Applied Mathematical Sciences Vol. 8, 2014, no. 135, 6783 - 6790
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ams.2014.49694

A Bidirectional Path Tracing Method for Global Illumination Rendering on GPU

Simone Celestino[1], Giuliano Laccetti[1], Marco Lapegna[1], Diego Romano[2]

[1] Department of Mathematics and Applications, University of Naples Federico II,
Via Cintia Monte S. Angelo, Naples, 80126, Italy
[2] ICAR-CNR, Via P. Castellino 111, Naples, 80131, Italy

Copyright © 2014 Simone Celestino, Giuliano Laccetti, Marco Lapegna and Diego Romano. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

This article proposes a new method for synthetic images rendering with Global Illumination on Graphics Processing Units (GPU). The method is based on Bidirectional Path Tracing in which we truncated path lenghts to improve GPU performance and introduced a number of Virtual Light Points to organize the Light Tracing pass. Results shows that a rendered image is accurate if confronted with the corresponding reference image, and it is sythetized at a reasonably interactive rate when the algorithm is implemented on a commodity GPU.

Mathematics Subject Classification: 68U05, 68W10

Keywords: computer graphics, global illumination, gpu

1 Introduction

Computer Graphics (CG) is about the creation of synthetic images, and one of the biggest challenges in this field is the creation of images representing *scenes* that should appear photorealistic to humans [8]. We call *rendering* the process that transforms a mathematical model representing a $3D$ scene into a $2D$ image. A photorealistic rendering simulates the transport of light and its interaction with the human eye. There are two main models for photorealistic rendering: *local illumination* and *global illumination* [3]. Local illumination

only considers the direct interaction with the lights (direct illumination), while global illumination also includes rays coming from other surfaces (indirect illumination), in order to reproduce optical effects such as shadows, refractions and reflections. *Graphic Processing Units* (GPUs) are devices designed to calculate real-time local illumination renderings taking advantage of the highly parallel nature of the problem. In a modern GPU the redering steps are executed by grouped programmable multiprocessors with several ALUs each, in which data is processed parallely during pipelined stages. Thanks to a growing availability of scientific computing programming tools on GPUs for the so-called *GPU Computing*, we decided to explore a different approach for rendering $3D$ scenes on GPU using global illumination. We focused on a method called *bidirectional path tracing*(BPT) [7] which seemed to be the most promising. It combines the ideas of two other methods for global illumination, namely *light tracing*(LT) and *path tracing*(PT) [3], considering both the energy getting out from the lights and the visibility of the scene from the observer. These methods are based on Monte Carlo integration [2]. An example of the latest advances in Monte Carlo rendering is presented by Delbracio et al. [1].
The idea behind our algorithm for BPT is to speed up the convergence of the classical PT algorithm. This is achieved by considering the light contribution from some "special" points which actively diffuse the light energy which themselves received from other sources. We call these *Virtual Light Points* (VLPs) recalling the idea presented in [9] where a *bidirectional instant radiosity* method is presented. The benefit of introducing such VLPs in the algorithm concerns the organization of data structures in such a way that an efficient multi threading code can be implemented on a GPU. In section 2 we will give a general overview of the model used for the global illumination problem. In section 3 we will describe the peculiarities of our parallel algorithm and in section 4 we will present some interesting results. We will close this paper with some conclusions in section 5.

