Real-Time Global Illumination on GPU

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

Vineet Goel
ATI Research, Orlando, FL

Abstract

We present a system for computing plausible global illumination solution for dynamic environments in real time on programmable graphics processors (GPUs). We designed a progressive global illumination algorithm to simulate multiple bounces of light on the surfaces of synthetic scenes. The entire algorithm runs on ATI's Radeon 9800 using vertex and fragment shaders, and computes global illumination solution for reasonably complex scenes with moving objects and moving lights in realtime.

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 capability of GPU[Purcell et al. 2002]. Researchers have proposed such algorithms to compute global illumination at faster rates[Keller 1998, Ma and McCool 2002]. We present such a re-engineered solution for computing global illumination.

Our algorithm takes advantage of the capability of the GPU for computing accurate global illumination in 3D scenes and is fast enough so that interactive and immersive applications with desired complexity and realism are possible. We simulate the transport of light in a synthetic environment by following the light emitted from the light source(s) through its multiple bounces on the surfaces of the scene. Light bouncing off the surfaces is captured by cube-maps distributed uniformly over the volume of the scene. Cube-maps are rendered using graphics hardware. Light is distributed from these cube-maps to the nearby surface positions by using a simple trilinear interpolation scheme for the computation of each subsequent bounce. Iterative computing 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.

Traditionally, cube maps are used for hardware simulation of the reflection of the environment on shiny surfaces [Blinn et al. 1976]. In recent years, researchers are using them for reflection of the environment on diffuse and glossy surfaces [Ramamoorthi and Hanrahan 2001, 2002; Sloan et al. 2002]. We extend the use of cube maps to global illumination computation.

Greger et al. [1998] used uniform grid for storage and interpolation of global illumination. The data structure and interpolation strategy used in our main algorithm is similar. The main difference is that our algorithm uses the uniform grid data structure for transport and storage of inter-reflected illumination and computes global illumination, whereas Greger et al. use the data structure simply for storage of illumination pre-computed using an off the shelve global illumination algorithm. In addition, we use spherical harmonic coefficients for compactly representing the captured radiance field at the sampled points. Implementation of our algorithm on programmable GPU provides us with solutions for reasonably complex scenes in fractions of seconds. This fast computation allows us to recompute global illumination for every frame, and thus support dynamic lights and the moving object's effect on the environment.

Ward's irradiance cache [Ward 1994] uses sampled

points in the scene for computing and storing scalar irradiance value for reuse by neighboring surface points through complex interpolation. The sample grid points used in our algorithm may as well be considered as radiance cache for reuse by neighboring surface points. The main difference is: we store the whole incoming radiance field at the grid points, not a scalar irradiance value. We use spherical harmonic representation for storing this radiance field. The spherical harmonic coefficients from the grid points are interpolated using a simple interpolation scheme.

The paper contains two main contributions. (1) We propose a progressive global illumination computation algorithm suitable for implementation on graphics hardware. (2) We take advantage of the programmability of the graphics pipeline to map the entire algorithm to programmable hardware. Mapping the algorithm to ATI's Radeon-9800 gives us more than 40 frames per second of interactive computation of global illumination in reasonably complex dynamic scenes.

2 Background

The steady-state distribution of light in any scene is described by the equation $L = L_e + \mathbf{T}L$, where L is the global illumination, the quantity of light of our interest, L^e is the light due to emission from light source(s), and $\mathbf{T}$ is the light transport operator. The recursive expansion of this equation gives rise to a Neuman series of the form $L = L_e + \mathbf{T}L_e + \mathbf{T}^2L_e + \mathbf{T}^3L_e + ...$ 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 $L = L_e + L_{direct} + \big(TL_{direct} + T^2L_{direct} + ...\big)$. The terms within the paranthesis in the right hand side of the expression represent the second and higher order bounces of light and they together make the $L_{indirect}$. Computation of L_{direct} is straight forward and is easily implemented in GPUs. Computation of $L_{indirect}$ is the most expensive step in any global illumination computation algorithm. In this paper we propose a GPU based method for computing fast and plausible approximation to the $L_{indirect}$ and hence to the global illumination in the scene. We achieve this by providing: (i) a spatial data structure to capture the lighting from the first bounce (and subsequent bounces) of light, (ii) spherical harmonic representation for capturing directional characteristics of bounced light, and (iii) spatial interpolation of the captured light for use as position dependent $L_{indirect}$ for hardware rendering. These three steps together account for one bounce of light and hence compute one term of $L_{indirect}$. Iteration of these steps add subsequent terms. Figure 2 shows the results for a diffuse scene illuminated by a lone spot light. Figure 2(a) shows the rendering of the scene using the results from the first bounce i.e. L_{direct}. Figures 2(d) and 2(e) show the rendering using additional one and two bounces respectively. Figure 2(b) and 2(c) show the incremental contribution of these individual bounces.

