Chapter 8

Ambient Aperture Lighting

Chris Oat[12] and Pedro Sander[13]
ATI Research

Figure 1: *A terrain is rendered in real-time and shaded using the Ambient Aperture Lighting technique described in this chapter.*

8.1 Abstract

A new real-time shading model is presented that uses spherical cap intersections to approximate a surface's incident lighting from dynamic area light sources. This method uses precomputed visibility information for static meshes to compute illumination, with approximate shadows, from dynamic area light sources at run-time. Because this technique relies on precomputed visibility data, the mesh is assumed to be static at render-time (i.e. it is assumed that the precomputed visibility data remains valid at run-time). The ambient aperture shading model was developed with real-time terrain rendering in mind (see Figure 1 for an example) but it may be used for other applications where fast, approximate lighting from dynamic area light sources is desired.

[12] coat@ati.com
[13] psander@ati.com

8.2 Introduction

Shadow mapping large terrains is frequently impractical for real-time applications because it requires transforming and rasterizing the terrain data multiple times per frame. Additionally, shadow maps are ill suited for terrain rendering as they don't allow for area light sources such as the sun and the sky. Finally, shadow maps exhibit aliasing artifacts particularly when used with large scenes such as terrains. Horizon mapping techniques for rendering self-shadowing terrains in real-time only allow point or directional light sources [Max88]. Precomputed Radiance Transfer allows for area light sources but assumes low-frequency lighting environments [Sloan02].

Figure 2: *[Left] A spherical cap is used to approximate a contiguous region of visibility (an aperture) on a point's upper hemisphere. [Right] The visible region acts as an aperture which restricts light such that it only reaches the point from visible (unoccluded) directions.*

Our approach allows for object self-shadowing from dynamic area light sources with the caveat that the object is not deformable (only rigid transformations are supported). Our technique stores a small amount of precomputed visibility data for each point on the mesh. The visibility data approximates contiguous regions of visibility over a point's upper hemisphere using a spherical cap (see Figure 2). This spherical cap acts as a circular aperture for incident lighting. The aperture is oriented on the hemisphere in the direction of average visibility and prevents light originating from occluded directions from reaching the point being rendered. At render time, area light sources are projected onto the point's upper hemisphere and are also approximated as spherical caps. Direct lighting from the area light source is computed by determining the area of intersection between the visible aperture's spherical cap and the projected area light's spherical cap. The overall diffuse response at the point is computed by combining the area of intersection with a diffuse falloff term described later.

The algorithm comprises of two stages. In a preprocessing stage, a contiguous circular region of visibility is approximated for each point on the surface of the model (Section 8.3). The data can be computed per texel and stored in a texture map, or per vertex and incorporated into the model's vertex buffer. Then, at rendering time, a pixel shader computes the amount of light that enters this region of visibility, and uses this result to shade the model (Section 8.4).

8.3 Preprocessing

Given a point – represented by a vertex on a mesh or by a texel in texture space – we wish to find its approximate visible region. The visible area at a point **x** is found by integrating a visibility function over the hemisphere. The visibility function evaluates to 1 for rays that don't intersect the scene and 0 otherwise. The percentage of visibility is then scaled by the area of a unit hemisphere and results in the solid angle of the visible region:

$$VisibleArea(x) = 2\pi \int_{\Omega} V(x,\omega) d\omega$$

This visible area is used as our aperture area. We do not store the area directly but instead store the arc length of a spherical cap of equivalent area (since this is what we need at run-time):

$$SphericalCapRadius = a\cos\left(\frac{-area}{2\pi} + 1\right)$$

The spherical cap's radius is a single floating point number that must be stored with the mesh (either per-vertex or per-texel).

The visible region's orientation on the hemisphere is determined by finding the average direction for which the visibility function evaluates to 1 (a.k.a. bent normal):

$$VisibleDir(x) = \int_{\Omega} V(x,\omega)\omega d\omega$$

This gives us an average direction of visibility and serves as our aperture's orientation on the surrounding hemisphere. This direction is stored as three floating point values (a three component vector) either per-vertex or per-texel.

8.4 Rendering

First we describe our rendering algorithm which, for each pixel, seeks to compute the intersection between the precomputed circular visibility region and the circular light source (projected area light). Then, we describe an optimization which significantly reduces computation time while achieving nearly identical qualitative results.

