Self-shadowing Bumpmap using 3D Texture Hardware

Tom Forsyth, Mucky Foot Productions Ltd. TomF@muckyfoot.com

Abstract

Self-shadowing bumpmaps add realism and depth to scenes and provide important visual cues. Previous techniques have shown self-shadowing at interactive rates, but at a substantial cost over standard normal mapped bump mapping. This paper uses volume textures on consumer graphics hardware, using few enough resources that they can be easily added to existing normal map bump mapping renderers. Also explored are the ways the technique can be scaled according to available memory constraints.

1. Introduction

Bumpmaps are common on today's consumer hardware, and widely accepted and used in applications to add surface detail without additional geometry. However, without self-shadowing they lack many of the visual cues important for immersion.

Horizon mapping[1] is a particular implementation of self-shadowing on bump-mapped surfaces. The angle from each point on the surface to the point's local horizon is encoded into a separate texture for each texel of the bumpmap. Sloan[2] presents a method of rendering horizon maps on consumer hardware where multiple horizon maps are stored and interpolated using multiple rendering passes.

These papers together provide a firm theoretical base for horizon mapping over curved surfaces. However, their implementations require considerable effort and rendering time over existing normal mapped rendering. Self-shadowing is currently not considered a required feature in rendering pipelines, and it needs to have a low impact on the rendering effort and speed to be used widely.

Kautz[3] introduces an alternative idea of using an ellipsoidal cone at each texel to approximate the set of unshadowed light directions. This also requires a large number of texel reads and/or rendering passes, and a lot of computation at each pixel.

2. Horizon Maps

Given a distant light illuminating a surface from a certain compass bearing, a horizon map stores the elevation from each point on the surface to the point's apparent horizon. If the elevation of the light is greater than this angle, the point is drawn lit, otherwise it is drawn shadowed. To generalise to a light coming from any bearing, multiple horizon maps are generated, each storing the horizon map for a light at a particular bearing. When rendering, the relevant maps are blended together to produce an interpolated horizon elevation for the given bearing, which is then compared with the light's elevation.

For curved surfaces, the techniques used in normal map bump mapping[4] transform the light into surface-local co-ordinates at each vertex. The surface-local bearing is then interpolated and at each pixel the relevant horizon maps are read and compared with the interpolated surface-local elevation.

The curvature of the geometry will affect how much and how far each bump casts shadows, and it is usually easiest to take this into consideration when generating the horizon maps. This problem is common to all self-shadowing bumpmap algorithms. In practice, if this effect is visible then the bumps concerned are usually large enough to be converted to actual geometry.

There is a discontinuity in the bearing/elevation parameterisation where the light passes directly overhead. However, near this discontinuity, very little if any of the surface will be in shadow, and the errors in interpolation will at least preserve this where they do go wrong. As long as the surface is tessellated to a sufficient degree, the errors in interpolation are not noticeable. This minimum degree of tessellation is a requirement for most other techniques used in illumination such as normal mapping, so this is not a harsh restriction.

3. Hardware Implementation

At each pixel, the closest few horizon maps must be blended together to provide an interpolated horizon elevation value. Sloan[2] uses multiple passes, one for each horizon map, and a set of basis textures to achieve this blending on conventional hardware. However, this requires a lot of fill rate and multiple rendering calls. Multi-texture hardware can be used to improve the speed of Sloan's method and reduce the number of passes, but all the texels of all the bearings are still being read, which is costly. The different numbers of textures supported by different hardware also makes the code difficult to write and maintain.

Three-dimensional volume textures are now common on consumer hardware. The 2D horizon maps, one for each light bearing, are placed in a stack to form a 3D texture volume, and the bearing of the light at each pixel is used as the third texture co-ordinate.

The addressing of this co-ordinate is set to a special wrap mode, so that the bearing value can wrap around from 0 to 1 correctly. This mode is supported by many vendors under Direct3D. At the triangle rasterisation set-up stage after any vertex shaders are applied, the texture co-ordinates of each vertex are truncated to the 0-1 range. Then a value of 1 may be added or subtracted to two of the vertices, so that they lie within 0.5 of the third vertex. The triangle is then rasterised using the standard wrap texture addressing mode. Thus a triangle with texture co-ordinates of 0.1, 0.2 and 0.9 will have the third vertex's texture co-ordinate wrapped to –0.1, so that rasterisation would take the shortest route if the texture were mapped to a cylinder[5]

Using this wrap mode, the addressing and filtering logic of the texture unit performs the selection and blending between horizon maps. This reduces the technique to a single pass, however many horizon maps are used, and at each pixel only the necessary two maps are sampled, reducing memory bandwidth substantially.

