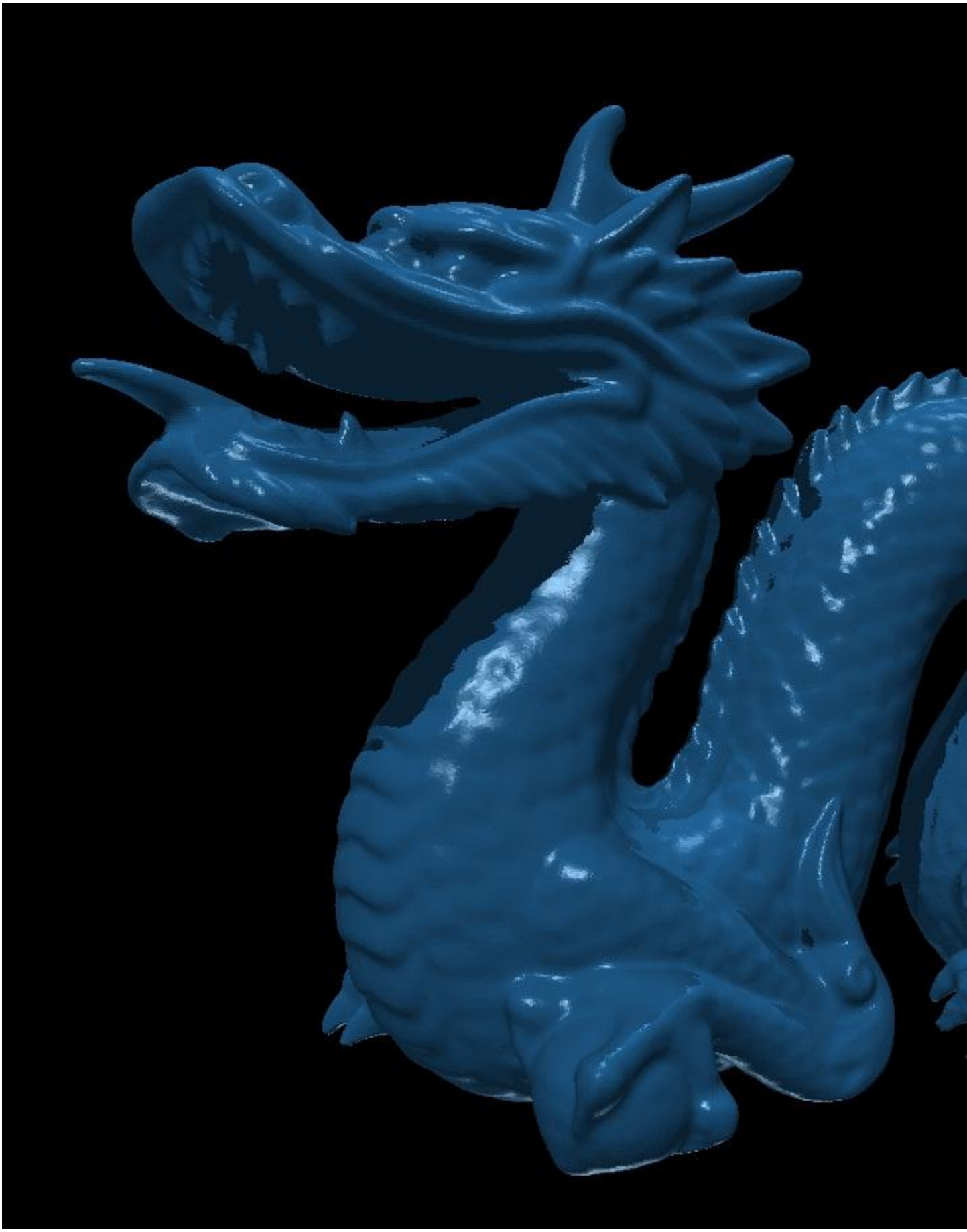
OpenCL Ray Tracer

CSSE 451
(Future) Dr. Taylor



Xinyu Cheng
Jacob Crouch
Spencer Murphy
Dylan Sturgeon

1. Introduction

Ray tracing is a technique for generating an image by tracing the path of a ray through pixels in a plane and simulating the effects of its environment. During Advanced Graphics, we had created a ray tracer that will perform on the CPU. This full functioning ray tracer taught the concepts but is not used in the real world because the CPU is not optimal for this kind of process. Instead, ray tracing is usually performed on the GPU. Our project is to recreate the ray tracer for the GPU. To do this we learned OpenCL and used it as the implementation language to create a ray tracer on the GPU.

