Real-Time Global Illumination on GPU

Mangesh Nijasure, Sumanta Pattanaik
U. Central Florida, Orlando, FL

Vineet Goel
ATI Research, Orlando, FL

Abstract

We have developed a system for computing a plausible global illumination solution for dynamic environments in real time on programmable graphics processors (GPUs). The entire algorithm runs on DirectX9 compatible graphics hardware using vertex and fragment shaders, and computes a global illumination solution for reasonably complex scenes with moving objects and moving lights at frame rates of approximately 10-40 fps. We present the salient steps of the algorithm that made this real-time global illumination evaluation possible.

Keywords: Global Illumination, Real Time Rendering, Programmable Graphics Hardware.

1 Introduction

Accurate lighting computation is one of the key elements to realism in rendered images. In the past two decades, many physically based global illumination algorithms have been developed for accurately computing the light distribution [Cohen et al. 1993; Shirley 2000; Sillion et al. 1994; Jensen 2001]. These algorithms simulate the propagation of light in a three-dimensional environment and compute the distribution of light to desired accuracy. Unfortunately, the computation times of these algorithms are high. It is possible to pre-compute the light distribution to render static scenes in real-time. However, such an approach produces inaccurate renderings of dynamic scenes where objects and/or lights may be changing.

The computation power of the programmable graphics processors (GPU) in commodity graphics hardware is increasing at a much faster rate than that of the CPU. It is often not possible and not even meaningful to use CPU algorithms for GPU processing. The key to harnessing the GPU power is to re-engineer the lighting computation algorithms to make better use of their SIMD processing capabilities. [Purcell et al. 2002]. Researchers have proposed such algorithms to compute global illumination at faster rates[Keller 1998, Ma and McCool 2002]. We have developed such a re-engineered solution for computing global illumination. The algorithm does not depend on any additional resources other than the programmable graphics hardware and the performance is tied only to the speed of the GPU.

We simulate the transport of light in a synthetic environment by following the light emitted from the light source(s) through multiple bounces on the surfaces of the scene. Light bouncing off the surfaces is captured as incident radiance maps at sample points distributed uniformly over the volume of the scene. Incident radiance maps are rendered and represented in a compact form using graphics hardware. The incident light from these maps is interpolated to the nearby surface positions for lighting computation for each subsequent bounce. Iterative computation of a small number of bounces gives us a plausible approximation of global illumination for scenes involving both moving objects and changing light sources in fractions of seconds. The algorithm was presented in brief in [Nijasure 2004].

In this paper we present the following ideas that made the implementation possible in real-time (10-40 frames per second) and we believe that individually they can be used in other GPU based lighting computation algorithms.

- Textured imposters for efficient computation of incident radiance maps,
- Spherical Harmonics (SH) representation of the incident radiance function from the radiance maps,
- Global illumination from SH representation, and
- Shadow cube-maps for indirect shadow.

Executables in both OpenGL and DirectX 9.0 of this implementation as well as a demo video are available for download from the website listed at the end of this paper.

2 Background

The steady-state distribution of light in any scene is described by the equation $L = L_e + \mathbf{T}L$, where L, the light due to global illumination, is the quantity of light of our interest, L_e is the light due to emission from the light source(s), and $\mathbf{T}$ is the light transport operator. The recursive expansion of this equation gives rise to a Neumann series of the form $L = L_e + \mathbf{T}L_e + \mathbf{T}^2L_e + \mathbf{T}^3L_e + \ldots$. In

this equation $\mathbf{T}L_e$ represents the first bounce of light after leaving the light source. This term is commonly known as L_{direct}. Substituting L_{direct} for $\mathbf{T}L_e$ we get the expression $L = L_e + L_{direct} + \left(\mathbf{T}L_{direct} + \mathbf{T}^2L_{direct} + ...\right)$. The terms within the parenthesis in the right hand side of the expression represent the second and higher order bounces of light and they together make $L_{indirect}$. Computation of L_{direct} is straight forward and is easily implemented on GPUs. Computation of $L_{indirect}$ is the most expensive step in any global illumination computation algorithm. We use an iterative method to compute a fast and plausible approximation to $L_{indirect}$ and hence the global illumination in the scene. The basic idea behind our approach is:

> We start with the evaluation of $\mathbf{T}L_{direct}$ from L_{direct} computed for the whole scene. In the next iteration we compute $\mathbf{T}^2L_{direct}$ from $\mathbf{T}L_{direct}$. We continue this process until we have evaluated a certain number of pre-specified indirect terms.

