Stream programming Framework for Global Ilumination Techniques Using a GPU

Federico J. Marino
Facultad de Ingeniería, Universidad de Buenos Aires
Av. Paseo Colón 850 (C1063ACV), Capital Federal, República Argentina
fedemarino@gmail.com

and

Horacio Abbate
Facultad de Ingeniería, Universidad de Buenos Aires
Av. Paseo Colón 850 (C1063ACV), Capital Federal, República Argentina
habbate@fi.uba.ar

Abstract

Stream processors are becoming an affordable alternative to implement hardware assisted rendering techniques which were usually relegated to offline usage. We built a stream processing framework based on the Stream Programming Model concepts, selected the Photon Mapping algorithm and an NVIDIA GPU (Graphics Processing Unit) as a test case implementation of a Global Illumination technique. We defined a set of C++ classes to encapsulate the components (kernels and streams) of this new paradigm, using OpenGL and Cg language. Our application combines the Photon Splatting method and the BVH (Bounding Volumes Hierarchy) acceleration structure into a rendering pipeline relying almost entirely on the GPU. Finally, we evaluated its performance using a Cornell Box model.

Keywords: Photon Mapping, GPU, Stream Programming, Photon Splatting, Cg, OpenGL, BVH, Real Time Rendering, Global Illumination.

Resumen

Los procesadores de streams están comenzando a ser una alternativa accesible para implementar técnicas de rendering asistidas por hardware que habitualmente estaban relegadas al uso offline. Nosotros elaboramos un marco de trabajo para procesamiento de streams basado en los conceptos del modelo de Stream Programming, seleccionamos el algoritmo de Photon Mapping y una GPU (Graphics Processing Unit) Nvidia para una implementación de un caso de prueba. Definimos un conjunto de clases en C++ para encapsular los componentes (kernels y streams) de este nuevo paradigma, usando OpenGL y el lenguaje Cg. Nuestra aplicación combina el método de Photon Mapping y una estructura de aceleración BVH (Bounding Volumes Hierarchy) en un pipeline de renderizado basado casi completamente en la GPU. Finalmente, evaluamos su desempeño usando un modelo de caja de Cornell.

Palabras claves: Photon Mapping, GPU, Stream Programming, Photon Splatting, Cg, OpenGL, BVH, Rendering en Tiempo Real, Iluminación Global.

1. BACKGROUND

Modern programmable graphics processors (GPUs) integrate multiple processing pipelines working in parallel, with a high-speed dedicated memory (texture memory) that can achieve a combined performance of many gigaflops. Their processing power grows at a higher rate that traditional CPUs, motivating a change of paradigm in the programming techniques used for compute-intensive and real-time applications.
GPUs consist of vertex and fragment processing units capable of handling high precision arithmetic. Vertex processors handle the transformations and projections of geometry vertices of the 3D model which are then provided to the rasterizer, which converts vector data (triangles) into bitmap data (fragments). The fragment processors do the shading of the triangles, applying an illumination model specified by a fragment program (written in a C-like language, called Cg [11]). They read data stored in textures and usually produce pixel colors as results.
Global illumination algorithms aim to simulate a physically correct illumination of a 3d scene through a numerical approximation to the solution of the rendering equation [1]. Some years ago, Photon Mapping appeared as a unified technique able to simulate most of the effects of light-surface interactions in an efficient and elegant way, covering a wider range of situations than previous approaches such as, Radiosity or Path-tracing. Figure 1 shows a sample scene with and without global illumination.
In order to adapt an algorithm like Photon Mapping to this type of hardware, we have to use the approach known as GPGPU (general purpose computing on GPU). The idea is to consider the inputs and outputs of the fragment processor as general data instead of image-specific. Data structures have to be mapped into textures and code supported by fragment or vertex programs written Cg. Although the programmability on recent GPU generations has improved, flow control and branching instructions have performance penalties. The lack of scatter operations (random access writes) is another limiting factor, especially for general algorithms like sort.

