CUDA PathTracer

Raimond-Hendrik Tunnel
Institute of Computer Science,
University of Tartu
Tartu

Abstract*—*This paper describes my CUDA implementation of a path tracing based rendering.

Keywords—parallel computation; CUDA; high performance computing; rendering; Cornell Box

I. INTRODUCTION (*HEADING 1*)

Realistic rendering has always been a goal in computer graphics. Rendering via the *standard graphics pipeline* usually will not include realistic effects like indirect illumination and ambient occlusion. Instead additional methods are considered, when a realistic effect is needed. For example *screen space ambient occlusion* or *environment mapping* can be done.

Ray tracing based rendering can achieve those basic realistic effects like indirect illumiantion and ambient occlusion by default. Unfortunately, ray tracing is quite expensive compared to the standard graphics pipeline.

II. PATH TRACING

Path tracing is a global illumination technique that uses ray tracing as its core. If we were to model the realistic scenario, then in order to calculate the reflectance of light for a given point, we would need to shoot a ray in every possible direction.

Kajiya proposed in his article *The Rendering Equation* a Monte-Carlo like probabilistic approach to solving that problem. For each point, the user sees, we send a ray in a random direction in the hemisphere defined by the surface normal. If we average together many such random samples, we will get a close enough approximation of reality.

III. CUDA

CUDA is a *General Purpose Graphics Processing Unit* (GPGPU) language for NVIDIA GPU-s. It consists of *host*, *device*, and *global* functions. Host functions are those that are ran on the CPU side that initializes the computation. *Device* and *global* functions are ran on the GPU, first being the parallel kernels and latter are, well, global functions that can be called from the *device* kernels.

NVIDIA GPU memory architecture consist of a global memory, shared memory and local memory. Computational architecture consists of blocks of *arithmetic-logic units* (ALU). Each ALU has their own local memory and ALU-s in the same block share the shared memory of that block. Global memory is, of course, shared among the entire card, and accessing it is slower than accessing the shared or local memory.

Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors).*

IV. IMPLEMENTATION

I implemented the path tracing algorithm in CUDA and C++, code is available from GitHub [2]. Initially I thought that it would be reasonable to use a CUDA, C++, OpenGL combination. Unfortunately, because I used the EENet cluster for the computation, using OpenGL as is was not an option. I researched a bit about VirtualGL, that is supposed to be for such purposes and should allow a frame image to be rendered and sent via the X11 protocol over SSH directly to me, so that I could see it with XMing. While discussing this idea with EENet cluster support, they did not seem to be able to provide that capability.

When implementing a parallel path tracing it is important to consider, what do we parallelize. The naïve approach is to parallelize individual pixels. Li explains that a better idea would be to parallelize by rays, because pixels that do not have a part of the scene visible, will waste GPU resources. [3]

I created a queue of all rays. The initial kernel will populate that queue with rays for each pixel and each iteration. This queue is in the global memory and takes a lot of space. With 512×512 frame and 300 iterations, a limit of 3.5 GB was reached. This is because the queue consists of tasks, each 48 bytes in size.

$$512 \cdot 512 \cdot 300 \cdot 48 = 3.8 \text{ GB} \quad (1)$$

The Tesla K20m GPU on the EENet cluster has 4 GB of global memory. This problem could be avoided if we were to resolve the rays in batches. For example, if we generate 3 GB worth of rays and then wait until they have been resolved (queue becomes empty), after that we continue with the ray generation.

After creating the queue of all rays, I use dynamic parallelism available since CUDA 3.5 to start new threads that will resolve the rays. This means that there will generally be more threads than there are pixels on the frame. CUDA 3.5 will allow 2^{31}-1 blocks in one grid dimension and maximum of 1024 threads in one block. This is a much larger number than the usual size of a frame.

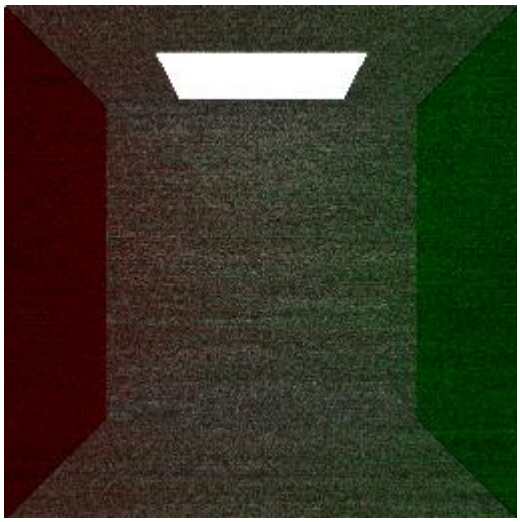
Each resolving ray will compute the intersection of the ray with the geometry using the Möller-Trumbore intersection algorithm [4]. It will find the Lambertian diffuse reflection value, multiplies it with the current value stored in the task. After that we generate a new random direction from the hemisphere defined by the surface normal, assign a ray

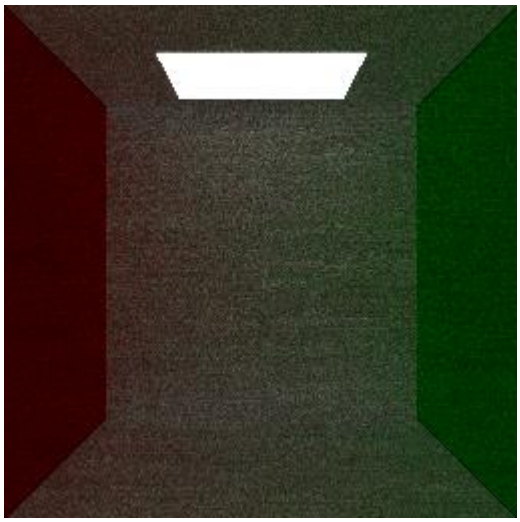
originating from our current hit point and the random direction back to the task. Finally we put the task back to the queue to be processed by another thread.