The blending available is not as controllable as Sloan's basis textures, but in practice the difference in quality for the same number of horizon maps is small. The extra efficiency of this method allows more horizon maps to be used for similar frame rates, improving the quality above Sloan's method. The disadvantage is an increase in storage space required for the extra horizon maps.

4. Shader Integration

Existing pipelines already transform the light vector into surface-local space at each vertex, then linearly interpolate this light vector across each polygon. The extra steps required to add self-shadowing are simple to add, and require no large changes.

4.1 Vertex Shader

The additional step required at each vertex is to calculate bearing and elevation from the surface-local light vector. This can be costly since it involves inverse trigonometry. Some pipelines have fast approximate equivalents, but even this is not actually necessary. The elevation value is only interpolated and then used in a boolean comparison, so it can be remapped within reason to any monotonic function. The tangent of the elevation angle is simple to calculate and works well. Although the tangent grows large at high elevations, as mentioned previously very little if any of the surface will be shadowed at these large values, so the values can be clamped quite early without noticeable errors. A maximum slope can be calculated for any given bumpmap heightfield, giving the maximum tangent that will need to be stored, and allowing a global scale to be applied to fit this into the limited texture-channel resolution. This allows fine detail for shallow or smooth surfaces, and coarse detail but high dynamic range for rougher surfaces.

An alternative to storing the tangent of elevation in the horizon map is storing the sine of elevation, which is the same as the dot(light,normal) value already calculated for use by normal-mapping. Although this should work equally well in theory, with the advantage of a fixed range (unlike the tangent), in practice the linear interpolation between horizon maps works visibly better when using the tangent. However, if speed is paramount, it is a good alternative.

Similarly, the bearing simply selects and interpolates between the nearest two horizon maps. The actual bearing function used must be continuous and have a range of 0 to 1. However, beyond that the errors inherent in sampling horizon values at a relatively small number of directions (typically 8 or 16 in a full circle) are larger than any errors from using a non-linear function for interpolation. I have used the following code with good results:

```
float angle;
if (vec.x >= vec.y) {
     if (vec.x >= -vec.y) {
          // +ve X quadrant. Map from 0.75 (y=-0.707) to 1.0 (y=+0.707)
          angle = vec.y * (0.25 / 1.414) + 0.75 + (0.707 * (0.25 / 1.414));
     } else {
          // -ve Y quadrant. Map from 0.5 (x=-0.707) to 0.75 (x=+0.707)
          angle = vec.x * (0.25 / 1.414) + 0.5 + (0.707 * (0.25 / 1.414));
     }
} else {
     if (vec.x >= -vec.y) {
        // +ve Y quadrant. Map from 0 (x=+0.707) to 0.25 (x=-0.707)
        angle = vec.x * (-0.25 / 1.414) + 0.0 + (-0.707 * (-0.25 / 1.414));
    } else {
        // -ve X quadrant. Map from 0.25 (y=+0.707) to 0.5 (y=-0.707)
        angle = vec.y * (-0.25 / 1.414) + 0.25 + (-0.707 * (-0.25 / 1.414));
    }
}
```

Using DirectX8.1 Vertex Shaders, this code can be reduced to seven instructions. A typical Vertex Shader for a curved-surface normal map renderer will use around thirty or forty instructions, so this is a very reasonable additional cost.

4.2 Pixel Shaders

The additional operations required for the pixel pipeline are equally simple. A standard normal map bump map pipeline will find the dot-product of the surface-local light vector with the normal map and add this lighting to the scene. Here, the horizon map volume texture for the surface is bound to a spare texture stage. The surface-local light bearing calculated at each vertex is used as the third texture co-ordinate to select and blend between the horizon map "slices" of this texture. The elevation read is compared with the light elevation calculated at each vertex. If the light's elevation is greater, the normal-mapped light's contribution is added to the pixel's lighting.