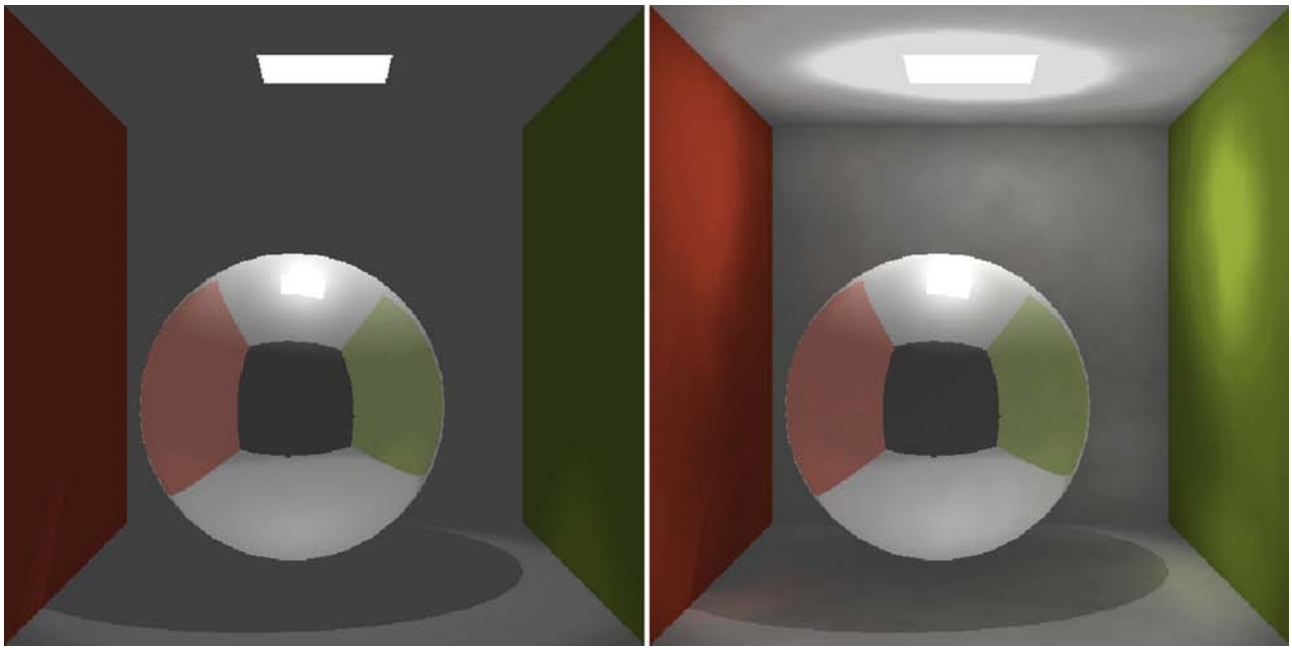

Figure 1: Sample Scene, without global illumination (left), with global illumination (right)

Photon mapping was developed by Henrik Wann Jensen [2] and can be classified as a density (of particles per surface area unit) estimation two-phase algorithm. The term photon has its origin in the physical term, but in this case it only represents a particle carrying a discrete amount of luminance energy on a specific direction. Other term used is radiance, which defines the radiant energy flux per area unit in a certain direction. In addition, the term irradiance refers to the integral of the radiance over a hemisphere (radiant flux emitted per unit area in all directions).
The first phase of Photon Mapping consists on distributing light particles (photons) from the source in random directions, through the 3d scene, until the maximum number of interactions is reached. On each photon-surface interaction, the Russian roulette technique is used to decide probabilistically whether photons should be reflected, refracted or absorbed.
The traced photons are stored in a data structure called photon map, including data such as hit coordinates on the surface, incident direction and photon's energy.
The second phase uses a standard ray tracing algorithm to trace primary rays from the viewer's eye into the scene through the pixels of the view plane (one ray per pixel). For each pixel, the amount of indirect illumination is calculated using the radiance estimate, which gathers the nearest neighbor photons of the surface hit point and sums its contribution to the radiance, applying the respective BRDF (Bidirectional Reflectance Distribution Function) [12] of the surface material.
One remarkable feature of Photon Mapping is that the illumination representation (photon map) is decoupled from geometry, so that illumination detail is not tied to 3D geometry level of detail.
The results of the method are biased, meaning that because they are based on a number of pre-computed particles, they will never converge to the real value of the integral it is attempting to solve; even though, the method is consistent.
There are several optimization strategies, such as irradiance caching proposed by Ward [3], who introduced the idea of re-using irradiance values in certain points of lambertian surfaces via interpolation.

2. ILLUMINATION TECHNIQUES ON GPUS

Thrane and Simonsen [7] proposed the BVH acceleration structure to speedup the ray tracing process. The idea is to organize the geometry in a binary tree forming a bounding volumes hierarchy. Therefore, many ray-triangle tests can be avoided by first testing the intersection with the ray and the box bounding a group of triangles. This structure has proved to be more efficient and easier to implement on a GPU compared to KD-trees or Uniform Grids.
Lavignotte and Paulin [6] presented a new image based method for computing an efficient global illumination solution using graphics hardware. To calculate the irradiance they propose the Photon Splatting technique, which distributes each photon's contribution over the pixels instead of gathering radiance from the neighborhood (which requires some complex search structure difficult to implement on a GPU). At the moment they presented their work, hardware limitations (8-bits per channel precision buffers) required complex workarounds which now have been surpassed.
Moelhave [4] implemented a ray tracing engine on the GPU, using a uniform grid acceleration structure. It was based on Pixel buffers extension, OpenGL and C++. That implementation was only capable of rendering direct illumination of diffuse surfaces without reflections, refractions or shadows support.

3. STREAM PROGRAMMING FRAMEWORK

Our approach is based on the concepts exposed by Purcell [5], the Stream Programming Model, the MRT (Multiple render targets) feature and the FBO (Frame Buffer Object) OpenGL extension. The MRT feature allowed us to output up to four 32-bit floats from a fragment program, reducing the

need of multiple runs of the same kernel to evaluate each output separately. The FBO extension is a replacement for Pixel Buffers (PBuffers) that allows an efficient and flexible use of the render-to-texture mode in OpenGL.

