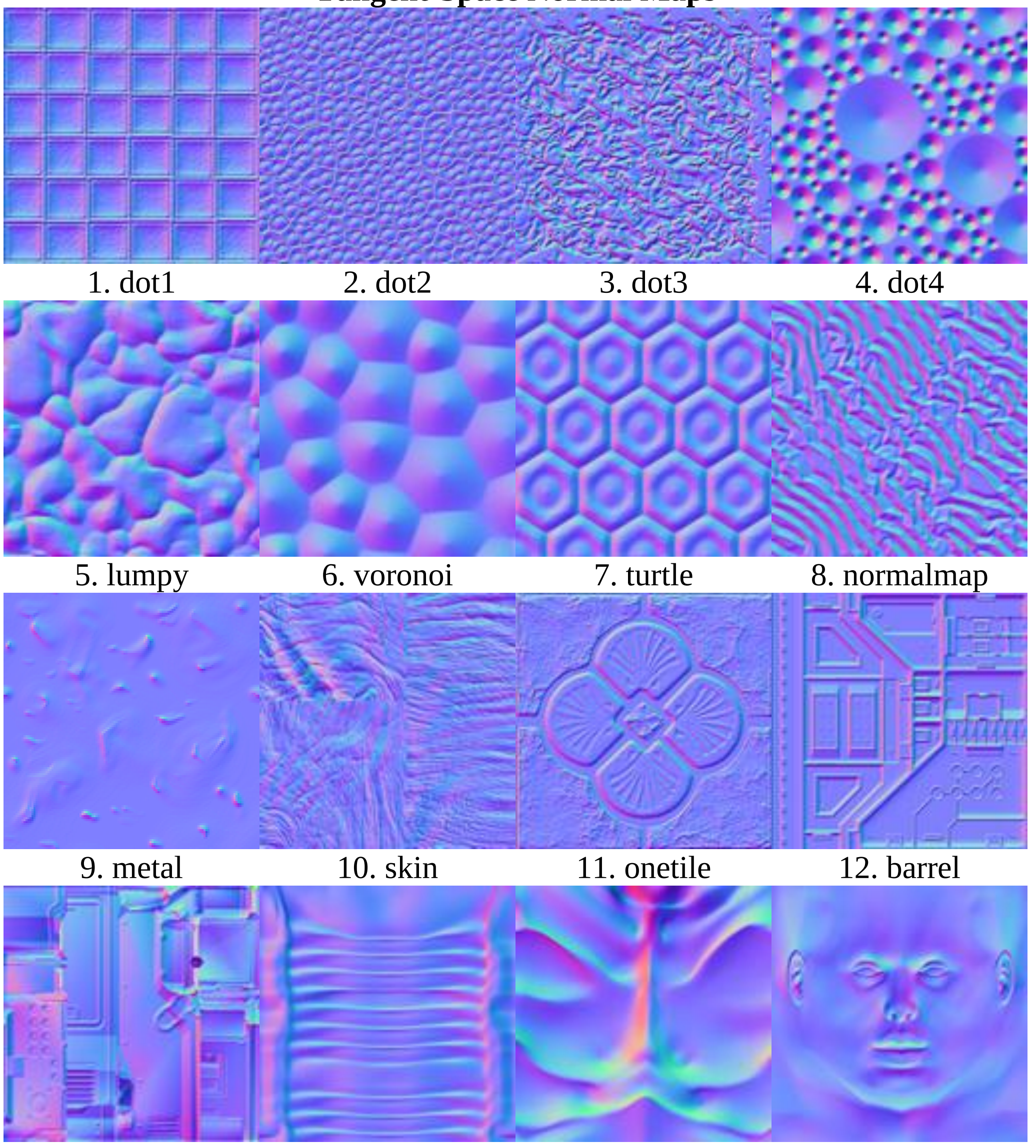


Object-Space PSNR
45
40
35
30
PSNR
1
2
3
4
Normal Map #
DXT1
DXT (re-normalized)
XY_Z DXT5
XY_Z DXT5 (re-normalized)

7.2 Tangent-Space

The tangent-space normal map compression techniques have been tested with the tangent-space normal maps shown below.

Tangent-Space Normal Maps



1. dot1 2. dot2 3. dot3 4. dot4

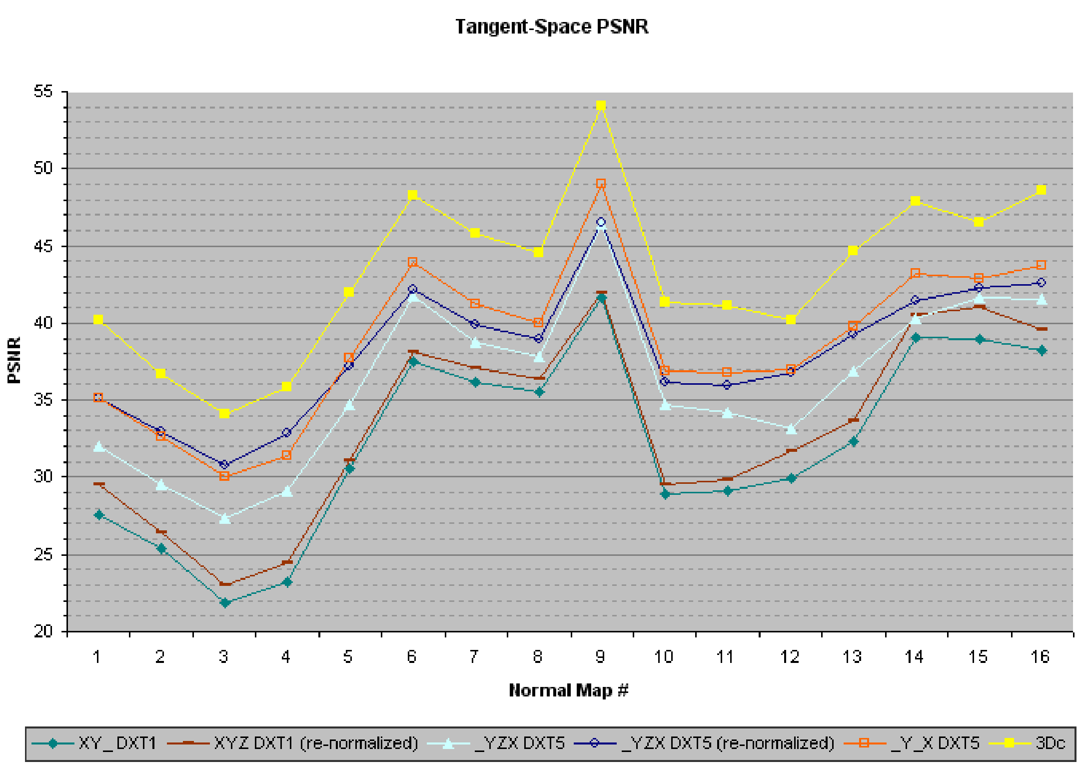
5. lumpy 6. voronoi 7. turtle 8. normalmap

9. metal 10. skin 11. onetile 12. barrel

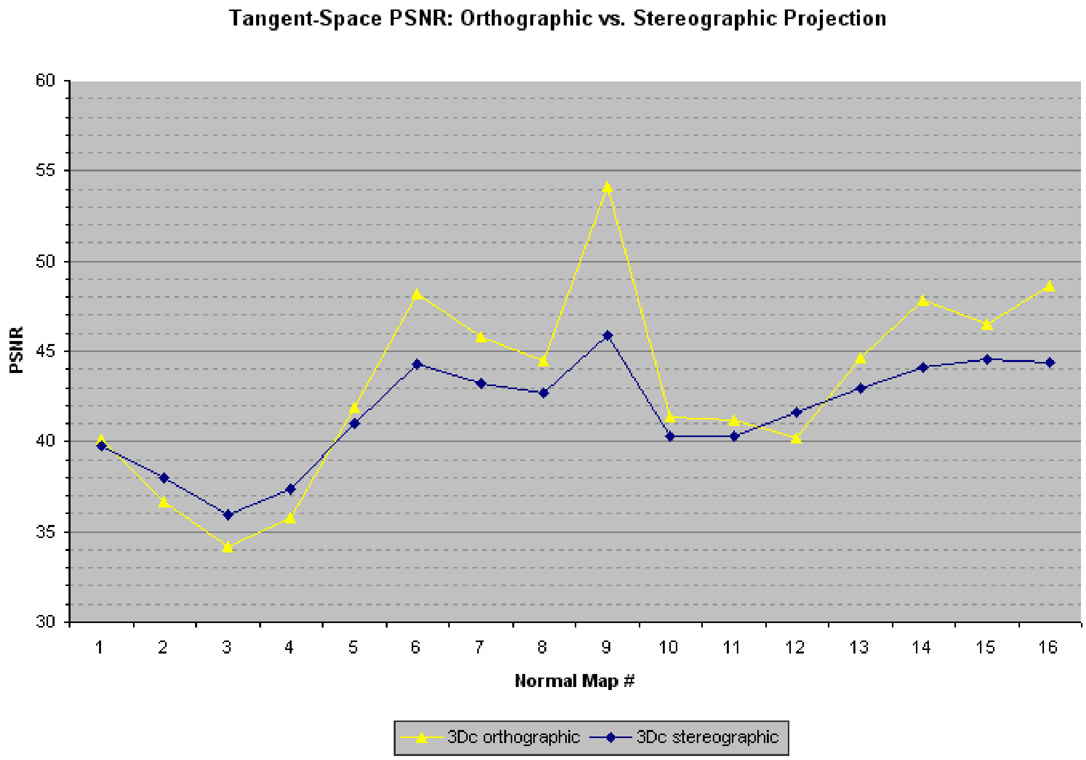
13. arcade 14. tentacle 15. chest 16. face

The Peak Signal to Noise Ratio (PSNR) has been calculated over the unweighted X, Y and Z values, stored as 8-bit unsigned integers.

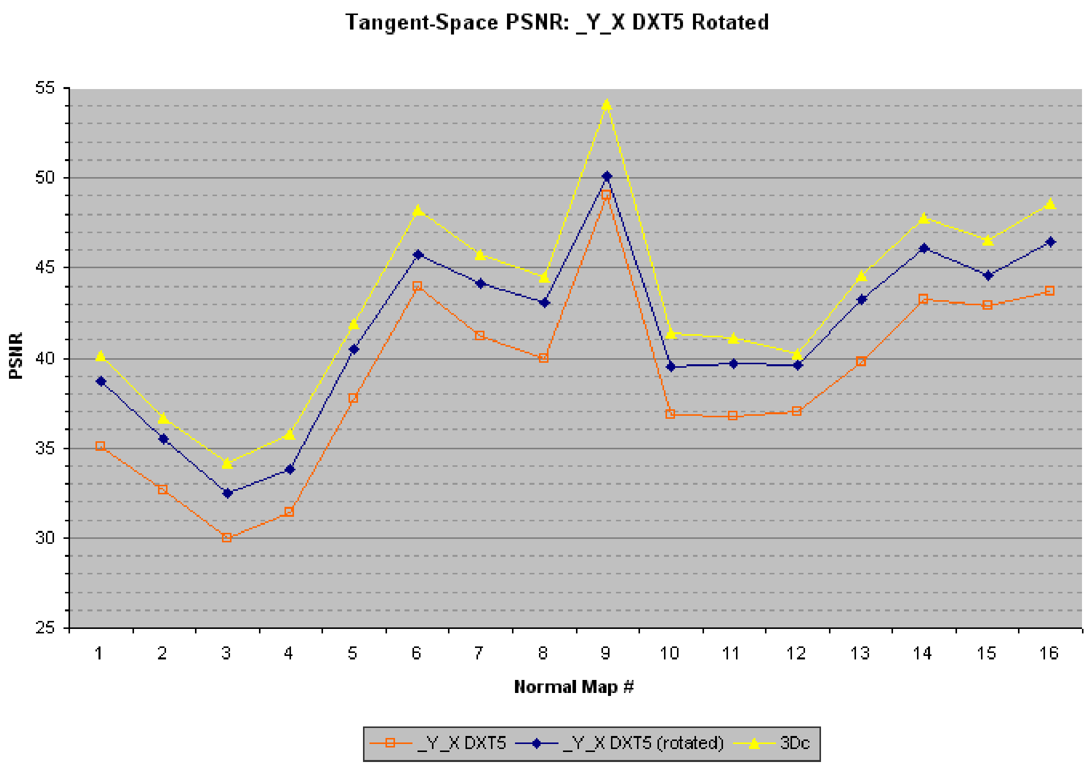
PSNR						
image	**XY_ DXT1**	**re-normalized XYZ DXT1**	**_YZX DXT5**	**re-normalized _YZX DXT5**	**_Y_X DXT5**	**3Dc**
01_dot1	27.61	29.51	32.00	35.16	35.07	40.15
02_dot2	25.39	26.45	29.55	32.92	32.68	36.70
03_dot3	21.88	23.05	27.34	30.77	30.02	34.13
04_dot4	23.18	24.46	29.16	32.81	31.38	35.80
05_lumpy	30.54	31.13	34.70	37.15	37.73	41.92
06_voronoi	37.53	38.16	41.72	42.16	43.93	48.23
07_turtle	36.12	37.06	38.74	39.93	41.22	45.76
08_normalmap	35.57	36.36	37.78	38.95	40.00	44.49
09_metal	41.65	41.99	46.37	46.55	49.03	54.10
10_skin	28.95	29.48	34.68	36.20	36.83	41.37
11_onetile	29.08	29.82	34.17	35.98	36.76	41.14
12_barrel	29.93	31.67	33.15	36.79	37.03	40.20
13_arcade	32.31	33.63	36.86	39.24	39.81	44.61
14_tentacle	39.03	40.47	40.30	41.39	43.23	47.82
15_chest	38.92	41.03	41.64	42.29	42.87	46.52
16_face	38.27	39.58	41.59	42.55	43.71	48.61