8.4.1 Intersecting Spherical caps

At rendering time, the spherical area light sources are projected onto the hemisphere and the radius of the light's enclosing spherical cap is computed. The amount of light that reaches a point is determined by computing the area of intersection between the precomputed spherical cap representing the aperture of visibility and the spherical light source as shown in Figure 3.

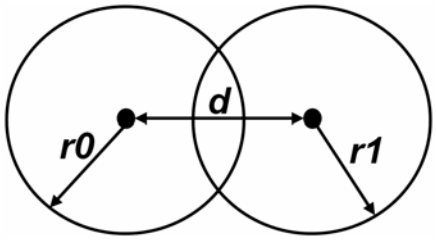


Figure 3: *The intersection area of two spherical caps is a function of the arc length of their radii (**r0**, **r1**) and arc length of the distance (**d**) between their centroids.*

The area of intersection for two spherical caps is computed as follows:

$$\begin{cases} 2\pi - 2\pi\cos(\min(r1,r0)) & \min(r1,r0) \le \max(r1,r0) - d \\ 0 & r0 + r1 \le d \\ \text{IntersectArea}(r0,r1,d) & \text{otherwise} \end{cases}$$

The first case, when ***min(r0,r1) <= max(r0,r1)-d*** occurs, the caps are fully intersected. This means that one cap is entirely inside of the other cap (or both caps are the same size and are perfectly aligned) and so we know the intersection area must be equal to the area of the smaller of the two caps. The next case occurs when the caps are far enough away from each other that we know that no intersection can possible occur (***r0+r1 <= d***), in this case the intersection area must be zero. Finally, if neither of the previous cases apply, there must be some partial intersection occurring so we must solve for the intersection area using our `IntersectArea` function which solves the following:

$$\begin{aligned} & 2\cos(r1)\arccos\left(\frac{-\cos(r0)+\cos(d)\cos(r1)}{\sin(d)\sin(r1)}\right) \\ & -2\cos(r0)\arccos\left(\frac{\cos(r1)-\cos(d)\cos(r0)}{\sin(d)\sin(r0)}\right) \\ & -2\arccos\left(\frac{-\cos(d)+\cos(r0)\cos(r1)}{\sin(r0)\sin(r1)}\right) \\ & -2\pi\cos(r1) \end{aligned}$$

which is a simplified form of the result from [Tovchigrechko01]. Once the area of intersection is found, the net diffuse response is approximated by scaling the area of intersection by a Lambertian coefficient (cosine of the angle between the intersection centroid and the geometric surface normal, explained in section 8.4.3).

8.4.2 Optimization

The spherical cap intersection function (*IntersectArea*) may be too expensive to solve directly for many mid-range GPUs. Instead of solving this expensive function directly it may be desirable to use a less expensive approximation. Because the intersection function exhibits a smooth falloff with respect to increasing cap distance (see Figure 4) a *smoothstep* function is an appropriate approximation:

$$(2\pi - 2\pi\cos(\min(r1, r0)))\ \ \text{smoothstep}(0,1,1-\frac{d-|r0-r1|}{r0+r1-|r0-r1|})$$

The *smoothstep* function here returns 1.0 when a full intersection occurs and 0.0 when no intersection occurs. The result of the *smoothstep* is then scaled by the area of the smaller of the two spherical caps so that the end result is a smooth transition between full intersection and no intersection. Empirically, we did not notice any significant qualitative difference between using the *smoothstep* approximation and the actual intersection function. Figures 5 and 6 compare the exact intersection function and its approximation for both small and large circular light source radii. Note that the differences, which can only occur in the penumbra region, are imperceptible.