Each step of the iteration requires the evaluation of an indirect term for every point in the scene. This requirement of computing $L_{indirect}$ for every point in the scene may be relaxed by computing it for every vertex in a well tessellated scene. Even with this change, it is impractical to compute $L_{indirect}$ for any reasonably complex scene in real-time. If rendering static scenes is the goal, then the actual computation of $L_{indirect}$ can be done offline and the pre-computed information can be used to render scenes in real time [Greger 1998]. But such a pre-computation strategy is not useful for dynamic scenes with moving objects and moving light sources. Thus it is very clear that if we want real-time performance in dynamic scenes we have to capture information only for a sparse number of sample points in the scene and interpolate this captured information to the surface point of interest. Such a sparse sampling strategy has been used by many researchers [Ward 1994, Greger 1998, Walter 1999, Tole 2002].

The outline of our algorithm is as follows:

1. Sample the scene volume at a discrete number of volumetric sample points.

   Repeat the following steps for the desired number of iterations.

2. Capture the incident light $L_{incident}$ at each sample point. This is the combination of L_{direct} and $L_{indirect}$ (from the previous iteration), arriving from all the surface points visible to that sample point.
   L_{direct} is computed using a local lighting model. $L_{indirect}$ is initialized to zero during the first iteration.

3. Use spherical harmonic transformation of the captured $L_{incident}$ to represent it in a compact fashion for easy interpolation.

4. Interpolate $L_{incident}$ to neighboring surface points of interest. The interpolated value will be a reasonable approximation of $L_{incident}$ at the surface point. Use this interpolated value for calculating $L_{indirect}$ in the subsequent iteration.

The accuracy of the $L_{indirect}$ computation depends on the resolution of the spatial data structure used for capturing the light as well as the number of the iterations executed. These factors can be choosen to suit the computing power and available time budget. The accompanying video shows the real-time computation of global illumination in the example scene with moving objects and moving lights using a spatial data structure of $4 \times 4 \times 4$ resolution and two iterations of the algorithm.

3 GPU Based Computation

In this section we describe the mapping of the salient steps of our algorithm to the GPU.

3.1 Use of Impostors for Efficient Computation of Incident Radiance Maps

Computation of the incident radiance function at the sample points is one of the most important and expensive steps in this algorithm. We use cubemap data structures for this capture. We divide the volume of the 3D scene space into a uniform volumetric grid of size $n_x \times n_y \times n_z$ and render a cube map from the center of each voxel to capture the incident radiance function due to the light reflected off the surfaces in the scene. A cubemaps is generated by rendering the entire scene in hardware 6 times.

The most expensive part of any hardware rendering is the rasterization of the scene which produces scene points visible through the cubemap pixels. Each iteration of our algorithm requires the computation of cubemaps. We minimize expensive rasterization during each iteration by making use of an observation that for each frame these scene points remain invariant in between the iterations. This is true even in dynamic scenes because the objects for any single frame can be safely considered to be static. Thus it is not necessary to process the scene geometry through the rendering pipeline in between the iterations of the algorithm. Hence we store the surface position, the surface normal and the surface material property from the first iteration, and use them in successive iterations. This strategy is similar to using impostors [Maciel 1995]. In addition to computing and storing the surface related information we also compute the light coming out of the surface points (due to direct light and indirect light computed so far). We use the Multiple Render Targets (*MRT*) feature provided by the GPU to store this information. The MRT feature allows the pixel shader

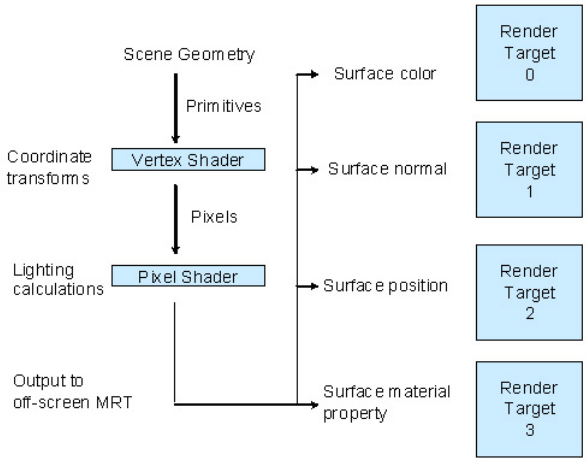


Figure 1. Use of multiple render targets during cubemap rendering.

to export up to 4 different colors (colors are replaced by the desired information) to different areas of memory.

