Ian Stewart
Alex Newman

CUDA Real-time Raytracer

Our project is a real-time, massively parallelized ray tracer which runs on modern Nvidia GPUs. It is written in CUDA, a language based on C/C++ with extensions to support highly parallel computation and memory operations on the CPU (host) and GPU (device).

Ray tracing produces high quality images of 3D scenes and is particularly good at producing convincing transparent and translucent effects. Unfortunately ray tracing is generally much slower than rasterization, both due to the nature of the computation (which is particularly bad at using memory efficiently), and due to the lack of research and development into ray tracing hardware when compared to rasterization. This is beginning to change as modern GPUs have become extremely powerful, with some having upwards of 2,000 programmable cores. We have developed a ray tracer that can use the incredible power of modern graphics processors reasonably efficiently to produce high quality, high resolution images in real time. Our implementation is comparable to the basic project 4 ray tracer, minus the area light feature. It features reflection and refraction, intersection with triangles, spheres, and planes, good shadow generation, and realistic shading.

We also developed a .obj parser to allow objects to be loaded into our scene. The parser is somewhat specific as it requires that the objects be entirely constructed out of triangles. The parser is also capable of creating 12-triangle bounding boxes around the objects as it parses them, although this feature could not be integrated into the application as a whole due to time constraints. At this time materials, non-polygon objects (spheres, planes), and lights must be hard-coded into the ray tracer, though it is entirely possible to animate them.

Our project required us to solve several challenges at the start in order for us to proceed. Although the ray tracing algorithm is fundamentally very parallelizable, converting it from a serial computation that runs on the CPU to a CUDA kernel which is run as one million parallel instances would be the first major challenge. We solved this by treating each group of 400 pixels (20x20) as a block of 400 threads, with each thread finding the color of a single pixel in the final frame. Moving structures from the CPU memory space to the GPU memory space and getting the output from a computation into host memory was another challenge which we resolved with good organization and semantics of host and device pointers. We also chose to use the SDL library to display the output from a given calculation in an X window. As CUDA is an extension of C/C++, we could not use any of the functions from project 4 without rewriting them to be compatible with C-style argument passing and memory management. Although both of us were a little rusty on C, we managed to get the basic structure down in a reasonable amount of time.

We encountered several difficulties in the development of our project. First, debugging CUDA kernel errors is difficult since it is not easy to read the state of a thread as it executes. There was a particularly difficult to resolve graphical error involving sphere intersections that took 2 days to resolve - it would have taken much less time to fix had it occurred in a Java environment such as Eclipse. We also had difficulties with implementing acceleration structures such as bounding volumes; we were unable to demonstrate these features. Finally, although CUDA allows programmers to utilize high-speed block memory within the GPU, we were unable to resolve graphical issues that developed while attempting to implement this feature. This is unfortunate, as we believe this would have significantly reduced per-frame render time; block memory is 5-6 times faster than global GPU memory in the best case.

We also encountered difficulties when working on the object parser, which was originally written using several C++ libraries which were incompatible with nvcc (CUDA compile) which required a rewrite of several functions relating to the parser. There were also issues relating to keeping track of the number of faces and vertices while parsing, as it is not possible to recover the size of a malloc'd memory segment;

this was resolved by storing the number of vertices and faces and computing the size of the segments when needed.

Alex primarily worked on developing the .obj parser, bounding boxes and creating the blender scenes, which are included in the objects folder. Ian primarily worked on CUDA development, creating the GPU kernels and converting the project 4 code to C/C++.

