Bump Map Shadows for OpenGL Rendering

Jan Kautz, Wolfgang Heidrich, Katja Daubert

MPI–I–2000–4-001 February 2000

Author's Address

Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
Germany
{kautz,heidrich,daubert}@mpi-sb.mpg.de

Keywords

Bump Mapping, Shadows, Frame Buffer Tricks, Texture Mapping.

1 Introduction

Bump maps are a commonly used technique to map more detail (i.e. bumps) onto simple geometry. Bump maps do not actually change the underlying geometry, only the lighting is changed as if the geometry was more complex. Interactive techniques usually use the Blinn-Phong model for the lighting of the bumps [Bli77] (local illumination only), but without shadows. Although techniques for shadowing bump maps have been developed [Max88], they are not commonly used. In particular they cannot be used for hardware rendering.

We propose a novel technique which adds shadows to bump maps, but acts only as an extension to bump mapping. One can use any traditional bump mapping technique and then simply add shadows on top of that.

We have implemented two slightly different methods: hard shadows and shadows with antialiased shadow boundaries[1]. The antialiased shadow method could be even seen as a fake soft shadow algorithm. As we assume parallel light, it is obvious that it does not correspond to real soft shadows.

We use texture mapping and frame buffer arithmetic to perform the shadow test and to write the result as a shadow mask into the stencil buffer. Then we render the bump map in a traditional way (dot product bump mapping in our case) just with the stencil test turned on so that the shadowed parts remain dark. The antialiased shadow algorithm on the other hand uses the alpha channel to modulate the intensities of the bump mapped surface.

In order to be able to do all this with graphics hardware, we have developed a new way to represent shadows for all possible light directions, such that the shadow test is so simple that current graphics hardware can be used to do this test at interactive rates (assuming a directional light source).

[1]From now on we will call it the *antialiased shadow algorithm* for simplicity.

2 The Ellipse Shadow Map

Our method precalculates for all possible incident light directions whether a certain pixel in the bump map is in shadow or not. We will explain this preprocessing step for a single pixel in the bump map. But obviously, this process has to be repeated for every pixel in the bump map.

First we send rays from the center of the pixel through its upper hemisphere, which is defined by the normal of the polygon being bump mapped. Rays either hit a bump in the bump map or leave the surface. Rays leaving the surface are *light* rays, i.e. light from that direction illuminates the pixel and rays hitting other bumps are *shadow* rays, i.e. no direct light can arrive from that direction.

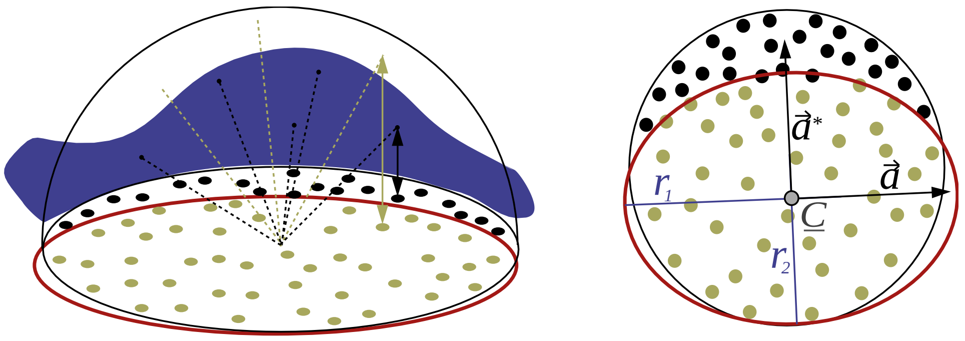

Figure 2.1: *For the shadow test we precompute 2D ellipses at each point of the bump map by fitting them to the projections of the test rays into the tangent plane. Light rays are drawn yellow and shadow rays are drawn black.*

Now we project the unit vectors of these rays into the tangent plane, see Figure 2.1. Then we fit a 2D ellipse that contains as many light directions as possible, without containing too many shadow directions. This ellipse is defined by its center $\underline{C}$, a direction $\vec{a} = (a_x, a_y)^T$ describing the major axis, a direction $\vec{a}^* = (-a_y, a_x)^T$ describing the minor axis, and two radii r_1 and r_2, for the extent of each axis.