2 Model

The rendering problem consists in measuring the light energy that each pixel of the final image should emit in direction of the viewer. Such quantity in photometry is represented by the *radiant flux*, which measures the radiance passing through an area A, and it can be calculated integrating the radiance emitted by all points that belong to the scene S over all the possible directions:

$$\Phi(A) = \int_S \int_\Omega L(x \to \Theta) \cos(N_x, \Theta) d\omega_\Theta dS_x \tag{1}$$

where $L(x \to \Theta)$ is the radiance emitted by x in direction Θ, Ω is the set of the incident directions and N_x is the normal to the surface in point x. For

reasons of concision, in our light model the energy striking a surface in a point is transmitted with energy leaving from the same point, therefore omitting scattering surfaces. The function which describes this behavior for a surface is called BRDF (Bidirectional Reflectance Distribution Function)[4] [3]. A BRDF can be defined as the ratio of the differential radiance reflected in a given direction Θ and the differential irradiance incident in a given direction Ψ.

$$f_r(x, \Psi \to \Theta) = \frac{dL(x \to \Theta)}{dE(x \leftarrow \Psi)} \tag{2}$$

A compact form of the rendering equation is:

$$L(x \to \Theta) = L_e(x \to \Theta) + L_r(x \to \Theta) \tag{3}$$

where $L(x \to \Theta)$ is the exitance radiance from x in direction Θ, $L_e(x \to \Theta)$ is the emitted radiance and $L_r(x \to \Theta)$ is the reflected radiance. Considering that:

$$dE(x \leftarrow \Phi) = L(x \leftarrow \Psi)\cos(N_x, \Psi)d\omega_\Psi \tag{4}$$

from definition (2), we have the extended formulation of the rendering equation:

$$L(x \to \Theta) = L_e(x \to \Theta) + \int_\Omega f_r(x, \Psi \to \Theta)L(x \leftarrow \Psi)\cos(N_x, \Psi)d\omega_\Psi \tag{5}$$

3 Method

Several methods in literature do not solve the rendering equation in a complete way but they implement approximations which produce a decent and reliable output (see Ray Tracing [12] or Radiosity [5] algorithms). But some limitations arise: e.g. Ray Tracing can simulate only specular surfaces because rays are reflected geometrically with a basic BRDF; Radiosity can simulate only diffusive surfaces because it implements only diffusive BRDFs. Our idea is to use a stochastic method in order to simulate every possible BRDF in the same scene , [10]. Bidirectional Path Tracing combines two different approaches into a single method. It generates random paths which simulate both light propagation starting from the light source, and light absorption in the image plane of the energy arriving from the scene in direction of the observer. This means that we need to calculate in two passes respectively a path for the emitted radiance and another path for the radiant flux in the pixels. Usually the two passes are independent and can be executed in arbitrary order. After the generation of both paths, the final points of the two are connected to create a longer random path which is constrained to start from a light source and to end into a pixel. By following the energy propagation along this bidirected path, and by repeating the procedure in a Monte Carlo integration fashion, the flux (1) can be estimated in each pixel. Since we want to exploit the SIMD nature of GPUs, in our formulation of the method we decided to prearrange

the order of calculation for the two passes of the BPT, by running LT first and PT later. Moreover we introduced the idea of Virtual Light Point: each hit point on a diffuse surface [5] during the LT pass is marked as a light source. In this way we effectively simulate the light propagation in the scene, and at the same time we delegate part of the light emission to the illuminated surfaces where it is modulated through their respective BRDFs. Therefore we can estimate the (5) in the VLPs with:

$$\langle L(x \to \Theta) \rangle = L_e(x \to \Theta) + \frac{1}{N} \sum_{i=1}^{N} \frac{L(y_i \leftarrow \Psi_i) f_r(y_i, \Psi_i \to \Theta_i) \cos(N_{y_i}, \Psi_i)}{p(\Psi_i)} \quad (6)$$

where x is a VLP, Θ is the direction in which the light is transmitted, N is the lenght of the random path in the LT, y_i are the hit points in the random path, Θ_i,Ψ_i are respectively the outcoming and incoming directions in y_i, and $p(\Psi_i)$ is a probability density function (PDF) on the domain of the possible incoming directions. In the second pass of the BPT algorithm, during the PT, for every step of a random path, the total radiance arriving at each hit point is approximated averaging the energy arriving from all the sources and the VLPs. That is:

$$\begin{aligned} \langle L(y \to \Theta) \rangle = & \frac{1}{M_s} \sum_{i=1}^{M_s} L_e(x_i \to \Psi_i) f_r(y, \Psi_i \to \Theta) \cos(N_y, \Psi_i) + \\ & + \frac{1}{M_v} \sum_{j=1}^{M_v} L(x_j \to \Psi_j) f_r(y, \Psi_j \to \Theta) \cos(N_y, \Psi_j) \end{aligned} \quad (7)$$

where y is a hit point of the random walk in the PT, Θ is the direction in which the light is transmitted, M_s is the number of light sources, M_v is the number of VLPs, x_i are points on light sources, Ψ_i are ray directions from x_i to y, x_j are VLPs and Ψ_j are ray directions from x_j to y. Hence the BPT algorithm in two passes can be summarized as follows:

- We generate a random light path from the light source. This path, $x_0, x_1, \ldots, x_l$, has a random length l controlled by russian roulette that determines if continue or not the path, through the evaluation of the PDF.

- Similarly, we trace a random path that starts from the midpoint of a pixel, and hits a point y_0 on a surface visible by the observer through the pixel. The path has length k (also controlled by russian roulette) and is composed by a set of points $y_0, y_1, \ldots, y_k$. At each step of this path, the radiance transmitted by y_i is calculated by applying the (7). At the same time an approximation of the flux (1) is calculated by estimating the radiance transmitted to y_0 along the path.

Listing 1: Pseudo-code of parallel BPT

```
1  //building the light paths in parallel
2  RadianceLightTracing<<numThreads>>(Lights)
3
4  //building the eye paths in parallel
5  RadiancePathTracing<<numThreads>>(Pixels)
```

This procedure should be reapeted undefinitely, tracing new random paths each time and averaging the calculated flux. This is necessary to reduce the error for the flux in each pixel, and therefore to improve the image quality. In listing 1, a schematic of the algorithm is presented. In the parallel GPU algorithm for both functions (listings 2 and 3), we assigned to each thread an independent path. In order to keep the workload balanced among the computing units, we decided to fix the lenght of the paths *a priori*, removing the russian roulette. To compensate the bias introduced truncating this lenght, we experimentally evaluated the visual quality of the results versus the execution time on the GPU of PC3 (Table 1). For this purpose we measured the Structural Similarity (SSIM) [11] of the rendered results against a reference image (obtained after 10 minutes of execution) within a critical time of 5 seconds, which is a present-day reasonable time frame for a real-time global illumination rendering. We observed that for LT best visual results are obtained when lenght is 1, that is when we consider only VLPs on directly illuminated surfaces. Longer paths did not improve visual quality, rather they worsened it. This can be explained because of time consuming calculations which produce contributions not visually relevant. For PT we noticed that an optimal value depends on the objects in the scene and the complexity of their BRDFs. For this reason we left it as a parameter of the scene which should be given in input.

Listing 2: Pseudocode of Parallel LT

```
1  RadianceLightTracing(Lights){
2   //start a ray from a Light
3   hit_point=NextRay(Lights)
4   for lenght {
5     if (BRDF(hit_point) == Diffuse){
6       radiance=ComputeRadiance(hit_point)
7       //save a VLP
8       SavePoint(hit_point,radiance)
9       hit_point=NextRay(hit_point)
10    }
11    else{ //e.g. glass or specular
12      hit_point=ComputeReflRefr(hit_point);
13    }
14  }
15 }
```

Listing 3: Pseudocode of Parallel PT

```
1  RadiancePathTracing(Pixels){
2     //start a ray from a pixel
3     (x,y)=extract(threadId);
4     hit_point=NextRay(Pixel(x,y));
5     for length{
6            if (BRDF(hit_point) == Diffuse){
7                   SampleLights(lights,VLPs);
8                   EvaluateFlux(Pixel(x,y));
9                //compute next ray
10                  hit_point=NextRay(Pixel(x,y));
11       }
12       else{ //glass or specular
13          hit_point=ComputeReflRefr(hit_point);
14       }
15    }
16 }
```