The accuracy of the $L_{indirect}$ computation depends on the the fineness of the spatial data structure used for capturing the light and the number of the iterations. It is possible to choose the resolution of the data structure, and the number of iterations to suit to the GPU power and available time budget. The accompanying video shows the real-time computation in the example scene with moving objects and moving light using a spatial data structure of $4 \times 4 \times 4$ resolution and two iterations of the algorithm.

In Section 3, we describe our algorithm. In Section 4, we describe the implementation of this algorithm on GPU. In Section 5, we provide the results of our implementation. In Section 6, we provide an extension to our basic algorithm to handle indirect shadow.

3 Algorithm

The characteristic steps of our progressive algorithm are as follows. Figure 1 illustrates each of these steps.

1. Divide the volume of the 3D scene space into a uniform volumetric grid. Initialize $L_{indirect}$ for the scene surfaces to zero. (see Figure 1a)

2. Render a cube map at each grid point to capture the incoming radiance field due to the light reflected off the surfaces of the scene. Carry out this rendering using L_{direct} and from the current value of $L_{indirect}$. (see Figure 1c)

3. Use spherical harmonic transformation to convert directional representation of the incoming radiance field captured in the cube maps into spherical harmonic coefficients.

4. For every surface grid point, interpolate the spherical harmonic coefficients from its neighboring grid points. The interpolated coefficients approximate the incident radiance at the surface point. Compute reflected light, L_{out}, due to this incident radiance. Replace the previous $L_{indirect}$ with L_{out}.

5. Repeat Steps (2) to (4) until the changes in $L_{indirect}$ are insignificant.

6. Render a view frame from L_{direct}, and from $L_{indirect}$ computed from step (5). (see Figure 1d)

Interactive rendering of a dynamic scene requires the computation of all the steps of the algorithm for each rendered frame.

3.1 Rendering Cube Maps

Cube maps (see Figure 1b) are the projections of an environment on the six faces of the cube with a camera positioned at the center of the cube and pointed along the centers of the faces. They capture a discrete representation of incoming radiance from all directions around their center point. Current GPUs use multiple and highly optimized pipelines for this type of rendering.

3.2 Spherical Harmonic Representation of Incoming Radiance Field

Spherical harmonic functions $Y_l^m(\omega)$, define an orthonormal basis over spherical directions [Arkken 1970] (see Appendix for the definition of these functions). Using these basis functions incoming radiance $L(\omega)$ can be represented as a number of coefficients L_l^m, such that Equation 1 is satisfied.

$$\begin{aligned} L(\omega) &= \sum_{l} \sum_{m=-l}^{l} L_l^m Y_l^m(\omega) \qquad (1) \\ L_l^m &= \int_0^{\pi} \int_0^{2\pi} L(\omega)\, Y_l^m(\omega)\; d\omega \end{aligned}$$

In Equation 1, l, the index of outer summation, takes values from 0 and above. The exact upper limit of l, and hence the number of coefficients required for accurate representation of $L(\omega)$ depends on the frequency content of the function. For low frequency functions, the number of such coefficients required is small. Thus, spherical harmonics coefficients provide a compact representation for such functions. Reflected radiance function is known to be a mostly low frequency function [Horn 1974]. This is the reason why spherical harmonic coefficients are often used for representing incident radiance fields [Ramamoorthi and Hanrahan 2001, 2002; Sillion et al. 1991; Sloan et al. 2002; Westin et al. 1992].

We convert the discrete representation of incoming radiance $L(\omega)$ captured on 6 faces of a cube map to spherical harmonic coefficients L_l^m, by computing weighted sums of the pixel values multiplied by $Y_l^m(\omega)$.

$$\begin{aligned} L_l^m &\approx \sum_{face=1}^{6} \sum_{i=1}^{size} \sum_{j=l}^{size} L_{face}(i,j) Y_l^m(\omega)\, A(\omega) \\ A(\omega) &= \int_{pixel_{ij}} d\omega \qquad (2) \end{aligned}$$

In Equation 2, *size* is the dimension of a cube map face, $L_{face}(i,j)$ is a radiance stored in pixel (i,j) of one of the faces, ω in $Y_l^m(\omega)$ represents the direction from the center of the cube to a pixel, and $A(\omega)$ is the solid angle occupied by the pixel on the unit sphere around the center of the cube.

