Stochastic Glossy Global Illumination on the GPU

Attila Barsi[*] TU Budapest
László Szirmay-Kalos[†] TU Budapest
Gábor Szijártó[‡] TU Budapest

Abstract

This paper presents an algorithm for the glossy global illumination problem, which runs on the Graphics Processing Unit (GPU). In order to meet the architectural limitations of the GPU, we apply randomization in the iteration scheme. Randomization allows to use that set of the possible light interactions, which can be efficiently computed by the GPU, and makes it unnecessary to read back the result to the CPU. Instead of tessellating the surface geometry, the radiance is stored in texture space, and is updated in each iteration. The visibility problem is solved by hardware shadow mapping after hemicube projection. The shooter of the iteration step is selected by a custom mipmapping scheme, realizing approximate importance sampling. The variance is further reduced by partial analytic integration.

1 Introduction

This paper proposes the GPU implementation of a stochastic global illumination algorithm. Unlike previous approaches, we emphasize here the application of randomization as a tool to solve general problems on the GPU. Randomization, also called Monte Carlo method, has proven to be very successful to solve high dimensional integration problems. On the other hand, randomization also allows to replace a computation by another simpler calculation which gives back the result of the original computation just in an average case. In this sense, randomization provides us enormous freedom to simplify or to restructure algorithms. In case of special hardware the goal of randomization is to trace back the computation to random steps, which can efficiently be carried out by the GPU hardware.

The paper is organized as follows. In section 2 we review the difficulties of porting algorithms to the stream processing architecture of the GPU. Section 3 discusses previous work on the hardware support for global illumination, and the new algorithm is placed among the existing methods. Section 4 presents the new method and its implementation details. Section 6 analyzes its performance, and finally the paper is closed with conclusions.

2 Porting algorithms to the GPU

When porting an algorithm to the GPU we have to implement three programs, one for the CPU, one for the vertex shader, and one for the pixel shader. Vertex and pixel shaders form a stream processing architecture, where CPU feeds the vertex shader, which only modifies data items. Vertex shader results are passed to fixed interpolators that linearly interpolate the values over the pixels of the triangles. The interpolated values are in turn sent to pixel shaders that can read textures, and can only write its target pixel stored either in the frame buffer or in the texture memory. Reading back any data to the CPU destroys pipeline efficiency, thus should be avoided. Meeting all these limitations is the real challenge of GPU programming.

[*]e-mail:blade80@freemail.hu
[†]e-mail: szirmay@iit.bme.hu
[‡]e-mail:szijarto.gabor@freemail.hu

In the following subsections we consider three particular problems due to the architectural limitations and their general solutions.

The graphics hardware uses the z-buffer algorithm for visibility checks. This algorithm processes the geometry (triangles) in an arbitrary order, but the valid visibility information is available only if the complete geometry has gone through the pipeline. When a point of a triangle is processed, it is not yet known whether or not this point is visible. Thus computations involving the visibility function require two passes. In the first pass the visibility function is computed, then in the second pass the computation can depend on this valid visibility information. This two pass approach resembles to shadow map methods [Coombe et al. 2004].

GPU algorithms are data driven, which means that the result can only be written into the pixel (or texel), which is at the end of the current processing pipeline. Thus we should know the target pixel even at the vertex shader where the pixel data will be written to. Consider the example of ray-shooting, which selects an origin and shoots a ray into the scene, and finally adds the carried power to that surface area, which is hit by the ray. Such shooting type algorithm cannot be implemented by the pixel shader since the location of the result (i.e. the receiver of the transfer) becomes known after the calculation, while a pixel shader is allowed to write only to its own pixel. However, in gathering a ray is shot from the ray origin, and the radiance of the hit point is transferred to the origin of the ray. Thus the location where the result should be written is known in advance, which meets the requirements of the GPU pipeline.

Considering this fact, gathering type global illumination algorithms may seem to be better for GPU implementation. However, it is known that shooting type approaches have potentially faster convergence [Sbert 1996; Bekaert 1999], thus — despite to the difficulties — shooting is still worth implementing on the GPU. We concluded that due to the temporarily invalid visibility information, computations based on visibility consist of at least two passes. The problem of shooting can be solved if in the second pass we set a transformation that visits potentially all possible result locations, and based on the visibility information of the previous pass, these result locations are either updated or left unchanged.

Because of the fixed register set, the amount of data arriving at a pixel shader is limited, thus many partial results cannot be computed and summed in a single pass. A usual trick to attack this problem is computing a single term of the sum in each pass, which is added with a copy image of the target buffer. This copy is mirrored at the end of each pass. This technique, which is called *ping-pong buffering*, can be used only in special circumstances where the number of added terms is rather limited, since the number of the required passes is equal to the number of terms to be summed.

This problem can be solved by Monte Carlo summation. The sum is approximated by a randomly selected term, which is divided by the selection probability. Thus the approximation can be obtained in a single pass no matter how many terms the sum has. The randomization of the summation introduces some random noise in the result, which can be eliminated by averaging the temporary results. This averaging, on the other hand, can be very efficiently implemented by the ping-pong technique. The previous result is read from one of the ping-pong buffer, is averaged with the actual result, and the result is written to the other ping-pong buffer.