4 Results

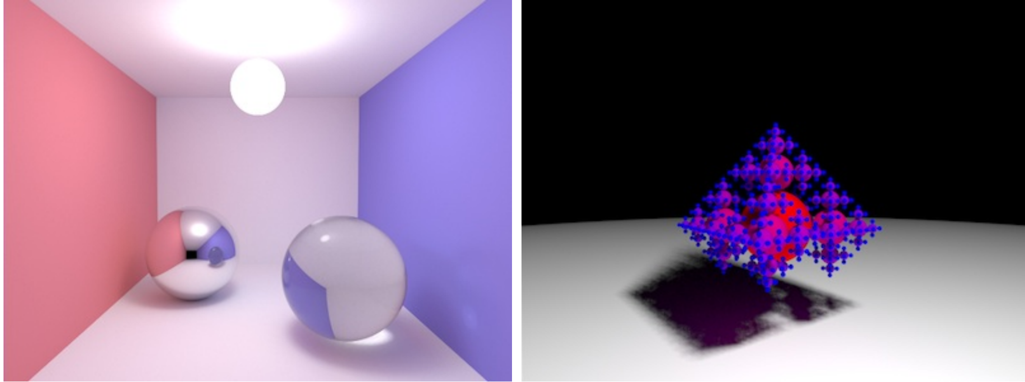

Figure 1: Reference Scene for simple test   Figure 2: Reference Scene for complex test

The architectures used for the tests are in Table 1. The development en-

	PC 1	PC 2	PC 3
CPU	Intel i5-650 @ @3,2GHz	Intel i5-650 @3,20GHz	Intel i7-3517U @1.9GHz
GPU:	Quadro FX 3800	GeForce 560 Ti	GeForce GT 780
Vendor	NVIDIA	NVIDIA	NVIDIA
Architecture	G80	Fermi	Kepler
Cuda Core	192	384	2304
SFU	48	32	96
Processor Clock	1,2Ghz	1,7 Ghz	1,06 Ghz
Memory Interface	256-bit	256-bit	384-bit
Amount of Memory	1023 MB	1023 MB	3071 MB

Table 1: CPU and GPU specs

vironment consists in Ubuntu 12.04 LTS Linux provided with Cuda 6.0 SDK (Cuda compilation tools V6.0.1) and Nvidia display drivers 331.62. This includes the latest OpenGL library.

We implemented two test scenes. The first is the classic Cornell box (Fig. 1). The second is a 3D representation of a *Sierpinski's triangle* modeled with 783 spheres of different dimensions (Fig. 2). We tested the quality of the rendered image by means of the SSIM when executed on PC3. After 1 second the rendered image has a SSIM of 0.93 for the complex scene (a value of 1.0 means a rendered image identical to the reference image), and 0.90 for the simple scene. This is due to the diffusive nature of the Cornell Box which requires more samples to visually represent the correct lighting, while the Sierpinski's triangle is immersed in an empty space and therefore the direct lighting is more relevant. After 5 secs both scenes were rendered with a SSIM of 0.97, while we measured a SSIM of 0.998 after 60 secs for the simple scene and 120 secs for the complex one. We confronted the performances on different architectures in

