Robust Diffuse Final Gathering on the GPU

Tamás Umenhoffer and László Szirmay-Kalos

Dept. of Control Engineering and Information Technology
Budapest University of Technology and Economics
Magyar Tudósok krt. 2., H-1117, Hungary
szirmay@iit.bme.hu

ABSTRACT

This paper presents a method to obtain the approximate indirect diffuse reflection on a dynamic object, caused by a diffuse or a moderately glossy environment. Instead of tracing rays to find the incoming illumination, we look up the indirect illumination from a cube map rendered from the reference point that is in the vicinity of the object. However, to cope with the difference between the incoming illumination of the reference point and of the shaded points, we apply a correction that uses geometric information also stored in cube map texels. This geometric information is the distance between the reference point and the surface visible from a cube map texel. The method computes indirect illumination albeit approximately, but providing very pleasing visual quality. The method fits very well the GPU architecture, and can render these effects interactively. The primary application area of the proposed method is the introduction of diffuse interreflections in games.

Keywords: Global illumination, GPU programming.

1 INTRODUCTION

Final gathering, i.e. the computation of the reflection of the indirect illumination toward the eye, is one of the most time consuming steps of realistic rendering. According to the rendering equation, the irradiance of point $\vec{x}$ can be expressed by the following integral

$$I(\vec{x}) = \int_S L^{in}(\vec{x} \leftarrow \vec{y}) \cdot v(\vec{x},\vec{y}) \cdot \frac{\cos^+ \theta_{\vec{x}} \cdot \cos^+ \theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}\, dy,$$

where S is the set of surface points, $L^{in}(\vec{x} \leftarrow \vec{y})$ is the incoming radiance arriving at point $\vec{x}$ from point $\vec{y}$, v is the visibility indicator between two points, and $\theta_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the illumination direction and the surface normals at points at $\vec{x}$ and $\vec{y}$, respectively. When the angles get larger than 90 degrees, their cosine should be replaced by zero, which is indicated by superscript $+$.

The evaluation of this integral usually requires many sampling rays from each shaded point. Ray casting finds *illuminating points* $\vec{y}$ for *shaded point* $\vec{x}$ at different directions (Figure 1), and the radiance of these illumination points is inserted into a numerical quadrature approximating the rendering equation. In practice, number P of shaded points is over hundred thousands or millions, while number D of sample directions is about a hundred or a thousand to eliminate annoying sampling artifacts. On the other hand, in games and in real-time systems, rendering cannot take more than a few tens of milliseconds. This time does not allow tracing $P \cdot D$, i.e. a large number of rays.

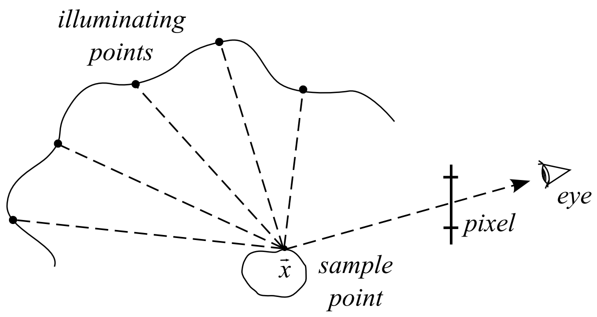


Figure 1: *Indirect illumination with sampling rays.*

To solve this complexity problem, we can exploit the fact that in games the dynamic objects are usually significantly smaller than their environment. Thus the global indirect illumination of the environment can be computed separately, since it is not really affected by the smaller dynamic objects. On the other hand, when the indirect illumination of dynamic objects is evaluated, their small size makes it possible to reuse illumination information obtained when shading its other points.

The first idea of this paper is to reuse illuminating points when the illumination on the dynamic object is computed. It means that first we obtain a set of virtual lights that represent the indirect illumination and calculate just the single reflection of these lights during final gathering (Figure 2). During this, we ignore complex self-shadowing of the dynamic object and apply just a simple test based on the normal vector and the illumination direction to determine whether the virtual light may illuminate the point.