OpenGL supports MRT through vendor specific extensions (GL_ATI_DRAW_BUFFERS for ATI cards) and DirectX offers this feature by simply specifying four output color targets (oC0 - oC3) in the pixel shader. The important issue here is that the surface normal and position require floating point precision and hence are rendered to off-screen render targets with a 32 bit IEEE floating point format. OpenGL supports floating point render targets as *pbuffers* or pixel-buffers (supported through the extension WGL_ARB_PBUFFER) and DirectX allows the creation of floating point texture surfaces that can be bound as a rendering surface or render target.

Figure 1 illustrates the multiple outputs from the pipeline during the first iteration. During successive iterations of the algorithm, render targets 1, 2 and 3 are bound as textures and the surface information is used to compute and update the color of the pixels of each cubemap to reflect the various bounces of light.

3.2 Spherical Harmonic Representation of $L_{incident}$

The cubemaps capture illumination from the surfaces visible from a grid point. This representation is converted to a compact format (a small number of coefficients) using spherical harmonic transformation. This transformation is carried out by evaluating the following equation for each coefficient, for every cubemap captured.

$$L_l^m \approx \sum_{face=1}^{6} \sum_{i=1}^{width} \sum_{j=1}^{height} L_{face}(i,j) Y_l^m(\omega_{ij}) A(i,j)$$

where Y_l^m is a spherical harmonics basis function, $A(i,j) = (x_i^2 + y_i^2 + z_i^2)^{-\frac{3}{2}}$ is the solid angle of the pixel(i,j) and ω_{ij} is the direction through the center of the pixel.

All the cubemaps are of the same size, and the products $Y_l^m(\omega_{ij}) A(i,j)$ are the same for each cubemap. Thus, we compute the products only once during the setup phase and store them in a texture. These product values are of the order 10^{-3} and are signed. So we store them in a floating point texture in OpenGL with vendor specific extensions (ATI cards expose this feature with GL_ATI_TEXTURE_FLOAT), and in DirectX by specifying a 32-bit floating point format option, during texture creation. There are M coefficients that need to be calculated, each coefficient requires a different Y_l^m and there are 6 faces in a cubemap. Thus we precompute $6M$ such textures.

In the spherical harmonics coefficient calculation, the first step is the multiplication of each pixel in the cubemap with its associated precomputed factor. This is done by rendering a screen aligned quad with the window size set to the size of the cubemap face. The face of the cubemap being processed is bound as one texture and the corresponding precomputed texture is bound as the other. The pixel shader fetches pixel values from each of the textures and writes out their product. The output is directed to a temporary off-screen render target which also has a 32-bit floating point format. Figure 2 shows this process, which is repeated for all the faces of each cubemap.

The final coefficients are calculated by summing up all the resulting pixels over all the faces. An important property that can be leveraged again is that all the six faces have the same dimensions. The final result involves only summation of pixels and thus the six faces are superimposed into a single texture. This process is illustrated in Figure 3. For each cubemap, six temporary textures are bound and added together in the pixel shader. The resulting texture is of the same size as that of a cubemap face.

The next step is to sum up each of the values in the texture generated in the previous step. In a CPU based algorithm, this would be trivial to do by using a couple of nested for loops and a running sum. Unfortunately the graphics hardware does not support such facilities due to the SIMD architecture and inability to perform arbitrary data export from the shaders. Hence, this summation is carried out indirectly as follows.

The texture that needs to be summed is bound as the source texture. The output render target is set to half the

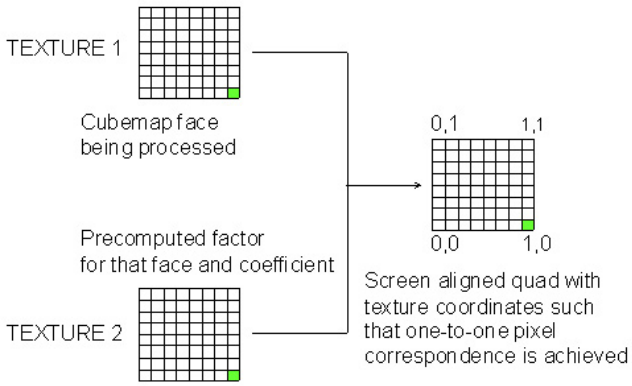


Figure 2. Multiplication of pixels by their respective factors.