Monte Carlo methods have been successfully applied to solve global illumination problems since they can efficiently approximate high dimensional integrals inherent in global illumination [Bekaert 1999; Keller 1998]. Note, however, that now randomization has a different objective. Randomization is regarded as a technique that can substitute a complex operation (summation or integration) by a much simpler technique, which can be executed by the available hardware.

3 Global illumination with hardware support

Global illumination algorithms aim at the solution of the rendering equation

$$L(\vec{x},\omega) = E(\vec{x},\omega) + R(\vec{x},\omega),$$

which expresses the radiance $L(\vec{x},\omega)$ of point $\vec{x}$ at direction ω as a sum of emission E and reflection R of the radiances of all points that are visible from point $\vec{x}$. The reflection of the radiance of visible points is expressed by an integral operator

$$(\mathcal{T}_{f_r} L) = \int_S L(\vec{y},\omega_{\vec{y}\to\vec{x}}) \cdot f_r(\omega_{\vec{y}\to\vec{x}},\vec{x},\omega) \cdot G(\vec{x},\vec{y})\; dy, \qquad (1)$$

which is also called as the *light transport operator* [Kajiya 1986]. In this equation S is the set of surface points, f_r is the BRDF and

$$G(\vec{x},\vec{y}) = v(\vec{x},\vec{y}) \cdot \frac{\cos\theta_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}$$

is the *geometric factor*, where $v(\vec{x},\vec{y})$ is the mutual visibility indicator, which is 1 if points $\vec{x}$ and $\vec{y}$ are visible from each other and zero otherwise, $\theta_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the surface normals and direction $\omega_{\vec{y}\to\vec{x}}$ that is between $\vec{x}$ and $\vec{y}$.

The solution of the rendering equation requires general purpose instructions and is thus usually computed on the CPU [Dutre et al. 2003]. Our goal is to take advantage of the huge computation power of the GPU for the solution of the rendering equation. In order to do so, we should transform the algorithm according to the capabilities of the GPU.

The CPU-based solution algorithms can be classified as random walk [Kajiya 1986] and iteration techniques. The GPU support of random walk algorithms has been examined in [Purcell et al. 2003]. Since iteration algorithms are conceptually closer to local illumination, which is originally supported by GPUs, we believe that iteration algorithms are better candidates for GPU implementation.

Iteration techniques are based on the fact that the solution of the rendering equation is the fixed point of the following iteration scheme:

$$R_m = \mathcal{T}_{f_r} L_{m-1} = \mathcal{T}_{f_r}(E + R_{m-1}).$$

If this scheme is convergent, then the solution can be obtained as a limiting value:

$$R(\vec{x},\omega) = \lim_{m\to\infty} R_m(\vec{x},\omega).$$

Iteration works with the complete radiance function, whose temporary version should be represented somehow. The classical approach is the *finite-element method*, which approximates the radiance function in a function series form. In the simplest diffuse case we decompose the surface to small elementary surfaces $A^{(1)},\ldots,A^{(n)}$ and apply a piece-wise constant approximation, thus the reflected radiance function is represented by the reflected radiance of these patches, that is by $R^{(1)},\ldots,R^{(n)}$. In the glossy case the radiance is also a function of the direction, which may be handled by applying a similar finite element decomposition in the directional domain [Sillion and Puech 1994], but this approach would increase the storage requirements considerably. Fortunately, applying randomization, we can solve the glossy global illumination problem without introducing any finite elements in the directional domain [Szirmay-Kalos 1999].

If these elementary surfaces are small, we can consider just a single point of them in the algorithms, while assuming that the properties of other surface points are similar. Surface properties, such as the BRDF and the emission can be given by directional functions $f_r^{(1)}(\omega^{in},\omega^{out}),\ldots,f_r^{(n)}(\omega^{in},\omega^{out})$, and $E^{(1)}(\omega^{out}),\ldots,E^{(n)}(\omega^{out})$, in each positional finite element. These directional functions can be defined by material properties, such as the diffuse and specular albedos and the shininess. We shall use a physically plausible diffuse-specular model [Neumann et al. 1999] defined by the diffuse and specular albedos and the shininess, and assume that the lights are diffuse, but other material and light models could also be used in the proposed method.

Iteration simultaneously computes the interaction between all surface elements, which has quadratic complexity in terms of the number of finite elements, and is hard to implement on the GPU. This problem can be attacked by special iteration techniques, such as Southwell iteration (also called progressive radiosity), hierarchical radiosity, or by randomization. Southwell iteration computes the interaction of the element having the highest unshot radiosity and all other surface elements [Cohen et al. 1988]. It is quite simple to implement but has also quadratic complexity [Szirmay-Kalos and Márton 1995]. Methods supporting progressive radiosity with the graphics hardware is as old as the method itself. The form factors between the shooter and all other patches can be computed with the hemicube method that identifies the visible patches by the z-buffer hardware [Cohen and Greenberg 1985].