2. Installation

1. Install an appropriate OpenCL SDK (we tested NVidia, AMD, and Intel). These can be found at the respective companies websites.
2. Set an environment variable
3. Install Visual Studio 2013
4. Download code - Github
5. Push the build button
6. Receive Image

3. OpenCL Research

OpenCL is a standard for cross-platform parallel programming, created by the Khronos Group. OpenCL is royalty-free, nonproprietary, and is supported by a wide range of platforms, including most modern CPUs and GPUs. Hardware developers implement the OpenCL standard on their devices and create software development kits (SDKs) for programmers to utilize the OpenCL standard. OpenCL can dramatically improve the performance of certain applications through its parallel processing capabilities. NVidia maintains an alternative parallel programming standard, known as CUDA, that is comparable to OpenCL but is proprietary and only works with NVidia GPUs.

3.1 Framework Overview

Nearly all modern devices implement a version of the OpenCL standard, including: processors from Intel and AMD and graphics processors from NVidia and AMD. Other manufacturers and compute devices implement the standard, but our implementation focused on CPUs and GPUs; specifically Intel CPUs and NVidia GPUs.

The OpenCL framework encompasses interactions between a host with available OpenCL enabled devices. The interactions are managed by a context and involve building applications for the devices to execute, exchanging data between the host and devices, and executing applications with the data. Hosts are typically CPU applications, written in a variety of languages because OpenCL bindings are available in several common languages. Our implementation utilized C++ for the host application.

Figure 1 provides a graphical representation of using the context. Host applications build OpenCL kernels by compiling OpenCL code into programs and then creating kernels. Kernels are the entry points for devices to execute the program. The host must specify any arguments to the kernel, which are likely to include memory objects, buffers or images. Memory objects describe data segments in device memory, that the host and device can optionally read or write. Hosts can then initiate program execution on the device by issuing commands, stored in a command queue to be executed. The asynchronous nature of program execution on devices necessitates a queue structure for commands.