3.3 Computation of $L_{indirect}$

Spherical harmonic coefficients of the incoming radiance field sampled at grid points are tri-linearly interpolated to any surface point s, inside a grid cell to estimate the incident radiance distribution $L(\omega)$ at s. Reflected radiance along any outgoing direction ω_o, due to this incoming radiance distribution is computed using Equation 3. This makes the $L_{indirect}$ of our algorithm.

$$\begin{aligned} L_{indirect}(\omega_o) &= \mathbf{T}L \\ &= \int_{\omega_i \in \Omega_N} L(\omega)\, \rho(\omega_i, \omega_o)\; \cos\theta_i\, d\omega_i \\ &= \int_{\omega_i \in \Omega_N} L(\omega)\, \widehat{\rho}(\omega_i, \omega_o)\; d\omega_i \\ &= \sum_{l} \sum_{m=-l}^{l} L_l^m \int_{\omega_i \in \Omega_N} Y_l^m(\omega)\; \widehat{\rho}(\omega_i, \omega_o)\; d\omega_i \\ &= \sum_{l} \sum_{m=-l}^{l} L_l^m\, T_l^m(\omega_o) \qquad (3) \end{aligned}$$

In Equation 3, 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 $\rho \cos\theta_i$. $T_l^m(\omega_o)$, the integration term of the right-hand side of the last equation, represents the coefficient of the transfer function that transfers the incoming radiance field at the surface to reflected radiance field. Note that in Equation 3, ω, the direction for the incoming light field, is normally defined with respect to a global co-ordinates system, where as ω_i, ω_o are defined with respect to a co-ordinate system local to the surface.

In the following part of this section we provide expressions for the evaluation of T_l^m for radially symmetric BRDFs, such as Lambertian model and Phong model. Such BRDFs consist of a single symmetric lobe of a fixed shape, whose orientation depends on a well defined central direction, $\mathbf{C}$. For Lambertian model this direction is the surface normal, $\mathbf{N}$ and for Phong lobe this direction is reflection vector $\mathbf{R}$ of ω_o. On reparameterization[1] [see Ramamoorthi and Hanrahan, 2002] these BRDFs become a 1D function of θ_i round this central direction $\mathbf{C}$, where $\theta_i = \cos^{-1}(\mathbf{C} \cdot \omega_i)$. Thus after reparameterization the BRDF function becomes independent of the outgoing direction. The spherical harmonic representation of this 1D function can be written as:

$$\widehat{\rho}(\theta_i) = \widehat{\rho}(\omega_i, \omega_o) = \sum_{l'} \widehat{\rho}_{l'} Y_{l'}^0(\omega_i).$$

[1] Note that for Lambertian BRDF no reparameterization is required.

where

$$\widehat{\rho}_{l'} = 2\pi \int_{\theta_i=0}^{\frac{\pi}{2}} \widehat{\rho}(\theta_i) Y_{l'}^0(\omega_i) \sin\theta_i d\theta_i.$$

Substituting this expression of $\widehat{\rho}()$ in the expression for $T_l^m(\omega_o)$ we get:

$$\begin{aligned} T_l^m = T_l^m(\omega_o) &= \int_{\omega_i \in \Omega_C} Y_l^m(\omega)\ \widehat{\rho}(\omega_i, \omega_o)\ d\omega_i \\ &= \sum_{l'} \widehat{\rho}_{l'} \int_{\omega_i \in \Omega_C} Y_l^m(\omega) Y_{l'}^0(\omega_i)\ d\omega_i. \end{aligned}$$

In this equation Ω_C is the hemisphere around the central direction $\mathbf{C}$. Note that because of the independence of $\widehat{\rho}$ to the outgoing direction after reprameterization, T_l^m's no more depend on the outgoing direction in the repameterized space.
Using the rotational property (to convert from global direction to local direction) [Ramamoorthi and Hanrahan, 2002] and the orthogonality of spherical harmonic functions the expression for T_l^m simplifies to:

$$T_l^m = \sqrt{\frac{4\pi}{2l+1}}\, \widehat{\rho}_l\, Y_l^m(\omega_C), \tag{4}$$

where ω_C is the unit direction along $\mathbf{C}$.

For reparameterized Phong BRDFs

$$\widehat{\rho}(\theta_i) = k_s \frac{(n+1)}{2\pi} (\cos\theta_i)^n$$

where n is the shininess and k_s is the specular reflection constant with value between 0 and 1, and thus

$$\begin{aligned} \widehat{\rho}_l &= (n+1)k_s \int_0^{\pi/2} (\cos\theta_i)^n Y_l^0(\theta_i) \sin\theta_i\, d\theta_i \\ &= k_s \sqrt{\frac{2l+1}{4\pi}} \begin{cases} \frac{(n+1)(n-1)(n-3)\ldots(n-l+2)}{(n+l+1)(n+l-1)\ldots(n+2)} & odd\ l \\ \frac{n(n-2)\ldots(n-l+2)}{(n+l+1)(n+l-1)\ldots(n+3)} & even\ l \end{cases} \end{aligned} \tag{5}$$

The coefficients $\widehat{\rho}_l$ fall off as a Gaussian with width of order $\sqrt{n}$. In other words, for specular surfaces with larger n values, we will get a better approximation of the function with a value proportional to $\sqrt{n}$ as the limit for l.