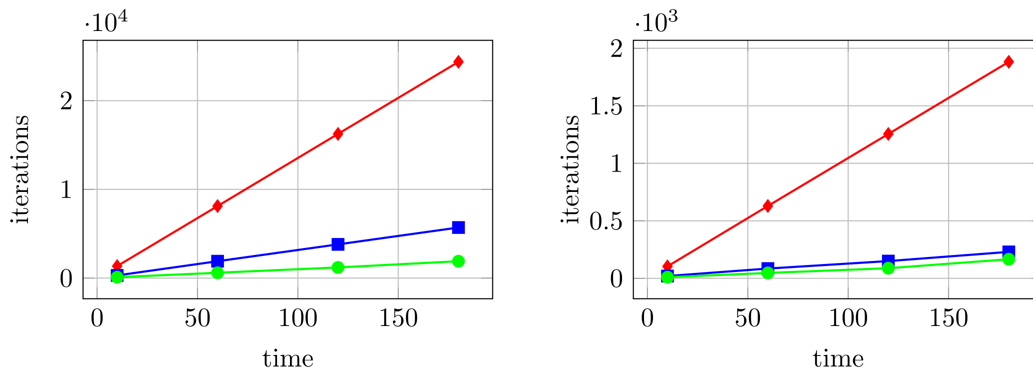

Figure 3: At left the result obtained with the simple scene, at right the result with the complex scene. Green line is for PC1, blue line is for PC2, red line is for PC3 as referred in Table 1.

the two charts in Fig. 3. The x-axis is for the execution time of the software, while the y-axis is for the number of iterations executed. The chart on the left - the one for the Cornell box - shows that after 3 minutes PC2 executed about 5700 iterations, while PC3 executed 24400 iterations. This means that the resulting image for the latter is almost 5 times more sampled than on PC2 in the same time. This difference clearly depends on the greater number of CUDA cores despite the slower clock rate of the GPU. About the *Sierpinski's triangle*, in our tests we noticed that performances for different architectures are proportional to the number of *Special Function Units* (SFUs) in the GPU. This behavior is due to the higher number of spheres in the scene, and therefore to the higher number of ray intersections to be tested. Since each intersectiont test is implemented using trigonometric functions, and those calculation on our GPUs are executed by SFUs, in this scope we were not able to take advantage of the total number of CUDA cores.

5 Conclusions and future work

Despite GPU devices are designed for local illumination rendering, both at professional and gaming level, we took advantage of the CUDA environment to produce photo-realistic images using a BPT method for global illumination. Results are encouraging both in terms of image quality and time to solution, especially on more recent hardware. To improve perfomances, in future we need to manage ray intersections bypassing the SFUs. We should also try to reorganize the parallel algorithm to avoid divergences among the threads on the GPU. Finally, in order to in order to harness more computing power in a system - as presented in [6] - we will spend an effort to design an hybrid multicore CPU/GPU quadrature algorithm.

References

[1] Mauricio Delbracio, Pablo Musé, Antoni Buades, Julien Chauvier, Nicholas Phelps, and Jean-Michel Morel. Boosting monte carlo rendering by ray histogram fusion. *ACM Trans. Graph.*, pages 8:1–8:15, 2014.

[2] Arnaud Doucet, Adam M Johansen, and Vladislav B Tadić. On solving integral equations using Markov chain Monte Carlo methods. *Applied Mathematics and Computation*, 216:2869–2880, 2010.

[3] P Dutre, K Bala, P Bekaert, and P Shirley. *Advanced global illumination*. 2003.

[4] AS Glassner. *Principles of digital image synthesis*. 1994.

[5] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer*, 18:213–222, 1984.

[6] Giuliano Laccetti, Marco Lapegna, Valeria Mele, and Diego Romano. A study on adaptive algorithms for numerical quadrature on heterogeneous gpu and multicore based systems. In *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, pages 704–713. Springer Berlin Heidelberg, 2014.

[7] Eric Lafortune. Mathematical models and Monte Carlo algorithms for physically based rendering. 1996.

[8] Eihachiro Nakamae and Katsumi Tadamura. Photorealism in computer graphics: Past and present. *Computers Graphics*, 19(1):119 – 130, 1995.

[9] Benjamin Segovia, Jean Claude Iehl, Richard Mitanchey, and Bernard Péroche. Bidirectional instant radiosity. In *Rendering Techniques*, pages 389–397, 2006.

[10] L Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum*, 18:C242–C244, 1999.

[11] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[12] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, 1980.

Received: May 1, 2014