The following graph uses the 3Dc format to show the quality difference between the orthographic and stereographic projections. The stereographic projection results in more consistent results but for most normal maps the quality is significantly lower.



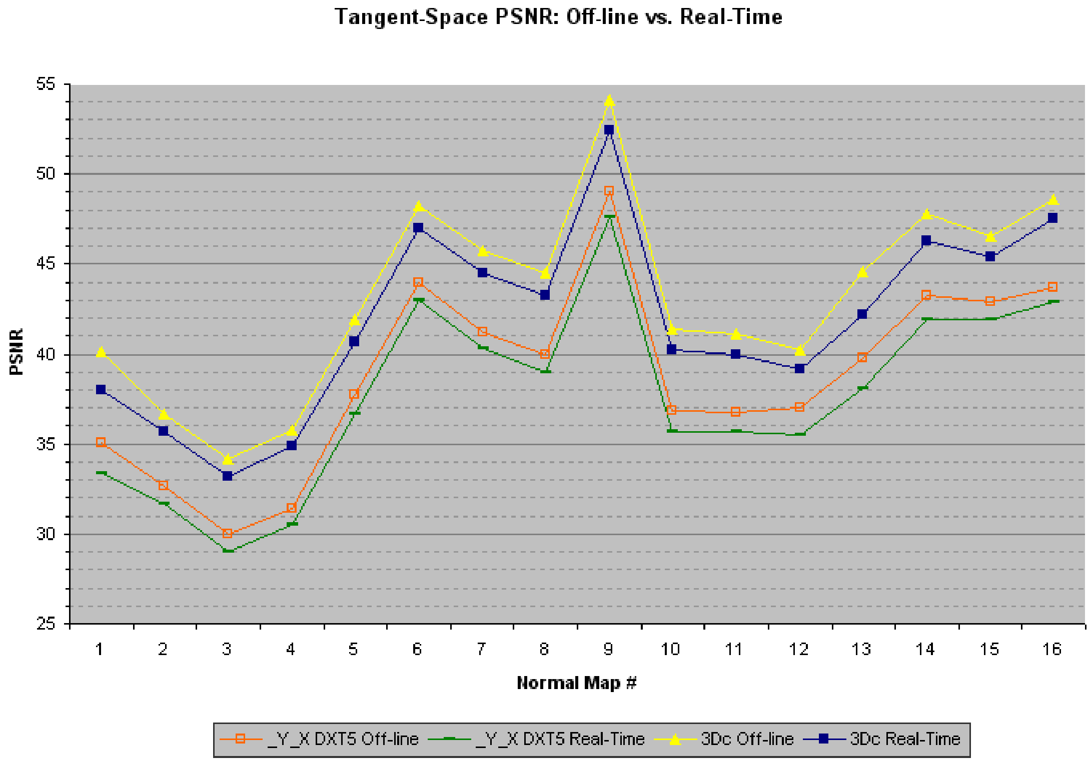
The following graph is only of theoretical interest, in that it shows the quality improvement from rotating the normals in a 4x4 block, and storing the rotation in one of the unused channels in the _Y_X DXT5 format. The graph shows the quality improvement for normal maps that are only point sampled, because filtering causes noticeable artifacts for texel samples between 4x4 blocks with different rotations.


Tangent-Space PSNR: _Y_X DXT5 Rotated
PSNR
55
50
45
40
35
30
25
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Normal Map #
_Y_X DXT5
_Y_X DXT5 (rotated)
3Dc

7.3 Real-Time Tangent-Space

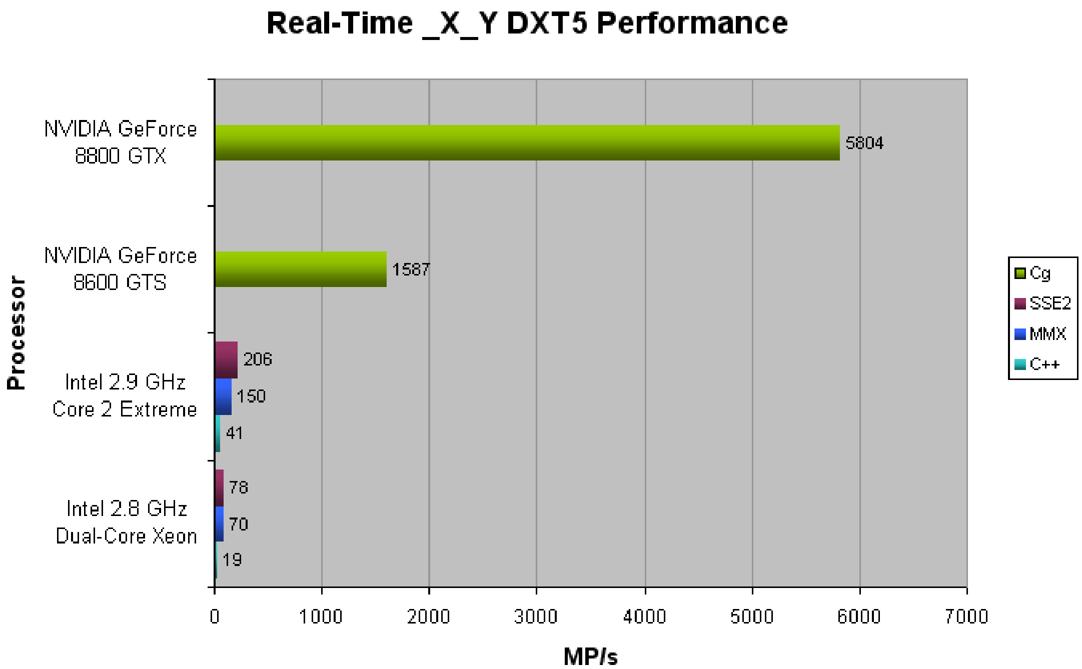
The real-time tangent-space normal map compressors have been tested with the same tangent-space normal maps shown above. The Peak Signal to Noise Ratio (PSNR) has been calculated over the unweighted X, Y and Z values, stored as 8-bit unsigned integers.

PSNR				
image	**off-line _Y_X DXT5**	**real-time _Y_X DXT5**	**off-line 3Dc**	**real-time 3Dc**
01_dot1	35.07	33.36	40.15	37.99
02_dot2	32.68	31.67	36.70	35.67
03_dot3	30.02	29.03	34.13	33.22
04_dot4	31.38	30.49	35.80	34.89
05_lumpy	37.73	36.63	41.92	40.63
06_voronoi	43.93	42.99	48.23	46.99
07_turtle	41.22	40.30	45.76	44.50
08_normalmap	40.00	38.99	44.49	43.26
09_metal	49.03	47.60	54.10	52.45
10_skin	36.83	35.69	41.37	40.20
11_onetile	36.76	35.67	41.14	39.92
12_barrel	37.03	35.51	40.20	39.11
13_arcade	39.81	38.05	44.61	42.18
14_tentacle	43.23	41.90	47.82	46.31
15_chest	42.87	41.95	46.52	45.38
16_face	43.71	42.85	48.61	47.53

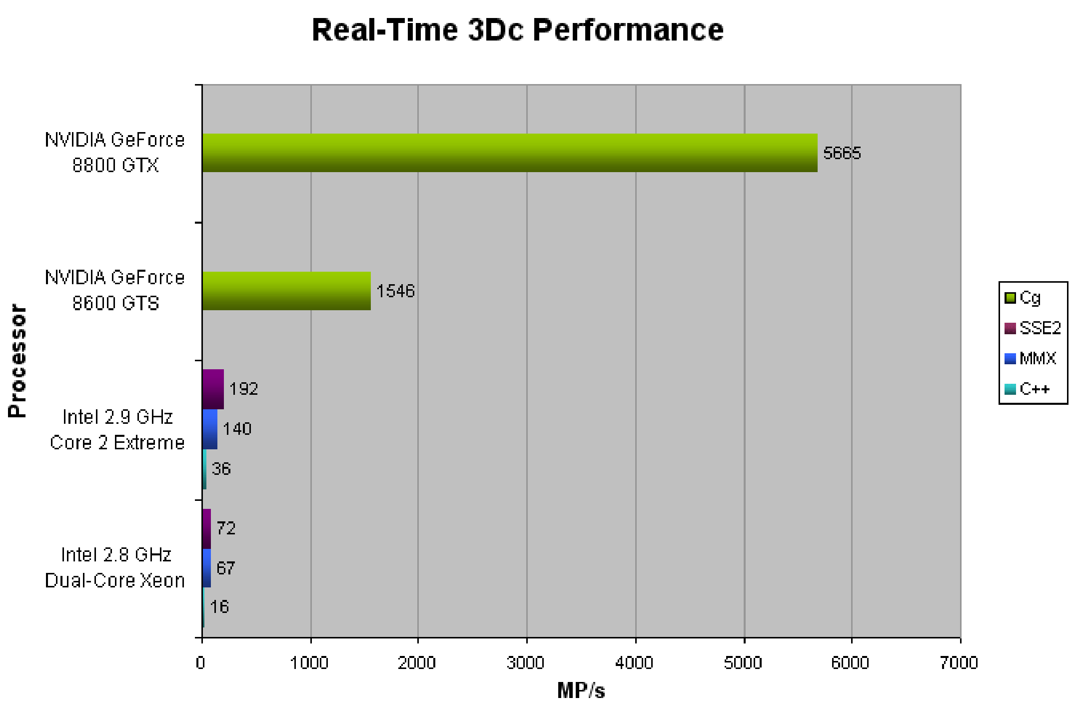


The performance of the SIMD optimized real-time compressors has been tested on an Intel® 2.8 GHz dual-core Xeon® ("Paxville" 90nm NetBurst microarchitecture) and an Intel® 2.9 GHz Core™2 Extreme ("Conroe" 65nm Core 2 microarchitecture). Only a single core of these processors was used for the compression. Since the texture compression is block based, the compression algorithms can easily use multiple threads to utilize all cores of these processors. When using multiple cores there is an expected linear speed up with the number of available cores. The performance of the Cg 2.0 implementations has also been tested on a NVIDIA GeForce 8600 GTS and a NVIDIA GeForce 8800 GTX.

The following figure shows the number of Mega Pixels that can be compressed to the _Y_X DXT5 format per second (higher MP/s = better).



The following figure shows the number of Mega Pixels that can be compressed to the 3Dc format per second (higher MP/s = better).