For Lambertian BRDFs,

$$\widehat{\rho}(\theta_i) = \frac{k_d}{\pi} \cos\theta_i$$

where k_d is the diffuse reflection constant and has a value between 0 and 1, and thus

$$\widehat{\rho}_l = 2k_d \int_0^{\pi/2} Y_l^0(\theta_i) \cos\theta_i \sin\theta_i\, d\theta_i$$

$$= \frac{k_d}{\pi} \begin{cases} \frac{\sqrt{\pi}}{2} & l = 0 \\ \sqrt{\frac{\pi}{3}} & l = 1 \\ 2\pi\sqrt{\frac{2l+1}{4\pi}} \frac{(-1)^{\frac{l}{2}-1}}{(l+2)(l-1)} \frac{l!}{2^l \left(\frac{l}{2}!\right)^2} & l \geq 2, even \\ 0 & l \geq 2, odd \end{cases} \tag{6}$$

Ramamoorthi and Hanrahan [2001] have shown that $l \leq 2$ captures the Lambertian BRDF with over 99% accuracy.

4 Mapping of Our Algorithm to Programmable Hardware

In this section, we describe the mapping of the algorithm described in Section 3, to DirectX 9 compatible GPU. A summary of this implementation is given in the steps below. A detail description follows in the following subsections.

1. *Render cube map at each of the $n_x \times n_y \times n_z$ grid points*: We use *floating point off-screen render targets* to render and to store the cube maps.

2. *Compute M spherical harmonic coefficients to represent Radiance captured at the grid points*: We compute the spherical harmonics coefficients in pixel shaders using multiple passes. These coefficients are stored as M, 3D texture maps of dimension $n_x \times n_y \times n_z$.

3. *Compute $L_{indirect}$*: We use a pixel shader (i) to tri-linearly interpolate the radiance coefficients L_l^m's from the 3D textures in hardware based on the 3D texture co-ordinates of the surface point visible through the pixel, (ii) to compute T_l^m's, the central direction specific coefficients shown in Equation 3, and finally (iii) to compute $L_{indirect}$ from the T_l^m's and the interpolated L_l^m's as per Equation 2.

4. Repeat steps 1-3 for a number of times as the time budget allows.

5. *Render image with lighting from global illumination*: We compute L_{direct} from local light model with shadow support. We add this L_{direct} to $L_{indirect}$ computed from step 4, and assign the result to the image pixel.

All the steps of our algorithm are implemented using fragment/vertex shaders. We have implemented the algorithm in both OpenGL and DirectX9. The executables[2] along with the test scene are available for download [Download 2003]. Note that the standard OpenGL library (Version

[2]The executables have been successfully tested on ATI-Radeon 9700s and 9800s. The OpenGL version uses vendor specific extensions and hence will run only on Radeon 9700s, and 9800s. However, the DirectX version should run on a programmable GPU providing full DirectX 9 functionality including the support of floating point render targets.

1.4) does not support many of the DirectX9 features. We have made use of ARB extensions and vendor specific extensions to make use of such features.

4.1 Initialization

We divide the volume of the 3D scene into a spatially uniform volumetric grid of dimension: $n_x \times n_y \times n_z$. Then we create M 3D texture maps of size $n_x \times n_y \times n_z$. These maps are used to store the M L_l^m values representing the incoming light field at the grid points. These texture maps are initialized with zeroes at the beginning of the computation for every frame. This amounts to starting the computation with no reflected light.

For every vertex of the scene we compute a normalized 3D texture co-ordinate that locates the vertex inside the volumetric 3D grid. These co-ordinates are computed in the vertex shader. The graphics pipeline automatically interpolates them for every surface point visible through the pixels and makes them available to fragment shader for fetching and interpolating any information stored in 3D textures.

4.2 Cube Map Generation

We generate cube maps at the grid points of our volumetric grid. Generating each cube map comprises six renderings of the scene for a given grid point. Instead of rendering into the framebuffer, we render them to off-screen render targets. In DirectX these render targets are textures and in OpenGL they are known as *pixel buffers* or *pbuffers*. These render targets are available in multiple formats. We set them to 32-bit IEEE floating-point format to carry out computations directly on the cube maps without any loss of precision.

