



Technical Report

Bump Map Compression

DEVELOPMENT

Bump Map Compression

This document describes several techniques for compressing bump maps.

Simon Green
devrelfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

10/21/2004



Almost all new video games are using bump mapping to some extent. While there is no doubt this improves the realism and visual complexity of scenes, the additional video memory consumed by having bump map textures for every surface can become problematic. The increasing popularity of the techniques that automatically generate normal maps from high resolution models further increases storage requirements because in this case normal maps cover the whole of each model without any reuse.

The DXT compression formats do a good job of compressing color textures, but were not designed for compressing normal map images. Now that programmable pixel shader hardware is commonplace, there are several different bump map storage formats available to developers, each with different trade-offs between storage and shader complexity.

This document is accompanied by an FX Composer project that you can use to experiment with the bump map compression techniques described here. This is available for download from the NVIDIA developer website.

Overview

Developers face several choices when implementing bump mapping.

- ❑ What bump map representation to use – a traditional RGB normalmap, a 2 component gradient map or a single channel heightfield? This is often dependent on what tools are used to generate the bump maps.
- ❑ Are there additional data channels that need to be stored with the bump map? For example a specular color map, specular exponent (shininess) map, or height map for parallax bump mapping.
- ❑ Which texture format to use? Signed or unsigned? 1, 2, 3 or 4 component? Compressed or uncompressed?
- ❑ If using normal maps, should the Z component be stored in the map, or calculated in the shader?

Bump Map Representations

The first choice developers face when implementing bump mapping is what representation to use for their bump maps. Most developers today use tangent-space normal maps, but there are other representations possible, as we discuss in the section “alternative bump mapping techniques” below.

Hand painted or photographically-derived bump maps usually start life as a greyscale heightfield image, where black indicates the minimum height and white indicates the maximum height. These heightfield images are typically processed using tools such as NVIDIA’s Photoshop normal map filter to produce a 3 channel RGB image. Normal maps are usually stored in a scaled and biased format so that the [-1, 1] range of the normal vectors is mapped to the [0, 255] color range.

Next we discuss the various ways in which normal maps can be compressed.

DXT Compression

The DXT (S3TC) texture formats use a form of lossy vector quantization to compress texture images by a ratio of 4:1 or 6:1. These formats are a standard part of Direct3D, and are available in OpenGL via the ARB_texture_compression and GL_EXT_texture_compression_s3tc extensions.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/ProgrammingGuide/GettingStarted/Direct3DTextures/compressed/compressedtextureformats.asp

http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_texture_compression.txt

http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_texture_compression_s3tc.txt

Although the DXT formats do a reasonably good job of compressing color images, they do not by default produce good results with typical normal map images. This is because the DXT compression algorithm makes the assumption that images have smoothly changing colors, whereas normal maps often have high frequency content (particularly when they are generated from geometry). The reference colors used in DXT are also stored in 16-bit format, which is not enough to accurately represent all possible normal directions. The block artifacts caused by the 4x4 blocks used in DXT compression tend to be much easier to see when the images are used in bump mapping. Compression artifacts are also amplified when compressed normal maps are used with a shader that includes specular reflections. The power function used in the specular reflection calculation multiplies the effect such that even a change in the least significant bit of the normal can cause a large difference in the final color.

Having said this, it is possible to get good results from DXT compression for normal maps using a few simple tricks.

DXT1 Format

The DXT1 format compresses RGB images by a factor of 6:1, for an average of 4 bits per texel. Unfortunately DXT1 does not usually produce good results for normal maps. The artifacts can be seen below in the form of rectangular blocks along high-contrast edges. Renormalizing the normal after the texture fetch improves the quality slightly, but the results are not acceptable for most bump maps.



Fig 1. Uncompressed

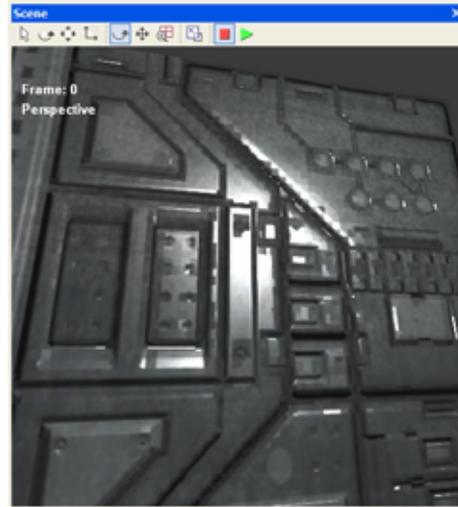


Fig 2. DXT1 compression



Fig 3. DXT1 compression with renormalization

DXT5 Format

The DXT5 format compresses an RGBA image by a factor of 4:1, using a byte per texel on average. The advantage of DXT5 over DXT1 is that it supports an alpha channel.