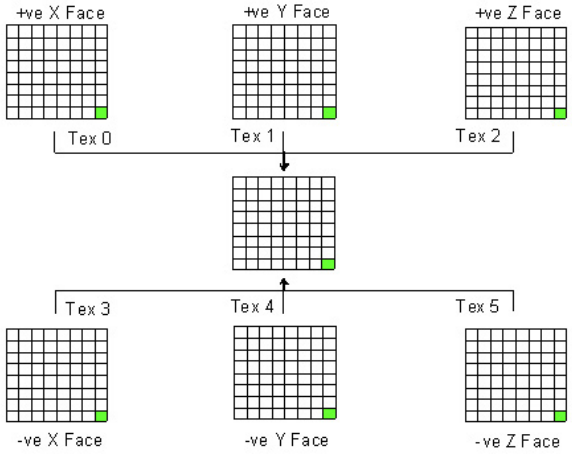


Figure 3. Accumulating the contributions from all the faces of the cubemap.

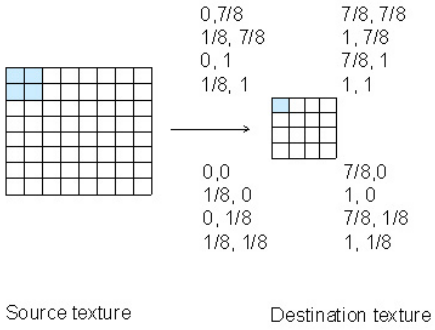


Figure 4. Reducing the source texture to quarter of its original size.

height and width of the original texture. The new screen aligned quad is set to have four texture coordinates associated with each vertex. These are calculated such that each pixel in the destination quad fetches four values from the source texture. These values are then summed up by the pixel shader.

Current generation GPUs allow the binding of 16 different source texture units and 8 unique texture coordinates. Given these limitations, the texture reduction process described below illustrates the sampling using 4 fetches at once. This technique is much like mip-map generation and is a widely accepted approach [GpuGems 04].

Figure 4 illustrates this procedure to reduce an 8×8 texture to a 4×4 one. The texture coordinates necessary for this particular case are as shown. For a cubemap size of 32×32, there are 5 such steps needed before the texture is reduced to a single coefficient value. This process needs to be repeated for each color channel (R, G and B channels), for each coefficient (M times), and for every cube map. The end results of all these multiple passes are $3M$ coefficient values for each cube map.

3.3 Computing $L_{indirect}$

The computation of $L_{indirect}$ at any point requires the application of the transport operator to the incident radiance function.

$$L_{indirect}(\omega_o) = \mathbf{T} L_{incident} = \sum_{l} \sum_{m=-l}^{l} L_l^m \, T_l^m(\omega_o) \quad (1)$$

where L_l^m and T_l^m are the spherical harmonic coefficients of the incident radiance function and the cosine modulated surface reflection function respectively. T_l^m values are precomputed and stored as textures. (We provide the derivation

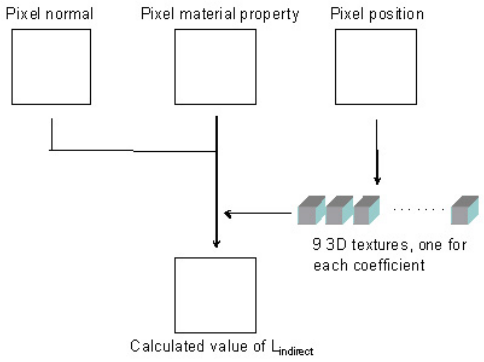


Figure 5. Calculating $L_{indirect}$.

of the above equation in the Appendix.[Ramamoorthy and Hanranhan 2001, 2002], and [Sloan et al 2002] provide a more detailed description.
The spherical harmonics coefficients calculated for the cube maps are assembled into 3D textures. The 3D texture data structure is chosen to make use of the fast dedicated trilinear interpolation hardware in the GPU. Current generation GPUs do not allow interpolation of floating point values and thus these coefficients are scaled and biased to fixed point values in order to maximize the performance obtained from hardware interpolation. When these values are fetched for use in the pixel shader, the reverse scaling and biasing is applied to get back the original floating point coefficients.