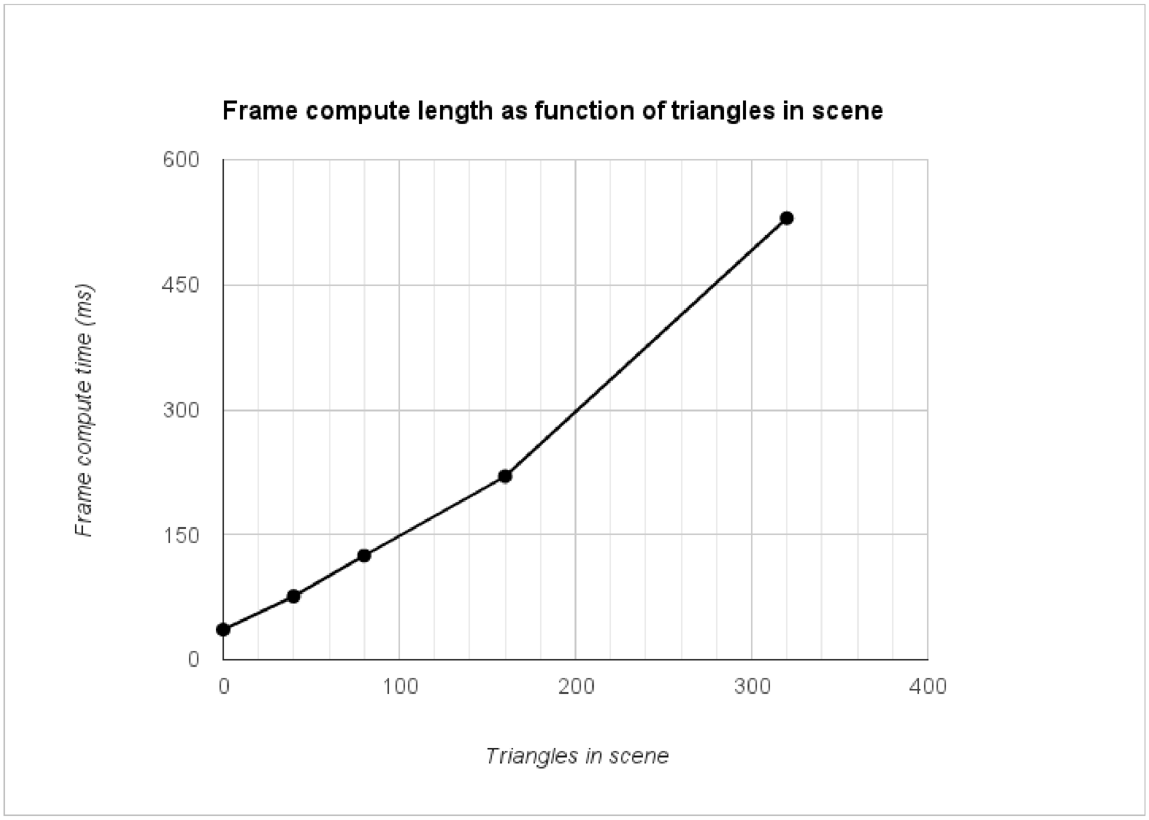
Further development of this project would focus on more efficient use of memory in the GPU (particularly in using faster block memory) and on acceleration structures. We want to implement something along the lines of a KD tree, bounding boxes or an intersection grid to improve the performance of the ray tracer as the number of faces increase.