The quadratic complexity can be reduced by hierarchical [Aupperle and Hanrahan 1993] and Monte Carlo [Shirley 1991; Neumann 1995] approaches. The limited capabilities of the hardware seem not to be appropriate for the implementation of demanding hierarchical approaches. Monte Carlo techniques, on the other hand, can greatly benefit from the hardware. Stochastic perspective ray bundles use the same elementary step as progressive radiosity [Szirmay-Kalos et al. 2003], while parallel ray-bundles [Neumann 1995] can be traced by depth peeling [Szirmay-Kalos and Purgathofer 1998] or can be made appropriate for normal z-buffer rendering by applying further randomization [Martinez et al. 2002]. All of these early attempts to use the hardware suffered from the limitation of the fixed pipeline, which did not allow the complete algorithm to be executed on the graphics card. Read-backs transferring data to the CPU, however, significantly reduced the speed. An exception is the instant radiosity [Keller 1997], which computed the last eye-step of a random-walk algorithm using the standard graphics hardware, thus it could eliminate the read-backs.

The emergence of programmable graphics hardware has made it possible to implement the complete algorithm without the performance penalty of read-backs. CPU algorithms usually decompose general surfaces to triangular patches. However, in GPU approaches this is not feasible since GPU processes patches independently thus the computation of the interdependence of patch data is difficult. Instead, the radiance function can be stored in a texture [Bastos et al. 1997; Nielsen and Christensen 2002]. An elementary surface area $A^{(i)}$ is the surface which is mapped onto texel i. Full matrix radiosity [Carr et al. 2003], progressive refinement [Nielsen and Christensen 2002], substructuring [Coombe et al. 2004], and final gathering with parallel ray bundles [Hachisuka 2004] have already been successfully implemented on the graphics hardware. However, as these algorithms have quadratic complexity they could achieve interactive frame rates on only simple models represented by at most 100×100 texture resolution. Due to the complicated operations, the CPU implementation has turned out to be not slower than the GPU algorithm in many applications [Carr et al. 2003].

Unlike these previous methods, this paper proposes the implementation of a Monte Carlo global illumination algorithm on the graphics hardware. The formal basis of such approaches is the stochastic iteration, which was originally proposed for the solution of the linear equations [Neumann 1995; Sbert 1996; Bekaert 1999], then extended for the solution of integral equations [Szirmay-Kalos 1999]. Stochastic iteration means that in the iteration scheme a random transport operator $\mathscr{T}^*_{f_r}$ is used instead of the light-transport operator $\mathscr{T}_{f_r}$. The random transport operator has to give back the light-transport operator in the expected case:

$$R_m = \mathscr{T}^*_{f_r}(E + R_{m-1}), \qquad E[\mathscr{T}^*_{f_r} L] = \mathscr{T}_{f_r} L.$$

Note that such an iteration scheme does not converge, but the iterated values will fluctuate around the real solution. To make the sequence converge, we obtain an image estimate at each iteration step, and compute the final result as the average of these image estimates:

$$C = \mathscr{M}E + \frac{1}{m} \cdot \sum_{k=1}^{m} \mathscr{M}R_k,$$

where $\mathscr{M}$ is the measuring operator computing the image seen from the camera from the actual radiance distribution. It means that the implementation of a stochastic iteration algorithm requires the storage of the evolving image (a value C for each pixel), and the temporary version of the reflected radiance function $R_k(\omega)$. The complexity of the representation of this function depends on the properties of the random transport operator, and its simplification is an important criterion to find a good randomization.

The core of all stochastic iteration algorithms is the definition of the random transport operator. We prefer those random operators, that can be efficiently computed on the GPU, introduce small variance, and does not require the complete storage of the radiance function for all points and directions.

In this paper we propose the implementation of the random hemicube shooting (perspective ray bundles) [Szirmay-Kalos et al. 2003] on the GPU. Unlike the CPU based method, we store both the actual irradiance and the accumulated radiance for the camera in textures, and minimize the data storage requirements. During the porting of the algorithm, we had to solve the problems of texture based non-diffuse radiance representation and importance sampling. We significantly simplified the data structures as needed by the GPU and stored the temporary reflected radiance and the evolving image in floating point textures. Note that although the new method solves the non-diffuse global illumination problem, thanks to randomization, it still uses two small RGBA textures. The GPU algorithm and the radiance representation are practically independent of the patch decomposition, surfaces describe only the geometry. We may call this approach as *patchless rendering*, since having detected the visibility between two points, all computations are done on texels independently of the surfaces. It allows us to work with the original geometry, no patch subdivision is necessary, but the finite element representation of the radiance can be high. We consider this as one of the main novelties of the method comparing to previous CPU based stochastic iteration algorithms.

On the other hand, with respect to previous GPU based radiosity algorithms, the main novelty is the introduction of randomization. Randomization allowed the rendering of non-diffuse scenes without any increase in required texture space. On the other hand, we can benefit the sub-quadratic complexity of Monte Carlo methods, which makes randomized approaches a definite winner for more complex scenes. Moreover, randomization can be regarded as a general tool to trace back the steps of general algorithms to those, which can be executed by the GPU.