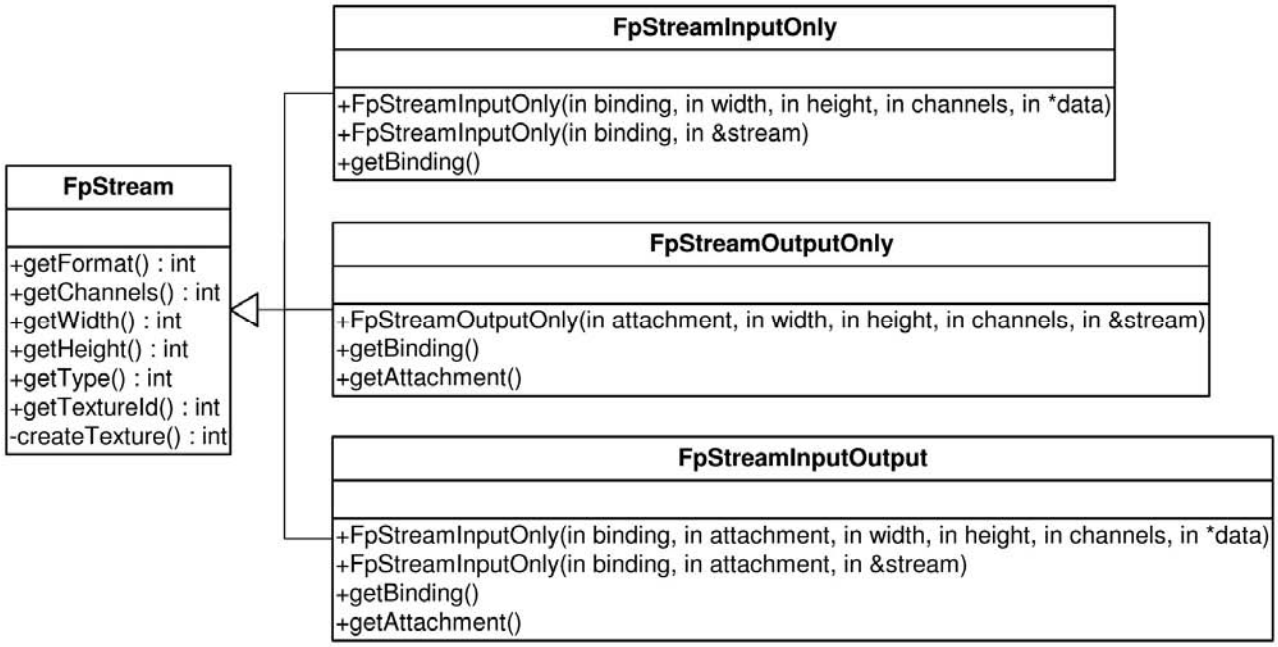

Figure 2. The FpStream class

One of the components of the Stream Programming Model is the kernel, which is basically a function call that performs a considerable amount of computations on a set of records known as streams. A kernel takes streams as inputs and produces streams as outputs.
We defined a set of C++ classes (FpStream and FpKernel) to encapsulate the render-to-texture mechanisms used to compute the results. The FpStream class (see figure 2) holds a 2D texture with up to 4 channels (RGBA) of 32 bits floating point precision, where application data structures fit in.

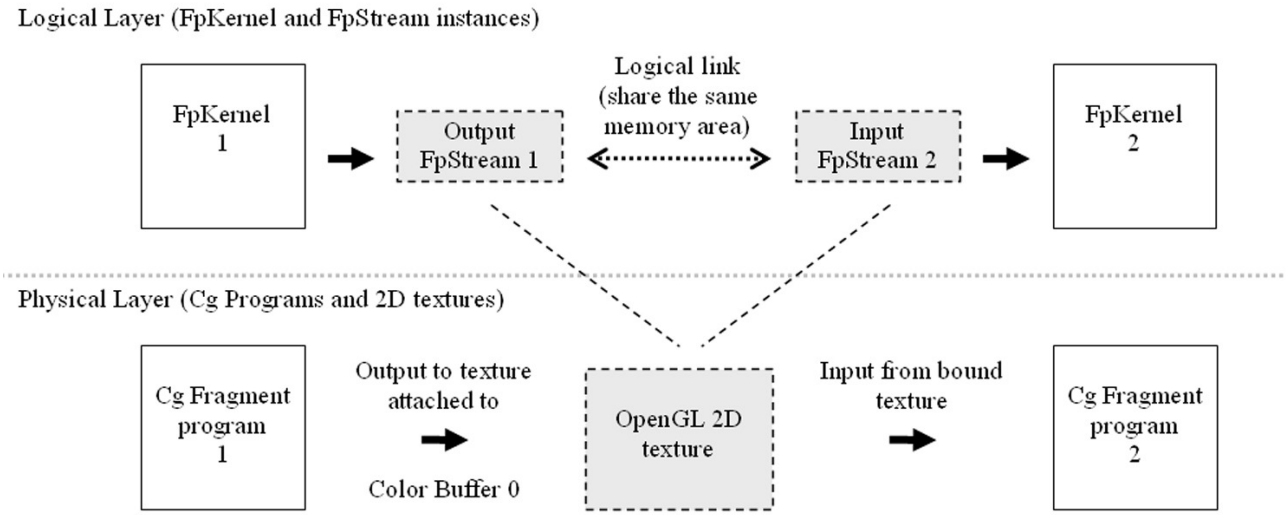

Figure 3. Framework layers.

Three child subclasses represent the modes in which the streams can be linked to a kernel: output, input or input-output modes. The constructor method of an input stream can receive an output stream as a parameter, providing a way to link one kernel output to other kernel's input or vice versa. The link is done logically, meaning that the actual data is represented only by one texture id, shared by those two streams that are used for writing and reading respectively (see figure 3).
It is also possible to have multiple input streams sharing the same physical texture, in cases where the same data serves as input for multiple kernels, in different contexts.
The FpKernel class (see figure 3) is associated to Cg fragment program. This class can represent both single-pass and multi-pass kernels. The first step is to bind the uniform parameters (global static parameters for a Cg program) to it. The second step is to bind the streams providing a "binding" name for the inputs which is the name used by the stream within the CG program, and specifying an "attachment" number for the outputs (this number corresponds to the output color buffer of the Cg program, which should be between 0 and 3).
Finally, the computation is carried out by the "run" (see figure 4 left) method which binds a FBO (Frame Buffer Object) instance, sets an orthographic projection frustum and then draws a rectangle (GL_QUAD) parallel to the view plane which covers it from the lower-left to the upper-right corner (see figure 4 right). The rasterizer converts the rectangle geometry into fragments, on which the Cg program runs and then the outputs are stored into the off-screen buffer (output streams).
Since GPUs process stream elements (fragments) independently, it's impossible to share the fragment processing results among them. This is due to the fact that fragment processors work in parallel running the same program over different parts of the stream in no particular order.