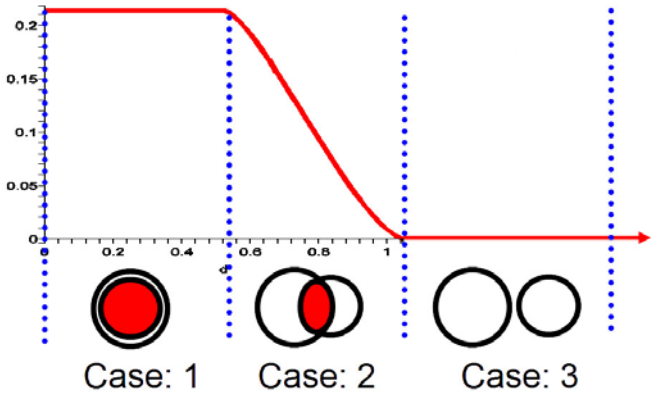
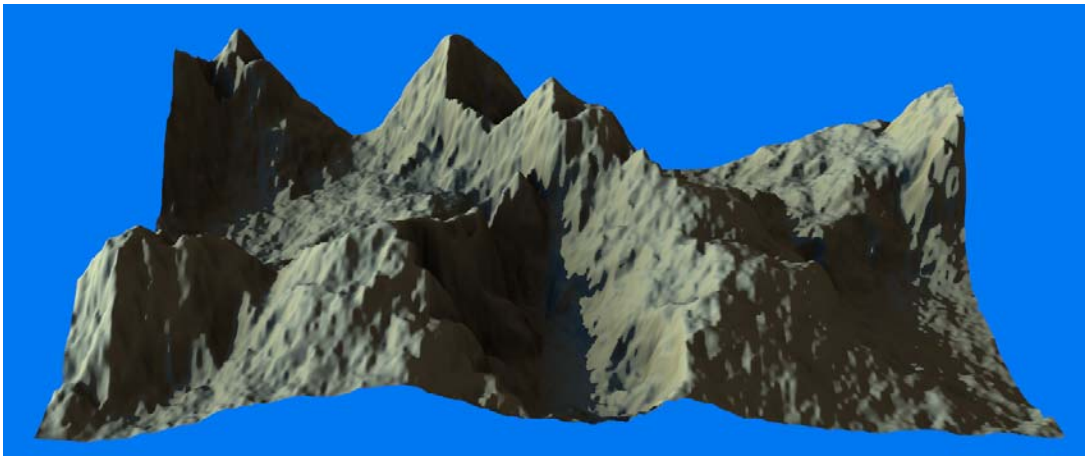
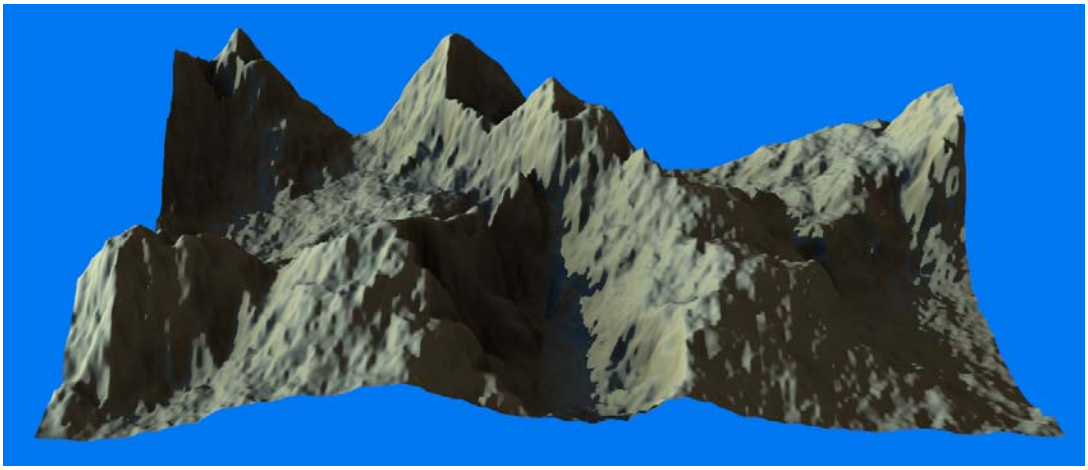


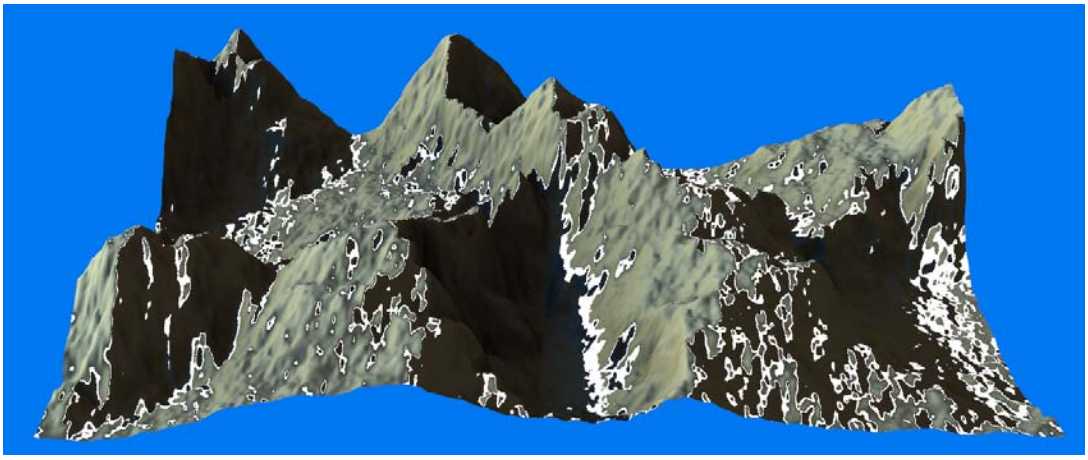
Figure 4: *Spherical cap intersection as a function of distance between spherical cap centers. Case 1: One cap entirely overlaps the over, full intersection occurs. Case 2: The full intersection function is evaluated to find the area of intersection between partially overlapping spherical caps. Case 3: The caps are too far apart for any intersection to occur, their intersection area is zero.*