4 The new GPU based global illumination algorithm

According to the rendering equation, the reflected radiance can be obtained by evaluating a surface integral. In a GPU algorithm, the surface is decomposed to small patches corresponding texels of the texture map. Let us first assume that the resolution of this texture map is high enough to make all elementary surfaces small, and thus we can check the mutual visibility of two elementary surfaces by inspecting only their centers. In this case the update of the reflected radiance representation in a single iteration can be approximated in the following way:

$$R_m^{(i)}(\omega) = \frac{1}{A^{(i)}} \cdot \int_{A^{(i)}} \int_{S} L_{m-1}(\vec{y}, \omega_{\vec{y}\to\vec{x}}) \cdot f_r^{(i)}(\omega_{\vec{y}\to\vec{x}}, \omega) \cdot G(\vec{x},\vec{y})\, dy dx \approx \sum_{j=1}^{n} L_{m-1}^{(j)}(\omega_{\vec{y}_j\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot G(\vec{y}_j,\vec{x}_i) \cdot A^{(j)}, \qquad (2)$$

where $\vec{y}_j$ and $\vec{x}_i$ are the centers of elementary surfaces $A^{(j)}$ and $A^{(i)}$, respectively.

In order to efficiently detect visibility between points $\vec{y}_j$ and $\vec{x}_i$, we run a GPU algorithm that is quite similar to depth buffer shadow methods. First a depth image is computed placing the camera at $\vec{y}_j$ and associating the pixels with the cells of a discretized hemicube. Then in the second pass, the center of every surface element $A^{(i)}$ is checked whether or not this surface element is farther from $\vec{y}_j$ than the closest surface projecting onto the same pixel as the center of $A^{(i)}$. If a surface element (i.e. a texel) turns out to be visible, then the contribution from the shooter is added, otherwise this contribution is zero.

Equation 2 — which has to be computed in every iteration — contains a sum for every finite element i and for each iteration. This sum is estimated randomly since the random estimator completely eliminates summation that would pose problems to the GPU. A surface element $A^{(j)}$ is selected with probability p_j, and $\vec{y}_j$ is set to its center. This randomization allows to compute the interaction between shooter surface element $A^{(j)}$ and all other receiver surface elements $A^{(i)}$, instead of considering all shooters and receivers simultaneously. Having selected shooter $A^{(j)}$ and its center point $\vec{y}_j$, the radiance of this point is sent to all those surface elements that are visible from here. The Monte Carlo estimate of the reflected radiance of finite element $A^{(i)}$ after this transfer is

$$R_m^{(i)}(\omega) = \frac{L_{m-1}^{(j)}(\omega_m^{(i)}) \cdot f_r^{(i)}(\omega_m^{(i)}, \omega) \cdot G_m^{(i)}}{p_j}, \qquad (3)$$

where direction $\omega_m^{(i)}$ points from the randomly selected shooter $\vec{y}_j$ of iteration m to point $\vec{x}_i$, and G_m is the geometry factor between them.

Due to the possible glossy reflections, the reflected radiance depends on viewing direction ω, thus its representation would require finite element decomposition in the directional domain as well. Note, however, that this dependence is caused by the BRDF, which makes it possible to store the irradiance by a single value. The iteration algorithm stores the *actual irradiance texture* caused by the last iteration step:

$$I_m^{(i)} = \frac{L_{m-1}^{(j)}(\omega_m^{(i)}) \cdot G_m}{p_j}. \tag{4}$$

Storing the shooter location of the previous iteration $\vec{y}_j$ in a global (uniform) parameter, the reflected radiance in an arbitrary direction ω can be obtained as

$$R_m^{(i)}(\omega) = I_m^{(i)} \cdot f_r^{(i)}(\omega_m^{(i)}, \omega). \tag{5}$$

The final image will be the average of the estimates computed for the eye direction. To obtain this average we compute the reflected radiance for the eye direction and add this result to *accumulation texture* value C:

$$C_m^{(i)} = C_{m-1}^{(i)} + I_m^{(i)} \cdot f_r^{(i)}(\omega_m^{(i)}, \omega_{eye}). \tag{6}$$

When the final result is displayed, we render the scene with the texture of values C divided by the number of iterations.

In order to realize this random transport operator, two tasks need to be solved, including the random selection of a texel identifying point $\vec{y}_j$, and the update of the irradiance at those texels which correspond to point $\vec{x}_i$ visible from $\vec{y}_j$ while also computing the contribution of this transfer onto the image.