8. Conclusion

Existing color texture compression formats can also be used to store normal maps, but the results vary. The latest graphics hardware also implements formats specifically designed for normal map compression. While decompression from these formats happens in real-time in hardware during rendering, compression to these formats may take a considerable amount of time. Existing compressors are designed for high-quality off-line compression, not real-time compression. However, at the cost of a little quality, normal maps can also be compressed real-time on both the CPU and GPU, which is useful for transcoding normal maps from a different format and compression of dynamically generated normal maps.

9. References

1. Simulation of Wrinkled Surfaces.
   James F. Blinn
   In Proceedings of SIGGRAPH, vol. 12, #3, pp. 286-292, 1978
   Available Online: http://portal.acm.org/citation.cfm?id=507101
2. Efficient Bump Mapping Hardware.
   Mark Peercy, John Airey, Brian Cabral
   Computer Graphics, vol. 31, pp. 303-306, 1997
   Available Online: http://citeseer.ist.psu.edu/peercy97efficient.html
3. S3 Texture Compression
   Pat Brown
   NVIDIA Corporation, November 2001
   Available Online: http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt
4. Compressed Texture Resources (Direct3D 9)
   Microsoft Developer Network
   MSDN, November 2007
   Available Online: http://msdn2.microsoft.com/en-us/library/bb204843.aspx
5. Block Compression (Direct3D 10)
   Microsoft Developer Network
   MSDN, November 2007
   Available Online: http://msdn2.microsoft.com/en-us/library/bb694531.aspx
6. Image Coding using Block Truncation Coding
   E.J. Delp, O.R. Mitchell
   IEEE Transactions on Communications, vol. 27(9), pp. 1335-1342, September 1979
   Available Online: http://scholarsmine.umr.edu/post_prints/01094560_09007dcc8030cc78.html
7. Bump Map Compression
   Simon Green
   NVIDIA Technical Report, October 2001
   Available Online: http://developer.nvidia.com/object/bump_map_compression.html

8. Normal Map Compression
ATI Technologies Inc
ATI, August 2003
Available Online: http://www.ati.com/developer/NormalMapCompression.pdf
9. Normal Map Compression
Jakub Klarowicz
Shader X2: Shader Programming Tips & Tricks with DirectX 9
Available Online: http://www.shaderx2.com
10. 3Dc White Paper
ATI Technologies Inc
ATI, April 2004
Available Online: http://ati.de/products/radeonx800/3DcWhitePaper.pdf
11. High Quality Normal Map Compression
Jacob Munkberg, Tomas Akenine-Möller, Jacob Ström
Graphics Hardware 2006
Available Online: http://graphics.cs.lth.se/research/papers/normals2006/
12. Tight Frame Normal Map Compression
Jacob Munkberg, Ola Olsson, Jacob Ström, Tomas Akenine-Möller
Graphics Hardware 2007
Available Online: http://graphics.cs.lth.se/research/papers/2007/tightframe/
13. Fast and Efficient Normal Map Compression Based on Vector Quantization
T. Yamasaki, K. Aizawa
In Proceedings of ICASSP (2006), vol. 2, pp. 2-12
Available Online:
http://www.ee.columbia.edu/~dpwe/LabROSA/proceeds/icassp/2006/pdfs/0200009.pdf
14. A Hybrid Adaptive Normal Map Texture Compression Algorithm
B. Yang, Z. Pan
In International Conference on Artificial Reality and Telexistence (2006), IEEE Computer Society, pp. 349-354
Available Online: http://doi.ieeecomputersociety.org/10.1109/ICAT.2006.11
15. Mathematical error analysis of normal map compression based on unity condition
Toshihiko Yamasaki, Kazuya Hayase, Kiyoharu Aizawa
IEEE International Conference on Image Processing, vol. 2, pp. 253-6, September 2005
Available Online: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1530039
16. Mathematical PSNR Prediction Model Between Compressed Normal Maps and Rendered 3D Images
Toshihiko Yamasaki, Kazuya Hayase, Kiyoharu Aizawa
Pacific Rim Conference on Multimedia (PCM2005) LNCS 3768, pp. 584-594, Jeju Island, Korea, Nov. 13-16, 2005
Available Online: http://www.springerlink.com/content/f3707080w8553g3l/

17. Real-time rendering of normal maps with discontinuities
Evgueni Parilov, Ilya Rosenberg, Denis Zorin
CIMS Technical Report, TR2005-872, August 2005
Available Online:
http://csdocs.cs.nyu.edu/Dienst/UI/2.0/Describe/ncstrl.nyu_cs%2FTR2005-872t
18. Mipmapping normal maps
M. Toksvig
Journal of Graphics Tools 10, 3, 65-71. 2005
Available Online: http://developer.nvidia.com/object/mipmapping_normal_maps.html
19. Frequency Domain Normal Map Filtering
Charles Han, Bo Sun, Ravi Ramamoorthi, Eitan Grinspun
SIGGRAPH 2007
Available Online: http://www.cs.columbia.edu/cg/normalmap/normalmap.pdf
20. ATI Compressonator Library
Seth Sowerby, Daniel Killebrew
ATI Technologies Inc, The Compressonator version 1.27.1066, March 2006
Available Online: http://www.ati.com/developer/compressonator.html
21. NVIDIA DDS Utilities
NVIDIA
NVIDIA DDS Utilities, April 2006
Available Online: http://developer.nvidia.com/object/nv_texture_tools.html
22. NVIDIA Texture Tools
NVIDIA
NVIDIA Texture Tools, September 2007
Available Online: http://developer.nvidia.com/object/texture_tools.html
23. Real-Time DXT Compression
J.M.P. van Waveren
Intel Software Network, October 2006
Available Online: http://www.intel.com/cd/ids/developer/asmo-na/eng/324337.htm
24. Real-Time YCoCg-DXT Compression
J.M.P. van Waveren, Ignacio Castaño
NVIDIA, October 2007
Available Online: http://news.developer.nvidia.com/2007/10/real-time-ycocg.html
25. GL_EXT_texture_compression_latc
Available Online: http://www.opengl.org/registry/specs/EXT/texture_compression_latc.txt
26. GL_EXT_texture_compression_rgtc
Available Online: http://www.opengl.org/registry/specs/EXT/texture_compression_rgtc.txt

Appendix A

```
/*
    Real-Time Normal Map Compression (C++)
    Copyright (C) 2008 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

typedef unsigned char   byte;
typedef unsigned short  word;
typedef unsigned int    dword;

#define INSET_COLOR_SHIFT       4       // inset color channel
#define INSET_ALPHA_SHIFT       5       // inset alpha channel

#define C565_5_MASK             0xF8    // 0xFF minus last three bits
#define C565_6_MASK             0xFC    // 0xFF minus last two bits

byte *globalOutData;

void EmitByte( byte b ) {
    globalOutData[0] = b;
    globalOutData += 1;
}

void EmitWord( word s ) {
    globalOutData[0] = ( s >>  0 ) & 255;
    globalOutData[1] = ( s >>  8 ) & 255;
    globalOutData += 2;
}

void EmitDoubleWord( dword i ) {
    globalOutData[0] = ( i >>  0 ) & 255;
    globalOutData[1] = ( i >>  8 ) & 255;
    globalOutData[2] = ( i >> 16 ) & 255;
    globalOutData[3] = ( i >> 24 ) & 255;
    globalOutData += 4;
}

word NormalYTo565( byte y ) {
    return ( ( y >> 2 ) << 5 );
}
```

```
void ExtractBlock( const byte *inPtr, const int width, byte *block ) {
    for ( int j = 0; j < 4; j++ ) {
        memcpy( &block[j*4*4], inPtr, 4*4 );
        inPtr += width * 4;
    }
}

void GetMinMaxNormalsBBox( const byte *block, byte *minNormal, byte
*maxNormal ) {

    minNormal[0] = minNormal[1] = 255;
    maxNormal[0] = maxNormal[1] = 0;

    for ( int i = 0; i < 16; i++ ) {
        if ( block[i*4+0] < minNormal[0] ) {
            minNormal[0] = block[i*4+0];
        }
        if ( block[i*4+1] < minNormal[1] ) {
            minNormal[1] = block[i*4+1];
        }
        if ( block[i*4+0] > maxNormal[0] ) {
            maxNormal[0] = block[i*4+0];
        }
        if ( block[i*4+1] > maxNormal[1] ) {
            maxNormal[1] = block[i*4+1];
        }
    }
}

void InsetNormalsBBoxDXT5( byte *minNormal, byte *maxNormal ) {
    int inset[4];
    int mini[4];
    int maxi[4];

    inset[0] = ( maxNormal[0] - minNormal[0] ) - ((1<<<< INSET_ALPHA_SHIFT )
+ inset[0] ) >> INSET_ALPHA_SHIFT;
    mini[1] = ( ( minNormal[1] << INSET_COLOR_SHIFT ) + inset[1] ) >>
INSET_COLOR_SHIFT;

    maxi[0] = ( ( maxNormal[0] << INSET_ALPHA_SHIFT ) - inset[0] ) >>
INSET_ALPHA_SHIFT;
    maxi[1] = ( ( maxNormal[1] << INSET_COLOR_SHIFT ) - inset[1] ) >>
INSET_COLOR_SHIFT;

    mini[0] = ( mini[0] >= 0 ) ? mini[0] : 0;
    mini[1] = ( mini[1] >= 0 ) ? mini[1] : 0;

    maxi[0] = ( maxi[0] <= 255 ) ? maxi[0] : 255;
    maxi[1] = ( maxi[1] <= 255 ) ? maxi[1] : 255;

    minNormal[0] = mini[0];
    minNormal[1] = ( mini[1] & C565_6_MASK ) | ( mini[1] >> 6 );

    maxNormal[0] = maxi[0];
    maxNormal[1] = ( maxi[1] & C565_6_MASK ) | ( maxi[1] >> 6 );
}
```

```
void InsetNormalsBBox3Dc( byte *minNormal, byte *maxNormal ) {
    int inset[4];
    int mini[4];
    int maxi[4];

    inset[0] = ( maxNormal[0] - minNormal[0] ) - ((1<<<< INSET_ALPHA_SHIFT )
+ inset[0] ) >> INSET_ALPHA_SHIFT;
    mini[1] = ( ( minNormal[1] << INSET_ALPHA_SHIFT ) + inset[1] ) >>
INSET_ALPHA_SHIFT;

    maxi[0] = ( ( maxNormal[0] << INSET_ALPHA_SHIFT ) - inset[0] ) >>
INSET_ALPHA_SHIFT;
    maxi[1] = ( ( maxNormal[1] << INSET_ALPHA_SHIFT ) - inset[1] ) >>
INSET_ALPHA_SHIFT;

    mini[0] = ( mini[0] >= 0 ) ? mini[0] : 0;
    mini[1] = ( mini[1] >= 0 ) ? mini[1] : 0;

    maxi[0] = ( maxi[0] <= 255 ) ? maxi[0] : 255;
    maxi[1] = ( maxi[1] <= 255 ) ? maxi[1] : 255;

    minNormal[0] = mini[0];
    minNormal[1] = mini[1];

    maxNormal[0] = maxi[0];
    maxNormal[1] = maxi[1];
}

void EmitAlphaIndices( const byte *block, const int offset, const byte
minAlpha, const byte maxAlpha ) {
    byte mid = ( maxAlpha - minAlpha ) / ( 2 * 7 );

    byte ab1 = maxAlpha - mid;
    byte ab2 = ( 6 * maxAlpha + 1 * minAlpha ) / 7 - mid;
    byte ab3 = ( 5 * maxAlpha + 2 * minAlpha ) / 7 - mid;
    byte ab4 = ( 4 * maxAlpha + 3 * minAlpha ) / 7 - mid;
    byte ab5 = ( 3 * maxAlpha + 4 * minAlpha ) / 7 - mid;
    byte ab6 = ( 2 * maxAlpha + 5 * minAlpha ) / 7 - mid;
    byte ab7 = ( 1 * maxAlpha + 6 * minAlpha ) / 7 - mid;

    block += offset;

    byte indices[16];
    for ( int i = 0; i < 16; i++ ) {
        byte a = block[i*4];
        int b1 = ( a >= ab1 );
        int b2 = ( a >= ab2 );
        int b3 = ( a >= ab3 );
        int b4 = ( a >= ab4 );
        int b5 = ( a >= ab5 );
        int b6 = ( a >= ab6 );
        int b7 = ( a >= ab7 );
        int index = ( 8 - b1 - b2 - b3 - b4 - b5 - b6 - b7 ) & 7;
        indices[i] = index ^ ( 2 > index );
    }
```

```
    EmitByte( (indices[ 0] >> 0) | (indices[ 1] << 3) | (indices[ 2] << 6) );
    EmitByte( (indices[ 2] >> 2) | (indices[ 3] << 1) | (indices[ 4] << 4) |
(indices[ 5] << 7) );
    EmitByte( (indices[ 5] >> 1) | (indices[ 6] << 2) | (indices[ 7] << 5) );

    EmitByte( (indices[ 8] >> 0) | (indices[ 9] << 3) | (indices[10] << 6) );
    EmitByte( (indices[10] >> 2) | (indices[11] << 1) | (indices[12] << 4) |
(indices[13] << 7) );
    EmitByte( (indices[13] >> 1) | (indices[14] << 2) | (indices[15] << 5) );
}

void EmitGreenIndices( const byte *block, const int offset, const byte
minGreen, const byte maxGreen ) {
    byte mid = ( maxGreen - minGreen ) / ( 2 * 3 );

    byte gb1 = maxGreen - mid;
    byte gb2 = ( 2 * maxGreen + 1 * minGreen ) / 3 - mid;
    byte gb3 = ( 1 * maxGreen + 2 * minGreen ) / 3 - mid;

    block += offset;

    unsigned int result = 0;
    for ( int i = 15; i >= 0; i-- ) {
        result <<= 2;
        byte g = block[i*4];
        int b1 = ( g >= gb1 );
        int b2 = ( g >= gb2 );
        int b3 = ( g >= gb3 );
        int index = ( 4 - b1 - b2 - b3 ) & 3;
        index ^= ( 2 > index );
        result |= index;
    }

    EmitUInt( result );
}

void CompressNormalMapDXT5( const byte *inBuf, byte *outBuf, int width, int
height, int &outputBytes ) {
    byte block[64];
    byte normalMin[4];
    byte normalMax[4];

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox( block, normalMin, normalMax );
            InsetNormalsBBoxDXT5( normalMin, normalMax );

            // Write out Nx into alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
```

```
            EmitAlphaIndices( block, 0, normalMin[0], normalMax[0] );

            // Write out Ny into green channel.
            EmitUShort( NormalYTo565( normalMax[1] ) );
            EmitUShort( NormalYTo565( normalMin[1] ) );
            EmitGreenIndices( block, 1, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}

void CompressNormalMap3Dc( const byte *inBuf, byte *outBuf, int width, int
height, int &outputBytes ) {
    byte block[64];
    byte normalMin[4];
    byte normalMax[4];

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox( block, normalMin, normalMax );
            InsetNormalsBBox3Dc( normalMin, normalMax );

            // Write out Nx as an alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
            EmitAlphaIndices( block, 0, normalMin[0], normalMax[0] );

            // Write out Ny as an alpha channel.
            EmitByte( normalMax[1] );
            EmitByte( normalMin[1] );
            EmitAlphaIndices( block, 1, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}
```