For the fitting process, we begin with the ellipse represented by the eigenvectors of the covariance matrix of all points corresponding to unshadowed directions. We then optimize the radii with a local optimization method.

Once we have computed this ellipse, the shadow test is simple. We have to project the global light direction into the local coordinate system of the shadow bump map, which yields the new unit vector $\vec{l}$. Then we have to transform this local light direction into the coordinate system defined by the axes of the ellipse:

$$l'_x := \begin{pmatrix} a_x \\ a_y \end{pmatrix} \circ \begin{pmatrix} l_x - C_x \\ l_y - C_y \end{pmatrix}, \tag{2.1}$$

$$l'_y := \begin{pmatrix} -a_y \\ a_x \end{pmatrix} \circ \begin{pmatrix} l_x - C_x \\ l_y - C_y \end{pmatrix} \tag{2.2}$$

A pixel is lit if the following equation holds, otherwise the pixel is in shadow:

$$1 - \frac{(l'_x)^2}{r_x^2} - \frac{(l'_y)^2}{r_y^2} \geq 0 \tag{2.3}$$

We will call this equation the *shadow test*.

3 Tricking Graphics Hardware into Doing the Shadow Test

We can make the shadow test from Equation 2.3 work with current graphics hardware if we rewrite it in the following way:

$$\left(\frac{l'_x}{r_x}\right)^2 + \left(\frac{l'_y}{r_y}\right)^2 \leq 1 \tag{3.1}$$

Then we substitute the dot products from Equation 2.1 and Equation 2.2 into this equation and we get:

$$\left(\frac{a_x l_x + a_y l_y - (a_x C_x + a_y C_y)}{r_x}\right)^2 + \left(\frac{-a_y l_x + a_x l_y - (-a_y C_x + a_x C_y)}{r_y}\right)^2 \leq 1$$

This can be rewritten with different dot products:

$$\begin{aligned}
\vec{L} &:= \begin{pmatrix} l_x \\ l_y \\ -1 \end{pmatrix}, \\
\vec{T_1} &:= \begin{pmatrix} a_x/r_x \\ a_y/r_x \\ (a_x C_x + a_y C_y)/r_x \end{pmatrix}, \qquad \vec{T_2} := \begin{pmatrix} -a_y/r_y \\ a_x/r_y \\ (-a_y C_x + a_x C_y)/r_y \end{pmatrix},
\end{aligned}$$

$$\left(\vec{L} \circ \vec{T_1}\right)^2 + \left(\vec{L} \circ \vec{T_2}\right)^2 \leq 1 \tag{3.2}$$

This way, we have expressed the shadow test with simple dot products, which can be computed with graphics hardware [WE98, NVI99].

Now we simply put $\vec{T_1}$ and $\vec{T_2}$ into texture maps. A pair of corresponding pixel in these two texture maps defines an ellipse. As we assume the light to be parallel, the vector $\vec{L}$ remains constant across a polygon, so it is not necessary to put it into a texture map. What is left to the graphics hardware, is to compute the dot products, to square the results, and to finally perform the comparison. The actual implementation of this depends on the underlying graphics hardware (i.e. supported OpenGL extensions).

The next section describes an implementation for an SGI machine using the OpenGL imaging subset and Section 5 desribes an implementation for an NVIDIA GeForce 256 using NVIDIA's register combiner extension.

Note that the equations above are only intended to show how we derived our method, but they will have to be rewritten anyway, because OpenGL only operates in the range between 0 and 1, but the vectors $\vec{T}_1$, $\vec{T}_2$, and $\vec{L}$ operate in a larger range.

4 Rendering using the Imaging Subset

In this section we describe how to use the imaging subset to compute the dot products and to square the results.

The imaging subset provides an extension called *Color Matrix*, which allows the application of a 4×4 matrix to RGBA pixel groups during the pixel transfer path. If we bind a texture map using a color matrix other than identity, the texture map will be altered according to the color matrix and the altered texture map is stored in the texture memory. Of course this matrix can also be used to compute dot products. The basic idea here is to rebind the texture maps $\vec{T}_1$ and $\vec{T}_2$ whenever $\vec{L}$ changes and to apply a color matrix to compute the appropriate dot products.