4.1 Random texel selection

The randomization of the iteration introduces some variance in each step, which should be minimized in order to get an accurate result quickly. According to importance sampling, the introduced variance can be reduced with a selection probability that is proportional to the integrand. Unfortunately, this is just approximately possible, and the selection probability is set proportional to the current power of the selected texel. Since the reflected radiance can be due to only a transfer from the previous shooter, the power can be expressed from the irradiance:

$$\Phi_m^{(j)} = \int_\Omega \int_{A^{(j)}} L_m(\vec{x}, \omega) \cdot \cos\theta \; dx d\omega =$$

$$(E^{(j)} \cdot \pi + I_m^{(j)} \cdot a^{(j)}(\omega_m^{(j)})) \cdot A^{(j)},$$

where $a^{(j)}(\omega^{in}) = \int_\Omega f_r^{(j)}(\omega^{in}, \omega) \cdot \cos\theta \; d\omega$ is the *albedo* of surface element $A^{(j)}$.

If the light is transferred on several wavelengths simultaneously, the luminance of the radiated power should be used. Thus the selection probability of elementary surface j is:

$$p_j = \frac{\mathscr{L}(\Phi^{(j)})}{\Phi}, \qquad \Phi = \sum_k \mathscr{L}(\Phi_k),$$

where $\mathscr{L}$ is the luminance of a spectrum represented by red, green and blue components. Substituting this selection probability into equation 4, we obtain:

$$I_m^{(i)} = \Phi \cdot \frac{E^{(j)} + I_{m-1}^{(j)} \cdot f^{(j)}(\omega_{m-1}^{(j)}, \omega_m^{(i)})}{\mathscr{L}(E^{(j)} \cdot \pi + I_{m-1}^{(j)} \cdot a^{(j)}(\omega_{m-1}^{(j)}))} \cdot G_m. \tag{7}$$

The random selection according to p_j can be supported by a *mipmapping* scheme. Mipmapping has originally been proposed for texture filtering. Later it was also used to find the maximum value in an image [Coombe et al. 2004]. In our approach, however, we use mipmapping to sample randomly, proportional to stored value $\mathscr{L}(\Phi^{(j)})$. A mipmap can be imagined as a quadtree, which allows the selection of a texel in $\log_2 \mathscr{R}$ steps, where $\mathscr{R}$ is the resolution of the texture. Each texel is the sum of the luminance of four corresponding texels on a lower level. The top level of this hierarchy has only one texel, which contains the average of the luminance of all elementary texels. Both generation and sampling require the rendering of a textured rectangle (also called a full screen quad) by $\log_2 R$ times.

The generated mipmap is used to sample a texel with a probability that is proportional to its luminance. First the luminance of the top level texel is retrieved from a texture and is multiplied by a random number uniformly distributed in the unit interval. Then the next mipmap level is retrieved, and the four texels corresponding to the upper level texel is obtained. The luminance of the four pixels are summed, the running sum (denoted by `cmax` in the program below) is compared to value `r` obtained on the higher level. When the running sum gets larger than the selection value from the higher level, the summing is stopped and the actual value is selected. A new selection value is obtained as the difference of the previous value and the luminance of all texels before the found texel (`r-cmin`). Then the same procedure is repeated in the next pass on the lower mipmap levels. This procedure terminates at a leaf texel with a probability that is proportional to its luminance.

The pixel shader of a pass of the mipmap based sampling, which selects according to random value `r` passed from the upper level and originally set randomly with uniform distribution in $[0, \Phi]$:

```
float cmin = 0, cmax = tex2D(texture, uv);
if(cmax >= r) {
   c = float3(r-cmin, uv.x, uv.y);
} else {
   cmin = cmax;
   float2 uv1 = float2(uv.x-rr, uv.y);
   cmax += tex2D(texture, uv1);
   if(cmax >= r) {
      c = float3(r-cmin, uv1.x, uv1.y);
   } else {
      cmin = cmax;
      uv1 = float2(uv.x, uv.y-rr);
      cmax += tex2D(texture, uv1);
      if(cmax >= r) {
         c = float3(r-cmin, uv1.x, uv1.y);
      } else {
         cmin = cmax;
         uv1 = float2(uv.x-rr, uv.y-rr);
         c = float3(r-cmin, uv1.x, uv1.y);
      }
   }
}
return c;
```

The `rr` parameter is the distance between two neighboring pixels in texture address space.

4.2 Update of the radiance texture

The points visible from $\vec{y}$ can be found by placing a hemicube around $\vec{y}$, and then using the z-buffer algorithm to identify the visible patches. Since it turns out just at the end, i.e. having processed all patches by the z-buffer algorithm, which points are really visible, the application of the random transfer operator requires two passes.

4.2.1 First pass: construction of the depth map

In the first pass the center and the base of the hemicube are set to $\vec{y}$ and to the surface at $\vec{y}$, respectively, then the scene is rendered computing the z coordinate of the visible points. These values are written into a texture, called the *depth map*. We also compute a tolerance value as the absolute value of the dot product between the surface normal and the view direction, so we can identify the faces that are close to being viewed at grazing angles from the shooter's point of view.

Note that this approach differs from earlier methods [Nielsen and Christensen 2002; Coombe et al. 2004] creating a map of patch indices and checking whether a patch associated with the pixel is the same as the id stored in the map. Working with depth values instead of patch indices allows tolerance to be incorporated, which can eliminate dot artifacts of previous methods.

4.2.2 Second pass: irradiance update

In the second pass we render into the rectangle of the irradiance texture. It means that the pixel shader visits each texel, and updates the stored actual irradiance (I) and the accumulating radiance (C) according to equations 4 and 6.

The *vertex shader* is set to map a point onto the corresponding texel having coordinates texx:

```
OUT.hpos.x = 2 * IN.texx.x - 1;
OUT.hpos.y = 1 - 2 * IN.texx.y;
OUT.hpos.z   = 0;
OUT.hpos.w   = 1;
OUT.texx = IN.texx;
```

Note that the first instruction not only makes the vertex coordinate equal to the texture coordinate, but also applies a transformation. This transformation is necessary because the vertex screen coordinates must be in [-1, 1], while the texture coordinates are expected in [0,1]. Direct3D has a texture space which defines the upper left corner as $(0,0)$ and the lower right corner as $(1,1)$, so we need to flip y coordinates.

The vertex shader also transforms the input vertex to camera space (x), as well as its normal vector xnorm to compute radiance transfer, determines homogeneous coordinates vch for the location of the point in the depth map.

```
x        = mul(position, modelview).xyz;
xnorm    = mul(xnorm, modelviewIT).xyz;
viscoord = mul(position, modelviewproj);
```

In equation 4, the geometric factor depends on the receiver point, thus its accurate evaluation could be implemented by the *pixel shader*:

```
float3 ytox = normalize(x); // dir y to x
float  xydist2 = dot(x, x); // |x - y|^2
float  cthetax = dot(xnorm, -ytox);
if (cthetax < 0) costhetax = 0;
float3 ynorm(0, 0, 1);
float  cthetay = ytox.z;
if (cthetay < 0) costhetay = 0;
float G = cthetax * cthetay / xydist2;
```

Note that we took advantage of the fact that $\vec{y}$ is the eye position of the camera, which is transformed to the origo by the modelview transform, and the normal vector at this point is transformed to axis z.

When a texel of the current radiance map and of the accumulating radiance map is shaded, it is checked whether or not the center of the surface corresponding this texel is visible from the shooter by comparing the depth values stored in the visibility map. We have to apply a tolerance based on the cosine of the viewing angle in order to avoid point artifacts. This tolerance is stored in the green channel of the visibility map. The pixel shader code responsible for converting homogeneous coordinates (vch) to Cartesian coordinates (vcc) and computing the visibility indicator is:

```
float3 vcc = vch.xyz / vch.w; // Cartesian
vcc.x = (vcc.x + 1) / 2;      // Texture space
vcc.y = (1 - vcc.y) / 2;
float2 depth = tex2D(depthmap, vcc).rg;
float vis = (abs(depth.r - vcc.z) <
            (eps1 - eps2 * depth.g));
```

To obtain the radiance transfer from shooter $\vec{y}$ to the processed point $\vec{x}$, first the radiance of shooter $\vec{y}$ is calculated from its irradiance stored in irradiance map irradmap according to its BRDF, and its emission stored in emissmap. Shooter's texture coordinates texy and previous shooter yprev are passed as uniform parameters:

```
float3 Iy = tex2D(irradmap, texy);
float3 Ey = tex2D(emissmap, texy);
float3 yin = normalize(yprev);  // yprev - y
float3 Ly = Ey +
            Iy * BRDF(texy, yin, ynorm, ytox);
```

The BRDF function reads the BRDF parameters from texture map brdfmap according to texture coordinates texy and computes a simple streched-Phong BRDF [Neumann et al. 1999].

The new irradiance at $\vec{x}$ is obtained from the radiance at $\vec{y}$ multiplying it with visibility vis and geometric factor G computed before, and divided by probability p passed as a uniform parameter. The emission and surface area of this texel are read from texture map emissmap. The luminance of the reflected power (lumPowx) at $\vec{x}$ is also computed to allow importance sampling in the subsequent iteration step, and stored in the alpha channel of the irradiance. Additionally, the contribution to the eye is also determined and the increase value is output in C.

```
float3 Ix = Ly * G * vis / p;
float4 Ex = tex2D(emissmap, texx);
float  Ax = Ex.a;           // surface area
float3 alb = Albedo(texx);  // albedo of x
float3 em  = float(1, 1, 1) * pi;
float  lumPowx = (dot(E, em) + dot(I, alb))/3;
OUT.I = float4(Ix, lumPowx);
float3 xtoe = normalize(eye - x);
float3 Lx = Ex +
            Ix * BRDF(texx, ytox, xnorm, xtoe);
OUT.C = C + Lx;
```

This pixel shader outputs two values, including the irradiance I and accumulating radiance toward the eye C, thus it requires the multiple render target option.

5 Variance reduction

Recall that we applied a finite element decomposition over the positional variation of the irradiance function. The decomposition was so fine that we could test the visibility for a single point of each finite element, thus we could eliminate all summations that would pose problems to the GPU. However, using the same level of surface tessellation for both visibility calculations and irradiance representation is not very effective since visibility information changes much more quickly than the relatively smoother irradiance function. This approach requires large textures, and small finite elements make it possible that two finite element centers get too close to each other, which is responsible for corner spikes. This problem can be solved if we use denser samples for visibility computations than for irradiance representation. It corresponds to merging different terms of equation 2 that correspond to elementary surfaces close to each other and therefore have similar radiance (a term is in fact a texel that represents a small surface area). Merging neighboring texels is a texture filtering, which replaces the texture by a higher mipmap level.

Note that unlike in classical radiosity algorithms and in [Coombe et al. 2004], we do not rely on the patch structure when the correspondence between the two levels of details are defined. The lower resolution texture can be imagined as a filtered version on a higher level in a mipmap structure. When this filtering is implemented, we have to be careful not to combine two texels that would correspond to two different surfaces. This problem is solved by looking up another texture generated during the preprocessing phase. This texture stores a patch id for each texels, and can be used to detect whether or not two texel values can be combined.

When combining several texels, the resulting radiance is the average of the radiance of individual texels, and the resulting geometry factor is the sum of the individual geometry factors. The sum (or the integral) of the geometry factors can be obtained using the point to disc approximation. Let us suppose that the merged texels correspond to surface area A with center $\vec{y}_j$. If the radiance of the texels are similar and their average is $\tilde{L}$, then

$$\int_A L(\vec{y}, \omega_{\vec{y}\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}\to\vec{x}_i}, \omega) \cdot G(\vec{x}_i, \vec{y})\, dy \approx$$

$$\tilde{L}(\omega_{\vec{y}_j\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot v(\vec{y}_j, \vec{x}_i) \cdot \frac{\cos\theta_{\vec{y}_j} \cdot \cos\theta_{\vec{x}_i}}{|\vec{y}_j - \vec{x}_i|^2 + A/\pi} \cdot A. \qquad (8)$$

Note that this merging can significantly reduce the variation of the terms since replacing elementary surfaces $A^{(j)}$ by larger surfaces A, the center $\vec{y}_j$ of the larger surface will be farther from receivers $\vec{x}_i$. On the other hand, the disc to point form factor approximation adds an A/π term to the denumerator, which cannot be as small as in the original approach.

6 Implementation results and further improvements

The proposed method has been implemented on an NV6800GT graphics card in DirectX/HLSL environment. The implementation has been tested with the Cornell box scene and we concluded that a single iteration requires less than 20 msec for a few hundred vertices and for 128×128 resolution radiance maps, while keeping the depth maps at 256×256 (the algorithm is pixel shader limited). Using 64×64 resolution radiance maps introduced a minor amount of shadow bleeding, but increased iteration speed by approximately 40%. Since we can expect converged images after 40 – 80 iterations for normal scenes, this corresponds to 0.5 – 1 frames per second, without exploiting frame-to-frame coherence. The algorithm (depending on the resolution) uses several render-to-surface objects and does one iteration in 22 – 30 passes. With faster, hardware optimized shadow calculation we should be able reduce the number of passes by 4 as well. In order to eliminate flickering, we should use the same random number generator in all frames. On the other hand, as in all iterative approaches, frame to frame coherence can be easily exploited. In case of moving objects, on the other hand, we can take the previous solution as a good guess to start the iteration. This trick not only improves accuracy, but also makes the error of subsequent steps highly correlated, which also helps eliminating flickering.

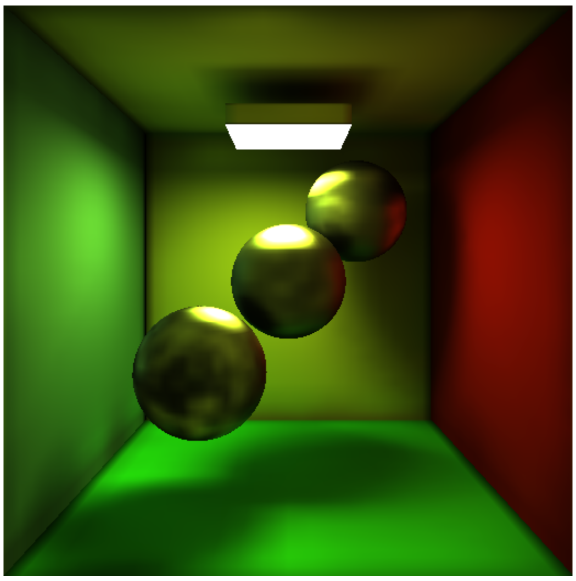

Figure 2: Specular objects in a diffuse room rendered with the proposed method in 9 sec on an Nvidia 6800GT graphics card.

7 Conclusions

This paper presented a stochastic glossy global illumination algorithm running entirely on the GPU. In order to port a global illumination algorithm — which should follow the interdependence of the patch radiances — onto the graphics hardware — which usually assumes patch independence — we used randomization. Randomization has turned out to be an efficient tool to modify algorithms to meet the capabilities of the underlying hardware. The final algorithm is fast, it can render moderately complex scenes interactively, and thus is an appropriate candidate to include global illumination effects in games. On the other hand, the algorithm is relatively simple and easy to implement.

8 Acknowledgement

This work has been supported by the National Scientific Research Fund (OTKA ref. No.: T042735), and the GameTools EU F6 project.

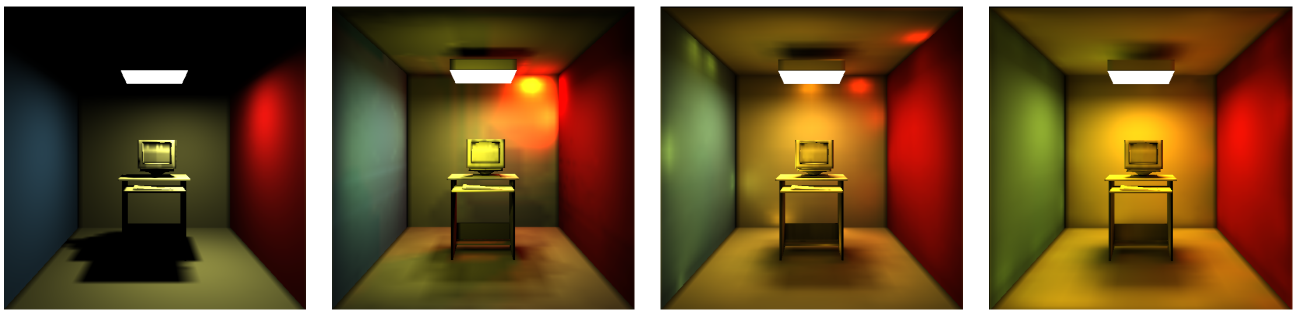

Figure 1: Image rendered with partial analytic integration and multiple importance sampling using 100 iteration. All objects are mapped to a single texture map of resolution 128 × 128. The rendering times are measured on an Nvidia 6800GT graphics card.

References

Aupperle, L., and Hanrahan, P. 1993. A hierarchical illumination algorithms for surfaces with glossy reflection. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 155–162.

Bastos, R., Goslin, M., and Zhang, H. 1997. Efficient radiosity rendering using textures and bicubic reconstruction. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics*.

Bekaert, P. 1999. *Hierarchical and stochastic algorithms for radiosity*. PhD thesis, University of Leuven.

Carr, N., Hall, J., and J., H. 2003. GPU algorithms for radiosity and subsurface scattering. In *Proc. of Workshop on Graphics Hardware*, 51–59.

Cohen, M., and Greenberg, D. 1985. The hemi-cube, a radiosity solution for complex environments. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, 31–40.

Cohen, M. F., Chen, S. E., Wallace, J. R., and Greenberg, D. P. 1988. A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, 75–84.

Coombe, G., Harris, M. J., and Lastra, A. 2004. Radiosity on graphics hardware. In *Graphics Interface*.

Dutre, P., Bekaert, P., and Bala, K. 2003. *Advanced Global Illumination*. A K Peters.

Hachisuka, T. 2004. Final gathering on GPU. In *ACM Workshop on General Purpose Computing on Graphics Processors*.

Kajiya, J. T. 1986. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, 143–150.

Keller, A. 1997. Instant radiosity. *SIGGRAPH '97 Proceedings*, 49–55.

Keller, A. 1998. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. Shaker-Verlag.

Martinez, R., Szirmay-Kalos, L., and Sbert, M. 2002. A hardware based implementation of the multipath method. In *Computer Graphics International*.

Neumann, L., Neumann, A., and Szirmay-Kalos, L. 1999. Compact metallic reflectance models. *Computer Graphics Forum (Eurographics'99) 18*, 3, 161–172.

Neumann, L. 1995. Monte Carlo radiosity. *Computing 55*, 23–42.

Nielsen, K., and Christensen, N. 2002. Fast texture based form factor calculations for radiosity using graphics hardware. *Journal of Graphics Tools 6*, 2, 1–12.

Purcell, T., Donner, T., Cammarano, M., Jensen, H., and Hanrahan, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 41–50.

Sbert, M. 1996. *The Use of Global Directions to Compute Radiosity*. PhD thesis, Catalan Technical University, Barcelona.

Shirley, P. 1991. Time complexity of Monte-Carlo radiosity. In *Eurographics '91*, Elsevier Science Publishers, 459–466.

Sillion, F., and Puech, C. 1994. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, Inc., San Francisco.

Szirmay-Kalos, L., and Márton, G. 1995. On convergence and complexity of radiosity algorithms. In *Winter School of Computer Graphics '95*, 313–322. http//www.iit.bme.hu/˜szirmay.

Szirmay-Kalos, L., and Purgathofer, W. 1998. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98*, 247–258.

Szirmay-Kalos, L., Antal, G., and B., B. 2003. Global illumination animation with random radiance representation. In *Rendering Symposium*.

Szirmay-Kalos, L. 1999. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum (Eurographics'99) 18*, 3, 233–244.