This approach has two advantages. On the one hand, instead of tracing $P \cdot D$ rays, we solve the rendering problem by tracing only D rays. Assuming that the same set of illuminating points are visible from each

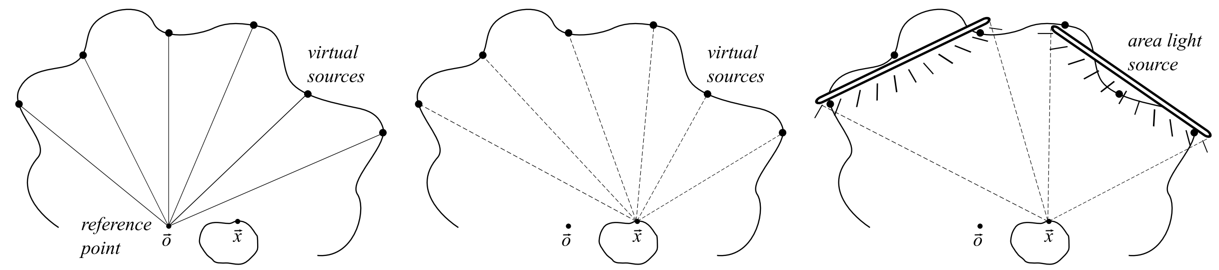


Figure 2: *The basic idea of the proposed method: first virtual lights sampled from reference point $\vec{o}$ are identified, then these point lights are grouped into large area lights. At shaded points $\vec{x}$ the illumination of a relatively small number of area lights is computed without visibility tests.*

shaded point, self-shadowing effects are ignored. However, while shadows are crucial for direct lighting, shadows from indirect lighting are not so visually important. Thus the user or the gamer finds this simplification acceptable.

Unfortunately, this simplification alone cannot allow real time frame rates. The evaluation of the reflected radiance at a shaded point still requires the evaluation of the irradiance and the orientation angle, and the multiplication with the diffuse reflectance for all directions. Although the number of rays traced to obtain indirect illumination is reduced from $P \cdot D$ to D, the illumination formula must be evaluated $P \cdot D$ times. These computations would still need too much time.

In order to further increase the rendering speed, we propose to carry out as much computation globally for virtual lights, as possible. Intuitively, global computation means that the sets of virtual light sources are replaced by larger homogeneous area light sources. Since the total area of these lights is assumed to be visible, the reflected radiance can be analytically evaluated once for a whole set of virtual light sources.

2 PREVIOUS WORK

Methods for reusing illumination information for multiple points of a shaded object can be classified according to whether they encode the irradiance of the receivers [17] or consider the environment as a set of virtual light sources [6].

Environment mapping [2] has been originally proposed to render ideal mirrors in local illumination frameworks, but can also be applied for glossy and diffuse reflections as well. The usual trick is the convolution of the angular variation of the BRDF with the environment map during preprocessing [11, 7] to obtain the irradiance of surfaces of all possible orientations. This step enables us to determine the illumination of an arbitrarily oriented surface patch with a single environment map lookup during rendering. A fundamental problem of this approach is that the generated environment map correctly represents the direction dependent illumination only at a single point, the reference point of the object. For other points, the environment map is only an approximation, where the error depends on the ratio of the distances between the point of interest and the reference point, and between the point of interest and the surfaces composing the environment (see figure 4).

One possible solution is to use multiple environment maps [4, 18, 10], which can be compressed using spherical harmonics [11, 8] or wavelets [18]. For example, Greger et al. [4] calculate and store the direction dependent illumination in the vertices of a bi-level grid subdividing the object scene. During run-time, irradiance values of an arbitrary point are calculated by tri-linearly interpolating the values obtained from the neighboring grid vertices. They reported good results even with surprisingly coarse subdivisions. This means that the smooth irradiance function is especially suitable for interpolation. While Greger et al. used a precomputed radiosity solution to initialize the data structures, Mantiuk et al.[9] calculated these values during run-time using an iterative algorithm that simulates the multiple bounces of light. Unfortunately, the generation and compression of many environment maps require considerable time which is not available during real-time rendering. Thus most of this computation should be done during preprocessing, which imposes restrictions on dynamic scenes.

The idea of approximating the indirect illumination by a finite set of virtual lights was born in the context of global illumination algorithms, such as in instant radiosity [6], and has been used many times in Monte Carlo algorithms [16, 12, 13, 14].

For the computation of diffuse interreflections on the GPU, Dachsbacher [3] considered shadow map lexels as virtual lights, while Lazányi [5] assigned virtual lights to texels of an environment map. Indeed, if only two-bounce indirect illumination is considered, a shadow map lexel identifies the point which is directly illuminated by the light source. Such points may indirectly illuminate other points, so can be considered as virtual lights.

Virtual light source algorithms suppose that virtual lights are point sources, and thus use the point-to-point form factor (also called geometry factor) when the irradiance is computed. Unfortunately, point-to-point form factor

$$\frac{\cos\theta_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}$$

is numerically unstable since it goes to infinity when the virtual light gets close to the shaded point, which results in bright spikes making the position of the virtual lights clearly visible (see figures 4 and 5). Another problem is that not only the position of the virtual light source is required, but the normal vector is also needed to compute $\cos\theta_{\vec{y}}$, which doubles the required storage space and slows down the generation of the virtual light sources.

