Interactive Horizon Mapping

Peter-Pike J. Sloan Michael F. Cohen
ppsloan@microsoft.com mcohen@microsoft.com
Microsoft Research

Abstract.
Shadows play an important role in perceiving the shape and texture of an object. While some previous interactive shadowing methods are appropriate for casting shadows on other geometry they can not be applied to bump maps (which contain no explicit geometry.) Horizon Mapping is a technique used to compute shadows for bump-mapped surfaces. We map the technique into modern graphics API's and extend it to account more accurately for the geometry of the underlying surface. We also use it to represent limited self-shadowing for pure geometry. In mapping the algorithm to hardware, we use a novel method to interpolate orientation in tangent space over the surface. We show results of self-shadowing at frame rates.

1 Introduction

Shadows provide important perceptual cues for understanding surface shape. However, it is challenging to display them while maintaining interactivity. An extensive amount of work has been undertaken to develop algorithms to generate shadows in general - see [11] for an excellent survey. Since shadows are essentially a visibility problem, it is not surprising that the most common interactive techniques for generating shadows are variants on the shadow zbuffer [10], (see [9, 12, 6] for implementations that function with modern graphics hardware).

Bump Mapping [2] is a technique to convey surface texture by perturbing the normal vectors of a surface. It is available on most current graphics hardware [5]. The advantage of bump mapping is that it provides a simple way to modulate the shading on a surface. However, since bump mapping does not define any explicit geometry, there are no actual bumps to cast shadows. Thus, interactive shadowing techniques that rely on an explicit representation of the geometry cannot be used to simulate shadows cast by the virtual bumps implied in bump mapping. A concurrent paper [4] presents a way to shadow bump maps by using implicit representations of an ellipse in the tangent plane.

This work is motivated by one particular technique for casting shadows: horizon mapping [7]. The idea behind horizon mapping is to precompute limited visibility from each point on a surface. In particular, the angle to the horizon is encoded in a discrete number of directions to represent at what height something becomes visible from that direction (i.e., pass over the horizon). This parameterization can be used to produce the self-shadowing of geometry as well. Accessibility maps defined by Miller [8] are similar to horizon maps. Accessibility maps only use a scalar value to represent the horizon. In contrast, horizon maps include more detail about the horizon.

In the remainder of the paper we show how horizon maps can be encoded and passed through current APIs to graphics hardware to produce real-time self-shadowing based on bump maps.

2 Bump and Horizon Mapping

Given a surface $P(u, v)$ parameterized on the unit square, a surface normal N is the cross product of the partials of the surface P_u and P_v. Given a bump map $F(u, v)$ a non-negative scalar function parameterized over the same domain, the surface normal N can be modified as follows (after dropping terms of first order):

$$P'_u = P_u + F_u N/|N|, P'_v = P_v + F_v N/|N|, N' = P'_u \times P'_v = N + D$$

where

$$D = (F_u N \times P_v - F_v N \times P_u)/|N|$$

is the perturbation of the normal. Max [7] slightly modfied this so that the bump maps behave properly over the surface. We will use normal mapping as outlined in [5]. This modification effectively deals with parameterization dependency for the illumination, but not for the shadowing.

As in [7] we define a mapping into the local coordinate system of the surface - this is done through the dual of the basis P_u, P_v, N where N is the ***unbumpmapped*** surface normal (the affine transform C^{-1} Max refers to). This is computed by building a matrix with the basis as columns and inverting it, the rows of this inverse are a scaled version of $N \times P_v$ and $P_u \times N$, along with N itself. A vector in this local frame has an associated orientation in the tangent plane θ and an angle with the normal ϕ.

We are interested in knowing at each pixel whether a light vector when transformed into the local coordinate frame is visible above the horizon. To discover this we build a *Horizon Map*, $\phi_{u,v,\theta}$. The horizon map is tabulated at discrete u, v parameter values, and in a set of directions θ, to represent the azimuth angle ϕ when a light would become visible. Typically u and v are sampled fairly densely (512x512), and θ is sampled more coarsely - 8 sample directions, (N, NE, E, etc.) in both our work and in [7].

In the case of curved surfaces, the precomputed horizon map is created in terms of the local geometry at each discrete (u, v) coordinate. This contrasts with [7] in which there is an essential assumption made that the underlying geometry of the surface is not known when the horizon maps are computed.