It is thus possible to use DXT5 to store normal maps with higher quality by moving the X component of the normal from the red channel to the alpha channel. This is the method used in Doom3. Since the alpha is compressed separately this produces much better results than the DXT1 format. We use swizzling (or a move instruction) in the shader to move the alpha channel back to the x component of the normal:

```
float3 normalMap = tex2D(normalSampler, IN.TexCoord.xy).agb*2-1;
```

It is also possible to reconstruct the Z component of the normal in the shader (as described below), which will provide even higher quality. It is worth noting that the green channel has better precision because the reference colors in DXT are stored in 5_6_5 format.

It may be possible to store other unrelated data in the spare channels of the DXT format. For example, if you have monochrome gloss and shininess maps, you could store these in the red and blue channels. If you are not using all the channels in a DXT format, it is best to zero out the unused color channels since this gives the compressor more flexibility in choosing the closest colors.



Fig 4. DXT5 compression



Fig 5. DX5 with generated Z

A8L8 Texture Format

Direct3D includes a two-component 8-bit texture format called “A8L8” (the equivalent signed format is “V8U8”). In OpenGL this format is exposed via the GL_HILO8_NV format which is part of the NV_texture_shader3 extension.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/d3d/enums/d3dformat.asp

http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_texture_shader3.txt

This format is intended for storing monochrome luminance images with an alpha channel, but we can also use it to store normal maps if we calculate the missing Z component in the shader as described below. This allows us to store an uncompressed bump map using only 2 bytes per texel.

If using the NVIDIA Photoshop texture tools, you should unselect the option “calculate luminance” in the Config window when saving in A8L8 format, otherwise the tool will attempt to average the RGB channels to calculate the luminance channel. If the option is unselected the tool uses the red channel as luminance.

Another alternative is to use the CxV8U8 format, which calculates the Z component automatically. GeForce 3 / 4 and GeForce FX GPUs include dedicated hardware for this format, so on these parts it may be faster than to use this format rather than deriving Z in the shader.

Deriving the Z Component of Unit Normals

Since tangent space normal maps always have a positive Z component, and we know the normals should always be unit length, it is possible to store the X and Y components of the normal in the 2 components of an A8L8 texture, and then reconstruct the Z component in the shader using this simple calculation:

```
length(N) = 1
sqrt(N.x*N.x + N.y*N.y + N.z*N.z) = 1
N.x*N.x + N.y*N.y + N.z*N.z = 1
N.z*N.z = 1 - N.x*N.x - N.y*N.y;
```

```
N.z = sqrt(1 - N.x*N.x - N.y*N.y);
```

This compiles to about 4 pixel shader instructions, so the additional performance cost is relatively small.



Fig. 6 – Uncompressed



Fig. 7 – A8L8 format with generated Z

Note that generating the Z component actually produces better results than the uncompressed normal map, because it generates a normal that is always unit length.

Renormalizing Normal Maps

Linear filtering of normal map textures can cause the normal to become less than unit length, particularly when the texture is heavily magnified. This can cause specular highlights to be missed. Renormalizing the normal after the texture fetch can dramatically improve the quality of bump mapping. This is particularly attractive on GeForce 6 series GPUs since these include a fast partial precision normalize instruction.

The diagram below illustrates how linear interpolation can lead to non unit-length normals.

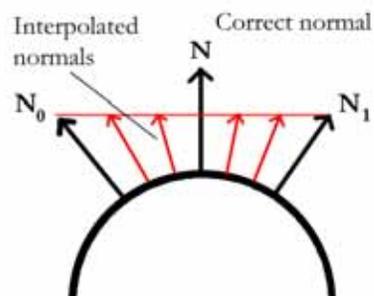




Fig. 8 – No normalization



Fig. 9 – With normalization

The only problem with renormalization is that it can cause aliasing of specular highlights. These artifacts are particularly noticeable when the viewpoint is moving. One solution to this problem is described in the NVIDIA paper “Mipmapping Normal Maps”, which is available on the developer website.

http://developer.nvidia.com/object/mipmapping_normal_maps.html

This technique uses the denormalization of normal vectors caused by interpolation as a measure of the minification of the texture. It uses this value to dim and broaden the size of the specular highlight to reduce aliasing. Note that this means this technique is incompatible with 2 component normal map formats that derive the Z component in the shader, since in this case the normals will always have a length of one.