Appendix B

```
/*
    Real-Time Normal Map Compression (MMX)
    Copyright (C) 2008 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

#define ALIGN16( x )                __declspec(align(16)) x
#define R_SHUFFLE_D( x, y, z, w )   (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | (
(y) & 3 ) << 2 | ( (x) & 3 ))

ALIGN16( static dword SIMD_MMX_dword_byte_mask[2] ) = { 0x000000FF,
0x000000FF };
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask0[2] ) = {
7<<<<<<<<<<<<<<<<<<<<<<<< INSET_ALPHA_SHIFT, 1 << INSET_COLOR_SHIFT, 1, 1 };
ALIGN16( static word SIMD_MMX_word_insetNormalDXT5ShiftDown[4] ) = { 1 << (
16 - INSET_ALPHA_SHIFT ), 1 << ( 16 - INSET_COLOR_SHIFT ), 0, 0 };
ALIGN16( static word SIMD_MMX_word_insetNormalDXT5QuantMask[4] ) = { 0xFF,
C565_6_MASK, 0xFF, 0xFF };
ALIGN16( static word SIMD_MMX_word_insetNormalDXT5Rep[4] ) = { 0, 1 << ( 16 -
6 ), 0, 0 };
ALIGN16( static word SIMD_MMX_word_insetNormal3DcRound[4] ) = { ((1<<<<
INSET_ALPHA_SHIFT, 1 << INSET_ALPHA_SHIFT, 1, 1 };
ALIGN16( static word SIMD_MMX_word_insetNormal3DcShiftDown[4] ) = { 1 << ( 16
- INSET_ALPHA_SHIFT ), 1 << ( 16 - INSET_ALPHA_SHIFT ), 0, 0 };
ALIGN16( static byte SIMD_MMX_byte_0[8] ) = { 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00 };
ALIGN16( static byte SIMD_MMX_byte_1[8] ) = { 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01 };
ALIGN16( static byte SIMD_MMX_byte_2[8] ) = { 0x02, 0x02, 0x02, 0x02, 0x02,
0x02, 0x02, 0x02 };
ALIGN16( static byte SIMD_MMX_byte_7[8] ) = { 0x07, 0x07, 0x07, 0x07, 0x07,
0x07, 0x07, 0x07 };
ALIGN16( static byte SIMD_MMX_byte_8[8] ) = { 0x08, 0x08, 0x08, 0x08, 0x08,
0x08, 0x08, 0x08 };
ALIGN16( static byte SIMD_MMX_byte_not[8] ) = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xFF, 0xFF };

void ExtractBlock_MMX( const byte *inPtr, int width, byte *block ) {
    __asm {
        mov         esi, inPtr
        mov         edi, block
```

```
        mov         eax, width
        shl         eax, 2
        movq        mm0, qword ptr [esi+0]
        movq        qword ptr [edi+ 0], mm0
        movq        mm1, qword ptr [esi+8]
        movq        qword ptr [edi+ 8], mm1
        movq        mm2, qword ptr [esi+eax+0]
        movq        qword ptr [edi+16], mm2
        movq        mm3, qword ptr [esi+eax+8]
        movq        qword ptr [edi+24], mm3
        movq        mm4, qword ptr [esi+eax*2+0]
        movq        qword ptr [edi+32], mm4
        movq        mm5, qword ptr [esi+eax*2+8]
        add         esi, eax
        movq        qword ptr [edi+40], mm5
        movq        mm6, qword ptr [esi+eax*2+0]
        movq        qword ptr [edi+48], mm6
        movq        mm7, qword ptr [esi+eax*2+8]
        movq        qword ptr [edi+56], mm7
        emms
    }
}

void GetMinMaxNormalsBBox_MMX( const byte *block, byte *minNormal, byte
*maxNormal ) {
    __asm {
        mov         eax, block
        mov         esi, minNormal
        mov         edi, maxNormal
        pshufw      mm0, qword ptr [eax+ 0], R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm1, qword ptr [eax+ 0], R_SHUFFLE_D( 0, 1, 2, 3 )
        pminub      mm0, qword ptr [eax+ 8]
        pmaxub      mm1, qword ptr [eax+ 8]
        pminub      mm0, qword ptr [eax+16]
        pmaxub      mm1, qword ptr [eax+16]
        pminub      mm0, qword ptr [eax+24]
        pmaxub      mm1, qword ptr [eax+24]
        pminub      mm0, qword ptr [eax+32]
        pmaxub      mm1, qword ptr [eax+32]
        pminub      mm0, qword ptr [eax+40]
        pmaxub      mm1, qword ptr [eax+40]
        pminub      mm0, qword ptr [eax+48]
        pmaxub      mm1, qword ptr [eax+48]
        pminub      mm0, qword ptr [eax+56]
        pmaxub      mm1, qword ptr [eax+56]
        pshufw      mm6, mm0, R_SHUFFLE_D( 2, 3, 2, 3 )
        pshufw      mm7, mm1, R_SHUFFLE_D( 2, 3, 2, 3 )
        pminub      mm0, mm6
        pmaxub      mm1, mm7
        movd        dword ptr [esi], mm0
        movd        dword ptr [edi], mm1
        emms
    }
}

void InsetNormalsBBoxDXT5_MMX( byte *minNormal, byte *maxNormal ) {
```

```
    __asm {
        mov         esi, minNormal
        mov         edi, maxNormal
        movd        mm0, dword ptr [esi]
        movd        mm1, dword ptr [edi]
        punpcklbw   mm0, SIMD_MMX_byte_0
        punpcklbw   mm1, SIMD_MMX_byte_0
        movq        mm2, mm1
        psubw       mm2, mm0
        psubw       mm2, SIMD_MMX_word_insetNormalDXT5Round
        pand        mm2, SIMD_MMX_word_insetNormalDXT5Mask
        pmullw      mm0, SIMD_MMX_word_insetNormalDXT5ShiftUp
        pmullw      mm1, SIMD_MMX_word_insetNormalDXT5ShiftUp
        paddw       mm0, mm2
        psubw       mm1, mm2
        pmulhw      mm0, SIMD_MMX_word_insetNormalDXT5ShiftDown
        pmulhw      mm1, SIMD_MMX_word_insetNormalDXT5ShiftDown
        pmaxsw      mm0, SIMD_MMX_word_0
        pmaxsw      mm1, SIMD_MMX_word_0
        pand        mm0, SIMD_MMX_word_insetNormalDXT5QuantMask
        pand        mm1, SIMD_MMX_word_insetNormalDXT5QuantMask
        movq        mm2, mm0
        movq        mm3, mm1
        pmulhw      mm2, SIMD_MMX_word_insetNormalDXT5Rep
        pmulhw      mm3, SIMD_MMX_word_insetNormalDXT5Rep
        por         mm0, mm2
        por         mm1, mm3
        packuswb    mm0, mm0
        packuswb    mm1, mm1
        movd        dword ptr [esi], mm0
        movd        dword ptr [edi], mm1
        emms
    }
}

void InsetNormalsBBox3Dc_MMX( byte *minNormal, byte *maxNormal ) {
    __asm {
        mov         esi, minNormal
        mov         edi, maxNormal
        movd        mm0, dword ptr [esi]
        movd        mm1, dword ptr [edi]
        punpcklbw   mm0, SIMD_MMX_byte_0
        punpcklbw   mm1, SIMD_MMX_byte_0
        movq        mm2, mm1
        psubw       mm2, mm0
        psubw       mm2, SIMD_MMX_word_insetNormal3DcRound
        pand        mm2, SIMD_MMX_word_insetNormal3DcMask
        pmullw      mm0, SIMD_MMX_word_insetNormal3DcShiftUp
        pmullw      mm1, SIMD_MMX_word_insetNormal3DcShiftUp
        paddw       mm0, mm2
        psubw       mm1, mm2
        pmulhw      mm0, SIMD_MMX_word_insetNormal3DcShiftDown
        pmulhw      mm1, SIMD_MMX_word_insetNormal3DcShiftDown
        pmaxsw      mm0, SIMD_MMX_word_0
        pmaxsw      mm1, SIMD_MMX_word_0
        packuswb    mm0, mm0
```

```
        packuswb    mm1, mm1
        movd        dword ptr [esi], mm0
        movd        dword ptr [edi], mm1
        emms
    }
}

void EmitAlphaIndices_MMX( const byte *block, const int channelBitOffset,
const int minAlpha, const int maxAlpha ) {
    ALIGN16( byte alphaBlock[16] );
    ALIGN16( byte ab1[8] );
    ALIGN16( byte ab2[8] );
    ALIGN16( byte ab3[8] );
    ALIGN16( byte ab4[8] );
    ALIGN16( byte ab5[8] );
    ALIGN16( byte ab6[8] );
    ALIGN16( byte ab7[8] );

    __asm {
        movd        mm7, channelBitOffset

        mov         esi, block
        movq        mm0, qword ptr [esi+ 0]
        movq        mm5, qword ptr [esi+ 8]
        movq        mm6, qword ptr [esi+16]
        movq        mm4, qword ptr [esi+24]

        psrld       mm0, mm7
        psrld       mm5, mm7
        psrld       mm6, mm7
        psrld       mm4, mm7

        pand        mm0, SIMD_MMX_dword_byte_mask
        pand        mm5, SIMD_MMX_dword_byte_mask
        pand        mm6, SIMD_MMX_dword_byte_mask
        pand        mm4, SIMD_MMX_dword_byte_mask

        packuswb    mm0, mm5
        packuswb    mm6, mm4

        packuswb    mm0, mm6
        movq        alphaBlock+0, mm0

        movq        mm0, qword ptr [esi+32]
        movq        mm5, qword ptr [esi+40]
        movq        mm6, qword ptr [esi+48]
        movq        mm4, qword ptr [esi+56]

        psrld       mm0, mm7
        psrld       mm5, mm7
        psrld       mm6, mm7
        psrld       mm4, mm7

        pand        mm0, SIMD_MMX_dword_byte_mask
        pand        mm5, SIMD_MMX_dword_byte_mask
        pand        mm6, SIMD_MMX_dword_byte_mask
```

```
    pand        mm4, SIMD_MMX_dword_byte_mask

    packuswb    mm0, mm5
    packuswb    mm6, mm4

    packuswb    mm0, mm6
    movq        alphaBlock+8, mm0

    movd        mm0, maxAlpha
    pshufw      mm0, mm0, R_SHUFFLE_D( 0, 0, 0, 0 )
    movq        mm1, mm0

    movd        mm2, minAlpha
    pshufw      mm2, mm2, R_SHUFFLE_D( 0, 0, 0, 0 )
    movq        mm3, mm2

    movq        mm4, mm0
    psubw       mm4, mm2
    pmulhw      mm4, SIMD_MMX_word_div_by_14

    movq        mm5, mm0
    psubw       mm5, mm4
    packuswb    mm5, mm5
    movq        ab1, mm5

    pmullw      mm0, SIMD_MMX_word_scale654
    pmullw      mm1, SIMD_MMX_word_scale123
    pmullw      mm2, SIMD_MMX_word_scale123
    pmullw      mm3, SIMD_MMX_word_scale654
    paddw       mm0, mm2
    paddw       mm1, mm3
    pmulhw      mm0, SIMD_MMX_word_div_by_7
    pmulhw      mm1, SIMD_MMX_word_div_by_7
    psubw       mm0, mm4
    psubw       mm1, mm4

    pshufw      mm2, mm0, R_SHUFFLE_D( 0, 0, 0, 0 )
    pshufw      mm3, mm0, R_SHUFFLE_D( 1, 1, 1, 1 )
    pshufw      mm4, mm0, R_SHUFFLE_D( 2, 2, 2, 2 )
    packuswb    mm2, mm2
    packuswb    mm3, mm3
    packuswb    mm4, mm4
    movq        ab2, mm2
    movq        ab3, mm3
    movq        ab4, mm4

    pshufw      mm2, mm1, R_SHUFFLE_D( 2, 2, 2, 2 )
    pshufw      mm3, mm1, R_SHUFFLE_D( 1, 1, 1, 1 )
    pshufw      mm4, mm1, R_SHUFFLE_D( 0, 0, 0, 0 )
    packuswb    mm2, mm2
    packuswb    mm3, mm3
    packuswb    mm4, mm4
    movq        ab5, mm2
    movq        ab6, mm3
    movq        ab7, mm4
```

```
    pshufw      mm0, alphaBlock+0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pmaxub      mm1, ab1
    pmaxub      mm2, ab2
    pmaxub      mm3, ab3
    pmaxub      mm4, ab4
    pmaxub      mm5, ab5
    pmaxub      mm6, ab6
    pmaxub      mm7, ab7
    pcmpeqb     mm1, mm0
    pcmpeqb     mm2, mm0
    pcmpeqb     mm3, mm0
    pcmpeqb     mm4, mm0
    pcmpeqb     mm5, mm0
    pcmpeqb     mm6, mm0
    pcmpeqb     mm7, mm0