To address the numerical instability, in [5] virtual lights were supposed to be small disks, which made the geometry factor bounded. However, this approximation is still quite far from being precise when the shaded point is close to the virtual light and the shaded point is not in the normal direction from the virtual light source.

In this paper we address the numerical instability of virtual lights and eliminate the need of storing normal vectors at these lights. The basic idea is that we always consider four virtual lights that are close to each other and assume that the surface is roughly planar between them. Thus instead of point sources we have a set of homogeneous area light sources. Since self-shadowing is ignored, the reflection of these area light sources can be computed analytically, in a numerically stable way.

3 THE NEW ALGORITHM

The first step of the algorithm is the generation of virtual lights that may illuminate the given dynamic object. To find these points, the scene is rendered from reference point $\vec{o}$ of the dynamic object, and the resulting images are put into an environment map. Not only the radiance of the visible points is evaluated but the distance between the reference point and the visible surface is found and stored in the alpha channel of the generated cube map. In each texel of the environment map a potential virtual light source is visible.

In order to estimate the integral of the rendering equation, the set of surface points visible from the reference point is partitioned according to the texels of the environment map. The surface between four points visible from the texel corners is approximated by a quadrilateral, thus the environment is assumed to be a list of quadrilaterals $S_i, i = 1,\dots,N$. After partitioning, the irradiance is expressed by the following sum:

$$I(\vec{x}) = \sum_{i=1}^{N} \int_{S_i} L(\vec{y}) \cdot \frac{\cos^{+}\theta_{\vec{x}} \cdot \cos^{+}\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}\, dy.$$

Since $dy \cos^{+}\theta_{\vec{y}}/|\vec{x}-\vec{y}|^2$ is the projection of area dy onto the surface of a unit hemisphere placed around $\vec{x}$, and factor $\cos^{+}\theta_{\vec{x}}$ represents a further projection from the sphere onto the tangent plane, the irradiance integral can also be evaluated on the tangent plane of the surface at $\vec{x}$:

$$I(\vec{x}) = \sum_{i=1}^{N} \int_{P_i} L(\vec{y}_p)\, dy_p$$

where P_i is the double projection of the visible surface, first onto the hemisphere then onto the tangent plane. Let us assume that $L(\vec{y}_p)$ is linear. In this case the irradiance is the product of the area of P_i and the average of the radiances stored at the four corner texels:

$$I(\vec{x}) = |P_i| \cdot \frac{L_i^1 + L_i^2 + L_i^3 + L_i^4}{4}$$

Let us consider a single term of this sum representing the radiance reflected from S_i.

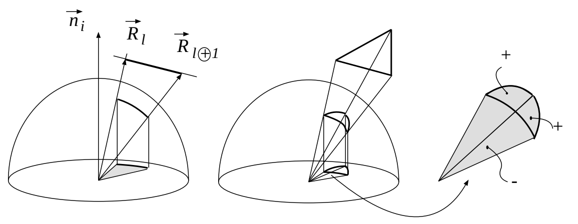


Figure 3: *Hemispherical projection of a planar polygon*

Area $|P_i|$ is in fact the polygon-to-point form factor [1]. Consider only one edge line of the polygon first, and two subsequent vertices, $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$, on it (figure 3). The hemispherical projection of this line is a half great circle. Since the radius of this great circle is 1, the area of the sector formed by the projections of $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$ and the center of the hemisphere is simply half the angle of $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$. Projecting this sector orthographically onto the equatorial plane, an ellipse sector is generated, having the area of the great circle sector multiplied by the cosine of the angle of the surface normal $\vec{n}_i$ and the normal of the segment $(\vec{R}_l \times \vec{R}_{l\oplus 1})$.