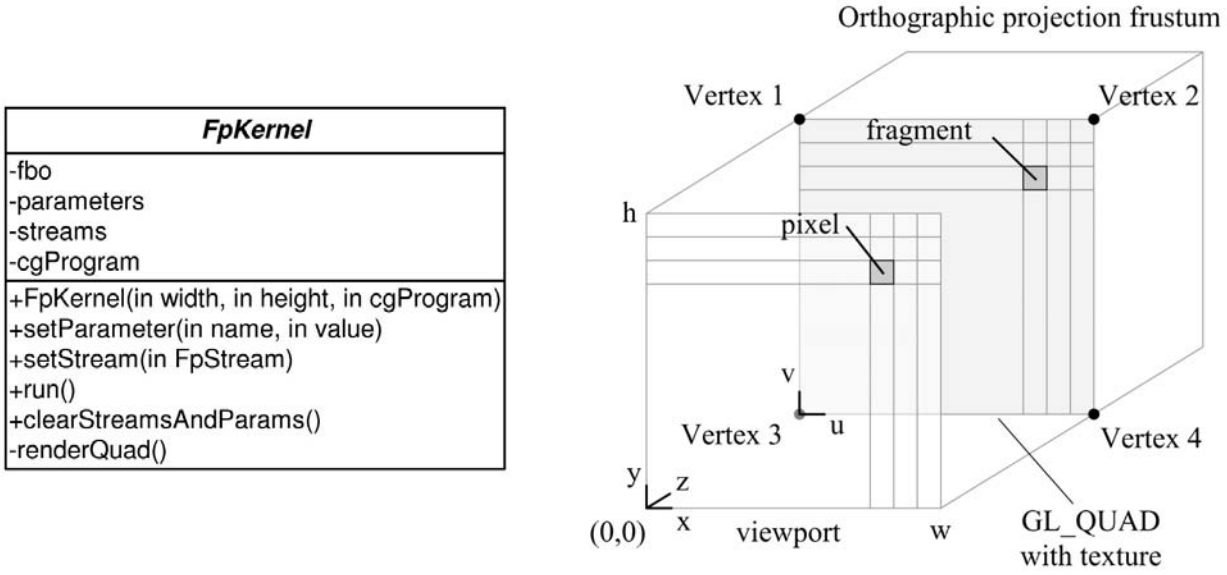

Figure 4. FpKernel class (left), Kernel computation (right).

When running multi-pass algorithms (which require many iterations of the same kernel), some elements of the streams could reach the end state of the process earlier than others. But current GPUs do not allow you to selectively disable the execution of the kernel on certain stream elements. Instead, you can only prevent the output value for a certain fragment to be written to the buffer, by using the discard instruction. To detect the condition when all stream elements have reached the end state, the OCCLUSION_QUERY extension is used. It returns the number of fragments that were updated in the output buffer or active in a certain pass. A loop over the kernel "run" method is done until this number reaches zero.

4. THE PHOTON MAPPING IMPLEMENTATION

Surface materials are modeled by the sum of two types of BRDF functions, ideal diffuse and specular reflections.
The light source is modeled as a directional spotlight, defined by an origin, direction, minimum and maximum angles which define two cones between which the intensity varies from maximum light power to total darkness.
In the photons tracing phase, the total power of the light is evenly spread on the generated photons. The color of the photons is filtered by the surface color they hit (color bleeding effect). Photons can be absorbed, diffusely reflected or specularly reflected when they hit a surface according to specularK or diffuseK coefficients. In figure 5 (left), photon A is absorbed after 2 reflections, photon B bounces into space and gets discarded, finally photon C is absorbed after 1 specular reflection.
During the rendering phase (see figure 5), direct illumination component is calculated using the Phong model [8] including the calculation of a shadow factor by tracing one shadow ray from the primary ray hit point to the light source. If the shadow ray hits a surface before light source, the shadow factor is zero.Otherwise, it is set to 1.0. To compute the indirect illumination, we estimate the irradiance at a surface point by splatting the photons (see point 4.2) and compute mirror-like reflections by tracing a secondary ray and adding its color according to the reflectivity coefficient of the material. The irradiance value is independent of the outgoing direction we are analyzing, so it is stored in a scalar variable which accumulates photons contributions.