    pshufw      mm0, SIMD_MMX_byte_8, R_SHUFFLE_D( 0, 1, 2, 3 )
    paddsb      mm0, mm1
    paddsb      mm2, mm3
    paddsb      mm4, mm5
    paddsb      mm6, mm7
    paddsb      mm0, mm2
    paddsb      mm4, mm6
    paddsb      mm0, mm4
    pand        mm0, SIMD_MMX_byte_7
    pshufw      mm1, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
    pcmpgtb     mm1, mm0
    pand        mm1, SIMD_MMX_byte_1
    pxor        mm0, mm1
    pshufw      mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    psrlq       mm1,  8- 3
    psrlq       mm2, 16- 6
    psrlq       mm3, 24- 9
    psrlq       mm4, 32-12
    psrlq       mm5, 40-15
    psrlq       mm6, 48-18
    psrlq       mm7, 56-21
    pand        mm0, SIMD_MMX_dword_alpha_bit_mask0
    pand        mm1, SIMD_MMX_dword_alpha_bit_mask1
    pand        mm2, SIMD_MMX_dword_alpha_bit_mask2
    pand        mm3, SIMD_MMX_dword_alpha_bit_mask3
    pand        mm4, SIMD_MMX_dword_alpha_bit_mask4
    pand        mm5, SIMD_MMX_dword_alpha_bit_mask5
    pand        mm6, SIMD_MMX_dword_alpha_bit_mask6
```

```
    pand        mm7, SIMD_MMX_dword_alpha_bit_mask7
    por         mm0, mm1
    por         mm2, mm3
    por         mm4, mm5
    por         mm6, mm7
    por         mm0, mm2
    por         mm4, mm6
    por         mm0, mm4
    mov         esi, globalOutData
    movd        [esi+0], mm0

    pshufw      mm0, alphaBlock+8, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pmaxub      mm1, ab1
    pmaxub      mm2, ab2
    pmaxub      mm3, ab3
    pmaxub      mm4, ab4
    pmaxub      mm5, ab5
    pmaxub      mm6, ab6
    pmaxub      mm7, ab7
    pcmpeqb     mm1, mm0
    pcmpeqb     mm2, mm0
    pcmpeqb     mm3, mm0
    pcmpeqb     mm4, mm0
    pcmpeqb     mm5, mm0
    pcmpeqb     mm6, mm0
    pcmpeqb     mm7, mm0
    pshufw      mm0, SIMD_MMX_byte_8, R_SHUFFLE_D( 0, 1, 2, 3 )
    paddsb      mm0, mm1
    paddsb      mm2, mm3
    paddsb      mm4, mm5
    paddsb      mm6, mm7
    paddsb      mm0, mm2
    paddsb      mm4, mm6
    paddsb      mm0, mm4
    pand        mm0, SIMD_MMX_byte_7
    pshufw      mm1, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
    pcmpgtb     mm1, mm0
    pand        mm1, SIMD_MMX_byte_1
    pxor        mm0, mm1
    pshufw      mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
    psrlq       mm1,  8- 3
    psrlq       mm2, 16- 6
    psrlq       mm3, 24- 9
```

```
        psrlq       mm4, 32-12
        psrlq       mm5, 40-15
        psrlq       mm6, 48-18
        psrlq       mm7, 56-21
        pand        mm0, SIMD_MMX_dword_alpha_bit_mask0
        pand        mm1, SIMD_MMX_dword_alpha_bit_mask1
        pand        mm2, SIMD_MMX_dword_alpha_bit_mask2
        pand        mm3, SIMD_MMX_dword_alpha_bit_mask3
        pand        mm4, SIMD_MMX_dword_alpha_bit_mask4
        pand        mm5, SIMD_MMX_dword_alpha_bit_mask5
        pand        mm6, SIMD_MMX_dword_alpha_bit_mask6
        pand        mm7, SIMD_MMX_dword_alpha_bit_mask7
        por         mm0, mm1
        por         mm2, mm3
        por         mm4, mm5
        por         mm6, mm7
        por         mm0, mm2
        por         mm4, mm6
        por         mm0, mm4
        movd        dword ptr [esi+3], mm0

        emms
    }

    globalOutData += 6;
}

void EmitGreenIndices_MMX( const byte *block, const int channelBitOffset,
const int minGreen, const int maxGreen ) {
    ALIGN16( byte greenBlock[16] );

    __asm {
        movd        mm7, channelBitOffset

        mov         esi, block
        movq        mm0, qword ptr [esi+ 0]
        movq        mm5, qword ptr [esi+ 8]
        movq        mm6, qword ptr [esi+16]
        movq        mm4, qword ptr [esi+24]

        psrld       mm0, mm7
        psrld       mm5, mm7
        psrld       mm6, mm7
        psrld       mm4, mm7

        pand        mm0, SIMD_MMX_dword_byte_mask
        pand        mm5, SIMD_MMX_dword_byte_mask
        pand        mm6, SIMD_MMX_dword_byte_mask
        pand        mm4, SIMD_MMX_dword_byte_mask

        packuswb    mm0, mm5
        packuswb    mm6, mm4

        packuswb    mm0, mm6
        movq        greenBlock+0, mm0
```

```
movq        mm0, qword ptr [esi+32]
movq        mm5, qword ptr [esi+40]
movq        mm6, qword ptr [esi+48]
movq        mm4, qword ptr [esi+56]

psrld       mm0, mm7
psrld       mm5, mm7
psrld       mm6, mm7
psrld       mm4, mm7

pand        mm0, SIMD_MMX_dword_byte_mask
pand        mm5, SIMD_MMX_dword_byte_mask
pand        mm6, SIMD_MMX_dword_byte_mask
pand        mm4, SIMD_MMX_dword_byte_mask

packuswb    mm0, mm5
packuswb    mm6, mm4

packuswb    mm0, mm6
movq        greenBlock+8, mm0

movd        mm2, maxGreen
pshufw      mm2, mm2, R_SHUFFLE_D( 0, 0, 0, 0 )
movq        mm1, mm2

movd        mm3, minGreen
pshufw      mm3, mm3, R_SHUFFLE_D( 0, 0, 0, 0 )

movq        mm4, mm2
psubw       mm4, mm3
pmulhw      mm4, SIMD_MMX_word_div_by_6

psllw       mm2, 1
paddw       mm2, mm3
pmulhw      mm2, SIMD_MMX_word_div_by_3
psubw       mm2, mm4
packuswb    mm2, mm2                       // gb2

psllw       mm3, 1
paddw       mm3, mm1
pmulhw      mm3, SIMD_MMX_word_div_by_3
psubw       mm3, mm4
packuswb    mm3, mm3                       // gb3

psubw       mm1, mm4
packuswb    mm1, mm1                       // gb1

pshufw      mm0, greenBlock+0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pmaxub      mm5, mm1
pmaxub      mm6, mm2
pmaxub      mm7, mm3
pcmpeqb     mm5, mm0
pcmpeqb     mm6, mm0
```

```
pcmpeqb     mm7, mm0
pshufw      mm0, SIMD_MMX_byte_4, R_SHUFFLE_D( 0, 1, 2, 3 )
paddsb      mm0, mm5
paddsb      mm6, mm7
paddsb      mm0, mm6
pand        mm0, SIMD_MMX_byte_3
pshufw      mm4, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
pcmpgtb     mm4, mm0
pand        mm4, SIMD_MMX_byte_1
pxor        mm0, mm4
pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psrlq       mm4,  8- 2
psrlq       mm5, 16- 4
psrlq       mm6, 24- 6
psrlq       mm7, 32- 8
pand        mm4, SIMD_MMX_dword_color_bit_mask1
pand        mm5, SIMD_MMX_dword_color_bit_mask2
pand        mm6, SIMD_MMX_dword_color_bit_mask3
pand        mm7, SIMD_MMX_dword_color_bit_mask4
por         mm5, mm4
por         mm7, mm6
por         mm7, mm5
pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psrlq       mm4, 40-10
psrlq       mm5, 48-12
psrlq       mm6, 56-14
pand        mm0, SIMD_MMX_dword_color_bit_mask0
pand        mm4, SIMD_MMX_dword_color_bit_mask5
pand        mm5, SIMD_MMX_dword_color_bit_mask6
pand        mm6, SIMD_MMX_dword_color_bit_mask7
por         mm4, mm5
por         mm0, mm6
por         mm7, mm4
por         mm7, mm0
mov         esi, gobalOutPtr
movd        [esi+0], mm7

pshufw      mm0, greenBlock+8, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pmaxub      mm5, mm1
pmaxub      mm6, mm2
pmaxub      mm7, mm3
pcmpeqb     mm5, mm0
pcmpeqb     mm6, mm0
pcmpeqb     mm7, mm0
pshufw      mm0, SIMD_MMX_byte_4, R_SHUFFLE_D( 0, 1, 2, 3 )
paddsb      mm0, mm5
paddsb      mm6, mm7
paddsb      mm0, mm6
```

```
        pand        mm0, SIMD_MMX_byte_3
        pshufw      mm4, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
        pcmpgtb     mm4, mm0
        pand        mm4, SIMD_MMX_byte_1
        pxor        mm0, mm4
        pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        psrlq       mm4,  8- 2
        psrlq       mm5, 16- 4
        psrlq       mm6, 24- 6
        psrlq       mm7, 32- 8
        pand        mm4, SIMD_MMX_dword_color_bit_mask1
        pand        mm5, SIMD_MMX_dword_color_bit_mask2
        pand        mm6, SIMD_MMX_dword_color_bit_mask3
        pand        mm7, SIMD_MMX_dword_color_bit_mask4
        por         mm5, mm4
        por         mm7, mm6
        por         mm7, mm5
        pshufw      mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        pshufw      mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
        psrlq       mm4, 40-10
        psrlq       mm5, 48-12
        psrlq       mm6, 56-14
        pand        mm0, SIMD_MMX_dword_color_bit_mask0
        pand        mm4, SIMD_MMX_dword_color_bit_mask5
        pand        mm5, SIMD_MMX_dword_color_bit_mask6
        pand        mm6, SIMD_MMX_dword_color_bit_mask7
        por         mm4, mm5
        por         mm0, mm6
        por         mm7, mm4
        por         mm7, mm0
        movd        [esi+2], mm7
        emms
    }

    globalOutData += 4;
}

void CompressNormalMapDXT5_MMX( const byte *inBuf, byte *outBuf, int width,
int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte normalMin[4] );
    ALIGN16( byte normalMax[4] );

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock_MMX( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox_MMX( block, normalMin, normalMax );
            InsetNormalsBBoxDXT5_MMX( normalMin, normalMax );
```

```
            // Write out Nx into alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
            EmitAlphaIndices_MMX( block, 0*8, normalMin[0], normalMax[0] );

            // Write out Ny into green channel.
            EmitUShort( NormalYTo565( normalMax[1] ) );
            EmitUShort( NormalYTo565( normalMin[1] ) );
            EmitGreenIndices_MMX( block, 1*8, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}

void CompressNormalMap3Dc_MMX( const byte *inBuf, byte *outBuf, int width,
int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte normalMin[4] );
    ALIGN16( byte normalMax[4] );

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock_MMX( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox_MMX( block, normalMin, normalMax );
            InsetNormalsBBox3Dc_MMX( normalMin, normalMax );

            // Write out Nx as an alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
            EmitAlphaIndices_MMX( block, 0*8, normalMin[0], normalMax[0] );

            // Write out Ny as an alpha channel.
            EmitByte( normalMax[1] );
            EmitByte( normalMin[1] );
            EmitAlphaIndices_MMX( block, 1*8, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}
```