(a) Exact result

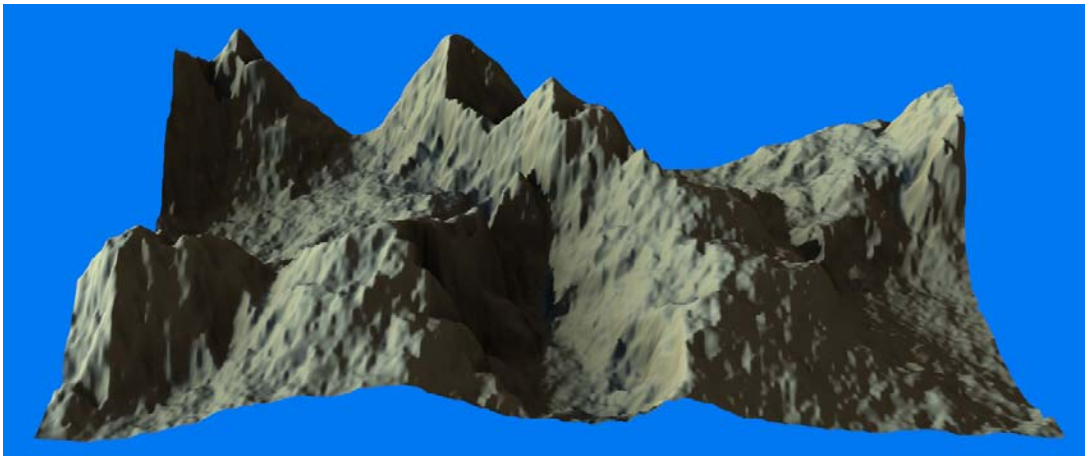


(b) Approximate results

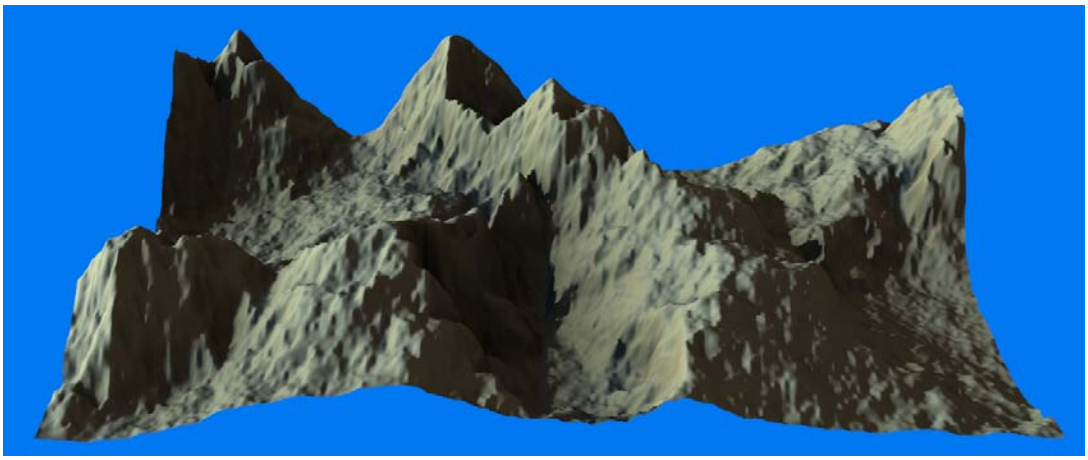


(c) Penumbra region color-coded in white

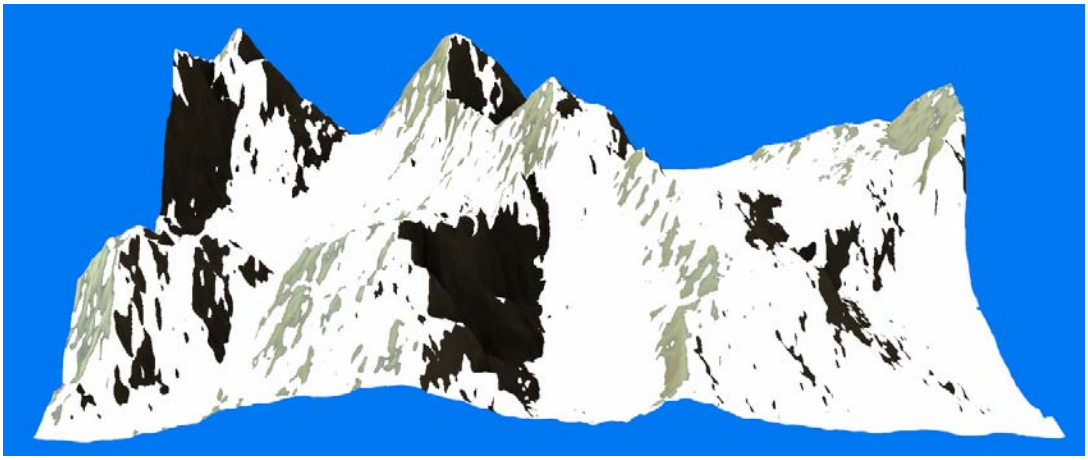
Figure 5: *Rendering results using a light source with a small radius.*



(a) Exact result



(b) Approximate results



(c) Penumbra region color-coded in white

Figure 6: *Rendering results using a light source with a large radius.*

```
// Approximate the area of intersection of two spherical caps
// fRadius0 : First cap's radius (arc length in radians)
// fRadius1 : Second caps' radius (arc length in radians)
// fDist : Distance between caps (radians between centers of caps)
float SphericalCapIntersectionArea(float fRadius0, float fRadius1, float fDist)
{
   float fArea;

   if ( fDist  <= max(fRadius0, fRadius1) - min(fRadius0, fRadius1) )
   {
      // One cap in completely inside the other
      fArea = 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }
   else if ( fDist >= fRadius0 + fRadius1 )
   {
      // No intersection exists
      fArea = 0;
   }
   else
   {
      // Partial intersection exists, use smoothstep approximation
      float fDiff = abs(fRadius0 - fRadius1);
      fArea=smoothstep(0.0,
                       1.0,
                       1.0-saturate((fDist-fDiff)/(fRadius0+fRadius1-fDiff)));
      fArea *= 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }

   return fArea;
}
```

Listing 1: *An HSLS function that computes the approximate area of intersection of two spherical caps using our smoothstep approximation for partially overlapping spherical caps.*

8.4.3 Our Friend Lambert

Knowing the area of intersection between the aperture's spherical cap and the light source's spherical cap isn't enough to compute lighting at the point, it really only tells us how much unoccluded light arrived at the point we're rendering. A given area of intersection will result in more or less diffuse reflection depending on its orientation relative to the point's surface normal. In other words, we must take Lambert's Cosine Law into account. The correct way to handle this would be to convolve the intersection area with a cosine kernel, but this would require solving an integral and is too expensive (if we wanted to solve integrals, we wouldn't be mucking about with spherical caps in the first place!). Instead, we account for the Lambertian falloff by first finding a vector from the point we're shading to the center of the area of intersection (i.e. the intersection region's centroid). A vector to the intersection region's centroid is estimated by averaging the aperture's orientation vector with the light's vector. We then compute the dot product between this vector and the point's geometric normal. This dot product is used to scale the incoming lighting and it results in a Lambertian falloff as the area of intersection approaches the horizon. For this approximation to be correct, we must

assume that the area above the intersection region's centroid is about the same as the area below the intersection region's centroid (see figure 7).