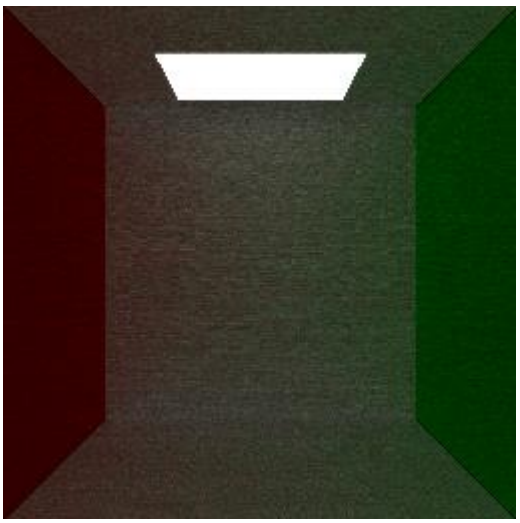
V. Results

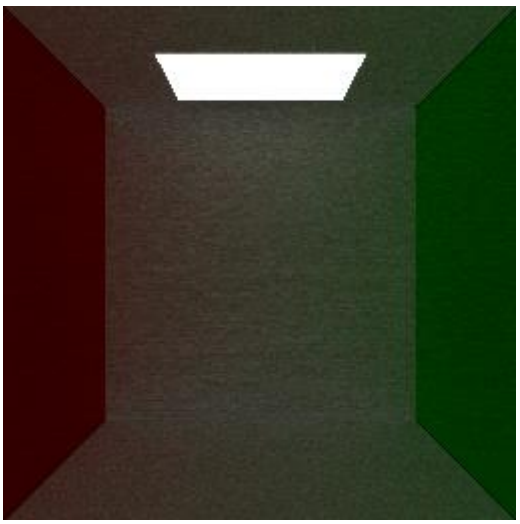
The recommended iteration count for the path tracing algorithm is around 800. I was able to generate a 256×256 image with 800 iterations in around 27 seconds. This is quite slow compared to other implementations [5].

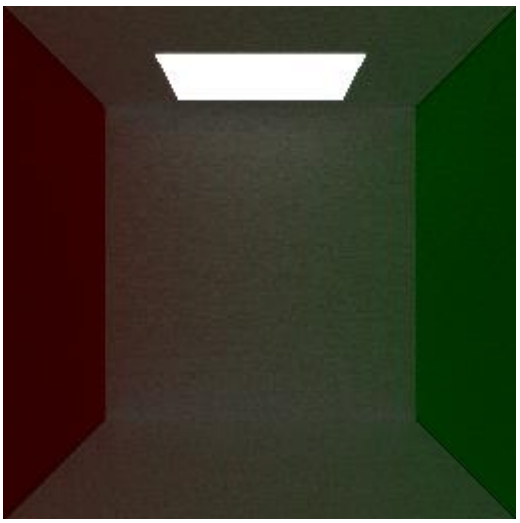
Following are the 256×256 images of the Cornell Box generated with iteration counts: 20, 40, 100, 200, 400 and 800. Rays do two bounces in the scene before terminating. As can be seen, indirect illumination of the red and green walls is visible from the bottom, back and top walls.

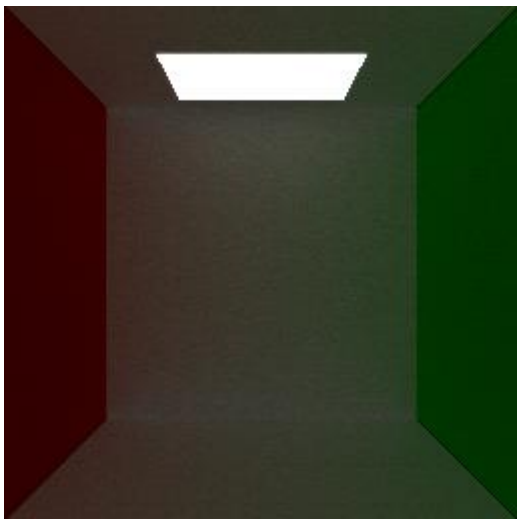












VI. Future Work

As mentioned, currently one problem is the global memory consumption from the queue. This can be improved by dividing the frame into batches of work.

Another problem, that can be seen from the result images, is an occurrence of black lines in the intersections of walls. This would require further investigation of the rays that happen to fall there.

Also the scene can be improved by adding differently shaped geometry with different material properties in it. Furthermore, the current idea of shooting a ray into a uniformly random direction, may not lead to fast enough convergence. There are ideas that propose for example to use a cosine weighed distribution of the actual reflection [6]. The current implementation is sometimes called the *Russian roulette* version.

References

[1] J. T. Kajiya, "The Rendering Equation", ACM, Dallas, August 1986, pp 143–150.

[2] GitHub, "Cuda PathTracer", https://github.com/jee7/cuda-pathtracer (accessed 18.01.2015)

[3] Y. K. Li, "Physically Based Shading and Path Tracing", http://cis565-fall-2012.github.io/lectures/09-24-Physically-Based-Shading-and-Pathtracing.pdf (accessed 18.01.2015)

[4] Wikipedia, "Möller-Trumbore Intersection Algorithm", http://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm (accessed 18.01.2015)

[5] GitHub, "Cuda Path Tracer", https://github.com/wuhao1117/CUDA-Path-Tracer (accessed 18.01.2015)

[6] Unknown author, "Bidirectional Path Tracing", https://graphics.stanford.edu/courses/cs348b-03/papers/veach-chapter10.pdf (accessed 18.01.2015)