Appendix C

```
/*
    Real-Time Normal Map Compression (SSE2)
    Copyright (C) 2008 Id Software, Inc.
    Written by J.M.P. van Waveren

    This code is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This code is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.
*/

#define ALIGN16( x )                __declspec(align(16)) x
#define R_SHUFFLE_D( x, y, z, w )   (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | (
(y) & 3 ) << 2 | ( (x) & 3 ))

ALIGN16( static dword SIMD_SSE2_dword_byte_mask[4] ) = { 0x000000FF,
0x000000FF, 0x000000FF, 0x000000FF };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask0[4] ) = {
7<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< INSET_ALPHA_SHIFT, 1 <<
INSET_COLOR_SHIFT, 1, 1, 1, 1, 1, 1 };
ALIGN16( static word SIMD_SSE2_word_insetNormalDXT5ShiftDown[8] ) = { 1 << (
16 - INSET_ALPHA_SHIFT ), 1 << ( 16 - INSET_COLOR_SHIFT ), 0, 0, 0, 0, 0, 0
};
ALIGN16( static word SIMD_SSE2_word_insetNormalDXT5QuantMask[8] ) = { 0xFF,
C565_6_MASK, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };
ALIGN16( static word SIMD_SSE2_word_insetNormalDXT5Rep[8] ) = { 0, 1 << ( 16
- 6 ), 0, 0, 0, 0, 0, 0 };
ALIGN16( static word SIMD_SSE2_word_insetNormal3DcRound[8] ) = { ((1<<<<
INSET_ALPHA_SHIFT, 1 << INSET_ALPHA_SHIFT, 1, 1, 1, 1, 1, 1 };
ALIGN16( static word SIMD_SSE2_word_insetNormal3DcShiftDown[8] ) = { 1 << (
16 - INSET_ALPHA_SHIFT ), 1 << ( 16 - INSET_ALPHA_SHIFT ), 0, 0, 0, 0, 0, 0
};
ALIGN16( static byte SIMD_SSE2_byte_0[16] ) = { 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
ALIGN16( static byte SIMD_SSE2_byte_1[16] ) = { 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 };
ALIGN16( static byte SIMD_SSE2_byte_2[16] ) = { 0x02, 0x02, 0x02, 0x02, 0x02,
0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02 };
ALIGN16( static byte SIMD_SSE2_byte_7[16] ) = { 0x07, 0x07, 0x07, 0x07, 0x07,
0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 };

void ExtractBlock_SSE2( const byte *inPtr, int width, byte *block ) {
    __asm {
        mov         esi, inPtr
        mov         edi, block
        mov         eax, width
```

```
        shl         eax, 2
        movdqa      xmm0, [esi]
        movdqa      xmmword ptr [edi+ 0], xmm0
        movdqa      xmm1, xmmword ptr [esi+eax]
        movdqa      xmmword ptr [edi+16], xmm1
        movdqa      xmm2, xmmword ptr [esi+eax*2]
        add         esi, eax
        movdqa      xmmword ptr [edi+32], xmm2
        movdqa      xmm3, xmmword ptr [esi+eax*2]
        movdqa      xmmword ptr [edi+48], xmm3
    }
}

void GetMinMaxNormalsBBox_SSE2( const byte *block, byte *minNormal, byte
*maxNormal ) {
    __asm {
        mov         eax, block
        mov         esi, minNormal
        mov         edi, maxNormal
        movdqa      xmm0, xmmword ptr [eax+ 0]
        movdqa      xmm1, xmmword ptr [eax+ 0]
        pminub      xmm0, xmmword ptr [eax+16]
        pmaxub      xmm1, xmmword ptr [eax+16]
        pminub      xmm0, xmmword ptr [eax+32]
        pmaxub      xmm1, xmmword ptr [eax+32]
        pminub      xmm0, xmmword ptr [eax+48]
        pmaxub      xmm1, xmmword ptr [eax+48]
        pshufd      xmm3, xmm0, R_SHUFFLE_D( 2, 3, 2, 3 )
        pshufd      xmm4, xmm1, R_SHUFFLE_D( 2, 3, 2, 3 )
        pminub      xmm0, xmm3
        pmaxub      xmm1, xmm4
        pshuflw     xmm6, xmm0, R_SHUFFLE_D( 2, 3, 2, 3 )
        pshuflw     xmm7, xmm1, R_SHUFFLE_D( 2, 3, 2, 3 )
        pminub      xmm0, xmm6
        pmaxub      xmm1, xmm7
        movd        dword ptr [esi], xmm0
        movd        dword ptr [edi], xmm1
    }
}

void InsetNormalsBBoxDXT5_SSE2( byte *minNormal, byte *maxNormal ) {
    __asm {
        mov         esi, minNormal
        mov         edi, maxNormal
        movd        xmm0, dword ptr [esi]
        movd        xmm1, dword ptr [edi]
        punpcklbw   xmm0, SIMD_SSE2_byte_0
        punpcklbw   xmm1, SIMD_SSE2_byte_0
        movdqa      xmm2, xmm1
        psubw       xmm2, xmm0
        psubw       xmm2, SIMD_SSE2_word_insetNormalDXT5Round
        pand        xmm2, SIMD_SSE2_word_insetNormalDXT5Mask
        pmullw      xmm0, SIMD_SSE2_word_insetNormalDXT5ShiftUp
        pmullw      xmm1, SIMD_SSE2_word_insetNormalDXT5ShiftUp
        paddw       xmm0, xmm2
        psubw       xmm1, xmm2
```

```
        pmulhw      xmm0, SIMD_SSE2_word_insetNormalDXT5ShiftDown
        pmulhw      xmm1, SIMD_SSE2_word_insetNormalDXT5ShiftDown
        pmaxsw      xmm0, SIMD_SSE2_word_0
        pmaxsw      xmm1, SIMD_SSE2_word_0
        pand        xmm0, SIMD_SSE2_word_insetNormalDXT5QuantMask
        pand        xmm1, SIMD_SSE2_word_insetNormalDXT5QuantMask
        movdqa      xmm2, xmm0
        movdqa      xmm3, xmm1
        pmulhw      xmm2, SIMD_SSE2_word_insetNormalDXT5Rep
        pmulhw      xmm3, SIMD_SSE2_word_insetNormalDXT5Rep
        por         xmm0, xmm2
        por         xmm1, xmm3
        packuswb    xmm0, xmm0
        packuswb    xmm1, xmm1
        movd        dword ptr [esi], xmm0
        movd        dword ptr [edi], xmm1
    }
}

void InsetNormalsBBox3Dc_SSE2( byte *minNormal, byte *maxNormal ) {
    __asm {
        mov         esi, minNormal
        mov         edi, maxNormal
        movd        xmm0, dword ptr [esi]
        movd        xmm1, dword ptr [edi]
        punpcklbw   xmm0, SIMD_SSE2_byte_0
        punpcklbw   xmm1, SIMD_SSE2_byte_0
        movdqa      xmm2, xmm1
        psubw       xmm2, xmm0
        psubw       xmm2, SIMD_SSE2_word_insetNormal3DcRound
        pand        xmm2, SIMD_SSE2_word_insetNormal3DcMask
        pmullw      xmm0, SIMD_SSE2_word_insetNormal3DcShiftUp
        pmullw      xmm1, SIMD_SSE2_word_insetNormal3DcShiftUp
        paddw       xmm0, xmm2
        psubw       xmm1, xmm2
        pmulhw      xmm0, SIMD_SSE2_word_insetNormal3DcShiftDown
        pmulhw      xmm1, SIMD_SSE2_word_insetNormal3DcShiftDown
        pmaxsw      xmm0, SIMD_SSE2_word_0
        pmaxsw      xmm1, SIMD_SSE2_word_0
        packuswb    xmm0, xmm0
        packuswb    xmm1, xmm1
        movd        dword ptr [esi], xmm0
        movd        dword ptr [edi], xmm1
    }
}

void EmitAlphaIndices_SSE2( const byte *block, const int channelBitOffset,
const int minAlpha, const int maxAlpha ) {
    __asm {
        movd        xmm7, channelBitOffset

        mov         esi, block
        movdqa      xmm0, xmmword ptr [esi+ 0]
        movdqa      xmm5, xmmword ptr [esi+16]
        movdqa      xmm6, xmmword ptr [esi+32]
        movdqa      xmm4, xmmword ptr [esi+48]
```

```
psrld       xmm0, xmm7
psrld       xmm5, xmm7
psrld       xmm6, xmm7
psrld       xmm4, xmm7

pand        xmm0, SIMD_SSE2_dword_byte_mask
pand        xmm5, SIMD_SSE2_dword_byte_mask
pand        xmm6, SIMD_SSE2_dword_byte_mask
pand        xmm4, SIMD_SSE2_dword_byte_mask

packuswb    xmm0, xmm5
packuswb    xmm6, xmm4

movd        xmm5, maxAlpha
pshuflw     xmm5, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufd      xmm5, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
movdqa      xmm7, xmm5

movd        xmm2, minAlpha
pshuflw     xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufd      xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
movdqa      xmm3, xmm2

movdqa      xmm4, xmm5
psubw       xmm4, xmm2
pmulhw      xmm4, SIMD_SSE2_word_div_by_14
movdqa      xmm1, xmm5
psubw       xmm1, xmm4
packuswb    xmm1, xmm1                           // ab1

pmullw      xmm5, SIMD_SSE2_word_scale66554400
pmullw      xmm7, SIMD_SSE2_word_scale11223300
pmullw      xmm2, SIMD_SSE2_word_scale11223300
pmullw      xmm3, SIMD_SSE2_word_scale66554400
paddw       xmm5, xmm2
paddw       xmm7, xmm3
pmulhw      xmm5, SIMD_SSE2_word_div_by_7
pmulhw      xmm7, SIMD_SSE2_word_div_by_7
psubw       xmm5, xmm4
psubw       xmm7, xmm4

pshufd      xmm2, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufd      xmm3, xmm5, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufd      xmm4, xmm5, R_SHUFFLE_D( 2, 2, 2, 2 )
packuswb    xmm2, xmm2                           // ab2