Given M sampled directions (8 in our implementation) for θ and the discrete domain coordinates u_i, v_j the horizon angle $\phi(u_i, v_j, \theta)$ for ANY direction at coordinates u_i, v_j can be found by interpolating between the discrete directions as follows:

$$\phi(u_i, v_j, \theta) = \sum_{k=1}^{M} B_k(\theta)\phi(i, j, k) \tag{1}$$

where $B_k(\theta)$ is a basis function for the k^{th} direction, which evaluates to 1 for the corresponding direction and linearly falls of to 0 for the neighboring directions. For example if the k^{th} direction is east, then the value is one when θ equals east and falls off to zero at NE and SE. In other words, the horizon angle ϕ is linearly interpolated between discrete directions, noting that since this is a radial function it wraps around to interpolate between $\phi(i, j, 1)$ and $\phi(i, j, M)$. Similarly, the function $\phi(u, v, \theta)$ is bilinearly interpolated across the parameters u and v.

To test the visibility of a light source, its direction (x, y, z) is transformed into the local tangent frame - a simple affine transform - to provide the local coordinates (u, v, θ). The angle θ could be determined by projecting the transformed vector onto the tangent

Fig. 1. Normal Map, North Horizon Map, North Basis Texture

plane. Instead, to take advantage of texture mapping hardware as explained later, the first two transformed coordinates are used as a lookup into a table of θ directions encoded as a texture. This is a key element to implementing the shadowing algorithm with current hardware.

3 General Self-Shadowing with Horizon Maps

The tangent plane parameterization and discretization used to compute shadows for bump maps can also be used to store global visibility information about a surface in general. The horizon angles are determined by shooting rays into the object, starting from the horizon and moving towards the surface normal until there is no intersection. This approach has the same limitations that horizon mapping does plus one additional limitation - the largest angle that can be represented in the horizon map is 90 degrees out of the tangent plane. This isn't that significant of a problem when determining shadows from bump maps, but can be a limitation when the gross scale of the features is more significant, for example, when the surface has undercuts.

4 Realizations on Commodity Graphics Hardware

We will now outline the runtime shadowing algorithm and how it is implemented on current graphics hardware. The input consists of:

- a surface geometry with a parameterization in u, v, and
- a scalar valued bump map, $F(u, v)$

4.1 Precomputation

A precomputation step produces:

- a vector valued perturbed normal map, $N'(u, v)$, from the bump map (see Figure 1).
- given M (in our case 8) directions in the tangent plane, $\theta_{k=1..M}$, M horizon maps, $\phi(u, v, \theta_k)$. With the 8 directions we will label each direction θ_k as N, NE, E, etc (see Figure 1). In fact, the 8 horizon maps are collected into only 2 maps by encoding 4 directions into the 4 color channels R, G, B, α. Thus, for example, the first map encodes direction N, NE, E and SE, while the other contains S, SW, W and NW.

- M *basis* maps, $B_k(s,t)$ representing the influence of direction θ_k (note this is independent of the parameterization) (see Figure 1). As in the case of the horizon maps, these are encoded in two maps containing 4 directions each.
- a 1D (arcos) mapping from $cos(\phi)$ to ϕ.

Finally, a per vertex pre-computation is carried out to invert the non-bump mapped local tangent frame $[P_u, P_v, N]^{-1} = [S^T, T^T, N^T]$. This will allow us to quickly transform the light direction onto the local tangent plane of the surface at each frame time. For a planar surface, the local tangent plane is the same for all vertices, but varies at each vertex over a curved surface.

4.2 At each frame time

Given the precomputation above, at each frame time, we first project the light direction onto the local tangent plane at each vertex. The light vector when dotted with the first two components of the inverted frame S and T yields the projection of the light vector into the coordinate space in the tangent plane resulting in the pair, (s,t). The light vector dotted with the normal at each vertex gives $cos(\phi_L)$.

The remainder of the computation is carried out per pixel and is done in hardware using multi-texturing and blending into the frame buffer.

First, set the transformations to render into UV space, (i.e., use (u,v) coordinates as vertex coordinates). Using a blending function to add, and multi-texturing to multiply rendering passes

- Set the multi-texturing to component-wise multiply the textures and sum the results and place them in the α channel
- Accumulate the contribution for the first 4 directions into the frame buffer
  - 1st texture is (E,NE,N,NW) basis map $B_1(s,t)$ (Note the s,t coordinates for each vertex are derived from the light direction.)
  - 2nd texture is (E,NE,N,NW) horizon map, $\phi_1(u,v)$
- Accumulate (i.e., add) the contribution for the next 4 directions
  - 1st texture is (W,SW,S,SE) basis map $B_2(s,t)$
  - 2nd texture is (W,SW,S,SE) horizon map, $\phi_2(u,v)$

The resulting α channel now represents the horizon angle, ϕ, in the direction of the light. Save results to a texture map which we will call $\phi(\theta_{LIGHT})$.