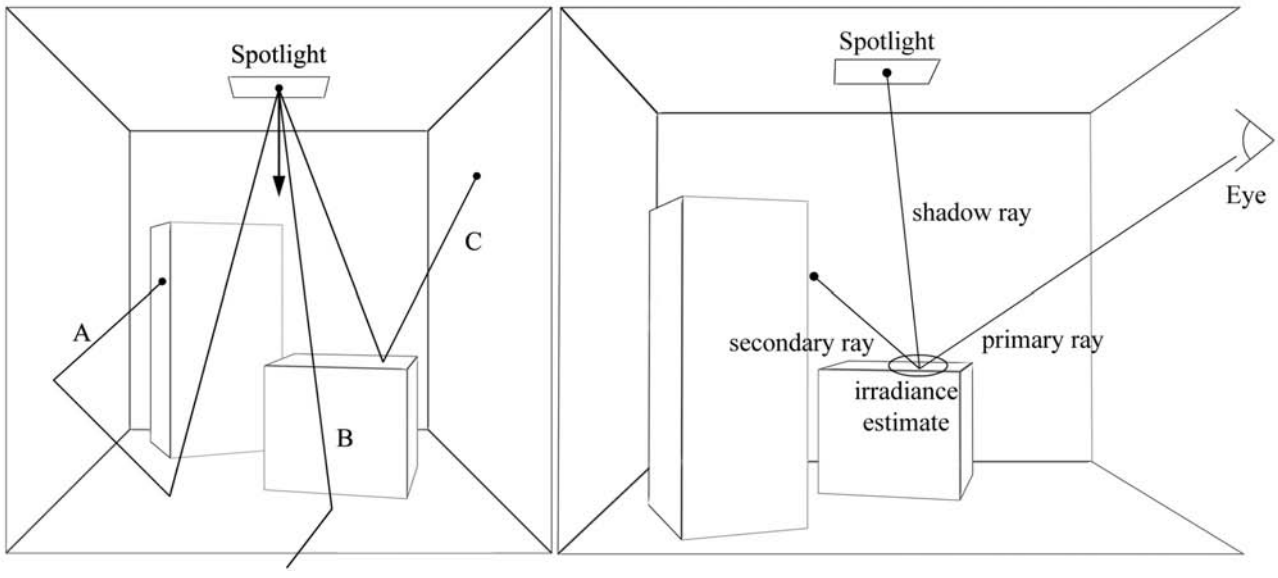

Figure 5. Photon tracing (left) & Rendering phase (right).

4.1. Photons and Rays Tracer

The process of tracing rays or photons is almost similar. The differences lay on the results of the interaction with the surface they hit, so basically, the algorithm and the acceleration structure can be shared for both purposes.
When using BVH, the geometry is organized in a binary tree, by recursively subdividing the space in two parts along one axis of the three axes (X, Y and Z), until each individual triangle is enclosed in the bounding box.

Actually, the tracing process does not require the tree itself, but an array specifying a sequence of nodes to traverse it, starting at the root node. Following that sequence a ray-triangle intersection should be found faster than using a uniform grid depending on the distribution of the triangles in the space.
The construction of the tree is done entirely on the CPU as a pre-process. In the case of a static scene (geometry does not change over time) no tree reconstruction is required.

4.2. The Splatting Process

This technique was presented by Stürzlinger and Bastos [9]. A splat has the shape of a disc and represents the photon contribution to indirect illumination in the surroundings of the hit point on a surface. For each photon stored, a splat should be placed centered on the hit point and oriented in the 3D space. After tracing the photons on the GPU, the output streams containing photons data are transferred and converted to arrays in the CPU memory, because geometry (triangles to represent the splats) can't be generated as an output from a fragment program. A CPU process scans the array of photons and draws a triangle holding the splat into the framebuffer using alpha blending mode, so that all the photon contributions are accumulated.
During the shading process of the triangle where the splat is supported, each fragment is tested, analyzing whether it belongs to the same surface that the photon has hit. If not, its contribution is discarded. In figure 6, it is observed that the photon contributes to the illumination of fragment A but not to the fragment B. The decision is based on comparisons between the normal vectors and the surface identifiers.

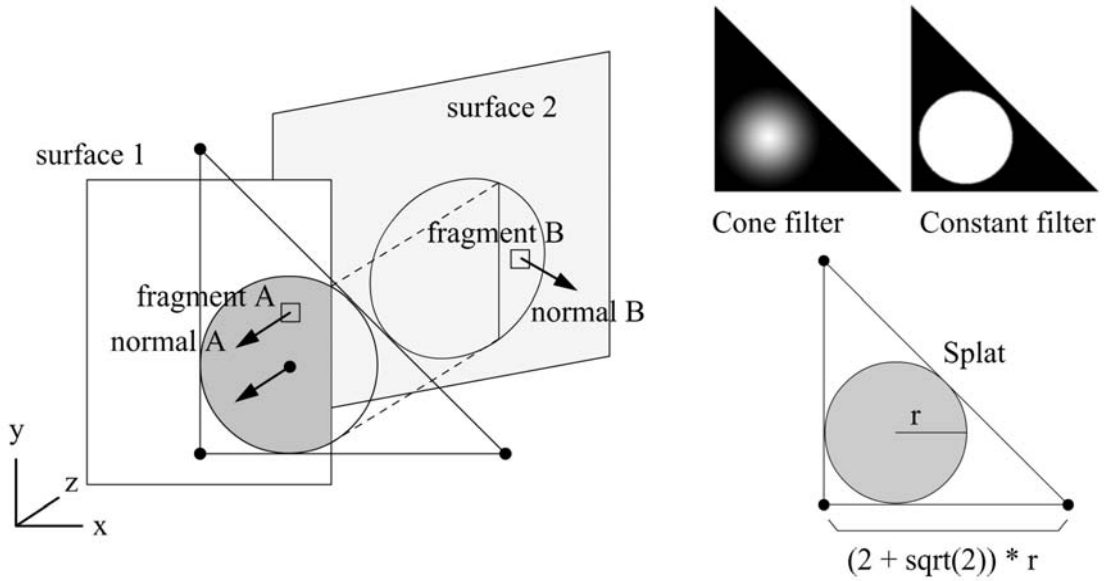

Figure 6. Splatting process (left), splat shape and filter types (right).