To compute $L_{indirect}$ for each surface point visible through the cubemap pixels, the appropriate normal, material property and position values are bound as input textures and the destination texture holds the calculated $L_{indirect}$ values. The calculation is done as follows. For each pixel, the position texture holds the 3D scene position. This scene position is converted to a volume co-ordinate for accessing the spherical harmonic coefficients stored in the 3D texture. This co-ordinate represents the scene position in normalized scene volume space. The spherical harmonic coefficients are fetched from their respective 3D textures using these normalized volume coordinates. Both OpenGL and DirectX require that the interpolation filters be set to linear interpolation so that the result provides trilinearly interpolated values from the 8 nearest neighbors of that point.

The surface normals, L_l^ms and T_l^ms are used to evaluate Equation (1) and thus compute the $L_{indirect}$ that would come out of that surface point. Figure 5 summarizes this step showing the needed input textures and the resulting value of $L_{indirect}$ stored in the destination texture. Figure 6 shows renderings from $L_{indirect}$ computed from two iterations.

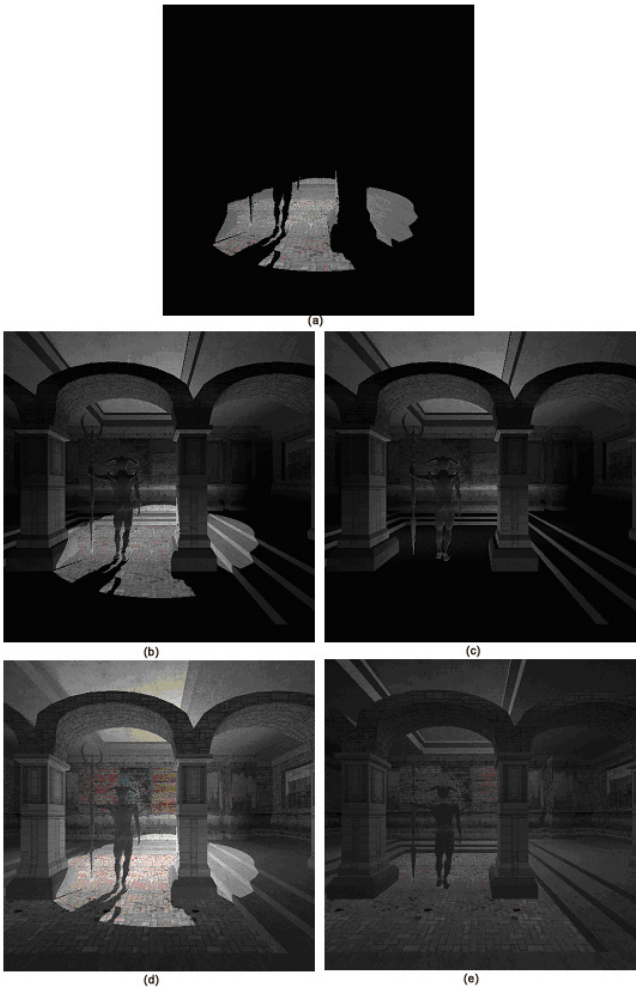


Figure 6. Iteration Results. The images were rendered using (a) L_{direct}only. (b) First iteration. i.e. $L_{direct} + TL_{direct}$ (c) Indirect light from the first iteration. i.e. TL_{direct} only (d) Second iteration. i.e. $L_{direct} + TL_{direct} + T^2L_{direct}$ (e) Indirect light from the second iteration. i.e. T^2L_{direct} only.

4 Implementation and Computation Times

All the steps of our algorithm are implemented using vertex and fragment shaders. We have implemented the algorithm in both OpenGL and DirectX 9. The executables along with the test scene are available for download from the website listed at the end of this paper. Note that the standard OpenGL library (Version 1.4) does not support many of the DirectX 9 features. We have made use of ARB extensions and vendor specific extensions to make use of such features.