Using the DX8.1 pixel shader pipeline, this requires no more than two extra instructions (and sometimes fewer) of the eight available. An example of a typical normal map bump map with self-shadowing is as follows:

```
;v0.rgb = bumpmapped light direction
;v1.rgb = bumpmapped light colour
;v0.a = bumpmapped light elevation
ps.1.1                               ;pixel shader version number
tex t0                               ;diffuse map
tex t1                               ;normal map
tex t2                               ;volume texture horizon map
texcoord t3                          ;non-bumpmapped vertex lighting
dp3_sat r1.rgb, t0_bx2, v0_bx2       ;perform normal-map lighting
mad r1.rgb, r1, v1, t3               ;add normal map light to vertex lighting
+sub r0.a, v0, t2_bias               ;r1.a = light_elev - (horizon_elev - 0.5)
cnd r0.rgb, r0.a, r1, t3             ;r1.a>0.5 ? combined light : just vertex lights
```

The last two instructions perform all the steps needed for self-shadowing. The "sub" instruction operates only on the alpha channel, and so can be run in parallel with the previous "mad" instruction (indicated by the "+" sign before it), and does not actually take up an extra instruction slot in this case.

5. Memory Use

While the shader additions are simple, even at 8 bits per texel the memory required to store the horizon maps is considerable.

Most hardware requires that volume textures be a power of two in every dimension. Typically, 8 bearings will be stored. For visibly higher quality close-up, 16 bearings can be used, but going to 32 does not result in much better quality in most cases. Using only 4 bearings is enough for some textures, especially general "roughness" maps without identifiable features. See figures 1 to 4.

[8 and 16 are the most important. If there is space, the 4-sample one would be good. If there is space left, or to make up a whole page/panel of illustrations or something, including the 32-sample one would be nice just to make the point that it's very similar to the 16-sample one]

The number of bearings sampled is simple to scale at runtime and requires no rendering pipeline changes except for the use of the correct volume texture. This allows good scalability of memory use according to distance and detail level.

The resolution of the texture in the other two dimensions is also easily altered without other rendering changes. Reducing the resolution of the horizon maps by a factor of two in each dimension has very little visible effect on quality, and reducing by a factor of four in each dimension has a visible but small effect on the smoothness of the shadow edge. The normal map performs most of the high-frequency lighting effects, while the horizon map is used to darken more substantial portions of the image. A reduction of eight times in each direction has a very visible impact on quality, and is unlikely to be acceptable.

Comparing typical memory sizes for textures gives the following table:

Type	Size	Format	Memory Required (kbytes)
Diffuse texture	512x512	32bpp ARGB	1024
	512x512	DXT1 - 4bpp compressed	128
Normal map	512x512	32bpp ARGB	1024
	512x512	8bpp indexed palette[6]	256
Horizon map	512x512x16	8bpp scalar	4096
	256x256x16	8bpp scalar	1024
	128x128x16	8bpp scalar	256
	128x128x8	8bpp scalar	128

Since horizon map data is typically composed of smooth gradients with occasional sharp edges, it compresses well using vector quantisation or schemes such as the existing DXTn/S3TC[7] formats. Although these formats do not currently support 8-bit scalar values, suitable variants of these schemes are likely to give good results.

The largest impact on the rendering pipeline is the memory required for the multiple horizon maps. Although represented using 8 bits per pixel, this can still be costly. Fortunately the method scales well according to the available memory, both in the number of directions the horizon map is sampled, and in the resolution of the horizon maps used. This allows the application to make easily controlled tradeoffs between visual quality and memory consumed.