The energy provided by each photon to each fragment is weighted by a function (known as filter) that depends on the distance to the center of the splat (see figure 6). Typically, a constant filter, a cone filter (intensity decays linearly until the radius is reached) or gaussian filter is used.
The contribution from all the splats is accumulated in the framebuffer. When a large number of splats are added, a smooth indirect illumination image is obtained (figure 7 shows the splatting process). The resulting texture is the called irradiance image and serves as input of the second phase of the algorithm (the rendering phase).

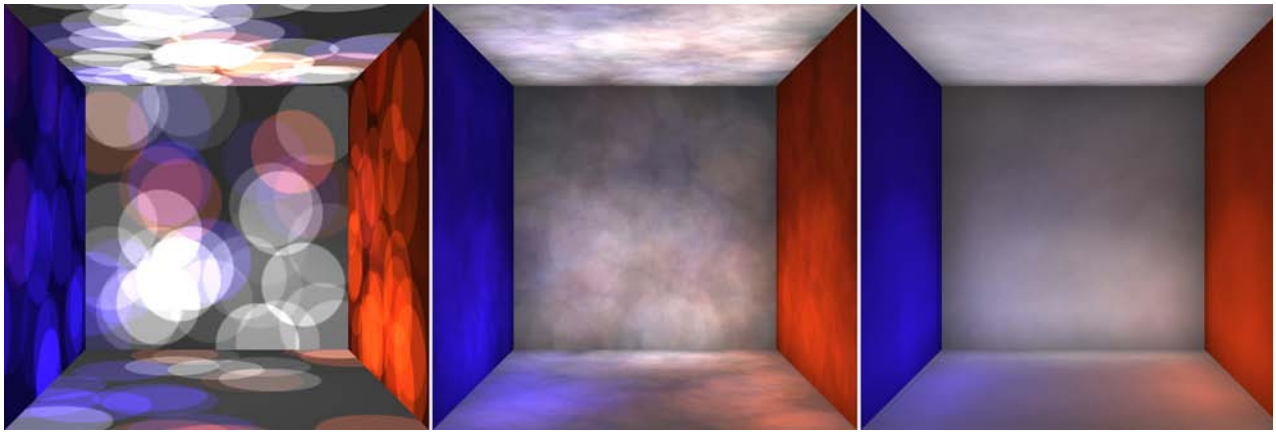

Figure 7. Splatting process with 256 photons (left), 4K photons (center), 64K photons (right)

5. THE PHOTON MAPPING ENGINE

A set of classes (SceneLoader, Tracer, PhotonMapper and RayTracer) represents the chain of processes of the algorithm, where the inputs are the scene geometry, illumination and view parameters, and the output is the synthesized image as a bitmap file (see figure 8). The SceneLoader class provides the geometry and builds the BVH tree, in the form of streams, from a 3DS format file (3DS Studio) and a plain text file that includes global parameters, surface materials, view frustum and light properties.

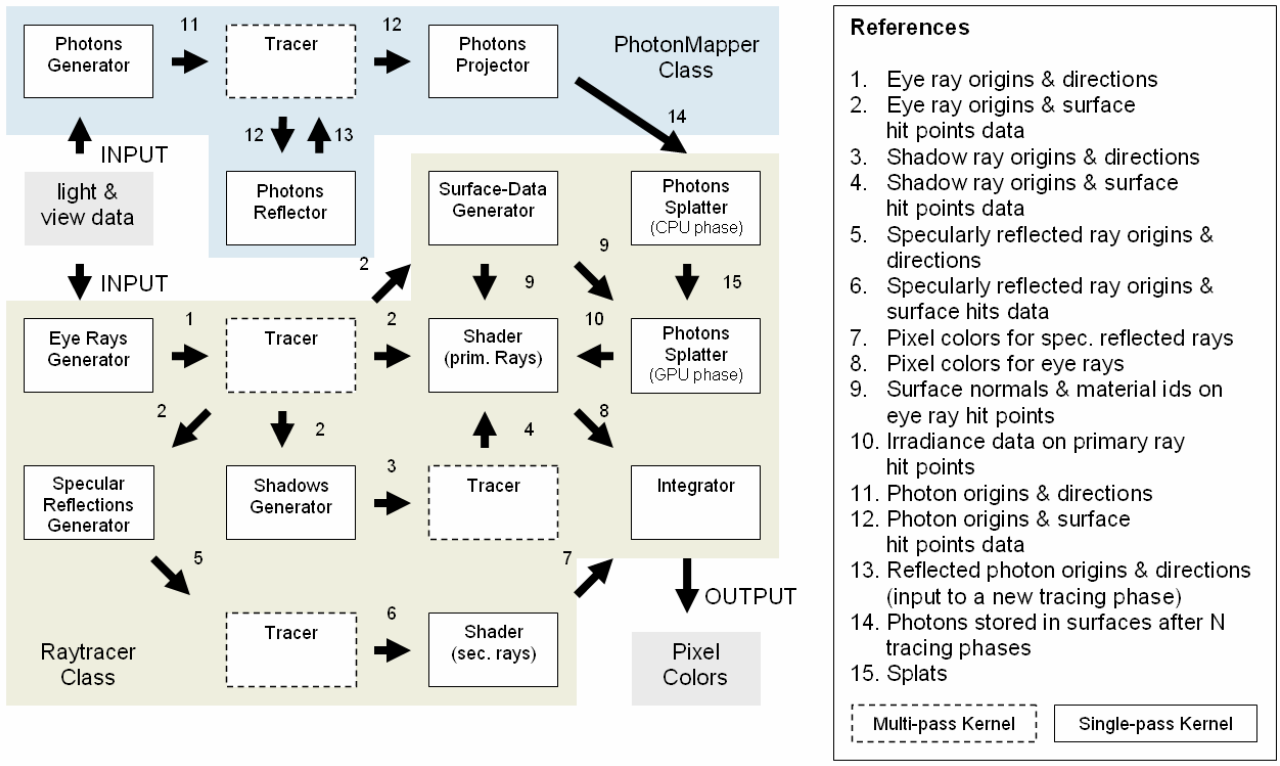

Figure 8. Classes layout.