The area of the doubly projected polygon can be obtained by adding and subtracting the areas of the ellipse sectors of the different edges, as is demonstrated in figure 3, depending on whether the projections of vectors $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$ follow each other clockwise when looking at them from the direction of the surface normal. This sign value provided by the dot product of the cross product of the two vertex vectors and the normal vector. Finally, the double projected polygon is a summation:

$$\sum_{l=0}^{L-1} \frac{1}{2} \cdot \text{angle}(\vec{R}_l, \vec{R}_{l\oplus 1}) \cdot \left(\frac{(\vec{R}_l \times \vec{R}_{l\oplus 1})}{|\vec{R}_l \times \vec{R}_{l\oplus 1}|} \cdot \vec{n}_i \right). \qquad (1)$$

4 IMPLEMENTATION

The proposed algorithm first computes an environment cube map from the reference point and stores the radi-

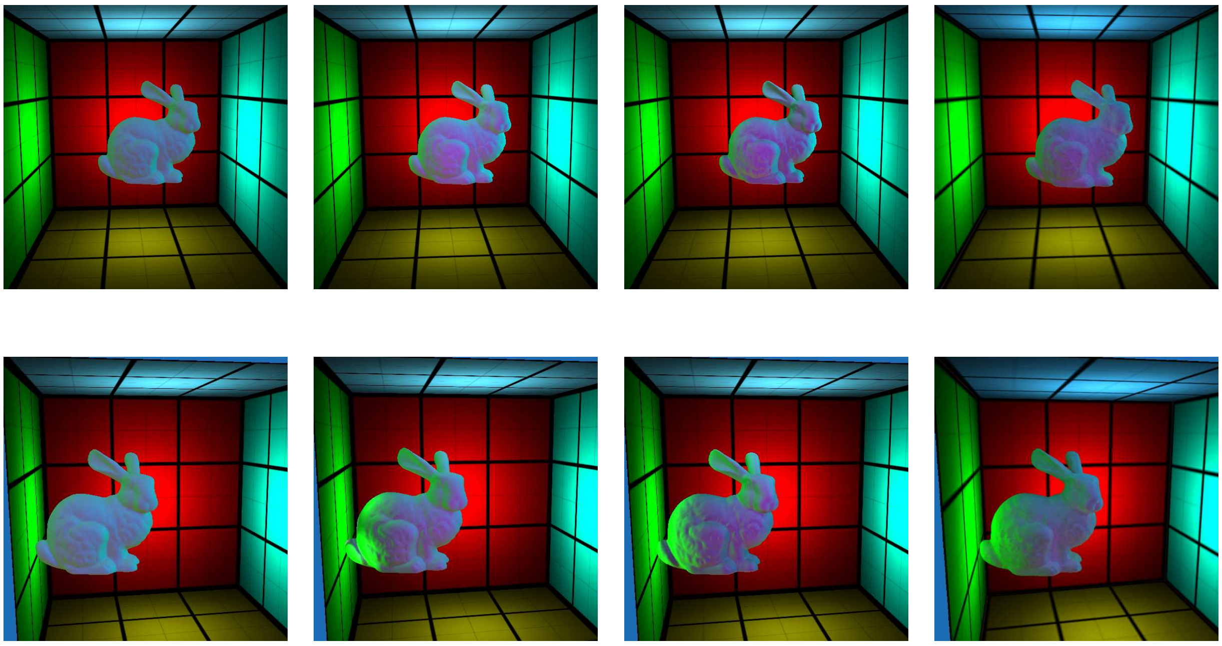


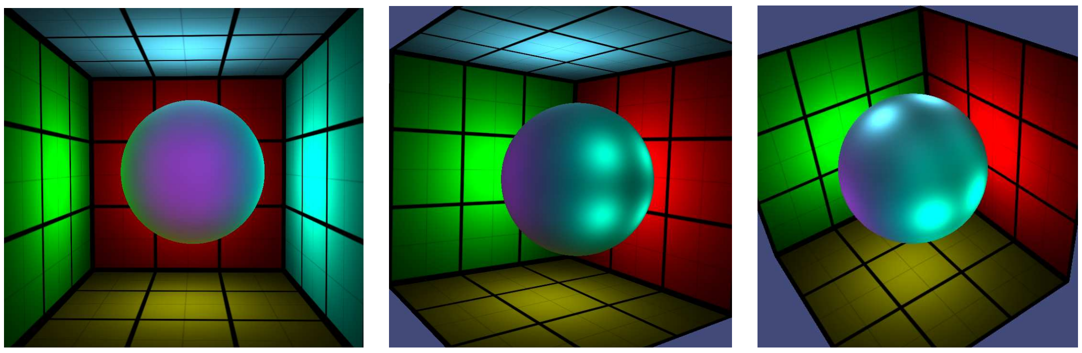
Figure 4: *The Stanford bunny model (about 25000 vertices and 50000 faces) in a colored box. On the right images diffuse reflections are calculated with only environment map convolution and one environment lookup. On the second column images point to point form factor while on the third column images polygon to point form factor is used. On the right column reference images rendered with Mental Ray are shown. The simple one environment map only produces pleasable results for points that are close to the environment reference point. The point to point form factor method can handle greater distances, but as the shaded surface gets close to the walls it gets unstable (see the bright green spikes on the bunny's tail and back). The polygon to point form factor has good results for any object positions. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 (SLI turned off), AMD Athlon64 Dual 4600+ processor.*

ance and distance values of the points visible in its pixels. We usually generate $6 \times 256 \times 256$ pixel resolution cube maps. Then the cube map is downsampled to have `M`×`M` pixel resolution faces (`M` is 4 or even 2). One texel of the low-resolution cubemap represents an area light source. Note that both radiance and distance values are averaged, thus finally we have larger lights having the average radiance of the small lights and placed at their average position.