Afterwards, we still have to square the result of the dot product. The imaging subset also allows to apply a color lookup table to the result of a color matrix multiplication. We use the lookup table to map the result of the dot product to its square. The color lookup table also works during the pixel transfer path, i.e. when we load a texture to texture memory the lookup table is applied (after the color matrix).

Unfortunately, OpenGL's [0,1]-range restriction forces us to modify the texture maps $\vec{T}_1$ and $\vec{T}_2$ and the shadow test from Equation 3.2.

4.1 Data Structures

Without limiting generality we assume that $r_x \leq r_y$. If this is not true, $\vec{a}$ can be exchanged with $\vec{a}^*$ to satisfy this assumption.

Now we multiply Equation 3.2 with r_x^2, which gives us two new texture maps:

$$\vec{T}_1^r := \begin{pmatrix} a_x \\ a_y \\ (a_x C_x + a_y C_y) \end{pmatrix}, \qquad \vec{T}_2^r := \begin{pmatrix} -a_y \cdot (r_x/r_y) \\ a_x \cdot (r_x/r_y) \\ (-a_y C_x + a_x C_y) \cdot (r_x/r_y) \end{pmatrix},$$

and a slightly changed shadow test:

$$r_x^2 - \left(\vec{L} \circ \vec{T}_1^r\right)^2 - \left(\vec{L} \circ \vec{T}_2^r\right)^2 \quad \geq \quad 0 \tag{4.1}$$

This ensures, that the components of both texture maps are in the [-1,1] range, assuming that the ellipsoid's center $\underline{C}$ does not lie outside the unit circle (i.e. $\|\underline{C}\| \leq 1$).

As you can see in Equation 4.1, we need to have access to r_x^2 somehow. Therefore we store r_x^2 in a separate third texture map $\vec{T}_3^r$.

4.1.1 Biasing

Now we need to bias the texture maps $\vec{T}_1^r$ and $\vec{T}_2^r$ so that they lie in [0,1]. For every component in the texture maps we apply the following mapping:

$$\text{newcomponent} = \frac{1}{2}(\text{component} + 1)$$

The component of the third texture map, which contains r_x^2 needs only to be divided by 2 (assuming the radius $r_x \leq \sqrt{2}$, which is very likely, because $r_x \leq r_y$). Now we have three texture maps $\vec{T}_1^r$, $\vec{T}_2^r$ and $\vec{T}_3^r$ that are all in the range [0,1].

4.1.2 Texture Maps

We have to use 4-component texture maps for $\vec{T}_1^r$ and $\vec{T}_2^r$, where the alpha component has to be set to 1, because of biasing issues which will become clearer in the next sections.

The third texture map can be a one-component texture containing only alpha values. These alpha values will be set to $\frac{1}{2}r_x^2$.

4.1.3 Color Matrix

Now we have to set the color matrix so that binding either texture map $\vec{T}_1^r$ or $\vec{T}_2^r$ computes the dot product $\vec{L} \circ \vec{T}_1^r$, or respectively $\vec{L} \circ \vec{T}_2^r$. So we simply put $\vec{L}$ into one column of the color matrix, paying attention though to correctly bias everything.

We use the following matrix (OpenGL notation, i.e. transposed):

$$M := \begin{pmatrix} 0 & 0 & 0 & \frac{1}{2}l_x \\ 0 & 0 & 0 & \frac{1}{2}l_y \\ 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & -\frac{1}{4}(l_x + l_y) + \frac{1}{2} \end{pmatrix}$$

Note, that choosing the last column of the matrix writes the result of the computation into the alpha channel. This is not the final result yet, as we still have to apply the lookup table, which will correct the remaining bias and square the result.

4.1.4 Color Lookup Table

After applying the color matrix, we apply the following color lookup table:

$$lookup(a) := \frac{1}{2}(4a-1)^2 \tag{4.2}$$

We would like to show now what happens, when we apply the matrix M and the lookup table to $\vec{T}_1^r$ (works equivalently with $\vec{T}_2^r$):

$$\begin{aligned} lookup(\vec{T}_1^{r/transpose} \bullet M) &= \frac{1}{2}\left((l_x \cdot a_x + l_y \cdot a_y - a_x \cdot c_x - a_y \cdot c_y)^2\right) \\ &= \frac{1}{2}(\vec{L} \circ \vec{T}_1^r)^2 \end{aligned}$$

This is exactly what we wanted. Note, that the factor $\frac{1}{2}$ is needed to overcome the [0,1] range limitations in the frame buffer.