The Tracer class is responsible for tracing both eye rays and photons through the scene using BVH acceleration structure. It receives the origin and direction vectors as inputs, and outputs the coordinates of the triangle hit points.
The PhotonMapper class uses the services of the tracer class. A kernel named Generator creates the photons, sampling the spotlight emission space (using random polar coordinates). Photons are traced up to a maximum number of bounces. After each tracing step, the Reflector kernel decides whether each photon should be reflected or absorbed based on the respective coefficients of the surface material. If a photon is reflected, its new instance is fed again to the Tracer. Finally, the absorbed photons are projected (Projector Kernel) in world space coordinates and transferred to the CPU memory (Raytracer instance).
The Raytracer class handles the splatting process (to generate the irradiance image) and rendering phase using the services of the Tracer class. It is responsible for combining all the illumination components (ambient, direct, indirect, mirror-like reflections and shadows) to deliver the final image.
Figure 9 shows sample images rendered with our application using 262K photons. The lighting on the walls exhibits low frequency noise due to the still insufficient number of photons.

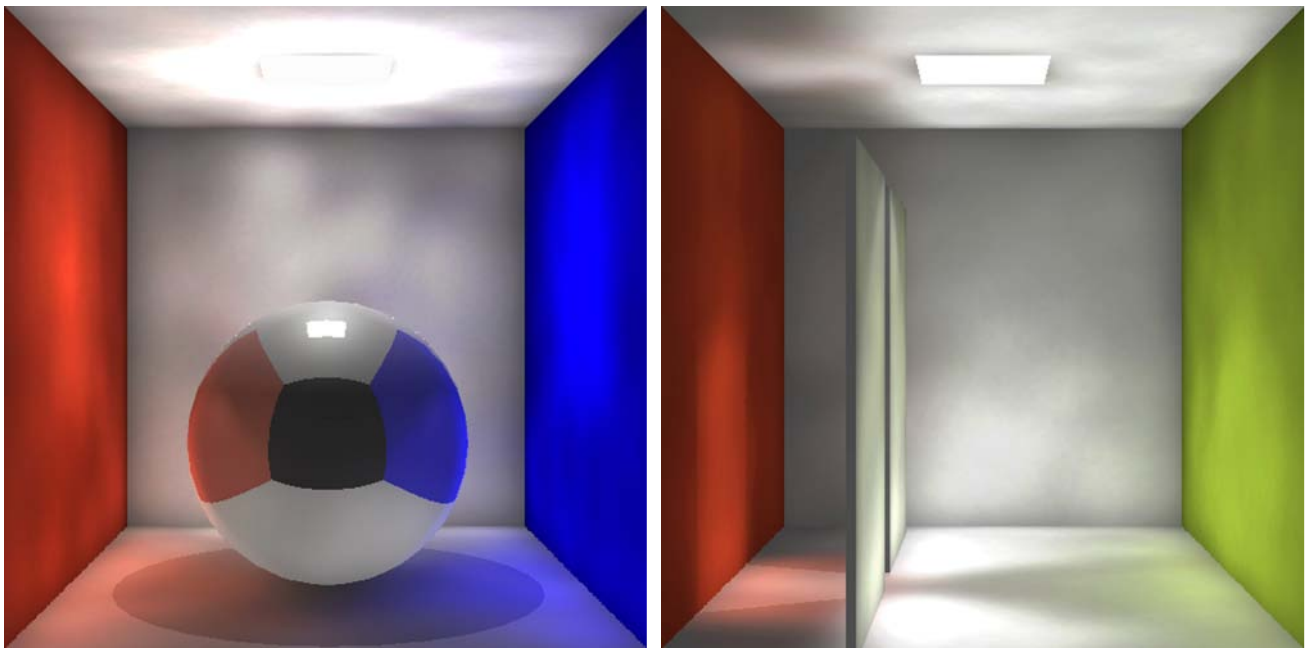

Figure 9. Sample scenes.

6. PERFORMANCE TESTS

A Cornell Box model was used to evaluate our implementation (see figure 10). All the surface materials are non-reflective (no mirror-like reflections) and have a diffuse reflection coefficient of 0.2 and a specular coefficient of 0.1, so, if 100 photons intersect a certain surface , 70% will be absorbed by it, 20 % will be diffusely reflected and the remaining 10% will be specularly reflected. Photons were traced up to a maximum of 3 bounces (4 tracing phases).
It's important to note that not all generated photons produce a splat, because some of them are reflected outside the scene and others are lost after being reflected past the photons bounces limit. Empirically, on this scene, the number of generated splats is between 70% and 75% of the generated photons. The number of splats varies, even on consecutive runs with the same input parameters because it depends on randomly generated numbers.

Figure 10. Scene used in performance tests (left), other sample scene (right).

The hardware and software platform used was Microsoft Windows XP SP2, Visual C++ 7.0, OpenGL 2.0, NVIDIA Cg 1.5 toolkit, an NVIDIA GeForce 6600 GTS 128MB AGP GPU, an AMD Athlon 64 3000+ CPU with 1GB DDR RAM and NVIDIA driver version 91.45. We used the fp40 and vp40 Cg profiles.
The following 27 parameters combinations were evaluated using the test scene:

- Image resolution: 128x128, 256x256 and 512x512 pixels.
- Splat radius: 0.5, 1.0 and 1.5 units (as a reference, the back wall is 10 units wide).
- Generated photons: 16384, 65535 and 262144 (approximately. 75% are splatted).