	Hall	Gallery
Triangles	8,000	80,000
Rasterization for cubemaps	67.3	86.2
Rendering cube maps using raster data	0.3	0.3
Computation of Coefficients and $L_{indirect}$ (2 Iterations)	11.1	11.3
Final frame Rendering	0.1	0.1
Total time for scene only with dynamic light source	20.8 (47.8 fps)	23.2 (42.49 fps)
Total time for scene with dynamic objects and dynamic light source	107 (9.3 fps)	110 (9.0 fps)

Table 1. Time spent in various steps of the algorithm in milliseconds.

We show the images from two scenes: “Hall” and “Gallery”. Figures 6-8 show the views from the “Hall” scene. Both the scenes are illuminated by one spot light source. Thus the illumination in the scene is due to the light incident on the floor and the inter-reflection of the light within the scene. The “Hall” scene has 8000 triangles and the “Gallery” scene has 80,000 triangles. We have used a uniform grid of dimension $4 \times 4 \times 4$, rendered cube maps at a resolution of 16×16 for each face, and computed 9 spherical harmonic coefficients. Table 1 lists the timing for various steps of the GPU implementation.

Thus we believe that a good approximation to global illumination can be achieved in about 100 ms. As we mentioned earlier a significant fraction of this time (86ms) is spent in rasterizing the scene for cube map computation. Carrying out rest of the iteration takes only about 20 ms. Thus increasing the number of iterations for capturing multiple bounces of light will not reduce the frame rate significantly.

Figure 7a shows the specular rendering effect in the



Figure 7. More Results. (a) The rendering of the "Hall" scene with a specular ball, specular character and a pinkish floor. (b) Rendering of the "Gallery" scene.

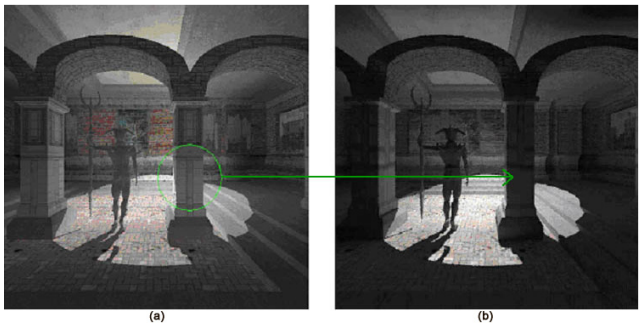


Figure 8. Light leakage problem during $L_{indirect}$ Computation. The image in the left (a) is computed using $L_{indirect}$ obtained through indiscriminate tri-linear interpolation. The image in the right (b) is computed using Radiance software. Note the difference in shade at the pillar (highlighted). The light from behind the pillar leaks on to the surface.

“Hall” scene with a specular ball. For this rendering we changed the surface property of the humanoid character to specular and the reflectance of the floor to uniform pink. Note that even with 9 spherical harmonic coefficients we are able to illustrate the specular behavior of the surfaces. Accuracy will improve with the number of spherical harmonic coefficients at an additional cost (see 3rd row of Table 1). Figure 7b shows the rendering of the “Gallery” scene. Note the color bleeding on the walls and the ceiling due to the reflection from the yellowish floor.

5 Shadowing in $L_{indirect}$ computation

For $L_{indirect}$ computation, our algorithm computes $L_{incident}$ only at a small number of volumetric sample

points and interpolates them to neighboring surface points. This interpolation is based on the assumption that the incident radiance function is smoothly varying in the voxel under consideration. However, this assumption breaks down in the presence of occluders. In such a situation indiscriminate tri-linear interpolation introduces error in the computation in the form of leakage of shadow or light. This problem is particularly severe when the volumetric sampling is sparse and hence must be addressed. The problem is highlighted in Figure 8. Incident light captured at sample point(s) behind the pillar is incorrectly propagated to the front of the pillar and thus increases the brightness of the surface of the pillar and arch shown in the image on the left. For comparison we show an accurate rendering computed offline using Radiance software [Ward-Larson 1998]. We handle this problem by a technique similar to shadow mapping [Williams 78, Reeves et al. 87] with very little extra effort. At the cube map rendering stage, we save the depth buffer value associated with each cube map face in a shadow cube map. For any surface point of interest we query the shadow cube maps associated with each of the eight neighboring grid points to find out if the surface point is visible or invisible from the grid points. Based on this finding we associate a weight of 1 or 0 to the grid point. Instead of a simple tri-linear interpolation we use a weighted tri-linear interpolation to account for the shadowing of the sample points . Current generation GPUs do not support weighted tri-linear interpolation. Figure 9 shows the results obtained using a CPU implementation of the weighted interpolation on our sample scene. Note that light leakage on to the arches and pillars in Figure 8a is suppressed in Figure 9. Note also that the lighting distribution on the arches and the pillars are now comparable to the distribution computed using Radiance software as shown in Figure 8b. Thus, we find a much better match between the images.
We are working on a method to map this extension to our hardware implementation. Given the limitations of GPUs, the solution would be to carry out this weighted interpolation manually in the shader. For each pixel that needs to be shaded, this would involve fetching 8 samples from each 3D texture (there are a total of 9 such textures - one for each SH coefficient). This would be followed by 7 linear interpolations in the shader to mimic the weighted trilinear interpolation. This process needs to be evaluated for each iteration as well. The overhead of executing 72 instructions per pixel, per coefficient is significant and does not allow the leveraging of the dedicated texture interpolation unit in the GPU. The comparison of the image in Figure 9 with the Radiance software generated image (8b) shows that the result obtained in two iterations of our algorithm with this modification is very close to the accurate solution.