4.1.5 Rendering with Hard Shadows

The rendering algorithm is quite simple now:

1. Bind texture map $\vec{T}_3^r$ (alpha := $\frac{1}{2}r_x^2$).
2. Set up color matrix M.
3. Set up color lookup table.
4. Bind texture map $\vec{T}_1^r$ (alpha := $\frac{1}{2}(\vec{L} \circ \vec{T}_1^r)^2$).
5. Bind texture map $\vec{T}_2^r$ (alpha := $\frac{1}{2}(\vec{L} \circ \vec{T}_2^r)^2$).
6. Render polygon with $\vec{T}_1^r$.
7. Set blending to *add*.
8. Render polygon with $\vec{T}_2^r$.
9. Enable the blend equation extension and set it to *subtract*.
10. Render polygon with $\vec{T}_3^r$.

11. Result in alpha channel of frame-buffer: $\frac{1}{2}\left(r_x^2 - (\vec{L} \circ \vec{T}_1^r)^2 - (\vec{L} \circ \vec{T}_2^r)^2\right)$
12. Set the alpha test to pass when alpha equals 0 (i.e. in shadow).
13. Set stencil operator to *increment*.
14. Call *copy pixels*, which will set the stencil bit to 1 for the regions that pass the alpha test.
15. Disable the alpha test.
16. Set stencil test to pass when the stencil bit is 0 (i.e. lit).
17. Draw bump map (we use [HS99] here).

As you can see, this is a fairly simple algorithm. The only disadvantage is that we have to do a *copy pixels* to set the stencil buffer according to the alpha channel, which is stored in the frame buffer. Unfortunately, it is not possible to use the alpha test instead, because it works with incoming fragments only! The next section shows a slightly different algorithm, which only uses the alpha channel, but results in antialiased shadows.

4.2 Rendering with Antialiased Shadows

It should be mentioned again that our antialiased shadow technique is not accurate. As we use directional light only, the resulting shadows should have exact shadow boundaries! But nonetheless antialiased shadows look much more convincing, because in reality there is no exactly parallel light.

The term $d := r_x^2 - (\vec{L} \circ \vec{T}_1^r)^2 - (\vec{L} \circ \vec{T}_2^r)^2$ basically tells you the distance of $\vec{L}$ from the ellipse's border. Values larger than zero stand for lit pixels and values smaller than zero mean the pixel is in shadow. If we use d as a measure how much a pixel is lit, we get antialiased shadows. We just have to multiply the bump map with d and we will get the ”right” answer. The closer d is to zero, the more the pixel is in shadow; the larger d is, the more the pixel is lit. It is a good idea to scale d by some factor to control the size of the penumbra (a factor of at least 2 yields good results).

Here is the modified rendering algorithm:

1. Bind texture map $\vec{T}_3^r$ (alpha := $\frac{1}{2} r_x^2$).
2. Set up color matrix M.
3. Set up color lookup table.

4. Bind texture map $\vec{T}_1^r$ (alpha := $\frac{1}{2}(\vec{L} \circ \vec{T}_1^r)^2$).

5. Bind texture map $\vec{T}_2^r$ (alpha := $\frac{1}{2}(\vec{L} \circ \vec{T}_2^r)^2$).

6. Render polygon with $\vec{T}_1^r$.

7. Set blending to *add*.

8. Render polygon with $\vec{T}_2^r$.

9. Enable the blend equation extension and set it to *subtract*.

10. Render polygon with $\vec{T}_3^r$.

11. Result in alpha channel of frame buffer: $\frac{1}{2}d := \frac{1}{2}\left(r_x^2 - (\vec{L} \circ \vec{T}_1^r)^2 - (\vec{L} \circ \vec{T}_2^r)^2\right)$