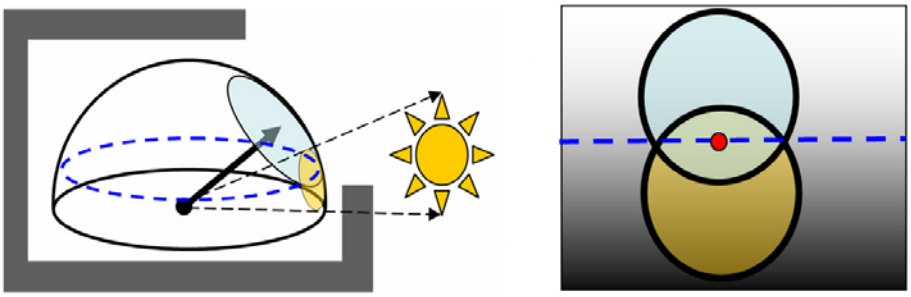


Figure 7: *[Left] The aperture's spherical cap is intersected with a light source's spherical cap. [Right] A rectangular plot of the two spherical caps along with the Lambertian falloff function (i.e. the cos(theta) term). Our approximation assumes that the intersection area above the intersection region's centroid (red) is the same as the intersection area below the region's centroid.*

This gives us a nice approximation of direct, diffuse lighting on our mesh but it doesn't handle indirect lighting at all and thus we require some additional terms to account for ambient lighting.

8.4.4 Ambient Light

What if light occupies only a portion of our aperture's area? Or none of the aperture's area? For outdoor lighting scenarios we should still get some indirect light scattering into our aperture and this needs to be accounted for in our shading model. The indirect sky light can be handled by filling the empty portion of our aperture with ambient light from the sky. Once the area of the light/aperture intersection is found, we can subtract this area from the aperture's total area to figure out how much of the aperture is unoccupied by the sun. This empty space can then be filled with indirect, ambient light. For outdoor rendering, we can use the average sky color for our ambient light. The average sky color could be found by sampling the lowest MIP level of a sky dome, for example, or it could just be derived based on time of day:

- Sky blue during the day
- Red or pink at sun set
- Black or deep blue at night

This works much nicer than the standard constant ambient term since it only applies to areas that aren't being lit directly and aren't totally occluded from the outside world. This method's advantage is that it doesn't destroy scene contrast by adding ambient light in areas that really should be dark because they're mostly occluded from the outside world.

8.5 Limitations

Ambient aperture lighting makes a lot of simplifying assumptions and approximations in order to reduce lighting computations when rendering with dynamic area light sources. There are certain cases where the assumptions break down and the shading model produces artifacts such as incorrect shadowing. One such assumption is that the visible region for any point on a mesh is both contiguous and circular. This is generally true for things like terrains where a point's visible region is more-or-less the horizon but it fails to handle the case of, for example, a sphere over a plane (see Figure 8).

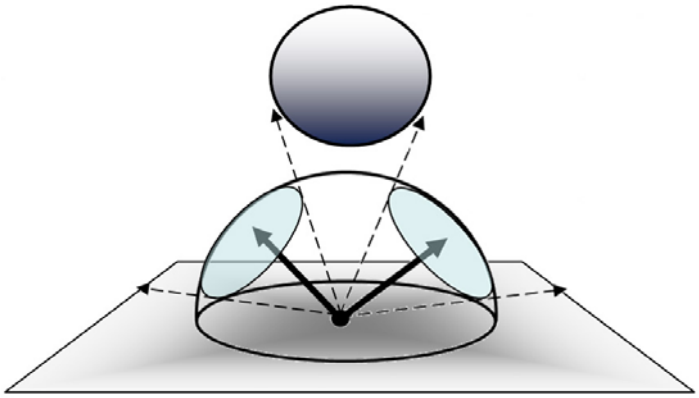


Figure 8: *Which way should the visibility aperture point? The visible region here is a band around the horizon which can't be closely approximated by a spherical cap.*

8.6 Conclusion

A real-time shading model that allows for dynamic area light sources was presented. This technique is well suited for outdoor environments lit by dynamic spherical area light sources (such as the sun) and is particularly useful for applications where fast terrain rendering is needed. Ambient aperture lighting makes several simplifying assumptions and mathematical approximations to reduce the computational complexity and storage costs associated with other real-time techniques that allow for dynamic area light sources.

8.7 Bibliography

MAX, N. L. *Horizon Mapping: Shadows for Bump-mapped Surfaces*. The Visual Computer 4, 2 (July 1988), 109-117.

SLOAN, P.-P., KAUTZ, J., SNYDER, J., *Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments*, SIGGRAPH 2002.

TOVCHIGRECHKO, A. AND VAKSER, I.A. 2001. *How common is the funnel-like energy landscape in protein-protein interactions?* Protein Sci. 10:1572-1583