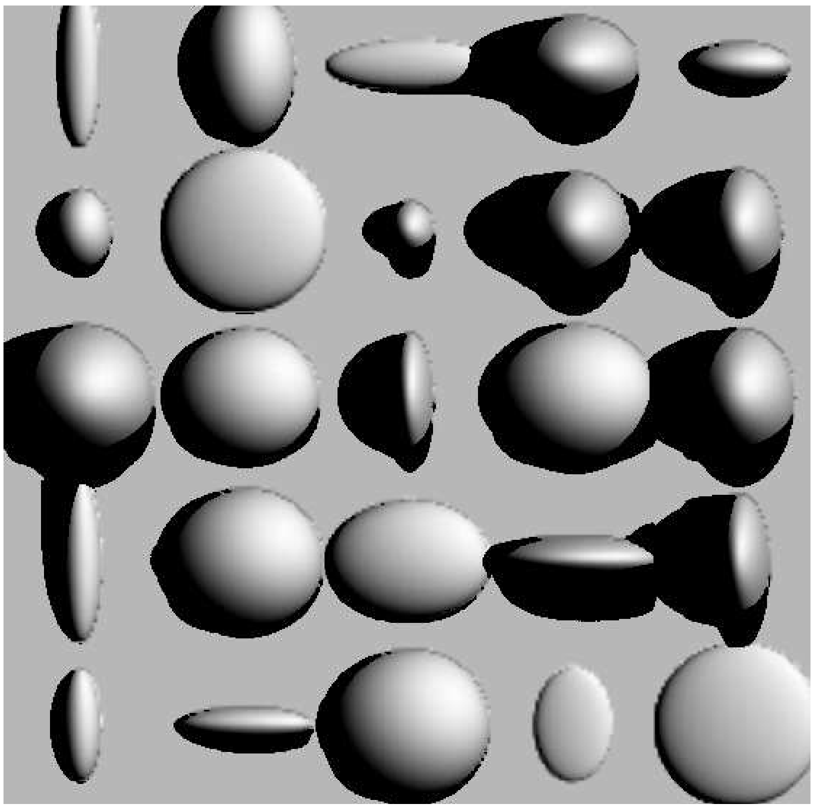

Figure 1 - 4 bearing samples

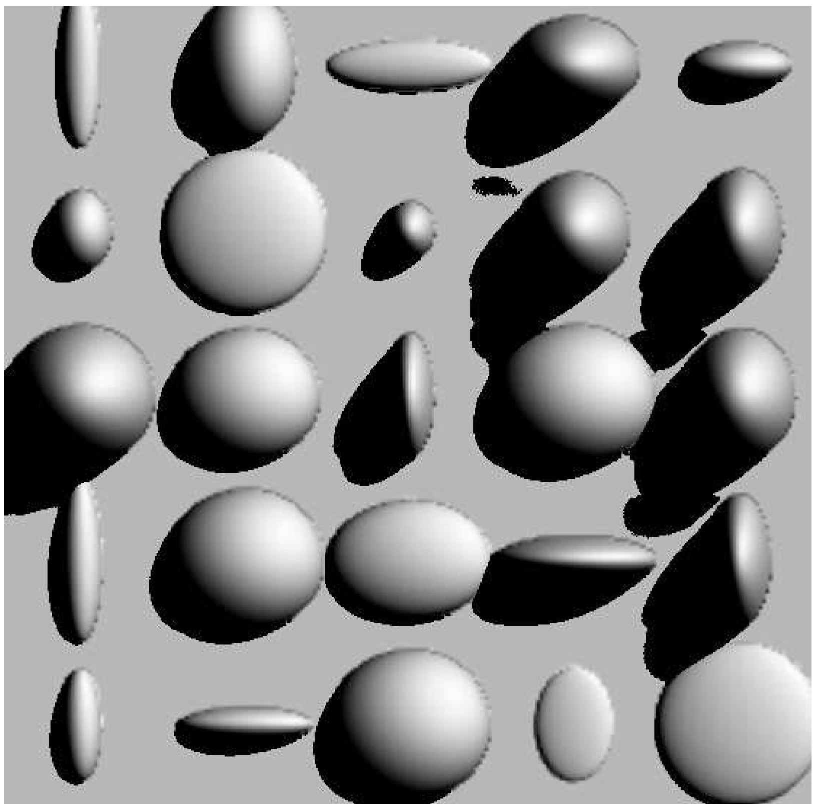
Figure 2 - 8 bearing samples

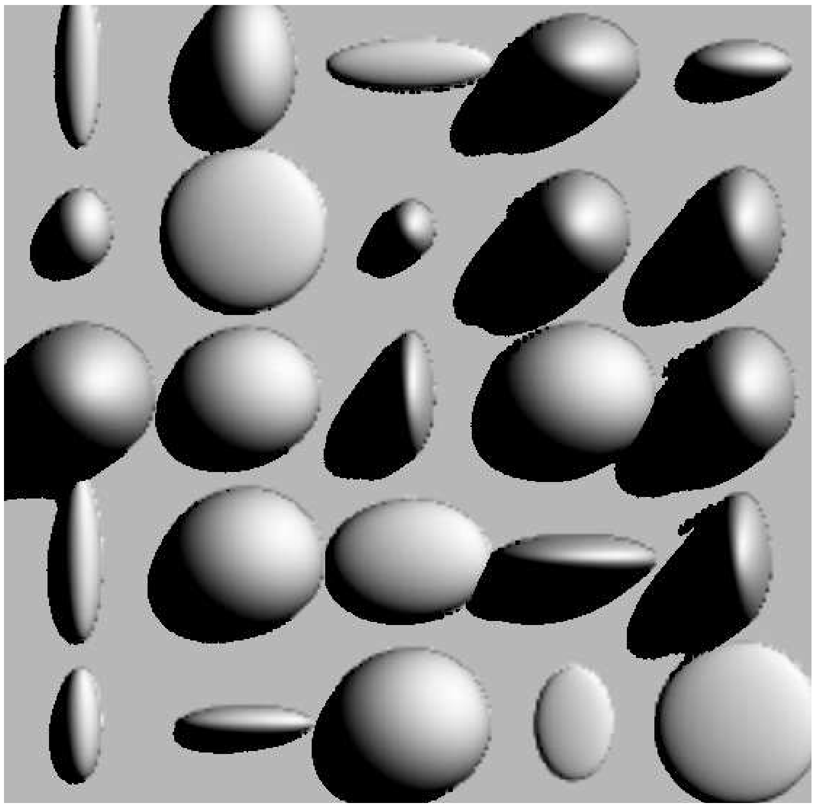
Figure 3 - 16 bearing samples

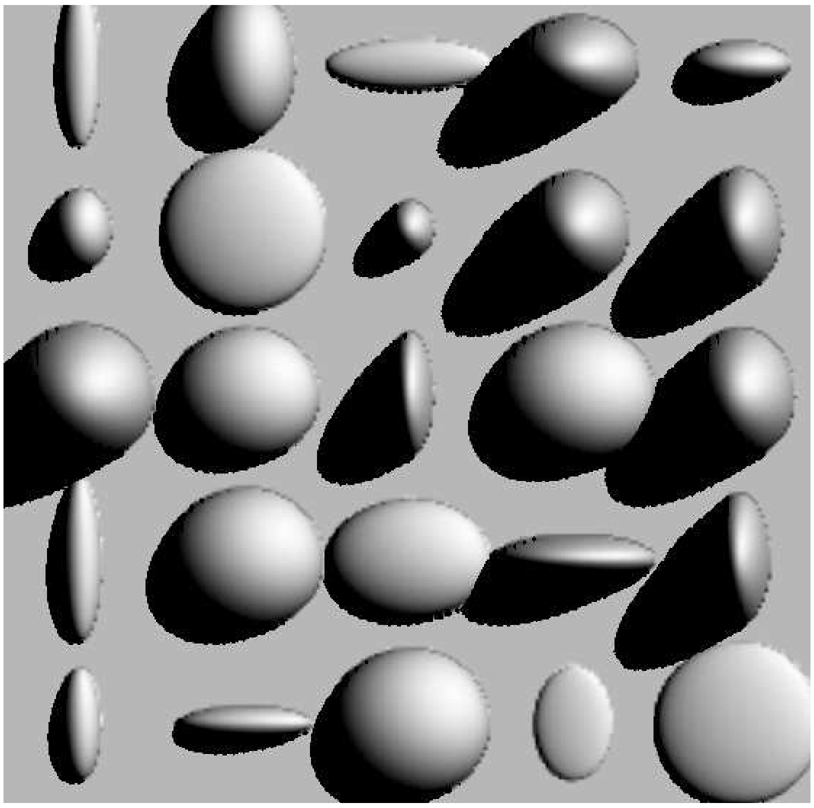

Figure 4 - 32 bearing samples

---

[1] Max, N. L. Horizon mapping: shadows for bump-mapped surfaces. The Visual Computer 4, 2 (July 1988), 109-117.

[2] Sloan, PP.; Cohen M. Interactive horizon mapping, Eurographics Rendering Workshop 2000, June 2000

[3] J. Kautz, W. Heidrich, K. Daubert, Bump Map Shadows for OpenGL Rendering, Technical Report MPI-I-2000-4-001, Max-Planck-Insitut fuer Informatik, Saarbruecken, 2000

[4] Heidrich, W., and Seidel, H.-P. Realistic, hardware-accelerated shading and lighting. Proceedings of SIGGRAPH 99 (August 1999), 171-178. ISBN 0-20148-560-5. Held in Los Angeles, California.

[5] Texture wrapping details: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wced3d/htm/_wcesdk_dx3d_Texture_Wrapping.asp

[6] Normal maps compress very badly with DXTn formats and have very visible compression artefacts. We have found that an 8-bit indexed palette format is far more effective and has very few visible artefacts.

[7] Compressed Texture Formats. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wced3d/htm/_wcesdk_dx3d_compressed_texture_formats.asp