4.3 Computing Spherical Harmonic Coefficients

This calculation involves a discrete summation of cube map pixel values as per Equation 2. We carry out this computation in fragment shader. In conventional CPU programming, this task is easily implemented using an iterative loop and a running sum. Current generation programmable hardware does not offer looping at the level of the fragment shader. We resort to a multi pass technique to implement this step.

In Equation 2, the $Y_l^m(\omega)$'s and A(ω)'s are the same for every cube map. We therefore compute them only once and store the product $Y_l^m(\omega)\,A(\omega)$ in the form of textures. We store a total of $6 \times M$ textures. Their values are small fractions (for a 32×32 texture the values are of the order of 10^{-3}), and are therefore stored in floating point format. (For OpenGL implementation we use vendor specific Texture_Float extension e.g. GL_ATI_texture_float, to store the texture in 32 bit IEEE floating point format.)

In the first pass, we multiply each pixel in the cube maps with its pixel specific factor, $Y_l^m(\omega)\,A(\omega)$. For this purpose, we use the cube map rendered into the render target as one texture and the pre-computed factor textures as the other. We use the fragment shader to fetch the corresponding values from the textures, and output their product to a *render target*. We apply this pass for each cube map in the grid.

In the next pass, we sum the products. This step seems to be an ideal step for making use of the *mip-map* generation capability of the hardware. Unfortunately, current hardware does not support *mip-map* computation for floating points textures. Therefore, we perform this step by repeatedly reducing the textures into quarter of their original sizes. We use a *render target*, quarter the size of the texture being reduced, as a rendering target to render a dummy quadrilateral with 16 texture coordinates associated with each vertex of the quadrilateral. These texture co-ordinates represent the position of sixteen neighboring pixels. The texture coordinates are interpolated at the pixel level. Sixteen neighboring pixels are fetched from the texture being reduced and summed up into a single pixel in the *render target*. This process is repeated until the texture size reduces to 1. The resulting value represents one of the spherical harmonic coefficients L_l^m. This procedure is carried out for each coefficient (M times). We assemble M 3D textures of dimension $n_x \times n_y \times n_z$ from the M coefficients of the grid points. (n_x, n_y, n_z represent the scene grid dimension.)

4.4 Computing $\mathbf{L}_{indirect}$

At the time of rendering cube maps or view frame, the 3D texture coordinate for the surface point, s, visible through each pixel is fetched from the pipeline and is used to index into the M 3D textures holding the coefficients. The graphics hardware tri-linearly interpolates texture values (L_l^m's) of eight neighboring grid-points nearest to s. The fragment shader fetches these M interpolated L_l^m coefficients. It evaluates Equation 4 for M T_l^m coefficients from the Y_l^m values computed along the central direction $\mathbf{C}$ and from the surface reflectance ρ_l's. For Lambertian surface $\mathbf{C}$ is the surface normal $\mathbf{N}$ and for Phong specular surface $\mathbf{C}$ is the reflected view vector $\mathbf{R}$. The ρ_l's for the surface material are pre-computed (using Equation 5 for Phong specular surfaces and Equation 6 for Lambertian surfaces) and stored with each surface. At present these coefficients and other reflectance parameters (k_d for Lambertian Surfaces, k_s for Phong surfaces) are passed to the fragment shader program as program parameters. Finally using Equation 3, the fragment shader evaluates $L_{indirect}$ from the L_l^m's and ρ_l's. The fragment shader also evaluates the local light model to get L_{direct} and assigns the sum of $L_{indirect}$ and L_{direct} to the rendered pixel as its color.

4.5 Shadow Handling for L_{direct} Computation

Note that shadow is not directly supported in the rendering pipeline. We use the shadow mapping technique [Williams 78, Reeves et al. 87] for handling shadows. We capture a light map for each light source and store them as depth textures. The vertex shader computes the light space coordinates for each vertex along with their eye space coordinates. At the pixel level, the fragment shader fetches the appropriate z-value (based on light space x and y coordinates) from the depth texture and tests for shadowing. Shadow maps are captured for every frame to handle the dynamic nature of the scene.