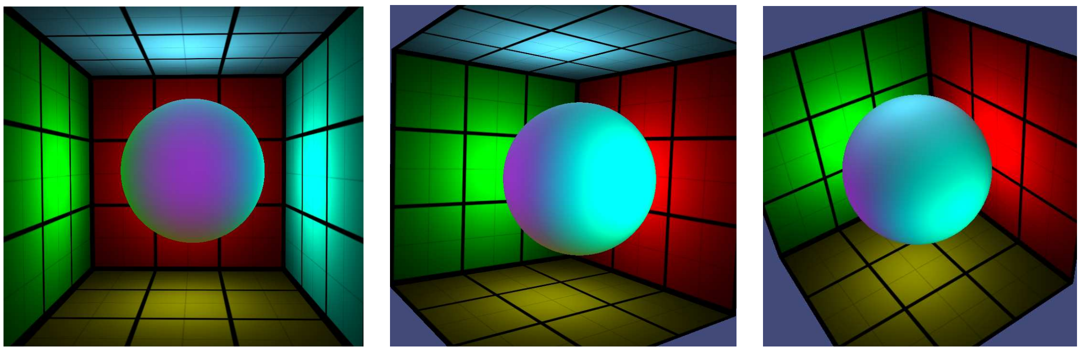
During final gathering for each texel in each cube face we should calculate the virtual light source's exact location and area and evaluate it's contribution to the reflected radiance. The following fragment shader code loops through the pixels of one cube face (in positive z direction), calculates pixel corner directions and calls the `getContr` function to evaluate texel contribution. The input parameters to this shader are: world space position (`pos`), world space normal (`N`), the reference point of the cubemap in world space (`referencePoint`), the diffuse coefficient (`kd`) and the cubemap sampler (`SmallEnvmap`). Our downsampled environment map had 4x4 pixel resolution(`M`).

```
float4 DiffuseReflectance(
            float4 hPosition   : POSITION,
            float3 pos         : TEXCOORD0,
            float3 N           : TEXCOORD1,
            uniform samplerCUBE SmallEnvMap,
            uniform float3 referencePoint,
            uniform float kd
                          )  :COLOR
{
 pos -= referencePoint;
 N = normalize(N);
  //reflected radiance
 float4 I = 0;
  //texel corner directions
 float3 L1, L2, L3, L4;
 float3 L; //texel center direction
 float4 Le; //texel color
 float width = 1.0 / M; //width of one texel
 float dWidth = 2.0 * width; //double width

 for (int x = 0; x < M; x++)
  for (int y = 0; y < M; y++)
  {
    //calculate texel coordinates
   float2 p;
   p.x = x * width;
   p.y = y * width;
   p = 2.0 * p - 1.0; //to range [-1,1]
```



150 FPS without SLI, 185 FPS with SLI turned on



32 FPS without SLI, 63 FPS with SLI turned on

Figure 5: *Sphere (about 2300 vertices and 2300 faces) in a colored box. Upper row with point to point form factor, bottom row with our polygon to point form factor. As the object gets closer to the walls the virtual light sources become visible in case of point to point form factor, while the polygon to point form factor method does not have this artifact. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 with SLI support, AMD Athlon64 Dual 4600+ processor.*

```
    //calculate texel corner directions
   L1 = float3(p.x, p.y, 1);
   L2 = float3(p.x + dWidth, p.y, 1);
   L3 = float3(p.x + dWidth, p.y + dWidth, 1);
   L4 = float3(p.x, p.y + dWidth, 1);
    //calculate texel center direction
   L = float3(p.x + width, p.y + width, 1);

    //read texel color
   Le = float4(texCUBE(SmallEnvMap, L).rgb,1);
    //get contribution from texel
   I += 0.5 * Le * getContr(L1, L2, L3, L4,
                            pos, N,
                            SmallEnvMap);
  }
  return kd * I;
}
```