This is followed by three rendering passes with the transformations set to draw into the current camera.

1. First Pass: Draw the model with ambient term only
2. Second Pass: Create a stencil that will only allow non-shadowed pixels to be rendered
   - Set the alpha test to only accept pixels that have non 0 alpha,
   - Set the stencil test to set a bit for any pixel that passes through, and
   - Set color mask to NOT write to color channels to preserve the ambient term of the first pass.
   - Set multi-texturing to subtract (note: negative values are clamped to 0)

- Draw using 2 sets of texture coordinates that will represent the angles to the light and the angle to the horizon. Subtracting the two texture values will yield positive values in the α channel where the surface sees the light and zero (negative values are clamped to zero) where it is in shadow. The two textures to perform this pass are:
  - a 1D texture ($cos(\phi)$->ϕ) (note: $cos(\phi)$ of the light was previous computed at each vertex). This result contains the angle off the normal to the light at each pixel.
  - the 2D texture $\phi(\theta_{LIGHT})$ computed before. This encodes the angle off the normal of the horizon in the direction of the light at each pixel.

3. Third pass to perform normal bump mapped rendering of the model where not in shadow.
   - Turn off alpha test
   - Set color mask to allow writing of color channels
   - Set stencil function to only draw pixels that have the stencil bit set.
   - Set blending to add into the frame buffer to accumulate with ambient term.
   - Draw using normal map - shades non-shadowed regions (i.e., standard bump mapping).

At this point we have an image that displays an ambient only term in shadowed regions and normal bump mapping in non-shadowed regions. We have found that a pleasing minor variant is to create lighter shadows by having the shadowed regions contain a toned down diffuse term rather than ambient only. This can be done in a fourth pass by first setting the stencil function to only draw pixels that do NOT have the stencil bit set and by then drawing the geometry one more time using normal bump mapping, but with a scaled-down diffuse term. This fourth pass can also be combined with the first ambient pass. Iterating on creating the light dependent horizon map and the resulting alpha test can be done if you have multiple lights.

This algorithm can be implemented in any hardware by not leveraging dot product fragment operations, but at a significant performance penalty. It can be coded in D3D using DX7 (using the DOTPRODUCT3 texture mode), but only 3 directions can be packed in a texture, however D3D support rasterizing directly into texture memory and in DX7 this particular function/API is supported on a wider variety of boards. DX8 will provide more general fragment processing than the register combiner extensions.

5 Results and Comments

We have computed normal maps and horizon maps for several surfaces. At 512x512 resolution in texture space it takes approximately two minutes to precompute the object space dependent horizon maps for 8 directions. The current implementation is written in OpenGL and leverages the NVidia register combining extensions [1] to do 4 directions in a single pass. The triangle rendering part of the code has not been optimized to use the vertex array extensions yet. Thus, the geometry can have a significant impact on performance. We have tested the self-shadowing algorithm on three objects: a simple plane (see Figure 2), a cylinder with 160 vertices (158 triangles) (Figure 4, and a tessellated BSpline surface (1600 vertices, 3042 triangles) (Figure 5).

The performance for the plane is around 42hz, 36hz for the cylinder and 33hz for the surface[1]. The main bottleneck is the copy-to-texture space, but the geometry code

[1]All timings were taken on a 733mhz PIII using a GeForce based video boards with 32MB of DDR SDRAM

paths are causing a performance hit as well.

The results are very encouraging for shadowing from bump maps. There are several changes to the graphics pipeline that would reduce the number of passes required. The most significant would be putting the alpha and stencil tests after the blending of fragments with the color already in the frame buffer. This way the frame buffer to texture memory copy could be eliminated. This would be done by writing the light direction in tangent space into the alpha channel when the ambient term is written, and then subtracting the contributions from each of the direction passes. This would also eliminate one rendering pass for the geometry. Having a more flexible vertex shader [3] would allow the (s, t) texture coordinates to not be computed on the fly and work optimally with the Transform and Lighting hardware. Also if hardware supported a dependent texture-read[2], the soft shadowing used by Max could be implemented as well.

The most obvious limitation of this method are that it cannot cast bumpy shadows onto other objects, and other objects cast shadows onto the receiving surface as if it was not bumped. For future work, it would be interesting to look at other ways of precomputing self-shadowing. The horizon map is a function over orientation in the tangent plane, there are other functions that could leverage this parameterization we are currently investigating applying it to mip-mapping of normal maps - representing variance as a function of orientation in the tangent plane.