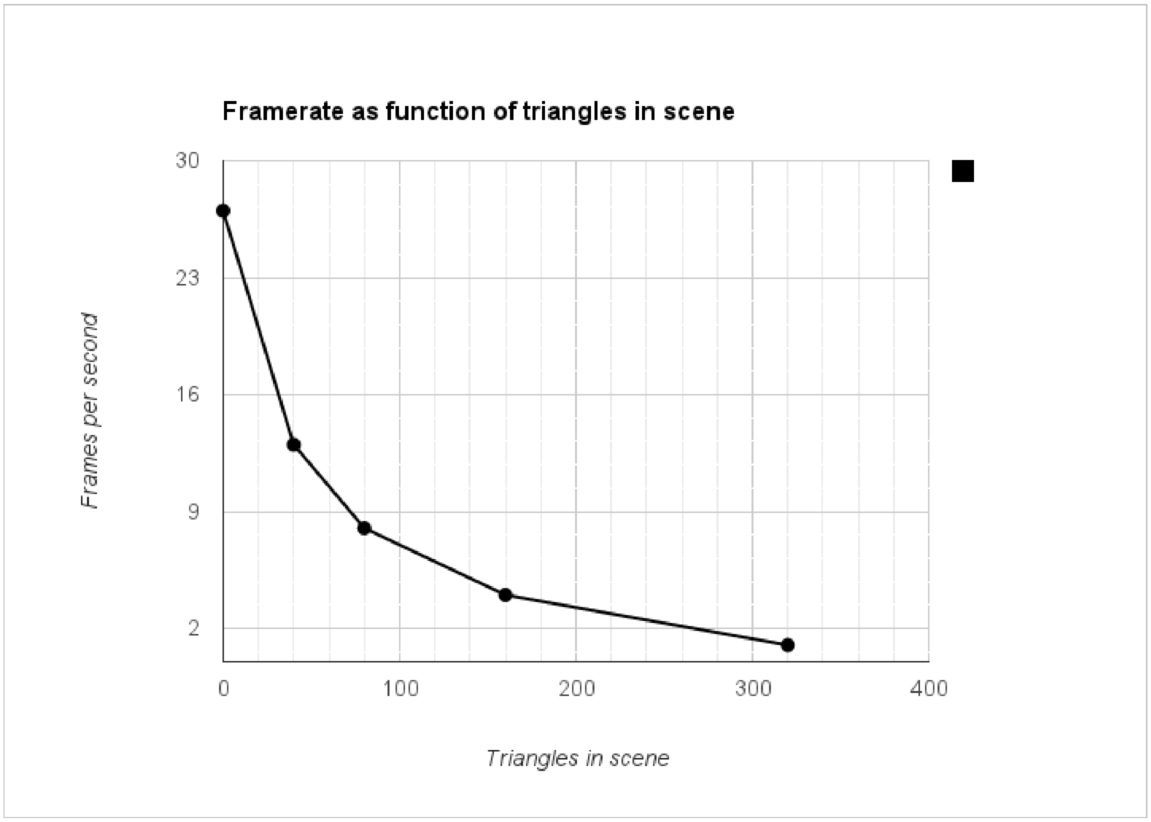
Notes on performance:
Maximum rendering speed with our current configuration is approximately 45-50 FPS with one plane and sphere at 1000 x 1000 pixels (1 M total threads in 2500 ‘blocks’ of 400 threads). This is a per-frame render time of 20-22 milliseconds. There is some overhead in the way we display our computed image; we copy each frame across the PCI bus and use SDL to copy the resulting data to the X window. A more efficient algorithm would probably map the output from each kernel call to an OpenGL texture and then draw that texture on an OpenGL quad. We considered this feature early on in development but chose to use SDL as a result of the relative complexity of binding memory on the GPU as both kernel memory and as texture data.

The render time as triangles, spheres, and planes are added grows roughly linearly. This is because each kernel traverses the memory holding these structures from start to end once. All tests and screenshots are run at 1000 x 1000 pixels. There is significant waste in the current ray tracing algorithm; even if we had simple bounding volumes the runtime would grow as a function of the number of triangle *mesh objects* in the scene as opposed to the number of triangles in the scene.

Here is data gathered from testing 0-320 triangles (4 diamonds). A test was attempted with 8 diamonds, and frames appeared to take around 1 second (1000ms). However, the program reported run times of about 230 ms, indicating that our method of determining run times is not 100% accurate, particularly as the number of triangles increases. Data in this set was gathered with three light sources and six planes.