The following code shows the `getContr` function which calculates the contribution of a single texel of the downsampled, low resolution cubemap `SmallEnvMap` to the illumination of the shaded point. It's input arguments are: the four pixel corner directions (`L1, L2, L3, L4`), the shaded point position and normal (`pos, N`) and the cubemap sampler (`SmallEnvMap`). This function calculates the exact position of pixel corners with the help of the stored distance values in the cubemap's alpha channel. Then evaluates the polygon-to-point form factor described above: calculates the four triangle areas given by the four edges of the texel and summs their signed values. The contribution from light sources located behind the tangent plane can be eliminated by ignoring the sums with negative values.

```
float4 getContr(float3 L1, float3 L2,
                float3 L3, float3 L4,
                float3 pos, float3 N,
                samplerCUBE SmallEnvMap)
{
  //texel corner distances and positions
  //from reference point
 float d;
 d = texCUBE(SmallEnvMap, L1).a;
 L1 = d * normalize(L1);
 d = texCUBE(SmallEnvMap, L2).a;
 L2 = d * normalize(L2);
 d = texCUBE(SmallEnvMap, L3).a;
 L3 = d * normalize(L3);
 d = texCUBE(SmallEnvMap, L4).a;
 L4 = d * normalize(L4);
```

```
  // corner directions from shaded point
 float3 r1 = normalize(L1 - pos);
 float3 r2 = normalize(L2 - pos);
 float3 r3 = normalize(L3 - pos);
 float3 r4 = normalize(L4 - pos);

 //calculate projected triangle areas
 float3 crossP = cross(r1, r2);
 float r = length(crossP);
 float dd = dot(r1, r2);
 float tri1 = acos(dd) * dot(crossP / r, N);

 crossP = cross(r2, r3);
 r = length(crossP);
 dd = dot(r1,r2);
 float tri2 = acos(dd) * dot(crossP / r, N);

 crossP = cross(r3, r4);
 r = length(crossP);
 dd = dot(r1,r2);
 float tri3 = acos(dd) * dot(crossP / r, N);

 crossP = cross(r4, r1);
 r = length(crossP);
 dd = dot(r1,r2);
 float tri4= acos(dd) * dot(crossP / r, N);

 //summation of triangle areas
 return max(tri1 + tri2 + tri3 + tri4, 0);
}
```

5 RESULTS

In order to demonstrate the results, we took a simple environment consisting of a colored cubic room. The first set of pictures shows the Stanford bunny model inside the room (figure 4). The images of the first column were rendered by the traditional environment mapping technique for diffuse materials where a precalculated convolution enables us to determine the irradiance at the reference point with a single lookup. This method has correct results only at the reference point and cannot deal with the position of the object, thus the bunny looks similar everywhere. The second column shows the point to point form factor. Although this method have pleasing results for points other than the reference point, it has artifacts for points too close to the virtual light sources. The right column shows our polygon-to-point form factor method which has correct results for arbitrary point positions. The difference between point to point and polygon-to-point factors can clearly be seen in the second set of pictures (figure 5). The first row shows point to point while the second row shows polygon-to-point form factor results. While the two methods show similar results when the object is in the center of the room there is a significant improvement in case of the polygon-to-point factor method when the sphere gets close to the walls.

The third set of pictures (figure 6) shows the Stanford Buddha model in the room with point to point (left) and polygon-to-point form factors (right). The images on the left show visual errors at surface points near the environment (see the keystone of the statue).

In figure 7 the Buddha model is placed in a dimly lit hallway. This environment was implemented in an actual game engine called OGRE. The scene has per pixel shading and reflections too.

In figure 8 the Buddha was placed in an other complex environment: a scientific laboratory. These pictures were also taken from an OGRE implementation.

6 CONCLUSIONS

This paper presented a GPU based method for computing diffuse reflections of the incoming radiance stored in environment maps. The environment map is considered as a definition of large area light sources whose reflections are obtained analytically without checking self-shadowing. The presented method runs in real-time and provides visually pleasing results.

ACKNOWLEDGEMENTS

This work has been supported by OTKA (T042735), GameTools FP6 (IST-2-004363) project, and by the National Office for Research and Technology (Hungary).

REFERENCES

[1] D.R. Baum, H.E. Rushmeier, and J.M. Winget. Improving radiosity solutions through the use of analytically determined form-factors. *CComputer Graphics (SIGGRAPH '89 Proceedings)*, pages 325–334, 1989.

[2] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[3] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 203–231, New York, NY, USA, 2005. ACM Press.