5 Results

We present the results from the implementation of our algorithm on a 1.5 GHz Pentium IV using a Radeon-9800 graphics card. We show the images from two scenes: "Hall" and "Art Gallery". Both the scenes are illuminated by one spot light source. Thus the illumination in the scene is due to the light spot on the floor and inter-reflection of the light bouncing off the floor. Hall scene has 8000 triangles and the art-gallery has 80,000 triangles. We used a uniform grid of dimension 4x4x4, rendered cube maps at a resolution of 16×16 for each face, and computed 9 spherical harmonic coefficients[3]. We use a simplified version (bounding box around objects with finer triangles) of the scene during cube map computation. Most expensive step in a cube map computation is the scene rasterization which must be repeated at every grid point. Our DirectX implementation takes about 80 msecs for this rasterization. Because of the static nature of our volumetric grid during the iterations, the surface points visible though the pixels of the cube map do not change from iteration to iteration. Hence we rasterize the scene for the cube maps and store the surface related information in multiple render targets at the begin of computation for every frame and use it for cube map rendering during every iteration. This significantly reduces the computation time of our algorithm. Further, during the movement of light source alone and during interactive walk-throughs, the surface information related to cube-map remain unchanged as well and hence do not require any re-rasterizaton. Using this strategy, we get a frame rate of greater than 40 fps for interactive walk-though with dynamic light in our test scenes. For dynamic objects, the spatial coherence across the frames is lost and hence the cube map related rasterization must be carried out at the begin of computation for every frame. In such cases the frame-rate reduces to about 10 fps. In the attached video, we have captured an interactive session where we move light sources and provide visual feedback in more than 40 frames per second.

Table 1 shows lists the timing for various steps of the implementation.

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

	Hall Scene (8,000 triangles)	Art Gallery (80,000 triangles)
Rasterization for cubemaps	67.3	86.2
Rendering cube maps using raster data	0.326	0.513
Computation of Coefficients and $L_{indirect}$(2 Iterations)	11.1	10.7
Final frame Rendering	0.144	0.8
Overheads	8.33	11.68
Total time for scene with dynamic light source	20.9 (47.8 fps)	23.7 (42.49 fps)
Total time for scene with dynamic objects	107 (9.3 fps)	110 (9.0 fps)

Figure 2 shows the renderings of Hall scene using our algorithm. Figure 2a shows a view rendered without any $L_{indirect}$. Figures 2b-c show views rendered with $L_{indirect}$, computed from one and two iterations of our algorithm respectively. Figure 2d shows $L_{indirect}$ from the 1^{st} iteration and Figure 2e shows the incremental contribution of the 2^{nd} iteration towards $L_{indirect}$. Figure 3 shows the comparison of results from two iterations of our algorithm (in Figure 3a), with the results computed using RADIANCE, a physically based renderer [Ward-Larson et al. 1998] (in Figure 3b). The comparison of the images in Figures 3a and 3b shows that the result obtained in two iterations of our algorithm is very close to the accurate solution. Thus we believe that a good approximation to global illumination can be achieved in about 100 ms. Table 1 lists the time spent in various steps of the algorithm. As we mentioned earlier a significant fraction of this time (86ms) is spent in rasterizing the scene for cube map computation. Rendering cube maps using this information and integration accounts for rest per iteration takes only about 7 ms. Thus increasing the number of iterations for capturing multiple bounces of light will not reduce the frame rate significantly.

Figure 4a shows the specular rendering effect in the "Hall" scene with a specular ball. For this rendering we

[3]Ramamoorthi and Hanrahan [2001] have shown that the reflected light field from a Lambertian surface can be well approximated using only 9 terms its spherical harmonic expansion: 1 term with $l = 0$, 3 terms with $l = 1$, and 5 terms with $l = 2$. For a good approximation for Phong specular surfaces the required number of coefficients is proportional to $\sqrt{n}$ where n is the shininess of the surface. For uniformity we have used 9 coefficients for both Lambertian and Phong surfaces.

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 2nd row of Table 1).

Figures 4b-d show the rendering of the "Art Gallery" scene from three different view points. Note the color bleeding on the walls and the ceiling due to the reflection from the yellowish floor.

The same algorithm rendered using a combination of GPU (for Cube map capture and for final rendering) and CPU (for spherical harmonic coefficients) takes 2 seconds for the scenes used in the paper. The main bottleneck in this implementation is the relatively expensive transfer data between GPU and CPU. Thus the GPU only implementation provides us with at least twenty times faster performance.

6 Handling of Indirect Shadow

In our algorithm so far, we did not describe the possible occlusion during the propagation of light from the grid points to the surface points. Indiscriminate tri-linear interpolation may introduce error in the computation in the form of leakage of shadow or light. We handle this problem by a technique similar to shadow mapping. At cube map rendering time, we capture the depth values of the surface points visible to the cube map pixels 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 point of interest is visible or invisible from the grid points. Based on this finding we associate a weight of 1 or 0 to the grid point. This requires modification of the simple tri-linear interpolation to weighted tri-linear interpolation. Current GPU does not support weighted tri-linear interpolation. In Figure 5 we show the results obtained using a CPU implementation of the weighted interpolation on our "Hall" scene. Figures 5a and 5b show a comparison of the results without and with indirect shadow. The light leakage on to the arches and pillars in Figure 5a is suppressed in Figure 5b. Compare the image in Figure 5b to image computed using Radiance shown in Figure 5c. The lighting distribution on the arches and the pillars are now comparable. Thus, we find a much better match between the images. We are working on a method to map this extension to our hardware implementation.