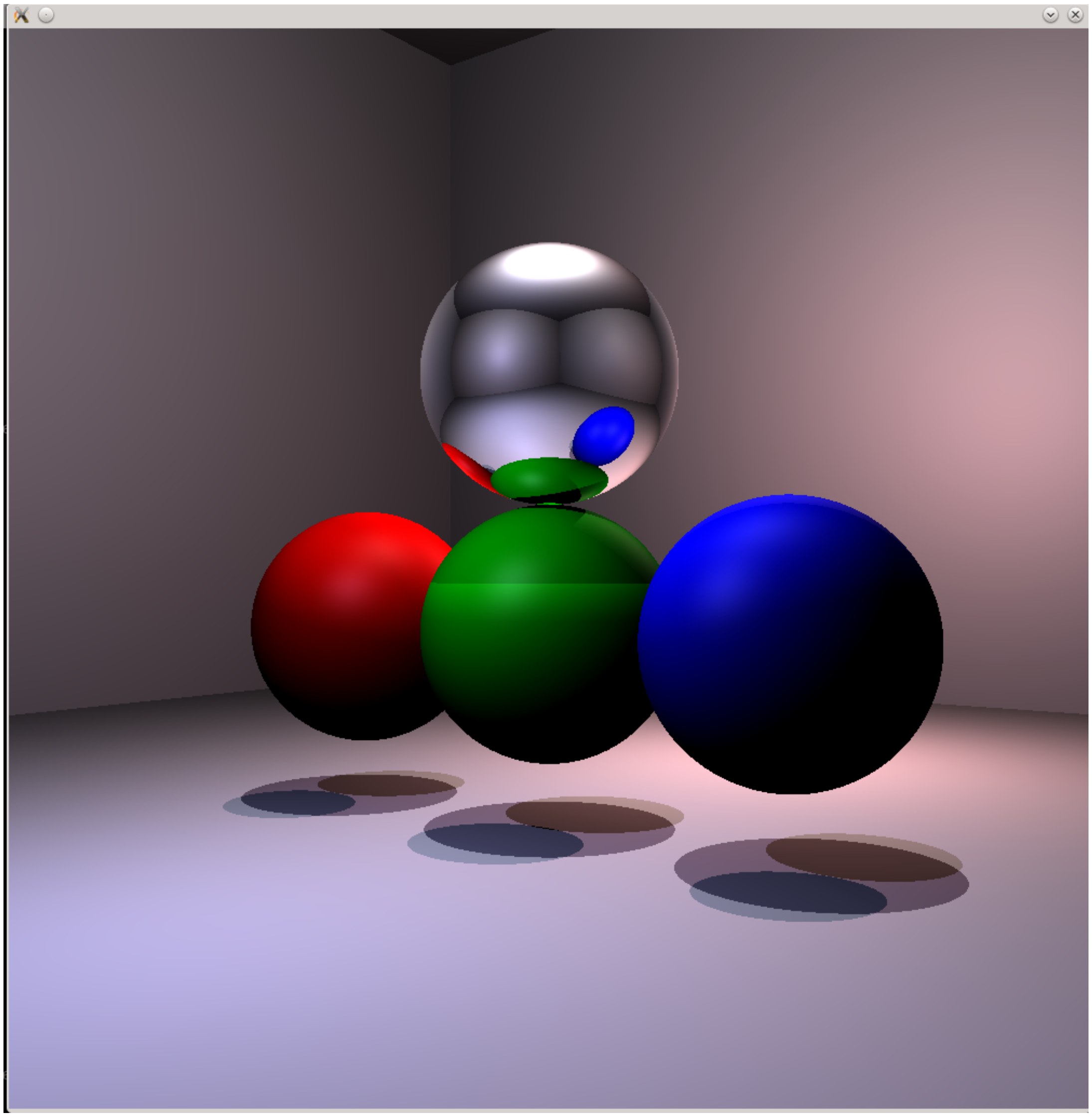
Triangles	Framerate (Frames/sec)	Frame length (ms)
0	27	36
40	13	76
80	8	125
160	4	220
320	1	530


Frame compute length as function of triangles in scene
600
450
300
150
0
0
100
200
300
400
Frame compute time (ms)
Triangles in scene