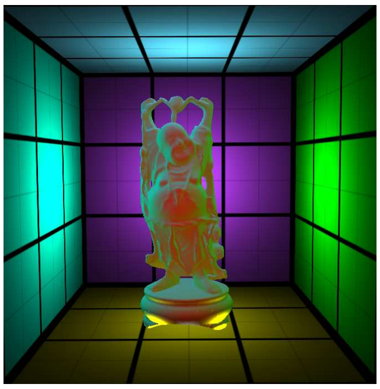
[4] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, March/April 1998.

[5] László Szirmay Kalos and István Lazányi. Indirect diffuse and glossy illumination on the gpu. In *SCCG 2006*, pages 29–35, 2006.

[6] A. Keller. Instant radiosity. In *SIGGRAPH '97 Proceedings*, pages 49–55, 1997.

[7] Gary King. Real-time computation of dynamic irradiance environment maps. In Parr M., editor, *GPU Gems II*, pages 167–170. Addison-Wesley, 2005.

[8] A.W. Kristensen, T. Akenine-Moller, and H.W. Jensen. Precomputed local radiance transfer for real-time lighting design. In *SIGGRAPH 2005*, 2005.


40 FPS without SLI
85 FPS with SLI turned on

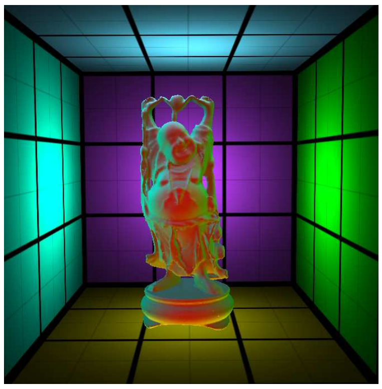

7 FPS without SLI
19 FPS with SLI turned on

Figure 6: *Diffuse Stanford Buddha (about 35000 vertices and 67000 faces) in a box. Left image with point to point form factor, right image with our polygon-to-point form factor method. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 with SLI support, AMD Athlon64 Dual 4600+ processor.*

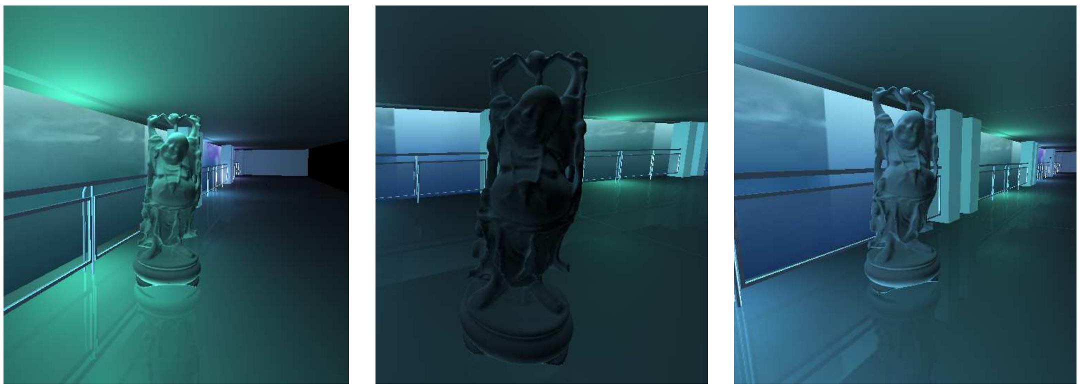


Figure 7: *Screen shots from Buddha in the hallway (the whole scene with the Buddha is about 38000 vertices and 70000 faces). OGRE implementation. 12 FPS, 800x600 resolution on NVIDIA GeForce 7800GT, AMD Athlon 64 3500+ processor.*

[9] R. Mantiuk, S. Pattanaik, and K. Myszkowski. Cube-map data structure for interactive global illumination computation in dynamic diffuse environments. In *International Conference on Computer Vision and Graphics*, pages 530–538, 2002.

[10] Mangesh Nijasure, Sumanta Pattanaik, and Vineet Goel. Real-time global illumination on the GPU. *Journal of Graphics Tools*, 2004. To appear.

[11] R. Ramamoorthi and P. Hanrahan. An efficient representation for irrandiance environment maps. *SIGGRAPH 2001*, pages 497–500, 2001.

[12] P. Shirley, C. Wang, and K. Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.

[13] I. Wald, C. Benthin, and P. Slussalek. Interactive global illumination in complex and highly occluded environments. In *14th Eurographics Symposium on Rendering*, pages 74–81, 2003.

[14] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: A scalable approach to illumination. In *SIGGRAPH 2005*, 2005.

[15] G. Ward. Adaptive shadow testing for ray tracing. In *Rendering Workshop '94*, pages 11–20, 1994.