packuswb    xmm3, xmm3                           // ab3
packuswb    xmm4, xmm4                           // ab4

packuswb    xmm0, xmm6

pshufd      xmm5, xmm7, R_SHUFFLE_D( 2, 2, 2, 2 )
pshufd      xmm6, xmm7, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufd      xmm7, xmm7, R_SHUFFLE_D( 0, 0, 0, 0 )
packuswb    xmm5, xmm5                           // ab5
packuswb    xmm6, xmm6                           // ab6
```

```
        packuswb    xmm7, xmm7                              // ab7

        pmaxub      xmm1, xmm0
        pmaxub      xmm2, xmm0
        pmaxub      xmm3, xmm0
        pcmpeqb     xmm1, xmm0
        pcmpeqb     xmm2, xmm0
        pcmpeqb     xmm3, xmm0
        pmaxub      xmm4, xmm0
        pmaxub      xmm5, xmm0
        pmaxub      xmm6, xmm0
        pmaxub      xmm7, xmm0
        pcmpeqb     xmm4, xmm0
        pcmpeqb     xmm5, xmm0
        pcmpeqb     xmm6, xmm0
        pcmpeqb     xmm7, xmm0
        movdqa      xmm0, SIMD_SSE2_byte_8
        paddsb      xmm0, xmm1
        paddsb      xmm2, xmm3
        paddsb      xmm4, xmm5
        paddsb      xmm6, xmm7
        paddsb      xmm0, xmm2
        paddsb      xmm4, xmm6
        paddsb      xmm0, xmm4
        pand        xmm0, SIMD_SSE2_byte_7
        movdqa      xmm1, SIMD_SSE2_byte_2
        pcmpgtb     xmm1, xmm0
        pand        xmm1, SIMD_SSE2_byte_1
        pxor        xmm0, xmm1
        movdqa      xmm1, xmm0
        movdqa      xmm2, xmm0
        movdqa      xmm3, xmm0
        movdqa      xmm4, xmm0
        movdqa      xmm5, xmm0
        movdqa      xmm6, xmm0
        movdqa      xmm7, xmm0
        psrlq       xmm1,  8- 3
        psrlq       xmm2, 16- 6
        psrlq       xmm3, 24- 9
        psrlq       xmm4, 32-12
        psrlq       xmm5, 40-15
        psrlq       xmm6, 48-18
        psrlq       xmm7, 56-21
        pand        xmm0, SIMD_SSE2_dword_alpha_bit_mask0
        pand        xmm1, SIMD_SSE2_dword_alpha_bit_mask1
        pand        xmm2, SIMD_SSE2_dword_alpha_bit_mask2
        pand        xmm3, SIMD_SSE2_dword_alpha_bit_mask3
        pand        xmm4, SIMD_SSE2_dword_alpha_bit_mask4
        pand        xmm5, SIMD_SSE2_dword_alpha_bit_mask5
        pand        xmm6, SIMD_SSE2_dword_alpha_bit_mask6
        pand        xmm7, SIMD_SSE2_dword_alpha_bit_mask7
        por         xmm0, xmm1
        por         xmm2, xmm3
        por         xmm4, xmm5
        por         xmm6, xmm7
        por         xmm0, xmm2
```

```
        por         xmm4, xmm6
        por         xmm0, xmm4
        mov         esi, globalOutData
        movd        [esi+0], xmm0
        pshufd      xmm1, xmm0, R_SHUFFLE_D( 2, 3, 0, 1 )
        movd        [esi+3], xmm1
    }

    globalOutData += 6;
}

void EmitGreenIndices_SSE2( const byte *block, const int channelBitOffset,
const int minGreen, const int maxGreen ) {
    __asm {
        movd        xmm7, channelBitOffset

        mov         esi, block
        movdqa      xmm0, xmmword ptr [esi+ 0]
        movdqa      xmm5, xmmword ptr [esi+16]
        movdqa      xmm6, xmmword ptr [esi+32]
        movdqa      xmm4, xmmword ptr [esi+48]

        psrld       xmm0, xmm7
        psrld       xmm5, xmm7
        psrld       xmm6, xmm7
        psrld       xmm4, xmm7

        pand        xmm0, SIMD_SSE2_dword_byte_mask
        pand        xmm5, SIMD_SSE2_dword_byte_mask
        pand        xmm6, SIMD_SSE2_dword_byte_mask
        pand        xmm4, SIMD_SSE2_dword_byte_mask

        packuswb    xmm0, xmm5
        packuswb    xmm6, xmm4

        movd        xmm2, maxGreen
        pshuflw     xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
        pshufd      xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
        movdqa      xmm1, xmm2

        movd        xmm3, minGreen
        pshuflw     xmm3, xmm3, R_SHUFFLE_D( 0, 0, 0, 0 )
        pshufd      xmm3, xmm3, R_SHUFFLE_D( 0, 0, 0, 0 )

        movdqa      xmm4, xmm2
        psubw       xmm4, xmm3
        pmulhw      xmm4, SIMD_SSE2_word_div_by_6

        psllw       xmm2, 1
        paddw       xmm2, xmm3
        pmulhw      xmm2, SIMD_SSE2_word_div_by_3
        psubw       xmm2, xmm4
        packuswb    xmm2, xmm2                          // gb2

        psllw       xmm3, 1
        paddw       xmm3, xmm1
```

```
    pmulhw      xmm3, SIMD_SSE2_word_div_by_3
    psubw       xmm3, xmm4
    packuswb    xmm3, xmm3                          // gb3

    psubw       xmm1, xmm4
    packuswb    xmm1, xmm1                          // gb1

    packuswb    xmm0, xmm6

    pmaxub      xmm1, xmm0
    pmaxub      xmm2, xmm0
    pmaxub      xmm3, xmm0
    pcmpeqb     xmm1, xmm0
    pcmpeqb     xmm2, xmm0
    pcmpeqb     xmm3, xmm0
    movdqa      xmm0, SIMD_SSE2_byte_4
    paddsb      xmm0, xmm1
    paddsb      xmm2, xmm3
    paddsb      xmm0, xmm2
    pand        xmm0, SIMD_SSE2_byte_3
    movdqa      xmm4, SIMD_SSE2_byte_2
    pcmpgtb     xmm4, xmm0
    pand        xmm4, SIMD_SSE2_byte_1
    pxor        xmm0, xmm4
    movdqa      xmm4, xmm0
    movdqa      xmm5, xmm0
    movdqa      xmm6, xmm0
    movdqa      xmm7, xmm0
    psrlq       xmm4,  8- 2
    psrlq       xmm5, 16- 4
    psrlq       xmm6, 24- 6
    psrlq       xmm7, 32- 8
    pand        xmm4, SIMD_SSE2_dword_color_bit_mask1
    pand        xmm5, SIMD_SSE2_dword_color_bit_mask2
    pand        xmm6, SIMD_SSE2_dword_color_bit_mask3
    pand        xmm7, SIMD_SSE2_dword_color_bit_mask4
    por         xmm5, xmm4
    por         xmm7, xmm6
    por         xmm7, xmm5
    movdqa      xmm4, xmm0
    movdqa      xmm5, xmm0
    movdqa      xmm6, xmm0
    psrlq       xmm4, 40-10
    psrlq       xmm5, 48-12
    psrlq       xmm6, 56-14
    pand        xmm0, SIMD_SSE2_dword_color_bit_mask0
    pand        xmm4, SIMD_SSE2_dword_color_bit_mask5
    pand        xmm5, SIMD_SSE2_dword_color_bit_mask6
    pand        xmm6, SIMD_SSE2_dword_color_bit_mask7
    por         xmm4, xmm5
    por         xmm0, xmm6
    por         xmm7, xmm4
    por         xmm7, xmm0
    mov         esi, globalOutData
    movd        [esi+0], xmm7
    pshufd      xmm6, xmm7, R_SHUFFLE_D( 2, 3, 0, 1 )
```

```
        movd        [esi+2], xmm6
    }

    globalOutData += 4;
}

bool CompressNormalMapDXT5_SSE2( const byte *inBuf, byte *outBuf, int width,
int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte normalMin[4] );
    ALIGN16( byte normalMax[4] );

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock_SSE2( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox_SSE2( block, normalMin, normalMax );
            InsetNormalsBBoxDXT5_SSE2( normalMin, normalMax );

            // Write out Nx into alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
            EmitAlphaIndices_SSE2( block, 0*8, normalMin[0], normalMax[0] );

            // Write out Ny into green channel.
            EmitUShort( NormalYTo565( normalMax[1] ) );
            EmitUShort( NormalYTo565( normalMin[1] ) );
            EmitGreenIndices_SSE2( block, 1*8, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}

void CompressNormalMap3Dc_SSE2( const byte *inBuf, byte *outBuf, int width,
int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte normalMin[4] );
    ALIGN16( byte normalMax[4] );

    globalOutData = outBuf;

    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {

            ExtractBlock_SSE2( inBuf + i * 4, width, block );

            GetMinMaxNormalsBBox_SSE2( block, normalMin, normalMax );
            InsetNormalsBBox3Dc_SSE2( normalMin, normalMax );

            // Write out Nx as an alpha channel.
            EmitByte( normalMax[0] );
            EmitByte( normalMin[0] );
```

```
            EmitAlphaIndices_SSE2( block, 0*8, normalMin[0], normalMax[0] );

            // Write out Ny as an alpha channel.
            EmitByte( normalMax[1] );
            EmitByte( normalMin[1] );
            EmitAlphaIndices_SSE2( block, 1*8, normalMin[1], normalMax[1] );
        }
    }

    outputBytes = outData - outBuf;
}
```