12. Set the blend function to: source factor = *destination alpha* and destination factor = *one*.

13. Render polygon with color (0.0, 0.0, 0.0, 1.0) (doubles alpha to d).

14. Render polygon with color (0.0, 0.0, 0.0, 1.0) (doubles alpha again to $2d$).

15. Reset blending.

16. Disable rendering into alpha channel.

17. Draw bump map (we use [HS99] here).

18. Enable rendering into alpha channel again.

19. Set the blend function to: source factor = *destination alpha* and destination factor = *destination alpha*.

20. Render polygon with color (1.0, 1.0, 1.0, 1.0) (multiplies frame buffer with alpha value, i.e. with $2d$).

21. Result in frame-buffer: Bumpmap $\cdot\ 2d$, which is exactly what we wanted!

This algorithm is more efficient than the algorithm mentioned before because the expensive *copy pixels* call is not necessary anymore.

5 Rendering using the GeForce 256

NVIDIA's GeForce 256 is the first card to provide a multitexturing unit, which operates in the range [-1,1]. Furthermore NVIDIA introduced a new extension to OpenGL, called register combiners. Register combiner offer some limited programmability in the multitexturing stage. The supported features include the computation of dot products, which makes it unnecessary for us to use the imaging subset (i.e. the color matrix)!

However, in practice problems arise with the computation of squared dot products in the multitexturing stage. The multitexturing unit uses bilinear filtering to produce smooth transitions between original texels (can be turned off, but will definitely result in block artefacts). These interpolated values are then used in the register combiner stage, e.g. to compute the square of the dot products, which can produce severe artefacts. This is especially true if the vectors are rapidly varying (i.e. rapid sign changes).

Consider the following example: you have a texture map containing a signed value: texel 1 contains the value -1 and texel 2 contains the value 1. Now you want to compute the square of this texture map. You render a polygon with this texture map using bilinear filtering (to avoid blockiness) and you use the register combiners to compute the square. If your polygon is exactly twice as big as your texture map, this will compute the following pixel values: pixel 1 becomes $(-1)^2$, pixel 2 becomes 0^2, and pixel 3 becomes 1^2, because the texels will be interpolated first and then squared! If squaring were done before interpolation, all three pixels would be exactly 1, as expected.

The previously defined texture maps $\vec{T}_1^r$ and $\vec{T}_2^r$ do contain rapid sign changes. Therefore it is desirable that interpolation is performed after the square-operations, which will avoid artefacts.

Fortunately we can ”pre-square” the dot products of Equation 4.1. This way we obtain new texture maps, which do not have rapidly varying sign changes.

5.1 Data Structures

We will expand Equation 4.1 and then regroup it differently so that only dot products are needed and the square operation can be done in advance:

$$r_x^2 - \left(T^r_{1,x} l_x + T^r_{1,y} l_y - T^r_{1,z}\right)^2 - \left(T^r_{2,x} l_x + T^r_{2,y} l_y - T^r_{2,z}\right)^2 \quad \geq \quad 0$$

becomes

$$r_x^2 - \left(L_1^n \circ T_1^n\right) - \left(L_2^n \circ T_2^n\right) \quad \geq \quad 0 \tag{5.1}$$

with

$$\begin{aligned}
\vec{L}_1^n \quad &:= \quad \begin{pmatrix} l_x^2 \\ l_x l_y \\ l_x \end{pmatrix}, \qquad \vec{T}_1^n := \begin{pmatrix} (T^r_{1,x})^2 + (T^r_{2,x})^2 \\ 2T^r_{1,x}T^r_{1,y} + 2T^r_{2,x}T^r_{2,y} \\ -2T^r_{1,x}T^r_{1,z} - 2T^r_{2,x}T^r_{2,z} \end{pmatrix}, \\
\vec{L}_2^n \quad &:= \quad \begin{pmatrix} l_y \\ l_y^2 \\ 1 \end{pmatrix}, \qquad \vec{T}_2^n := \begin{pmatrix} -2T^r_{1,y}T^r_{1,z} - 2T^r_{2,y}T^r_{2,z} \\ (T^r_{1,y})^2 + (T^r_{2,y})^2 \\ (T^r_{1,z})^2 + (T^r_{2,z})^2 \end{pmatrix}.
\end{aligned}$$

So we have two new texture maps $\vec{T}_1^n$ and $\vec{T}_2^n$, which can be computed from the old texture maps $\vec{T}_1^r$ and $\vec{T}_2^r$. The original light vector cannot be used anymore, instead two constant vectors $\vec{L}_1^n$ and $\vec{L}_2^n$ are needed.