[16] G. J. Ward. The RADIANCE lighting simulation and rendering system. *Computer Graphics*, 28(4):459–472, 1994.

[17] K. Zhou, Y. Hu, S. Lin, B. Guo, and H.-Y. Shum. Precomputed shadow fields for dynamic scenes. In *SIGGRAPH 2005*, 2005.

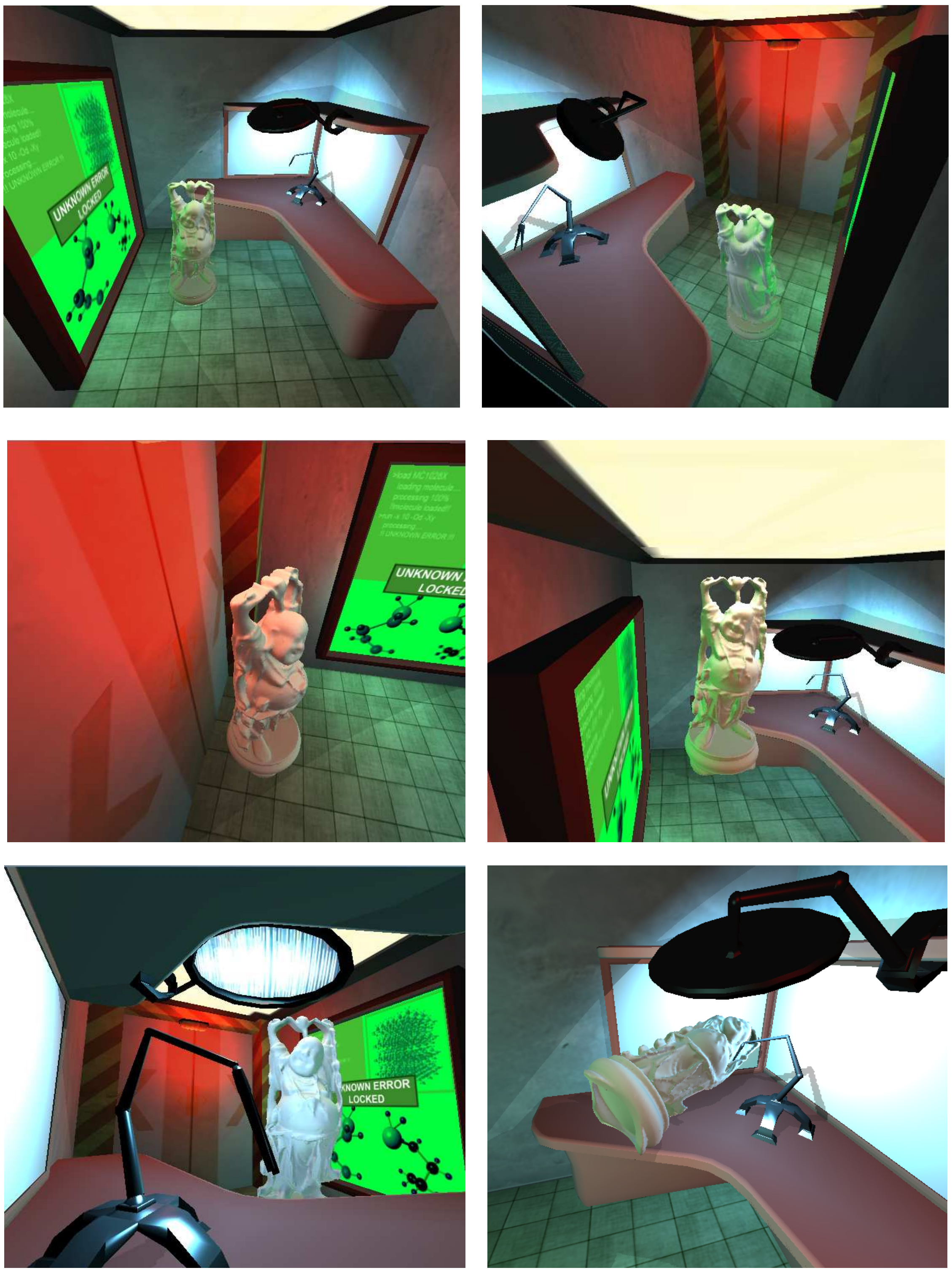


Figure 8: *Screen shots from Buddha in a laboratory (the whole scene with the Buddha is about 38000 vertices and 70000 faces). OGRE implementation. 15 FPS, 800x600 resolution on NVIDIA GeForce 7800GT, AMD Athlon 64 3500+ processor.*