7 Conclusion and Future work

In this paper we presented a progressive global illumination algorithm that is designed to take advantage of current graphics hardware features. We implemented the whole algorithm in a programmable graphics card. Using this implementation we are able to render dynamic scenes with global illumination in 10 to 40 frames per second. Our algorithm does not depend on any additional resources other than programmable graphics hardware. Thus the speed-up of this algorithm is simply tied to the speed-up of the hardware.

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 change. Pre-computed BRDF coefficients will be stored in a 2D table as textures and will be used during the $L_{indirect}$ computation. We are currently extending our implementation to support general BRDFs in our scene.

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. Currently, we are experimenting with a dynamic grid structure where grid is laid out in the viewing frustum volume and the grid moves dynamically with the view. Another strategy will be to adaptively distribute the grid points in the scene. This strategy will be closer to the irradiance cache approach, and is likely to be computationally less complex for larger environments.

There are numerous potential applications of our algorithm, such as virtual reality, interactive lighting design, and interactive modeling. We are creating a plug-in for 3D Studio Max to provide a 'modeling while rendering' experience to the designers.

References

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

[2] BLINN, J. F. AND NEWELL, M. E. 1976. Texture and reflection in computer generated images. In *Communications of the ACM*, 19:542-546.

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

[4] COHEN, M. F., CHEN, S. E., WALLACE, J. R. AND GREENBERG, D. P. 1988. A Progressive Refine-

ment Approach to Fast Radiosity Image Generation, *Proceedings of ACM SIGGRAPH '88*, 75-84, August 1988.

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

[6] Download. 2003. http://www.cs.ucf.edu/graphics/GPUassistedGI/Executables.zip

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

[8] HORN, B. K. P. 1974. Determining lightness from an image. In Computer Graphics and Image Processing, 3(1), 277-299.

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

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

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

[12] 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.

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

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

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

[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] SILLION, F. X., ARVO, J., WESTIN, S. AND GREENBERG, D. P. 1991. A Global Illumination Solution for General Reflectance Distribution. In *Proceedings of SIGGRAPH 91*, 187-196.

[19] 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.

[20] 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.

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

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

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

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

8 Appendix: Spherical Harmonic Functions

Spherical harmonic functions are harmonic basis functions defined on a sphere. A spherical harmonic function Y_l^m for $l \geq 0$ and $-l \leq m \leq +l$ is given by the following equation.

$$Y_l^m(\theta,\phi) = (-1)^m \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)\, e^{im\phi} \quad (7)$$

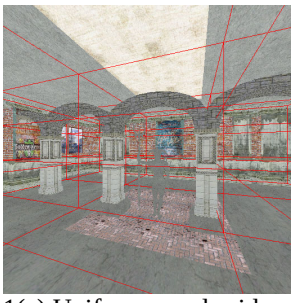
where P_l^m is an associated Legendre polynomial. The normalization of the functions is chosen such that

$$\int_0^{2\pi}\int_0^{\pi} Y_l^m(\theta,\phi)\, Y_{l'}^{m'}(\theta,\phi) \sin\theta \, d\theta \, d\phi = \delta_{mm'}\delta_{ll'}$$

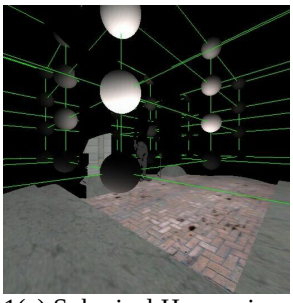
Given a unit vector $\mathbf{C}$ whose cartesian components are (x,y,z) and polar angles are (θ,ϕ), the Y_l^m function values up to $l \leq 2$ along $\mathbf{C}$ are given in the following equation both

using the angles and using the cartesian components[Arfken 1970].