Appendix D

```
/*
    Real-time DXT1 & YCoCg DXT5 compression (Cg 2.0)
    Copyright (c) NVIDIA Corporation.
    Written by: Ignacio Castano

    Thanks to JMP van Waveren, Simon Green, Eric Werness, Simon Brown

    Permission is hereby granted, free of charge, to any person
    obtaining a copy of this software and associated documentation
    files (the "Software"), to deal in the Software without
    restriction, including without limitation the rights to use,
    copy, modify, merge, publish, distribute, sublicense, and/or sell
    copies of the Software, and to permit persons to whom the
    Software is furnished to do so, subject to the following
    conditions:

    The above copyright notice and this permission notice shall be
    included in all copies or substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
    EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
    OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
    NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
    HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
    WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
    OTHER DEALINGS IN THE SOFTWARE.
*/

// vertex program
void compress_vp(float4 pos : POSITION,
                      float2 texcoord : TEXCOORD0,
                      out float4 hpos : POSITION,
                      out float2 o_texcoord : TEXCOORD0
                      )
{
    o_texcoord = texcoord;
    hpos = pos;
```

```
}

typedef unsigned int uint;
typedef unsigned int2 uint2;
typedef unsigned int4 uint4;

void ExtractColorBlockXY(out float2 col[16], sampler2D image, float2
texcoord, float2 imageSize)
{
#if 0
    float2 texelSize = (1.0f / imageSize);
    texcoord -= texelSize * 2;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            col[i*4+j] = tex2D(image, texcoord + float2(j, i) *
texelSize).rg;
        }
    }
#else
    // use TXF instruction (integer coordinates with offset)
    // note offsets must be constant
    //int4 base = int4(wpos*4-2, 0, 0);
    int4 base = int4(texcoord * imageSize - 1.5, 0, 0);
    col[0] = tex2Dfetch(image, base, int2(0, 0)).rg;
    col[1] = tex2Dfetch(image, base, int2(1, 0)).rg;
    col[2] = tex2Dfetch(image, base, int2(2, 0)).rg;
    col[3] = tex2Dfetch(image, base, int2(3, 0)).rg;
    col[4] = tex2Dfetch(image, base, int2(0, 1)).rg;
    col[5] = tex2Dfetch(image, base, int2(1, 1)).rg;
    col[6] = tex2Dfetch(image, base, int2(2, 1)).rg;
    col[7] = tex2Dfetch(image, base, int2(3, 1)).rg;
    col[8] = tex2Dfetch(image, base, int2(0, 2)).rg;
    col[9] = tex2Dfetch(image, base, int2(1, 2)).rg;
    col[10] = tex2Dfetch(image, base, int2(2, 2)).rg;
    col[11] = tex2Dfetch(image, base, int2(3, 2)).rg;
    col[12] = tex2Dfetch(image, base, int2(0, 3)).rg;
    col[13] = tex2Dfetch(image, base, int2(1, 3)).rg;
    col[14] = tex2Dfetch(image, base, int2(2, 3)).rg;
    col[15] = tex2Dfetch(image, base, int2(3, 3)).rg;
#endif
}

// find minimum and maximum colors based on bounding box in color space
void FindMinMaxColorsBox(float2 block[16], out float2 mincol, out float2
maxcol)
{
    mincol = block[0];
    maxcol = block[0];

    for (int i = 1; i < 16; i++) {
        mincol = min(mincol, block[i]);
        maxcol = max(maxcol, block[i]);
    }
}

void InsetNormalsBBoxDXT5(in out float2 mincol, in out float2 maxcol)
```

```
{
    float2 inset;
    inset.x = (maxcol.x - mincol.x) / 32.0 - (16.0 / 255.0) / 32.0;      //
ALPHA scale-bias.
    inset.y = (maxcol.y - mincol.y) / 16.0 - (8.0 / 255.0) / 16;      //
GREEN scale-bias.
    mincol = saturate(mincol + inset);
    maxcol = saturate(maxcol - inset);
}

void InsetNormalsBBoxLATC(in out float2 mincol, in out float2 maxcol)
{
    float2 inset = (maxcol - mincol) / 32.0 - (16.0 / 255.0) / 32.0;  //
ALPHA scale-bias.
    mincol = saturate(mincol + inset);
    maxcol = saturate(maxcol - inset);
}


uint EmitGreenEndPoints(in out float ming, in out float maxg)
{
    uint c0 = round(ming * 63);
    uint c1 = round(maxg * 63);

    ming = float((c0 << 2) | (c0 >> 4)) * (1.0 / 255.0);
    maxg = float((c1 << 2) | (c1 >> 4)) * (1.0 / 255.0);

    return (c0 << 21) | (c1 << 5);
}

#if 1

uint EmitGreenIndices(float2 block[16], float minGreen, float maxGreen)
{
    const int GREEN_RANGE = 3;

    float bias = maxGreen + (maxGreen - minGreen) / (2.0 * GREEN_RANGE);
    float scale = 1.0f / (maxGreen - minGreen);

    // Compute indices
    uint indices = 0;
    for (int i = 0; i < 16; i++)
    {
        uint index = saturate((bias - block[i].y) * scale) * GREEN_RANGE;
        indices |= index << (i * 2);
    }

    uint i0 = (indices & 0x55555555);
    uint i1 = (indices & 0xAAAAAAAA) >> 1;
    indices = ((i0 ^ i1) << 1) | i1;

    // Output indices
    return indices;
}

#else
```

```
uint EmitGreenIndices(float2 block[16], float minGreen, float maxGreen)
{
    const int GREEN_RANGE = 3;

    float mid = (maxGreen - minGreen) / (2 * GREEN_RANGE);

    float yb1 = minGreen + mid;
    float yb2 = (2 * maxGreen + 1 * minGreen) / GREEN_RANGE + mid;
    float yb3 = (1 * maxGreen + 2 * minGreen) / GREEN_RANGE + mid;

    // Compute indices
    uint indices = 0;
    for (int i = 0; i < 16; i++)
    {
        float y = block[i].y;

        uint index = (y <= yb1);
        index += (y <= yb2);
        index += (y <= yb3);

        indices |= index << (i * 2);
    }

    uint i0 = (indices & 0x55555555);
    uint i1 = (indices & 0xAAAAAAAA) >> 1;
    indices = ((i0 ^ i1) << 1) | i1;


    // Output indices
    return indices;
}

#endif


uint EmitAlphaEndPoints(float mincol, float maxcol)
{
    uint c0 = round(mincol * 255);
    uint c1 = round(maxcol * 255);

    return (c0 << 8) | c1;
}

uint2 EmitAlphaIndices(float2 block[16], float minAlpha, float maxAlpha)
{
    const int ALPHA_RANGE = 7;

    float bias = maxAlpha + (maxAlpha - minAlpha) / (2.0 * ALPHA_RANGE);
    float scale = 1.0f / (maxAlpha - minAlpha);

    uint2 indices = 0;

    for (int i = 0; i < 6; i++)
    {
        uint index = saturate((bias - block[i].x) * scale) * ALPHA_RANGE;
```

```
        indices.x |= index << (3 * i);
    }

    for (int i = 6; i < 16; i++)
    {
        uint index = saturate((bias - block[i].x) * scale) * ALPHA_RANGE;
        indices.y |= index << (3 * i - 18);
    }

    uint2 i0 = (indices >> 0) & 0x09249249;
    uint2 i1 = (indices >> 1) & 0x09249249;
    uint2 i2 = (indices >> 2) & 0x09249249;

    i2 ^= i0 & i1;
    i1 ^= i0;
    i0 ^= (i1 | i2);

    indices.x = (i2.x << 2) | (i1.x << 1) | i0.x;
    indices.y = (((i2.y << 2) | (i1.y << 1) | i0.y) << 2) | (indices.x >>
16);
    indices.x <<= 16;

    return indices;
}

uint2 EmitLuminanceIndices(float2 block[16], float minAlpha, float maxAlpha)
{
    const int ALPHA_RANGE = 7;

    float bias = maxAlpha + (maxAlpha - minAlpha) / (2.0 * ALPHA_RANGE);
    float scale = 1.0f / (maxAlpha - minAlpha);

    uint2 indices = 0;

    for (int i = 0; i < 6; i++)
    {
        uint index = saturate((bias - block[i].y) * scale) * ALPHA_RANGE;
        indices.x |= index << (3 * i);
    }

    for (int i = 6; i < 16; i++)
    {
        uint index = saturate((bias - block[i].y) * scale) * ALPHA_RANGE;
        indices.y |= index << (3 * i - 18);
    }

    uint2 i0 = (indices >> 0) & 0x09249249;
    uint2 i1 = (indices >> 1) & 0x09249249;
    uint2 i2 = (indices >> 2) & 0x09249249;

    i2 ^= i0 & i1;
    i1 ^= i0;
    i0 ^= (i1 | i2);

    indices.x = (i2.x << 2) | (i1.x << 1) | i0.x;
    indices.y = (((i2.y << 2) | (i1.y << 1) | i0.y) << 2) | (indices.x >>
```

```
16);
    indices.x <<= 16;

    return indices;
}

// compress a 4x4 block to DXT5nm format
// integer version, renders to 4 x int32 buffer
uint4 compress_NormalDXT5_fp(float2 texcoord : TEXCOORD0,
                      uniform sampler2D image,
                      uniform float2 imageSize = { 512.0, 512.0 }
                      ) : COLOR
{
    // read block
    float2 block[16];
    ExtractColorBlockXY(block, image, texcoord, imageSize);

    // find min and max colors
    float2 mincol, maxcol;
    FindMinMaxColorsBox(block, mincol, maxcol);
    InsetNormalsBBoxDXT5(mincol, maxcol);

    uint4 output;

    // Output X in DXT5 green channel.
    output.z = EmitGreenEndPoints(mincol.y, maxcol.y);
    output.w = EmitGreenIndices(block, mincol.y, maxcol.y);

    // Output Y in DXT5 alpha block.
    output.x = EmitAlphaEndPoints(mincol.x, maxcol.x);

    uint2 indices = EmitAlphaIndices(block, mincol.x, maxcol.x);
    output.x |= indices.x;
    output.y = indices.y;

    return output;
}


// compress a 4x4 block to LATC format
// integer version, renders to 4 x int32 buffer
uint4 compress_NormalLATC_fp(float2 texcoord : TEXCOORD0,
                      uniform sampler2D image,
                      uniform float2 imageSize = { 512.0, 512.0 }
                      ) : COLOR
{
    //imageSize = tex2Dsize(image, texcoord);

    // read block
    float2 block[16];
    ExtractColorBlockXY(block, image, texcoord, imageSize);

    // find min and max colors
    float2 mincol, maxcol;
    FindMinMaxColorsBox(block, mincol, maxcol);
    InsetNormalsBBoxLATC(mincol, maxcol);
```

```
    uint4 output;

    // Output Ny as an alpha block.
    output.x = EmitAlphaEndPoints(mincol.y, maxcol.y);

    uint2 indices = EmitLuminanceIndices(block, mincol.y, maxcol.y);
    output.x |= indices.x;
    output.y = indices.y;

    // Output Nx as an alpha block.
    output.z = EmitAlphaEndPoints(mincol.x, maxcol.x);

    indices = EmitAlphaIndices(block, mincol.x, maxcol.x);
    output.z |= indices.x;
    output.w = indices.y;

    return output;
}

uniform float3 lightDirection;
uniform bool reconstructNormal = true;
uniform bool displayNormal = true;
uniform bool displayError = false;
uniform float errorScale = 4.0f;

uniform sampler2D image : TEXUNIT0;
uniform sampler2D originalImage : TEXUNIT1;

float3 shadeNormal(float3 N)
{
    float3 L = normalize(lightDirection);
    float3 R = reflect(float3(0, 0, -1), N);

    float diffuse = saturate(dot (N, L));
    float specular = pow(saturate(dot(R, L)), 12);

    return 0.7 * diffuse + 0.5 * specular;
}

// Draw reconstructed normals.
float4 display_fp(float2 texcoord : TEXCOORD0) : COLOR
{
    float3 N;
    if (reconstructNormal)
    {
        N.xy = 2 * tex2D(image, texcoord).wy - 1;
        N.z = sqrt(saturate(1 - N.x * N.x - N.y * N.y));
    }
    else
    {
        N = normalize(2 * tex2D(image, texcoord).xyz - 1);
    }

    if (displayError)
    {
```

```
        float3 originalNormal = normalize(2 * tex2D(originalImage,
texcoord).xyz - 1);

        if (displayNormal)
        {
            float3 diff = (N - originalNormal) * errorScale;
            return float4(diff, 1);
        }
        else
        {
            float3 diff = abs(shadeNormal(N) - shadeNormal(originalNormal)) *
errorScale;
            return float4(diff, 1);
        }
    }
    else
    {
        if (displayNormal)
        {
            return float4(0.5 * N + 0.5, 1);
        }
        else
        {
            return float4(shadeNormal(N), 1);
        }
    }
}

// Draw geometry normals.
uniform float4x4 mvp : ModelViewProjection;
uniform float3x3 mvit : ModelViewInverseTranspose;

void display_object_vp(float4 pos : POSITION,
                      float3 normal : NORMAL,
                      out float4 hpos : POSITION,
                      out float3 o_normal : TEXCOORD0)
{
    hpos = mul(pos, mvp);
    o_normal = mul(normal, mvit);
}

float4 display_object_fp(float3 N : TEXCOORD0) : COLOR
{
    N = normalize(N);
    return float4(0.5 * N + 0.5, 1);
}
```