References

1. Nvidia web page. htt://www.nvidia.com.
2. Blinn, J. F. Simulation of wrinkled surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78) 12*, 3 (August 1978), 286–292. Held in Atlanta, Georgia.
3. D.McCool, M., and Heidrich, W. Texture shaders. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 1999), 117–126. Held in Los Angeles, California.
4. Heidrich, W., Daubert, K., Kautz, J., and Seidel, H.-P. Illuminating micro geometry based on precomputed visibility. *Proceedings of SIGGRAPH 2000* (July 2000).
5. 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.
6. Heidrich, W., Westermann, R., Seidel, H.-P., and Ertl, T. Applications of pixel textures in visualization and realistic image synthesis. *1999 ACM Symposium on Interactive 3D Graphics* (April 1999), 127–134. ISBN 1-58113-082-1.
7. Max, N. L. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (July 1988), 109–117.
8. Miller, G. Efficient algorithms for local and global accessibility shading. *Proceedings of SIGGRAPH 94* (July 1994), 319–326. ISBN 0-89791-667-0. Held in Orlando, Florida.
9. Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., and Haeberli, P. E. Fast shadows and lighting effects using texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 92) 26*, 2 (July 1992), 249–252. ISBN 0-201-51585-7. Held in Chicago, Illinois.
10. Williams, L. Casting curved shadows on curved surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78) 12*, 3 (August 1978), 270–274. Held in Atlanta, Georgia.
11. Woo, A., Poulin, P., and Fournier, A. A survey of shadow algorithms. *IEEE Computer Graphics & Applications 10*, 6 (November 1990), 13–32.
12. Zhang, H. Forward shadow mapping. *Eurographics Rendering Workshop 1998* (June 1998), 131–138. ISBN 3-211-83213-0. Held in Vienna, Austria.

[2]This is available in DX7, but only the Matrox G400 and ATI Rage6 currently supports it

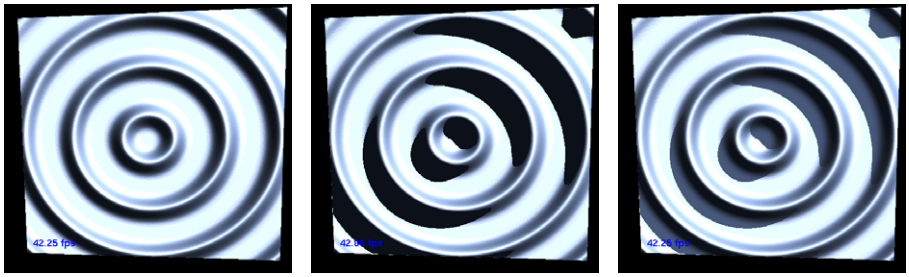

Fig. 2. Plane no shadow, dense shadow, light shadow

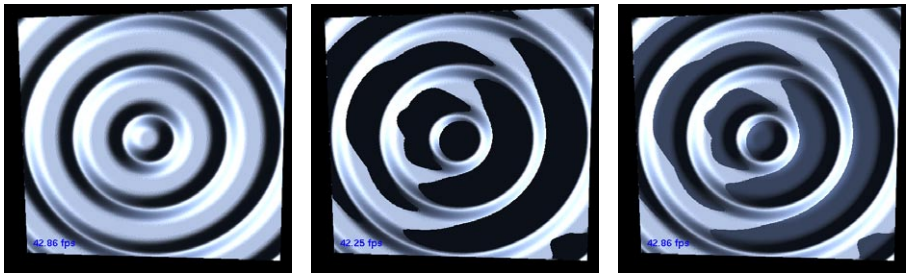

Fig. 3. Different light angle and more extreme bump height, tough case for 8 directions

Fig. 4. Cylinder no shadows, cylinder dense shadows, cylinder light shadows

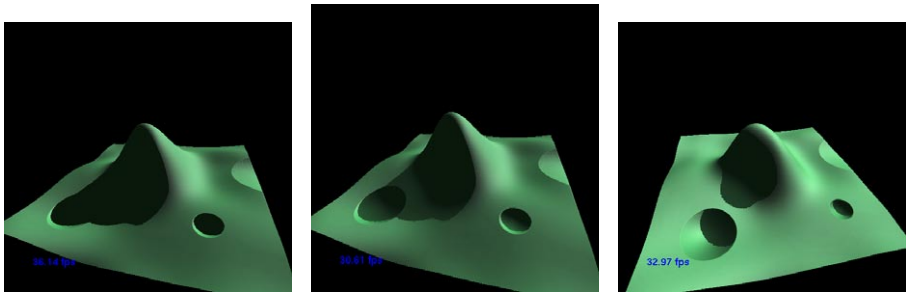

Fig. 5. Surface dense shadow, light shadow, different view light shadow