We use the shadow mapping technique for handling shadows during L_{direct} computation. Shadow maps are captured for every frame to handle the dynamic nature of the scene.

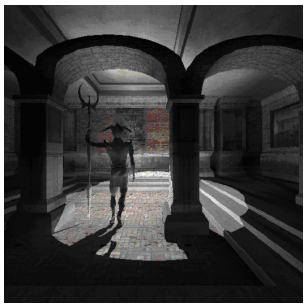


Figure 9. Result using indirect shadow handling approach in the CPU. Note that the results are closer to the results computed using Radiance software.

6 Discussion

In this presentation we have restricted our experiment to radially symmetric BRDFs. However, it should be noted that our algorithm is equally applicable to scenes with arbitrary non-diffuse surfaces. None of the steps of the algorithm will change to support such surfaces, very little of the framework will need changes. Pre-computed BRDF coefficients will be stored in a 2D table as textures and will be used during the $L_{indirect}$ computation.

The $4 \times 4 \times 4$ grid dimension used in our implementation is not sufficient for larger environments. Increasing the spatial grid resolution to handle environments with larger spatial extent will easily outgrow the computational resources of a single graphics processor. While this problem may be handled by using a hardware system supporting multiple graphics cards, a better strategy would be to use a combination of non-uniform grid spacing particularly along the depth axis, and to restrict the volume of the grid structure to a reasonable extent. Another strategy would be to adaptively distribute the grid points in the scene. This is closer to the irradiance cache approach, and is likely to be computationally less expensive for larger environments.

7 Web Information

Executables in both OpenGL and DirectX 9 of this implementation, the test scene as well as a demo video are available for download at http://www.acm.org/jgt/papers/NijasurePattanaikGoel05. The executables have been successfully tested on ATI's Radeon 9700 and 9800 series. The OpenGL version

uses vendor specific extensions and hence will only run on ATI cards. However the DirectX version will run on any programmable GPU with full DX9 support including floating point off-screen render targets.

References

[1] ARFKEN, G. 1970. *Mathematical Methods for Physicists*, Academic Press, New York.

[2] COHEN, M. F. AND GREENBERG, D. P. 1985. The hemi-cube: a radiosity solution for complex environments. *Proceedings of 12th International ACM conference on Computer Graphics and Interactive Techniques*, 31-40.

[3] COHEN, M. F. AND WALLACE, J. R. 1993. *Radiosity and Realistic Image Synthesis*. A. P. Professional.

[4] GPUGEMS 2004. Chapter 37. Addison-Wesley.

[5] GREGER, G., SHIRLEY, P., HUBBARD, P. AND GREENBERG, D. P. 1998. The Irradiance Volume. *IEEE Computer Graphics & Applications*, 18(2), 32-43.

[6] JENSEN, H. W. 2001. *Realistic Image Synthesis using Photon Mapping*. AK Peters.

[7] KELLER, A. 1997. Instant Radiosity. *Proceedings of ACM SIGGRAPH 1997*, 49-56.

[8] MA, V. C. H. AND McCOOL, 2002. Low Latency Photon Mapping Using Block Hashing. In *Proceedings of Eurographics Workshop on Graphics Hardware*. 89-98, 2002.