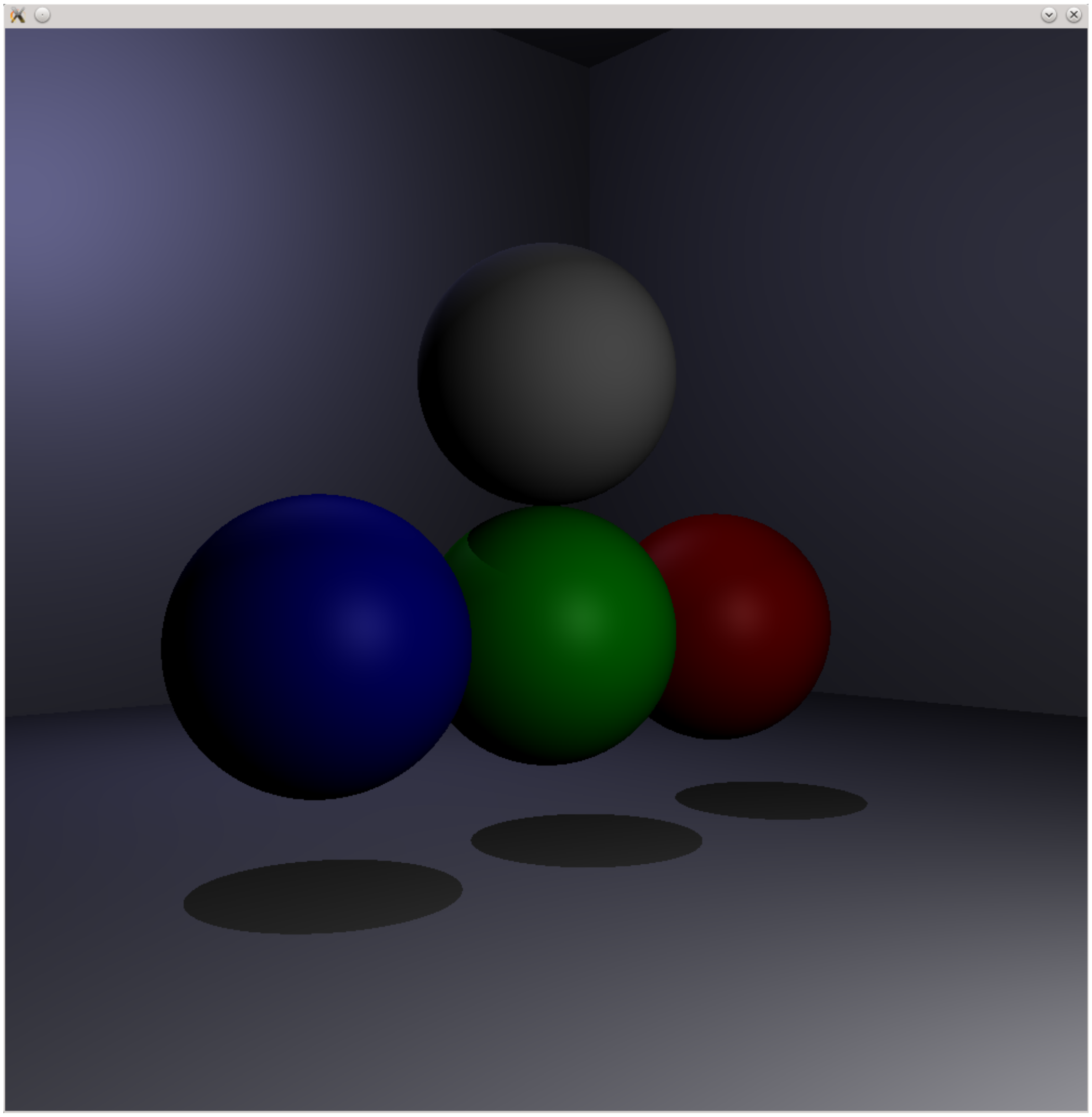

Framerate as function of triangles in scene
30
23
16
9
2
0
100
200
300
400
Frames per second
Triangles in scene

Screenshots:

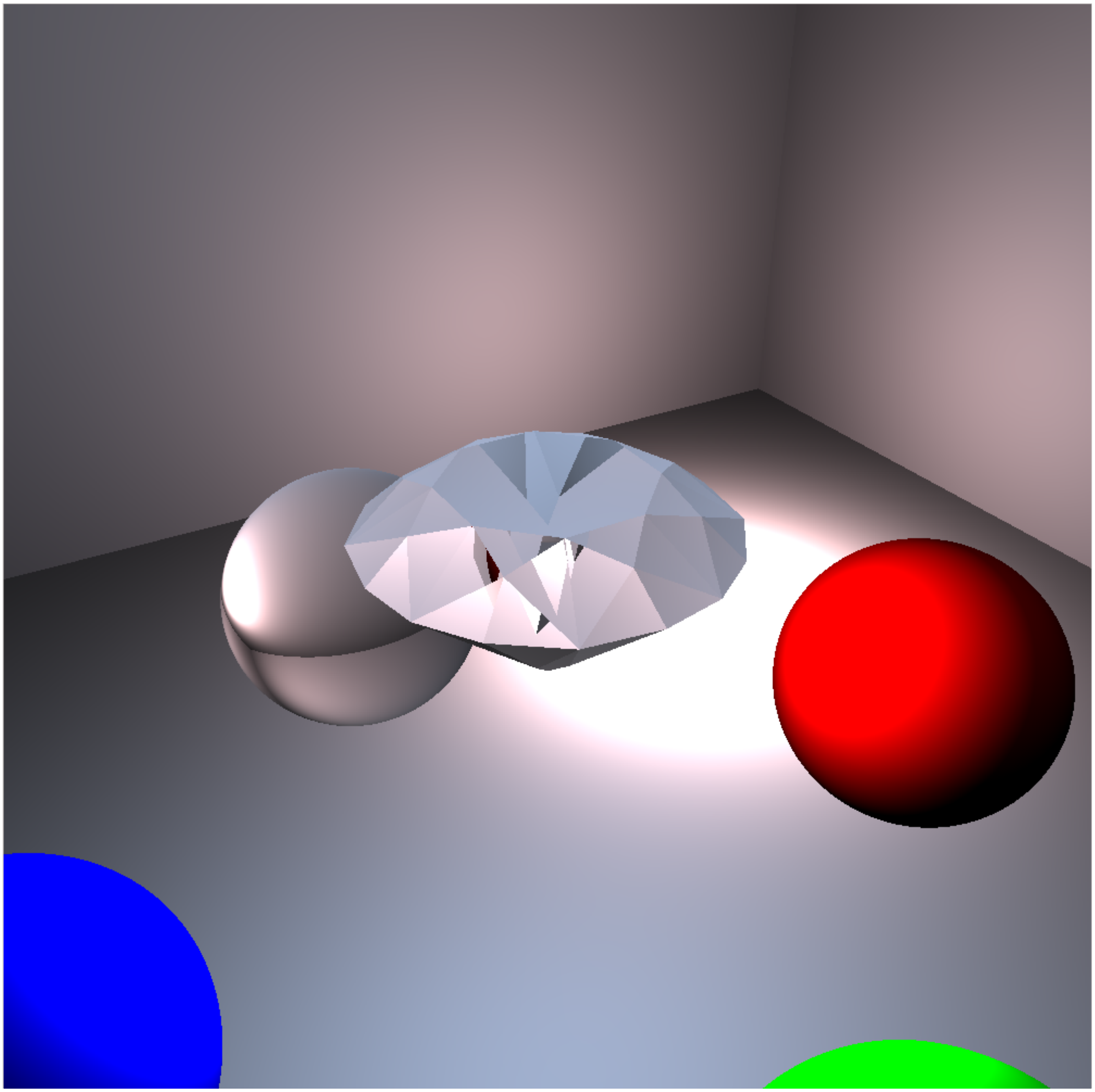
Scene with six planes, four spheres, and three lights. Top sphere shows reflection of room. Also observe proper shadow calculation on the floor below the spheres, as well as on the top of the green sphere. Walls of room demonstrate realistic shading. Runs at 25 frames/second on GTX 670 & Intel i5. More basic scenes with no reflective objects and fewer lights run about twice as fast.



Here is a more basic scene. With one light, six planes and four spheres we observe a frame rate of 29 FPS, or about 35 ms/frame.



A more complex scene featuring 80 triangles (facets), four spheres, six planes and three lights. We observe an average framerate of 8 FPS, or 125ms/frame. This scene demonstrates realistic refraction through multiple refractive objects with different indices of refraction (the diamond has an IOR of 2.417, the sphere is glass and has an IOR of 1.45). This scene is being rendered with five levels of refraction, although reflection has been disabled due to the structure of the ray tracing kernel. A recording of this demo is located in CUDARaytracer/Renders.



(Final comments on next page)

Code has been attached in a zip file along with this document. It is also available for download at https://github.com/Ian-Stewart/CUDARaytracer

Screenshots are viewable in-browser on github or by downloading the latest version of the repository. They can be found in CUDARaytracer/Renders/CUDA. There is also a short video demo in CUDARaytracer/Renders, which should play in VLC or any equivalent video player. Source files are located in CUDARaytracer/src and precompiled binaries of the above scenes can be found in CUDARaytracer/bin

Additional resources used throughout program development
https://www.udacity.com/course/cs344 - Introduction to parallel programming

http://www.libsdl.org/ - SDL graphics library

http://www.friedspace.com/cprogramming/SDLTest.c - Example SDL program. Our project uses a modification of the setpixel function described here to draw output from CUDA. Not sure how much you could really modify this and still get the job done, but it's worth mentioning nonetheless.

http://docs.nvidia.com/cuda/cuda-samples/index.html - was used early on during experiements with binding CUDA output to an OpenGL texture.