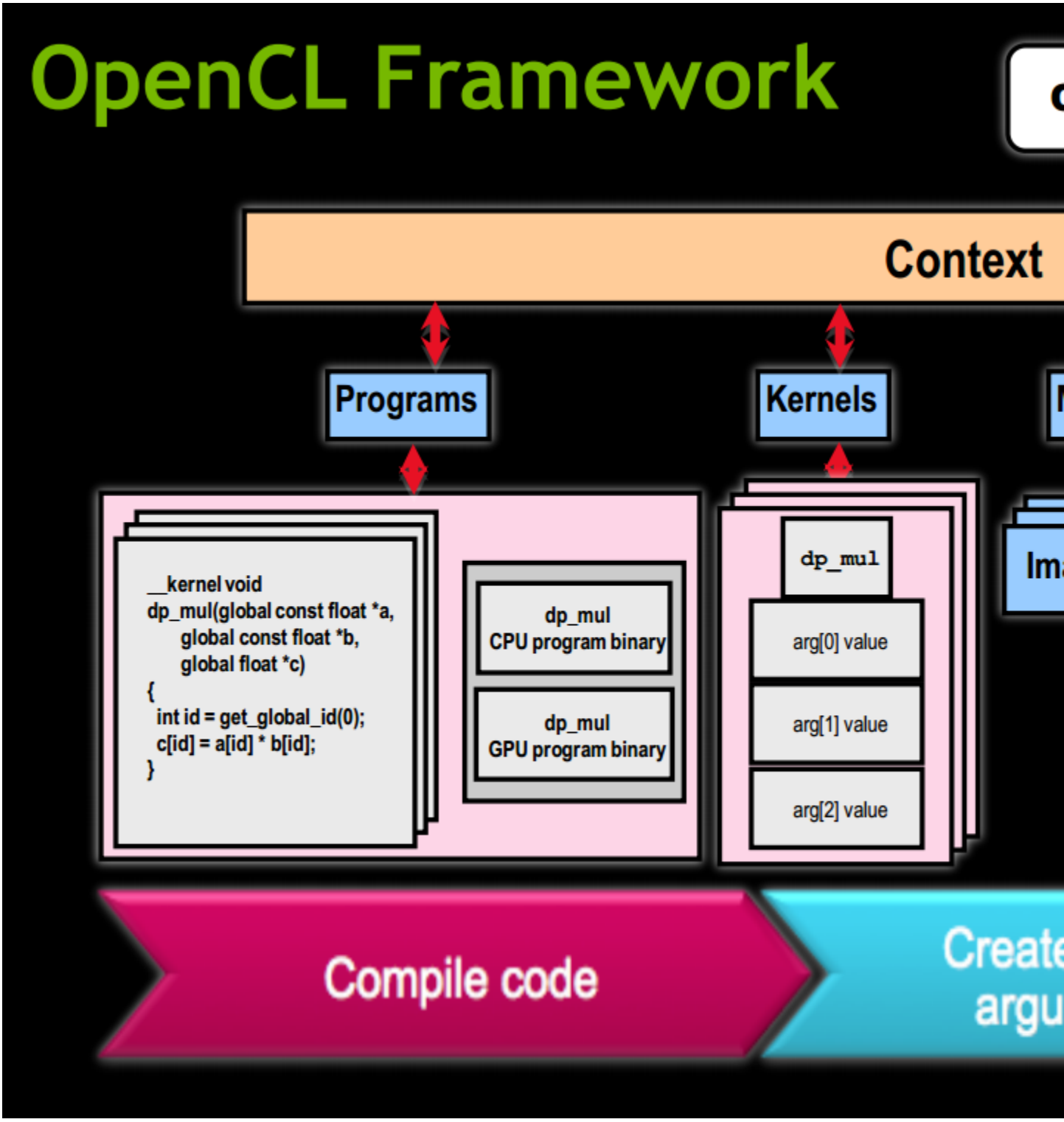


Figure 1. Graphical OpenCL framework overview. Highlights the use of contexts for the host and device to interact through building programs, sharing data, and executing programs with data.

3.2 OpenCL Code

OpenCL is a language derived from ISO C99. It is almost identical to C with built-in data types and libraries specific to the the OpenCL framework. It is very similar to shaders in OpenGL. Vector types are key built in data types allowing for built-in vector math functions useful for graphics such as normalize, dot, length, and other functions. It provides specification for the different memory types described later. As it is very parallelised, OpenCL provides built-in functions to accommodate synchronization.

While OpenCL provides many features, there are some restrictions. Function pointers are not allowed in OpenCL. Because of the difference in address space between the host and device, pointers can not be used in arguments because they would not actually point to the desired location. This means that all data has to passed to the device instead of references. Recursion is also not allowed. Memory can not be dynamically allocated, so all arrays and structures must be of a fixed size.

3.3 Building OpenCL Kernels

In order to build this code into a kernel, it must first be loaded in and built. Using the context, a program is created using the OpenCL code as the source. The program is then built before creating a kernel. The SDK providers that we tested with also included kernel builder software to test the validity of the OpenCL files. This made it much easier to test and debug the OpenCL code. Once it is built, a kernel is made from the program given the entry method.

Now that there is a kernel, the arguments need to be set. Array arguments must first be created as memory objects before they can be used. These can either be simple one dimensional buffer objects or more complex two or three dimensional image objects that are accessed using special built-in functions. These memory buffers can be read or written on the host or device as long as they are set to be read or written. Created buffers and other arguments must be set as arguments of the kernel. After all arguments are created and set, the kernel is ready to execute.

3.4 OpenCL Execution

OpenCL kernels execute on a hardware device that must implement a version of the OpenCL standard. The standard defines an interface to interact with the device, including requesting execution of programs. Devices consist of a collection of compute units which are themselves a collection of processing elements. The processing elements, or work-items, execute instructions in lock step with every other processing element within its containing compute unit, or work-group. OpenCL uses the work-item/work-group terminology as software constructs and processing elements/compute units as hardware constructs. It is possible for a single compute unit to execute multiple work-groups, but that does not impact the following discussion.

The host application invokes kernels on devices as a multi-dimensional collection of work-items. Some algorithms benefit from the concept of N-dimensional work spaces, but our implementation does not benefit from the availability of multi-dimensional work spaces. Work-items describe a specific instance of kernel execution; the number of times the kernel must execute to accomplish the desired task. Work-groups combine multiple work-items which must be executed in parallel using single instruction multiple data (SIMD) or single program multiple data (SPMD); essentially, multiple computation devices execute identical code using different data to obtain results faster than single threaded execution.

Hosts submit work to devices via a command queue within a context. Devices work out of a command queue, either in order or out of order. Work is often submitted asynchronously, because the host application need not pause while devices fulfill its requests; although it is possible to block for devices to finish submitted work through blocking instructions, events that record the status of submitted requests, and barriers entered into the queue that force all requests to be fulfilled before passing the barrier.

3.5 OpenCL Errors and Error Handling

The interface between hosts and devices includes error reporting and handling, because any request could fail for variety of reasons, some recoverable and others not. When requests fail, they set an error code, through copying the code into memory or returning a value, that can be interpreted. During our adventures with OpenCL, our favorite such error was the CL_INVALID_VALUE error that nearly any instruction could generate because a value it tried to use appeared to be invalid.