Fig.10 – Bump map showing specular aliasing

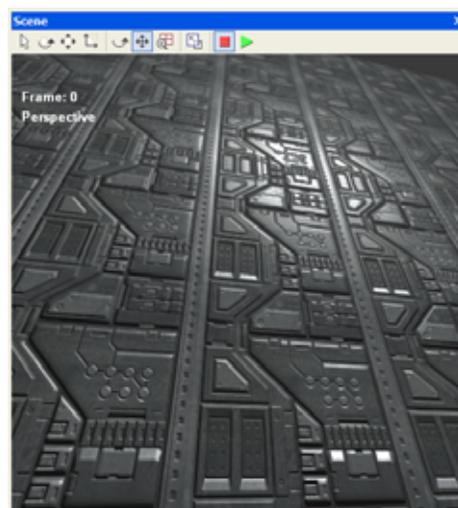


Fig 11. With “Toksvig” factor

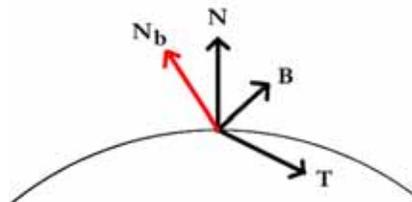
Alternative Bump Mapping Techniques

Another approach to reducing bump map storage is to not use tangent-space bump mapping at all. The original motivation for tangent-space bump mapping was that it requires little calculation at the per-pixel level. The tangent-space light and half angle vectors can be calculated per-vertex and interpolated, and then since the normal we fetch from the normal map is already in tangent space and approximately unit-length, all that is required per-pixel is two dot products and a power function to produce the diffuse and specular reflection terms.

Now that pixel shader hardware is more flexible and much higher performance, other approaches are possible. Bump mapping as originally described by [Blinn] does not use normal maps at all, but instead uses the gradient of the bump map heightfield to perturb the normal in the direction of the surface derivatives.

We can implement this method in a pixel shader as shown below. The bump map texture is stored as a two-component image containing the gradients of the bump map heightfield in X and Y. We can use the A8L8 texture format for this purpose, which only requires 2 bytes per texel. For the best quality the gradients should be scaled up so that they make the best use of the available 8-bit range. This is important because 8-bit textures are usually only interpolated at 8-bit precision. For testing purposes you can just use the X and Y components of a normal map without too many problems.

The shader simply perturbs the interpolated normal in the direction of the binormal and tangent vectors based on the values from the bump map. This shader is more expensive than regular tangent-space bump mapping since it requires a few more multiplies and a normalization. But an advantage of this method is that it allows finer control over the magnitude of the bump mapping effect via the “bumpScale” parameter. It may be interesting to animate this control, or even control the scale per-pixel via another texture channel. A similar effect can be achieved with tangent space bump mapping by scaling the tangent and binormal vectors.



```

// fetch bump map
float2 bumpMap = tex2D(normalSampler, IN.TexCoord.xy).rg*2-1;
// perturb normal in tangent directions
float3 Nb = IN.Normal + bumpScale*(bumpMap.x * In.Tangent +
                                   bumpMap.y * IN.Binormal);

Nb = normalize(Nb);

```

Strictly speaking the interpolated normal, tangent and binormal vectors should also be renormalized per-pixel, but in practice it is difficult see any difference.

It is also possible to use just a single channel heightfield as the bump map representation, and then use 3 texture lookups to estimate the gradients based on the neighboring heights. This is probably the most efficient bump map storage format, but requires a significant amount of work in the shader.



Fig 12. Tangent space bump mapping



Fig 13. Blinn-style bump mapping

Recommendations

For applications that want the best quality and can afford the additional pixel shader instructions to generate Z, we recommend using the A8L8 or V8U8 formats, which require 2 bytes per texel. This format is often faster than DXT5, despite the larger footprint.

For applications that want the best possible compression and do not mind some artifacts, we recommend using the DXT5 format and storing the X component in alpha. This requires just 1 byte per texel. It has the additional advantage that you can use the spare channels for other data. For improved quality you can renormalize the normal in the shader.

Bibliography

J. F. Blinn, Simulation of Wrinkled Surfaces, In Proceedings SIGGRAPH 78, pp. 286-292, 1978

M. J. Kilgard, A Practical and Robust Bump-mapping Technique for Today's GPUs Game Developers Conference, Advanced OpenGL Game Development. 2000.

M. Peercy, A. Airey, B. Cabral, Efficient Bump Mapping Hardware, In proceedings of SIGGRAPH 97, August 3-8, pp 303-306. 1997.



ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce are trademarks of NVIDIA Corporation.

Microsoft, Windows, the Windows logo, and DirectX are registered trademarks of Microsoft Corporation.

OpenGL is a trademark of SGI. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Copyright NVIDIA Corporation 2004



nVIDIA.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com