$$
\begin{aligned}
Y_0^0(\theta,\phi) &= \frac{1}{\sqrt{4\pi}} \\
Y_1^1(\theta,\phi) &= -\sqrt{\frac{3}{8\pi}}\sin\theta e^{i\phi} \\
Y_1^0(\theta,\phi) &= \sqrt{\frac{3}{4\pi}}\cos\theta \\
Y_1^{-1}(\theta,\phi) &= \sqrt{\frac{3}{8\pi}}\sin\theta e^{-i\phi} \\
Y_2^2(\theta,\phi) &= \sqrt{\frac{5}{96\pi}}3\sin^2\theta e^{2i\phi} \\
Y_2^1(\theta,\phi) &= -\sqrt{\frac{5}{24\pi}}3\sin\theta\cos\theta e^{i\phi} \\
Y_2^0(\theta,\phi) &= \sqrt{\frac{5}{4\pi}}\left(\frac{3}{2}\cos^2\theta - \frac{1}{2}\right) \\
Y_2^{-1}(\theta,\phi) &= \sqrt{\frac{5}{24\pi}}3\sin\theta\cos\theta e^{-i\phi} \\
Y_2^{-2}(\theta,\phi) &= \sqrt{\frac{5}{96\pi}}3\sin^2\theta e^{-2i\phi}
\end{aligned}
$$


1(a) Uniform voxel grid


1(b) Sample cubemap


1(c) Spherical Harmonics

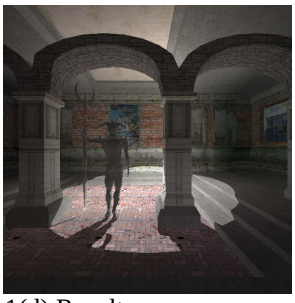
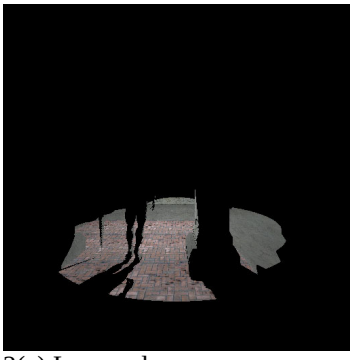
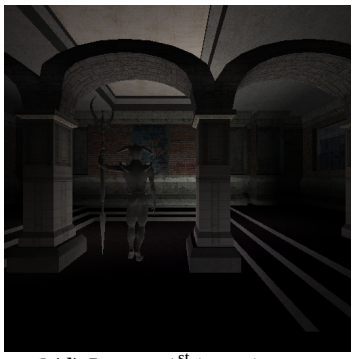

1(d) Result

Figure 1. *Salient steps in the algorithm*


2(a) L_{direct} only

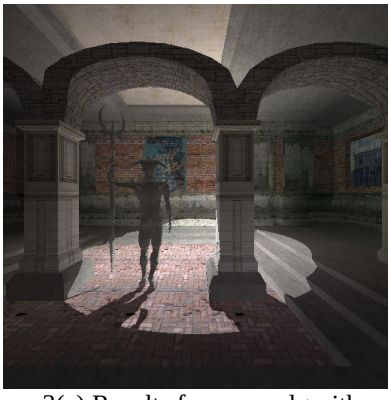

2(b) L_{direct}+ $L_{indirect}$ one iteration


2(c) L_{direct}+ $L_{indirect}$ two iterations


2(d) $L_{indirect}$ 1st iteration


2(e) $L_{indirect}$ 2nd iteration

Figure 2. *Contributions from individual iterations*

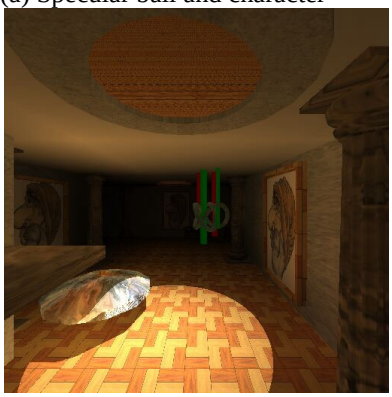

3(a) Results from our algorithm


3(b) Results from Radiance

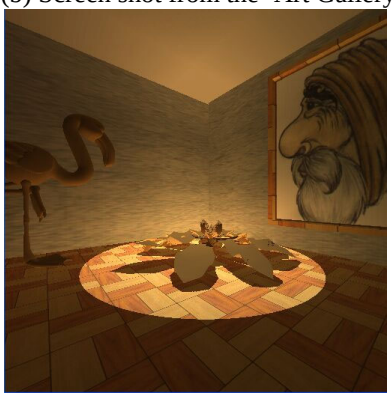
Figure 3. *Results from the 'Hall' scene*

4(a) Specular ball and character

4(b) Screen shot from the ‘Art Gallery’



4(c) Screen shot from the ‘Art Gallery’



4(d) Screen shot from the ‘Art Gallery’

Figure 4. *Some more results*

5(a) Without indirect shadow handling

5(b) With indirect shadow handling

5(c) Results from Radiance

Figure 5. *Handling Indirect shadows*