5.1.1 Texture Maps

This time biasing is less complicated, so we can move r_x^2 into the alpha component of T_1^n avoiding the need for a third texture map. This is even required, because the GeForce 256 only has two texture mapping units.

5.1.2 Biasing

Now we need to bias the texture maps $\vec{T}_1^n$ and $\vec{T}_2^n$ and the constant vectors $\vec{L}_1^n$ and $\vec{L}_2^n$ so that they lie in [0,1] (only the multitexturing stage supports signed values). For every component in the texture maps and in the two constant vectors we apply the following mapping:

$$\text{newcomponent} = \frac{1}{2}(\text{component} + 1)$$

This mapping can be reversed by the register combiner extension, so that the dot products can be computed with signed arithmetic.

5.2 Rendering with Antialiased Shadows

We will only show the antialiased shadow algorithm here, which is probably more interesting to most people. The rendering algorithm works as follows:

1. Enable multitexturing.
2. Enable register combiner extension.
3. Disable rendering to RGB channels (render to alpha channel only).
4. Load texture map $\vec{T}_1^n$ into texturing unit 0.
5. Load texture map $\vec{T}_2^n$ into texturing unit 1.
6. Set ConstantColor0 to $\vec{L}_1^n$.
7. Set ConstantColor1 to $\vec{L}_2^n$.
8. Set up combiner stage 0:
   (a) Spare0 $\leftarrow$ ConstantColor0 $\circ$ Texture0   (Spare0rgb = $\vec{L}_1^n \circ \vec{T}_1^n$).
   (b) Spare1 $\leftarrow$ ConstantColor1 $\circ$ Texture1   (Spare1rgb = $\vec{L}_2^n \circ \vec{T}_2^n$).
9. Set up combiner stage 1:
   (a) Spare0Alpha $\leftarrow$ (Spare0Blue * 1.0) + (Spare1Blue * 1.0)
   (Spare0Alpha = $\vec{L}_1^n \circ \vec{T}_1^n + \vec{L}_2^n \circ \vec{T}_2^n$)
10. Set up final combiner stage:
    (a) Output rgb $\leftarrow$ 0 and alpha $\leftarrow$ Spare0Alpha.
11. Render polyon (alpha = $\vec{L}_1^n \circ \vec{T}_1^n + \vec{L}_2^n \circ \vec{T}_2^n$).
12. Disable multitexturing.
13. Disable register combiners.
14. Enable the blend equation extension and set it to *subtract*.
15. Enable 2D texturing (GL_REPLACE).
16. Texture $\vec{T}_1^n$ is active (r_x^2 in alpha component).
17. Render polygon.

18. Result in alpha channel of frame buffer: $d := r_x^2 - \vec{L}_1^n \circ \vec{T}_1^n - \vec{L}_2^n \circ \vec{T}_2^n$.

19. Set the blend function to: source factor = *destination alpha* and destination factor = *one*.

20. Disable 2D texturing.

21. Render polygon with color (0.0, 0.0, 0.0, 1.0) (doubles alpha to $2d$).

22. Reset blending.

23. Disable rendering into alpha channel and enable rendering to RGB channels.

24. Draw bump map (we use dot product bump mapping here).

25. Enable rendering into alpha channel again.

26. Set the blend function to: source factor = *destination alpha* and destination factor = *destination alpha*.

27. Render polygon with color (1.0, 1.0, 1.0, 1.0) (multiplies frame buffer with alpha value, i.e. with $2d$).

28. Result in frame-buffer: Bumpmap $\cdot$ $2d$, which is exactly what we wanted!

Although this rendering algorithm might seem complicated, it is actually a straight forward implementation of Equation 5.1. If the register combiners had one more stage, the subtraction could also be done in the same pass, which would also allow the scaling to be done in the final combiner stage!

This algorithm could be combined with the hard shadow algorithm from Section 4.1.5 to render bump maps with hard shadows on a GeForce 256.