OpenCL errors are complicated through the asynchronous nature of execution: errors generated on the device can be received by the host at any proceeding point in its execution. Before learning about this potential for error, we spent a substantial amount of time searching for bugs in the wrong places: believing the host application had errors when actually the device was generating errors.

4. GPU Ray Tracer Implementation

4.1 Work in CPU & GPU

The first work in CPU is loading objects and materials, through which we could get the list of primitives and all the properties of objects and materials.

Besides, the CPU also handles the creation of camera, scene and tree based on the data we obtained from loading objects and materials.

Then, intersections and color calculations execute in GPU.

4.2 Data Passing

The idea for data passing between CPU and GPU is based on the OpenCL kernel command queue.

In CPU, we create buffers for data like triangles, nodes and materials. These buffers are input data for our GPU calculation.

The calculation results will be assigned to output buffer. In CPU, we can read back data in output buffer after the OpenCL functions are executed.

4.3 Multi Kernel

There are two kernels working in the GPU device: Intersection kernel and Color kernel.

For intersection, we pass tree nodes, triangles, camera and scene resolution to the kernel function. The output will be put into hitPoints, which is a list of hit points we get from all intersections.

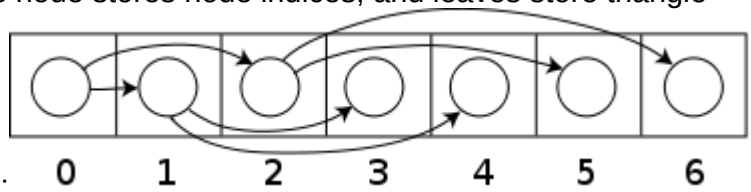
For color calculation, we pass the hitPoints, materials, lights, triangles and nodelists to the GPU. The output is the colors.

4.4 No Recursion

Recursion does not work in OpenCL. In order to traverse our tree on GPU, we created a stack. Besides, the reflections are implemented using a while loop.

4.5 Flat Trees

Pointers are not allowed in OpenCL, neither. Hence we use a flat tree to represent our BVH tree. It is actually a list of nodes. The node stores node indices, and leaves store triangle indices. Here is a picture for flat tree:



5. Results

The ray tracer was tested using three standard test scenes: a dragon, approximately 870,000 triangles; a spherical model, approximately 1,000 triangles; and a cornell box scene with a reflective floor, 30 triangles. All performance testing was performed with a Windows PC using the NVidia the CUDA toolkit version 5.5 with a i5 4670k processor, quad core 3.4GHz, and a NVidia GTX 760 GPU, OpenCL version 1.1. The OpenCL ray tracer was executed on the GPU, with the host application on the CPU. The comparison application is a CPU side only ray tracer implemented in C++ by project member Dylan Sturgeon. The team wanted to compare the GPU and CPU execution by running the OpenCL ray tracer on the CPU, but could not get the Intel OpenCL SDK to install nor the NVidia SDK to recognize the CPU as a platform.

Put a table in here.

Table 1. Performance testing results of OpenCL ray tracer on GPU.

The GPU implemented improved the performance of the ray tracer significantly. The team was hoping for a much larger performance improvement, but considering learning OpenCL was the first priority of the project, the code could likely be improved. Since it was certainly faster than the CPU implementation, the project is a success.

6. Future Work

6.1 Optimization

There are several ways to optimize this ray tracer in the future. Our tree used spatial splitting. This is an outdated algorithm and could definitely be improved. Additionally, the data passing could be improved drastically. The hit points array specifically is passed around too much. It is passed to the device from the CPU, read back from the CPU, and then sent across again. Similar unneeded passing also occurs with our lists of shapes and materials. We could also improve our memory reference locality. This could be accomplished by using less global memory. Also, by picking contiguous chunks of memory would speed up our kernels when they attempt to reference this memory.

6.2 Spheres

We would have liked to be able to model spheres. Luckily, they aren't too important because we could model spheres as a set of a lot of triangles. However, this setback meant that we were not easily able to model the "spheres" scene with three small floating spheres and one large gray one.

6.3 Animation

Some more improvements can be found through animations. Providing movement to the scenes would have been a very nice feature to our ray tracer. We did experiment a bit and added in the ability to move the camera to create a 360 degree view of a static scene. However, we would like to add more camera translations in as well.

6.4 Textures

Textures are something we did not support but would like to in the future. Being able to add an image onto a model is something that makes the ray tracer much better at modeling the

real world. Currently, we get the visual information from materials but would like to change to textures.