[9] MACIEL, P. W. C. and SHIRLEY, P. 1995. Visual Navigation of Large Environments Using Textured Clusters. *Proc. 1995 Symposium on Interactive 3D Graphics*. pp. 95–102, 1995.

[10] NIJASURE, M., PATTANAIK, S. N. and GOEL, V. 2003. Interactive Global Illumination in Dynamic Environments Using Commodity Graphics Hardware. Short paper in *Pacific Conference on Computer Graphics and Applications 2003*, 450-454, 2003.

[11] PURCELL, T. J., BUCK, I., MARK, W. R. AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. In *ACM Transactions on Graphics* (Proceedings of ACM SIGGRAPH 2002). 21(3), 703-712, 2002.

[12] RAMAMOORTHI, R. AND HANRAHAN, P. 2001. An Efficient Representation for Irradiance Environment Maps. In *Proceedings of SIGGRAPH 2001*, 497-500.

[13] RAMAMOORTHI, R. AND HANRAHAN, P. 2002. Frequency space environment map rendering. In *ACM Transactions on Graphics* (Proceedings of SIGGRAPH 2002), 21(3), 517-526.

[14] REEVES, W. SALESIN, D. AND COOK, R. 1987. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH 87*, 283-291.

[15] SAITO T. and TAKAHASHI T. 1990. Comprehensible rendering of 3-D shapes. In *Intl. Conference on Computer Graphcis and Interactive Techniques*, pp.197-206.

[16] SHIRLEY, P. 2000. *Realistic Ray Tracing*. A. K. Peters.

[17] SILLION, F. X. AND PUECH, C. 1994. *Radiosity and Global Illumination*. Morgan Kaufman Publishers, San Francisco.

[18] SLOAN, P., KAUTZ, J. AND SNYDER, J. 2002. Precomputed Radiance Transfer for Real-time rendering in Dynamic, Low-Frequency Lighting Environments. In *ACM Transactions on Graphics* (Proceedings of SIGGRAPH 2002), 21(3), 526-536.

[19] TOLE, P., PELLACINI, F., WALTER, B. J. AND GREENBERG, D. P. 2002. Interactive global illumination in dynamic scenes. In *ACM Transactions on Graphics* (Proceeding of ACM SIGGRAPH 2002), 21 (3), 537-546.

[20] WALTER, B. J., DRETTAKIS, G. AND PARKER, S. 1999. Render Cache for Interactive Rendering. In *Proceedings of 10th Eurographics Rendering Workshop.*

[21] WARD, G. J. 1994. The RADIANCE Lighting Simulation and Rendering System. In *Proceedings of SIGGRAPH 94 conference*, 459-472.

[22] WARD-LARSON, G. J. AND SHAKESPEARE, R.A. 1998. Rendering with Radiance: the Art and Science of Lighting Visualization. Morgan Kaufman Publishers.

[23] WILLIAMS, L. 1978. Casting Curved Shadows on Curved Surfaces. In *Proceedings of SIGGRAPH 78 conference*, 270-274.

8 Appendix: Derivation of Equation 1

$$
\begin{aligned}
L_{indirect}(\omega_o) &= \mathbf{T}L \\
&= \int_{\omega_i \in \Omega_N} L(\omega)\, \rho(\omega_i, \omega_o)\ \cos\theta_i\, d\omega_i \\
&= \int_{\omega_i \in \Omega_N} L(\omega)\, \widehat{\rho}(\omega_i, \omega_o)\ d\omega_i \\
&= \sum_l \sum_{m=-l}^{l} L_l^m \int_{\omega_i \in \Omega_N} Y_l^m(\omega)\ \widehat{\rho}(\omega_i, \omega_o)\ d\omega_i \\
&= \sum_l \sum_{m=-l}^{l} L_l^m\, T_l^m(\omega_o)
\end{aligned}
$$

In the above expression the domain of integration Ω_N is the hemisphere around the surface normal N, ω_i is the incoming direction in Ω_N, $\rho(\omega_i, \omega_o)$ is the BRDF (Bi-Directional Reflectance Distribution Function) at the surface point s, and $\widehat{\rho}$ is $\rho \cos\theta_i$. For radially symmetric BRDFs (for example Lambertian surfaces and Phong specular surfaces) $\widehat{\rho}$ is only a function of incoming direction [Ramamoorthi 2002]. Thus $T_l^m(\omega_o)$ is independent of the outgoing direction.