Globally, it is observed that the implementation is scalable as the rendering time increases sub linearly vs image resolution, splat radius and number of photons (see table 1). It is expected that with future GPU generations, the increased number of processing units will produce a direct performance boost, as more streams will be processed in parallel per clock cycle.
Table 2, shows a second set of tests, using a splat radius of 1.25 units and 512x512 image resolution. The results show that the splatting and tracing phases have an important share in the total computational cost specially when using 262144 generated photons where the quality of the solution seems acceptable (figure 10 left corresponds to that case).

Table 1. Rendering Time

Generated photons	**Total rendering time (seconds) according to image resolution (pixels)**								
	Splat radius = 0.5			**Splat radius = 1.0**			**Splat radius = 1.5**		
	128x 128	**256x 256**	**512x 512**	**128x 128**	**256x 256**	**512x 512**	**128x 128**	**256x 256**	**512x 512**
16384	4,87	5,09	6,17	5.47	5.11	6.78	4.86	5.19	6.65
65535	5,87	6,56	7,37	5.89	6.25	8.22	6.01	6.53	9.48
262144	10,67	11,43	12,33	10.69	11.,22	15.89	10.71	12.34	20.43

Table 2. Splatting and tracing phase incidence.

Generated Photons	Time (seconds)		
	Total	Photons Tracing	Photons Splatting
1024	6.81	2.78	0.21
4096	6.89	2.81	0.27
16384	7.42	2.98	0.62
65535	9.42	3.71	1.67
262144	16.92	7.26	5.51

7. CONCLUSIONS AND FUTURE WORK

As seen in the tests results, the splatting phase consumes much of the rendering time (table 2), so it would be advisable to concentrate future efforts on optimizing it. One way to do it is to study the benefits of using rectangles (GL_QUAD) or points (GL_POINT) instead of triangles (GL_TRIANGLES) for representing the splats, thus, less time would be wasted processing fragments that are outside the splat disc.

Another interesting issue would be to investigate the use of the "geometry instancing" feature, currently only supported under Direct3D. That feature allows us to generate geometry, by clonning and transforming geometry that is already stored in GPU memory (in our case, the splats), instead of creating them one by one by reading the photon's list.

In addition, it may be useful to analyze the possibility of running the process of photon tracing and ray tracing simultaneously using two GPUs connected in the same PC using a SLI (Scalable Link Interface) configuration, as well as taking advantage of the idle periods of the CPU while the GPU is working on relatively long processes, perhaps via multiple threads.

The next step towards building interactive Photon Mapping engine with real time feedback would be to study the way to reuse calculations between consecutive frames. For example, if lighting conditions do not change, photons do not need to be retraced. When lighting conditions change partially, only some of the photons could be retraced using the strategy called Selective Photon Tracing [10].

Finally, new emerging technology such as NVIDIA CUDA (Compute Unified Device Architecture) seems to be a more elegant and powerful base platform to build a stream processing application like the one we have studied. This new architecture has an API specifically designed for general-purpose computation, where code can be written directly in C language. It also allows the use of scatter instructions (that limitation in our implementation forced us to run part of the splatting process on the CPU, generating costly memory transfers through the AGP bus).

REFERENCES

[1] J. T. Kajiya, "The Rendering Equation", Proceedings of ACM SIGGRAPH 86, 1986, pages 143-150.

[2] H. W. Jensen, "Realistic Image Synthesis using Photon Mapping", A K Peters, 2001. ISBN 1568811470.

[3] G. J. Ward, F.M Rubinstein and R. D. Clear, "A ray tracing solution for diffuse interreflection", Computer Graphics (Proc. SIGGRAPH '88) 22(4): 85-92 (August 1988).

[4] T. Moelhave, "Implementing a Ray Tracer on a GPU", http://moelhave.dk/gpu-ray-tracer/, Jan. 2005.

[5] T. J. Purcell, "Ray Tracing on a Stream Processor", PhD thesis, Stanford University, 2004.

[6] F. Lavignotte, and M. Paulin, “Scalable Photon Splatting for Global illumination”, Graphite 2003 (International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia), Melbourne, Australia, ACM SIGGRAPH, pp. 1-11.
[7] N. Thrane, L. O. Simonsen, “A Comparison of Acceleration Structures for GPU Assisted Ray Tracing”, Master’s thesis, University of Aarhus, 2005.
[8] Phong, B.T.: Illumination for computer generated pictures. Comunnications of ACM 18, 6, ISSN 0001-0782, (Junio 1975),pp 311 – 317.
[9] W. Stürzlinger, R Bastos, “Interactive Rendering of Globally Illuminated Glossy Scenes”, Eurographics Rendering Workshop ‘97, June 1997, pp. 93-102, ISBN 3-211-83001-4.
[10] K. Dmitriev, S. Brabec, K. Myszkowski, H-P. Seidel, ”Interactive global illumination us-ing selective photon tracing”. In Thirteenth Eurographics Workshop on Rendering (2002), P. Debevec and S. Gibson, Eds.
[11] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, “Cg: a system for programming graphics hardware in a C-like language”. ACM Transactions on Graphics 22, 3.pp 896–907.
[12] F. E. Nicodemous, J. C. Richmond, J. J. Hsia, L. W. Ginsberg and T. Limperis, “Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), Oct. 1977.