6 Results

Here we would like to show a few results. The first test scene consists of only one polygon with a bump map. We show three different versions, one without shadows, one with hard shadows, and one with antialiased shadows, see Figures 6.1 – 6.3.

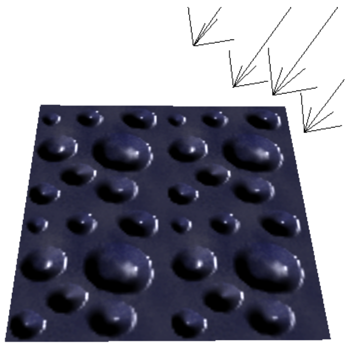
Figure 6.1: *Single bump mapped polygon without shadows.*

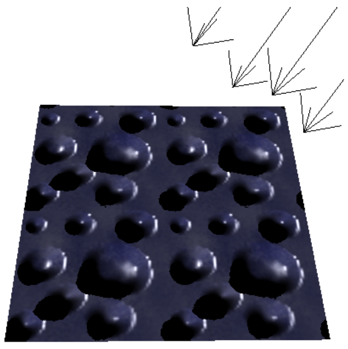
Figure 6.2: *Single bump mapped polygon with hard shadows.*

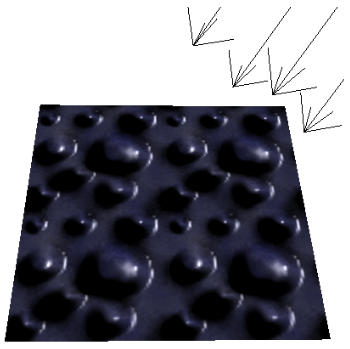
Figure 6.3: *Single bump mapped polygon with antialiased shadows.*

Figure 6.4 and Figure 6.5 show a bump mapped terrain, first without shadows and then with antialiased shadows. Note, that we have only one bump map and only one ellipse shadow map for the whole terrain. This is possible if the appropriate texture coordinates are assigned at each vertex of the terrain. Lighting could also be done with OpenGL instead of bump maps, but in both cases we can apply our technique and add shadows.

Frame rates for the terrain model (961 polygons) with a diffuse bump map and antialiased shadows are at about 17Hz on a GeForce 256. So even with a higher polygon count, this method still achieves interactive frame rates.

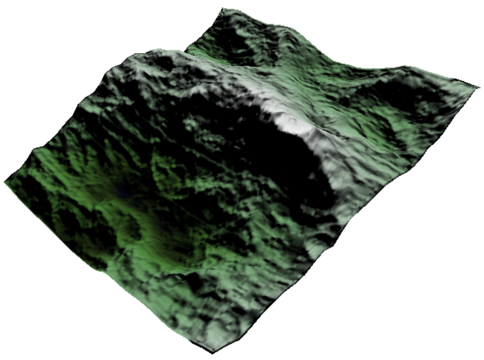

Figure 6.4: *Bump mapped terrain without shadows.*

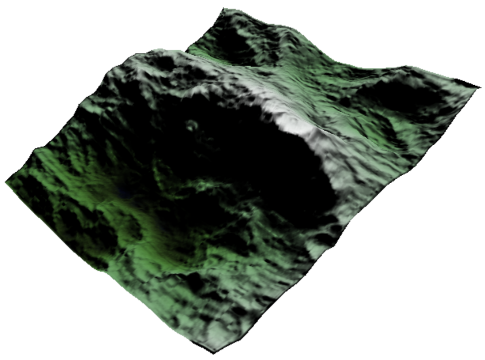

Figure 6.5: *Bump mapped terrain with antialiased shadows.*

7 Conclusions

We have presented a method that adds shadows to bump maps. The rendering algorithm is simple, once a precomputed ellipse shadow map has been generated. Rendering can be done at interactive rates using standard OpenGL with either the imaging subset or NVIDIA's register combiner extension.

Bibliography

[Bli77] J. Blinn. Models of light reflection for computer synthesized pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, July 1977.

[HS99] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August 1999.

[Max88] N. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[NVI99] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, October 1999. Available from http://www.nvidia.com.

[WE98] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *"Computer Graphics (SIGGRAPH '98 Proceedings)"*, pages 169–178, July 1998.

MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP