# Stream Processing in Global Illumination

Attila Barsi*
Gábor Jakab †

Department of Control Engineering and Information Technology
Budapest University of Technology
Budapest / Hungary

## Abstract

This paper presents the implementation of the stochastic radiosity algorithm on the graphics hardware. We store the radiosity function in texels of the floating point pbuffer. The radiosity function is updated in each iteration. When converged, the radiosity function is mapped onto surfaces by traditional texture mapping. Our goal is to enable interactive radiosity style rendering of scenes with moving objects and/or light sources.

**Keywords:** Stochastic radiosity, Cg programming, graphics hardware

## 1 Introduction

We witness two kinds of advances of the graphics cards and their processing units (GPU). On the one hand, their speed is improving constantly, outperforming the Moore law. The average computation time of a pixel, including transformations local illumination, projection, clipping, texturing, blending and visibility determination, is about a few nanoseconds, i.e. close to the time of a single memory cycle. This incredible speed is the result of the massive pipelining, the parallelization and the special ALUs along the pipeline. For example, a GeForce 3 GPU may operate with 800 pipeline levels, while the Intel P4 processor has at most 20. The pipeline is broken to four parallel channels at difficult parts, and ALUs along the pipes can handle four floating point values in parallel and execute complex operations such as the multiplication of a $4 \times 4$ matrix and a 4-dimensional vector.

On the other hand, the fixed pipeline has been developed further and has been turned to be partially programmable, thus the burnt in algorithms can be replaced by user specified ones. Two phases of the pipeline have become programmable, the vertex and pixel processing units (figure 1). Vertex processing, which was originally responsible for vertex transformation and local illumination computations at the vertices, can be controlled by a *vertex shader* program. Pixel processing, which originally computed texturing, can be governed by a *pixel shader* program. The programmability together with the high speed have made many researchers think of how special, non-local illumination and even non-graphics algorithms can be ported from the CPU to the GPU. Examples include ray-shooting [8], Voronoi diagrams, FFT and the solution of linear equations, etc. (see http://www.gpgpu.org). In [12] the method of solving Fredholm type integral equations of the second kind by the GPU is presented.

This paper discusses the implementation tricks of this method. The selected integral equation is the rendering equation describing the diffuse global illumination problem. Thus we use the graphics hardware, which was designed for local illumination rendering, to solve the global illumination problem.

When porting an algorithm onto the GPU we have to implement three programs, one for the CPU usually in C++, one for the vertex shader, and one for the pixel shader. Vertex and pixel shaders can be programmed in assembly, in Cg, the OpenGL shading language or in the high level shading language of DirectX. We used the Cg programming language [1].

### 1.1 Limitations and capabilities of the graphics hardware

The difficulty of the porting comes from the following restrictions. 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 pixel shaders that can read textures and can write only their target pixelś color (and/or depth). Only pixel shaders can access memory (textures), but their program size, instructions and the number of texture accesses are limited.

If the target pixel could only be in the frame buffer, then the limited number quantization levels (frame buffers usually have 8 bit precision per color channel) may pose problems.

Fortunately modern GPUs also come with a *pixel buffer* (or pbuffer), a virtual frame buffer that we can render images to. The pbuffer can store floating point values and it is also a possibility to bind the pixel buffer to a texture. It can be double buffered, and thus it can manage two images. The application of a pixel buffer in OpenGL is the

*ba351@ural2.hszk.bme.hu
†jg330@ural2.hszk.bme.hu

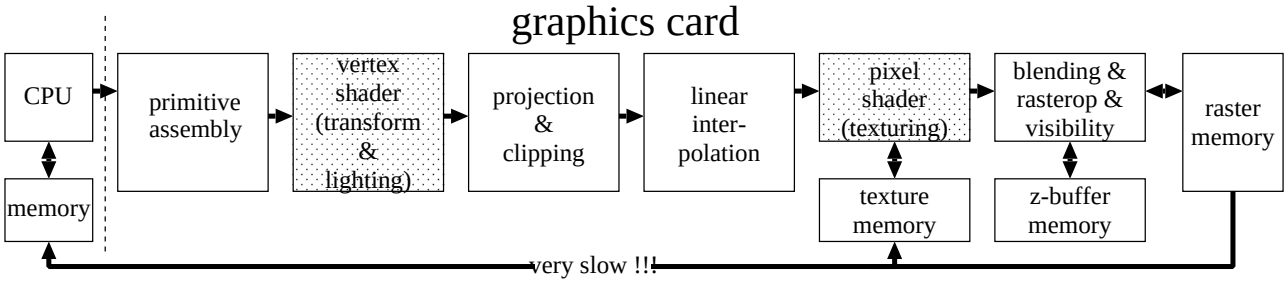


Figure 1: GPU architecture including vertex and pixel shaders

following.

1. Create a pixel buffer. Initialize it with size, pixel and texture parameters.

2. Change the rendering context from the frame buffer to the pixel buffer.

3. Render geometry.

4. Change the rendering context from the pixel buffer to the frame buffer.

When the content of the pbuffer is used later on, we can bind it to a texture. Nothing can be rendered to the pixel buffer until it is released. When the pbuffer is released, we can render into it again, and create dynamic texture effects.

The advantages of rendering into such a buffer are twofold. There is no need to upload the texture information to the CPU memory through the AGP bus, which would be slow compared to the GPUs video memory read, and unnecessary if we only need to render a texture. There is also the possibility of rendering 32 bit floating point precision images to this buffer. These textures are useful when we are rendering high dynamic range images and using high precision texture mapping.

Reading back any data to the CPU either from the frame buffer or from the pbuffer destroys pipeline efficiency, thus should be avoided. Meeting all these limitations is the real challenge of GPU programming.

# 2 Solution of the rendering equation

Global illumination algorithms aim at the solution of the rendering equation. For the sake of simplicity, let us assume that the surfaces are diffuse. In this case, the rendering equation

$$L(\vec{x}) = L^e(\vec{x}) + (\mathcal{T}_{f_r} L)(\vec{x})$$

expresses the radiance $L(\vec{x})$ of point $\vec{x}$ as a sum of the emission $L^e(\vec{x})$ and the reflection of all point radiances that are visible from here. The total reflection of the radiance of visible points is expressed by an integral operator

$$(\mathcal{T}_{f_r} L)(\vec{x}) = \int_S v(\vec{x},\vec{y}) \cdot L(\vec{y}) \cdot f_r(\vec{x}) \cdot \frac{\cos\theta'_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}\, d\vec{y}, \tag{1}$$

which is also called as the *light transport operator*. In this equation $S$ is the set of surface points, $v(\vec{x},\vec{y})$ is the mutual visibility indicator, which is 1 if points $\vec{x}$ and $\vec{y}$ are visible from each other and zero otherwise, $f_r$ is the BRDF and $\theta'_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the surface normals and the direction between $\vec{x}$ and $\vec{y}$.

The solution of the rendering equation requires general purpose instructions and is thus usually computed on the CPU. However, this is rather slow, and the requirements of interactive rendering cannot be met. 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 transform the algorithm according to the capabilities of the GPU.

The CPU-based solution algorithms can be classified as random walk [6] and iteration techniques. The GPU support of random walk algorithms has been examined in [8]. 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:

$$L_m = L^e + \mathcal{T}_{f_r} L_{m-1}.$$

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

$$L(\vec{x}) = \lim_{m\to\infty} L_m(\vec{x}).$$

Iteration simultaneously computes interaction between all surface elements, which is hard to implement in the GPU. The first attempt to port radiosity algorithm applied therefore Southwell iteration, which reduced the number of iterations [5]. This algorithm inherited the quadratic complexity of the original progressive radiosity and is quite complicated to implement.

We think that porting should also include the transformation of the original algorithm, to take into account the capabilities of the GPU. We propose that randomization offers a general strategy for such transformations. The formal basis of such approaches is the stochastic iteration, which was originally proposed for the solution of the linear equations, was presented in [7, 9, 4], then extended for the solution of integral equations [10, 11]. Stochastic iteration means that in the iteration scheme a random transport operator $\mathcal{T}^*_{f_r}$ is used instead of the light-transport operator $\mathcal{T}_{f_r}$. The random transport operator has to give back the light-transport operator in the expected case:

$$L_m = L^e + \mathcal{T}^*_{f_r} L_{m-1}, \qquad E[\mathcal{T}^*_{f_r} L] = \mathcal{T}_{f_r} L.$$

Note that such an iteration scheme does not converge, but the iterated values will fluctuate around the real solution. To make the sequence converge, all previous iterated values are averaged:

$$\tilde{L}_m = \frac{1}{m} \cdot \sum_{i=1}^{m} L_i = \frac{1}{m} \cdot (L^e + \mathcal{T}^*_{f_r} \tilde{L}_{m-1}) + \left(1 - \frac{1}{m}\right) \cdot \tilde{L}_{m-1}. \tag{2}$$

Iteration works with the complete radiance function, thus its temporary version should be stored somehow. CPU algorithms usually use finite-element methods based on the decomposition of surfaces to triangular patches. Since in stream processing patches are processed independently, we store the radiance function in textures similar to the method of [3].

Having introduced the basic concepts we discuss the appropriate selection of the random iteration scheme.

# 3 Perspective ray-bundle shooting on the GPU

Perspective ray-bundle shooting chooses a point randomly and sends the radiance of this point from here towards all directions [2, 12]. If point $\vec{y}$ is selected with probability density $p(\vec{y})$, then the random transport operator is

$$(\mathcal{T}^*_{f_r} L)(\vec{x}) = \frac{1}{p(\vec{y})} \cdot v(\vec{x}, \vec{y}) \cdot L(\vec{y}) \cdot f_r(\vec{x}) \cdot \frac{\cos\theta'_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x} - \vec{y}|^2}. \tag{3}$$

It is easy to prove that this random operator really gives back the real transform operator in the expected case. In order to realize this random transport operator on the radiance function stored in a texture, two tasks need to be solved, including the random selection of a texel identifying point $\vec{y}$, and the update of the radiance at those texels which correspond to points $\vec{x}$ visible from $\vec{y}$.

## 3.1 Random texel selection

According to importance sampling, it is worth setting the selection probability proportional to the integrand. Unfortunately, this is just approximately possible, and the point selection probability is set proportional to the radiance of the selected texel. If the light is transferred on several wavelengths simultaneously, the luminance of the radiated power should be used. Thus the selection probability of point $\vec{y}$ is:

$$p(\vec{y}) = \frac{\mathcal{L}(L_j)}{\Phi}, \qquad \Phi = \sum \mathcal{L}(L_i) \Delta S_i,$$

where texel $j$ corresponds to surface point $\vec{y}$, $\mathcal{L}$ is the luminance of a spectrum represented by red, green and blue components, $\Phi$ is the luminance of the integrated radiance, and $\Delta S_i$ is the area corresponding texel $i$. Note that if uniform parametrization is used then $\Delta S_i$ is similar to all texels.

In the current implementation we read back the radiosity map and apply a linear search to locate a random pixel. First the sum of the total luminance of all pixels is computed. This value is multiplied by a random value distributed in the unit interval. Then the luminance of the array elements is started to be summed, and the running sum is compared to the random luminance. When the running sum gets greater than the random value, the texel is found. It is easy to see that this selection scheme finds a texel proportionally to its luminance.

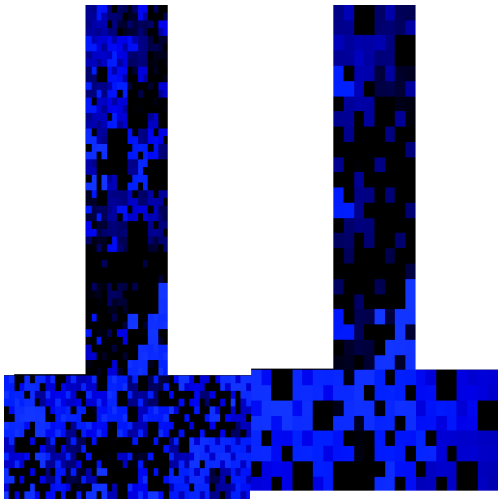

Figure 2: Visibility array showing the corresponding patches for each pixel. Patch index is encoded in texel color. (We are using floating point textures.)

When the texel is obtained, the surface point corresponding to it should be found. To support this operation we use a texture, called visibility array, that stores the patch index in each texel (figure 2).

When we implemented this scheme and made measurements we got shocking results. This selection algorithm took almost 10 times longer than all other elements of rendering, thus this is the bottleneck of the algorithm.

After that we replaced this method by a non-linear search with a GPU algorithm. The random selection is supported by the *mipmapping hardware*. When the texture

corresponding to the current radiance function is updated, the GPU is asked to compute the complete mipmap hierarchy. The mipmap can be imagined as a quad-tree, which allows the selection of a texel in $\log_2 R$ steps, where $R$ is the resolution of the texture. The top level of this hierarchy contains the average of all texels. The luminance of this value is multiplied by a random number uniformly distributed in the unit interval and also by four since on the next level this texel is decomposed to four texels. Then the next mipmap level is retrieved and the four texels corresponding to the upper level texel are obtained. The luminance values of the four pixels are summed, the running sum is compared to the value obtained on the higher level. When the running sum gets larger, the summing is stopped. A new selection value is obtained as the difference of the previous value and the luminance of all texels before the found texel. Then the same procedure is repeated recursively on the lower mipmap levels. This procedure terminates at a leaf texel with a probability that is proportional to its luminance.

Unfortunately the opengl does not support the automatic mipmap generation for floating point textures what we used for radiosity array so we have to generate mipmap levels with a fragment program in ten iterations, because we used radiosity array with a resolution $1024 \cdot 1024$. It is a bit slow. We use the following cg program to generate one mimap level:

```
float3 col=tex2D(radiosity_map,
float2(texcoord.x,texcoord.y-recres)).rgb;

float3 col1=tex2D(radiosity_map,
float2(texcoord.x-recres,texcoord.y-recres) ).rgb;

float3 col2 =tex2D(radiosity_map,
float2(texcoord.x-recres,texcoord.y) ).rgb;

float3 col3= tex2D(radiosity_map, texcoord).rgb;

return col + col1 + col2 + col3;
```

Where *recres* means the reciprocal of the current mipmap resolution. The value of recres is $1/1024$ in the first and after the tenth iteration it is $1$. The different mipmap levels are stored in different textures. We also made a fragment program which implements the random choice from the mipmap what we explained in the section above.The pixel shader program:

```
bool isRandom=0;

float2 retval; float2 retval2=float2(texcoord.x-
    recres,texcoord.y);
float2 retval3=float2(texcoord.x,texcoord.y
    -recres);

float2 retval4=float2(texcoord.x-recres,
texcoord.y-recres);

float3 col1=tex2D(texture,texcoord).rgb;

float3 col2=tex2D(texture,retval2).rgb;

float3 col3=tex2D(texture,retval3).rgb;

float3 col4=tex2D(texture,retval4).rgb;
float colorsum;

float colorsum1=col1.r+col1.g+col1.b;

float colorsum2=col2.r+col2.g+col2.b;

float colorsum3=col3.r+col3.g+col3.b;

float colorsum4=col4.r+col4.g+col4.b;
if(colorsum1>rand_times_last_max) {
    colorsum=colorsum1;
    retval=texcoord;
    isRandom=1;
} if(colorsum2>rand_times_last_max){
    colorsum=colorsum2;
    retval=retval2;
    isRandom=1;
} if(colorsum3>rand_times_last_max){
    colorsum=colorsum3;
    retval=retval3;
    isRandom=1;
} if(colorsum4>rand_times_last_max){
    colorsum=colorsum4;
    retval=retval4;
    isRandom=1;
} if(!isRandom){
    colorsum=max(colorsum1,colorsum2);
    colorsum=max(colorsum,colorsum3);
    colorsum=max(colorsum,colorsum4);
    if(colorsum==colorsum1) retval=texcoord;
    if(colorsum==colorsum2) retval=retval2;
    if(colorsum==colorsum3) retval=retval3;
    if(colorsum==colorsum4) retval=retval4;
} return float3(retval.x,retval.y,colorsum );
```

In the first iteration *randtimeslastmax* equals the full luminance of all pixels. This function chooses one texel for the next iteration, returns with the coordinates of this texel and the full luminance of this texel. These values will be passed for the next iteration as *texcoord.x*, *texcoord.y*, *randtimeslastmax*. This algorithm will stop after the tenth iteration.

## 3.2 Update of the radiosity texture

The points visible from $\vec{y}$ can be found by placing a hemisphere or a hemicube around $\vec{y}$, and then using the z-buffer algorithm to identify the visible patches. Both hemicube and hemisphere approaches have advantages and disadvantages. If hemicube is used, then we have to render the whole scene five times, while hemisphere requires only one rendering. However, the GPU works with triangles, but the hemispherical projection results in shapes with elliptical boundaries. The approximation of these elliptical segments is accurate only if the scene is highly tessellated. Thus we can conclude that the hemisphere requires only one rendering but needs higher tessellation. We used hemispherical projection in our implementation.

Note that hemispherical projection requires specialized vertex processing since the local illumination pipeline uses a rectangular and not a hemisphere window. 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. In the first pass the center and the base of the hemisphere are set to $\vec{y}$ and to the surface at $\vec{y}$, respectively, then the scene is rendered assuming of a point equal to the depth value of this point. Having computed the image, the result is stored in a *visibility texture*, called *vismap* (figure 3).

Figure 3: Depth image taken from the shooter point

To demonstrate how the hemispherical projection can be done with the GPU, the vertex program is shown in the followings, where *modelview* is the eye transform of the camera put at $\vec{y}$, *IN.position* is the current vertex, and *OUT.hpos* contains the normalized pixel coordinates and the depth value. Based on the front (*fp*) and back clipping distances (*bp*) the depth value is normalized to the [-1..1] interval for z-buffering and the *OUT.zdepth* normalized to the [0..1] interval to serve as the color value later in the pixel shader:

```
float3 dir  = mul(modelview, IN.pos).xyz;
OUT.hpos.xy = normalize(dir).xy;
OUT.hpos.z  = -(2*dir.z + bp + fp) / (bp - fp);
OUT.hpos.w  = 1;
OUT.zdepth  = -dir.z / bp; // [0,1]
```

In the second pass the same transformation is carried out. The BRDF and emission values are also passed, and now we render into the rectangle of the radiosity texture (figure 4).

The vertex shader is set according to this transformation and also prepares the value of the randomly transported radiance in variable according to equation 3:

$$Lref = Lshoot \cdot \frac{\cos\theta'_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x} - \vec{y}|^2},$$

$$Lshoot = \frac{L(\vec{y})}{p(\vec{y})} = \frac{L(\vec{y})}{\mathcal{L}(L(\vec{y}))} \cdot \Phi.$$

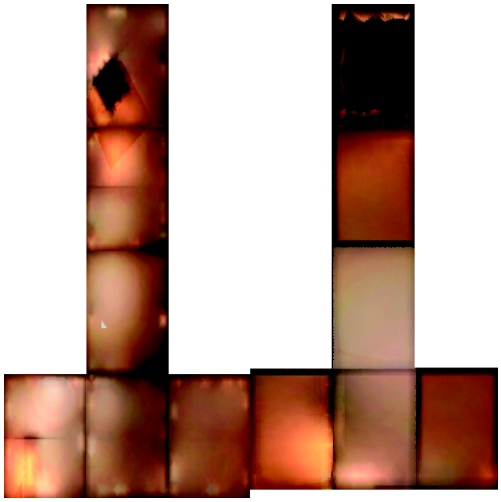

Figure 4: Radiosity map

The CPU determines *Lshoot* and passes it to the vertex shader, which computes *Lref* in the eye space where $\vec{y}$ is in the origo:

```
OUT.hpos.xy  = 2 * IN.texcoord.xy - float2(1,1);
OUT.hpos.z   = 0;
OUT.hpos.w   = 1;
OUT.texcoord = IN.texcoord;
float3 x     = mul(modelview, IN.pos).xyz;
OUT.zdepth   = -x.z / bp;
               // y is in the origo of eye space
float3 ytox  = normalize(x); // dir from y to x
float xydist2 = dot(x, x);  // |x - y|^2
float cthetay = -ytox.z; // normal(y) = axis -z
float3 tnorm  = mul(modelviewIT, IN.normal).xyz;
float cthetax = dot(tnorm, -ytox);
OUT.emission  = IN.emission;
OUT.brdf      = IN.brdf;
OUT.Lref      = Lshoot*cthetax*cthetay/xydist2;
OUT.viscoord.xy = (ytox.xy + float2(1,1)) / 2;
```

Note that the normal is transformed with the inverse transpose of the camera transform, because this way we can ignore any translation.

When a pixel is shaded, it is checked whether this pixel has the same depth as stored in the visibility map (figure 3) at the corresponding texel, that is, whether or not this pixel will be kept by the z-buffer [5]. The pixel shader code responsible for this check, for the multiplication with the visibility indicator and the BRDF (equation 3), for the update of the radiosity texture, and also for the averaging operation (equation 2), is the following:

```
float z = tex2D(vis_map, viscoord).x;
float visible = (abs(zdepth - z) < 0.05);
return tex2D(rad_map, texcoord).rgb * (1-1/m) +
            (emission + visible * Lref*brdf)/m;
```

The program gets the visibility texture coordinates (*viscoord*) and the patch index, *emission*, *BRDF* and the texture coordinate (*texcoord*) of the target point, as well as the prepared transported radiance *Lref* from the vertex shader. Iteration number *m* is passed directly from the CPU as a uniform parameter.

### 3.3 Presenting the results

The final results can be seen from an arbitrary viewpoint if the vertex and pixel shaders are set to the normal operation (figure 5). The vertex shader computes the modelview-projective transformation of the vertices, while the pixel shader finds the texel corresponding to the computed texture coordinates.

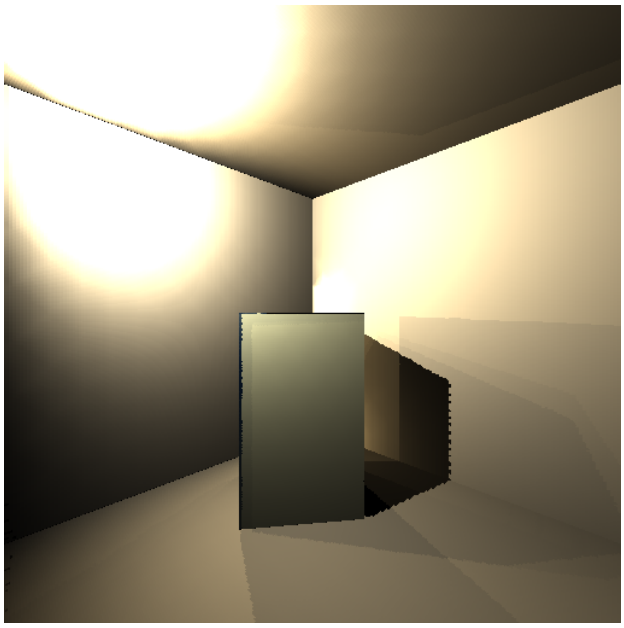

Figure 5: The rendered scene after 50 iterations

## 4 Implementation results and conclusions


Figure 7: Running time of the proposed algorithm (pbuffer size is $512 \cdot 512$)

The proposed method has been implemented on an ATI Radeon 9800 Pro graphics card in C++/OpenGL/Cg environment. In the current implementation the random selection is made by the CPU and all other operations by the GPU. The implementation has been tested with the Cornell box scene and we concluded that a single iteration requires 120-150 msec for ten to ten thousand vertices and for $512 \cdot 512$ resolution radiosity maps. This is comparable to the CPU implementation. Having looked more deeply into the time requirements of different phases, we concluded that this embarrassing time is due to the reading back of the pbuffer. If we can use automatic mipmap generation and for floating point textures our algorithm will be much more faster. We can speed up our algorithm if we read the pbuffer in the first few and also in every tenth iteration, but the huge time of the search remains. According to our measurements, the running time can be reduced to its tenth, i.e. an iteration cycle would need only 20-30 msec. This would be 4-5 times faster than the optimized CPU implementation using only the classical graphics pipeline for visibility checking.

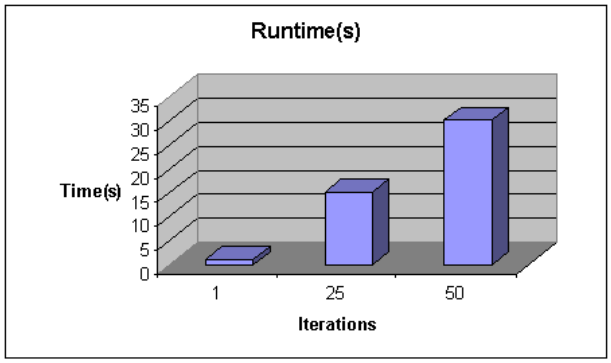


Figure 8: Running time of the proposed algorithm (pbuffer size is $1024 \cdot 1024$)

The drawback of this algorithm comes from the texture based radiosity representation, which can result in dot artifacts. This problem will be attacked by interpolation similarly to [3].

## 5 Summary

A method is presented to perform global illumination calculations on modern graphics hardware. This should reduce the time to perform radiosity precomputations. Pixel buffer readback from the graphics hardware is identified as the major bottleneck of the proposed approach.

## 6 Acknowledgements

This work has been supported by the National Scientific Research Fund (OTKA ref. No.: T042735). The scenes have been modelled by Maya that was generously donated by AliasWavefront.

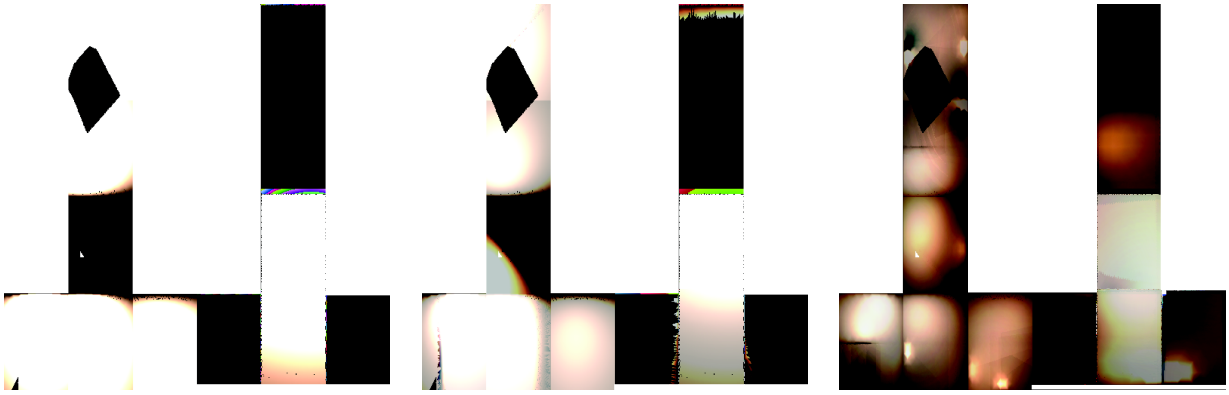

Figure 6: The radiosity map after 1, 4 and 50 iterations

## References


[1] http://developer.nvidia.com/page/cg_main.html.

[2] Gy. Antal, L. Szirmay-Kalos, F. Csonka, and Cs. Kelemen. Multiple strategy stochastic iteration for architectural walkthroughs. *Computers & Graphics*, 27:285–292, 2003.

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

[4] Ph. Bekaert. *Hierarchical and stochastic algorithms for radiosity*. PhD thesis, University of Leuven, 1999.

[5] G. Coombe, M. J. Harris, and A. Lastra. Radiosity on graphics hardware. Technical report, Univ. of North Carolina, UNC TR03-020, 2003.

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

[7] L. Neumann. Monte Carlo radiosity. *Computing*, 55:23–42, 1995.

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

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

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

[11] L. Szirmay-Kalos. *Photorealistic Image Synthesis Using Ray-Bundles*. D.Sc. Dissertation, Hungarian Academy of Sciences, 2000. www.iit.bme.hu/~szirmay/diss.html.

[12] L. Szirmay-Kalos, G. Szijártó. *Stochastic Radiosity on the Graphics Hardware*. KEPAF Conference. Miskolctapolca. 2004.