# ECE 6122 Final Project Report

Vaibhav Dedhia, Dan Kilanga, Theodore LaGrow, Chidiebere Okoli, Rory Rudolph

# I. Introduction

The idea behind the project was to make a high resolution animated video. We were inspired by pixar and dived into ray tracing. We wanted to generate a scene of objects with a physics engine and at each frame apply ray tracing to the objects in the scene. After processing the full scene, we stitch together the resulting images for a video. This method is not computationally efficient so we added a component of parallelization to the project for both the rendering speed and the ray trace marching.

# II. Setup

## Prerequisite Libraries for Compilation

This project is intended to work on Linux, primarily Ubuntu systems, and has not been fully tested on Windows. In order to compile properly, a few library dependencies are needed.

| Library | Package | Description |
|---|---|---|
| OpenGL | libgl1-mesa-dev | Graphics rendering library |
| OpenGL Extension Wrangler | libglew-dev | Helper for loading the proper OpenGL extensions |
| GLM | libglm-dev | OpenGL math library |
| Bullet Physics | libbullet-dev | Physics simulation library |
| FreeType | libfreetype6-dev | Used for uploading and displaying fonts |
| Assimp | libassimp-dev | Handles loading of Wavefront .obj and .mtl files |
| GLFW3 | libglfw3-dev | Windowing and user input (among other things) |
| FFmpeg | ffmpeg | Not a library, per se, but the actual application |

These libraries can be installed via the `apt-get` package manager like this:

```
$ sudo apt-get install libgl1-mesa-dev libglew-dev libglm-dev
libbullet-dev libfreetype6-dev libassimp-dev libglfw3-dev ffmpeg
```

## Compilation

**NOTE: You have to run the program from the bin directory** as the file paths for the 10-20 Wavefront object files are hard-coded. We know this is bad, but we didn't want to run install scripts or the likes to make sure the paths are in our "company" directory (/opt/ourteam), etc. It's a pretty cool program, so

please forgive us for this;  you should be fine compiling and running from the **ece-6122-final-project/bin/** directory:

```
$ tar xvf ece-6122-final-project.tar.gz
$ cd ece-6122-final-project/
$ cd bin/
$ cmake ..
```

## Running the Simulation

There are two main scenes/simulations to run -- each equally cool -- but first familiarize yourself with the program options:

```
$ ./ece-6122-final-project --help
Usage: ./ece-6122-final-project [OPTIONS...]

Options
  -f, --frames <NUM>        Total number of frames to simulate
                            Default=unlimited for scene 1
                            Default=300 for scene 2 (raytracing)
  -t, --threads <NUM>       Number of threads to use. Default=1
  -r, --raytrace            Enable raytracing. Default=disabled
  -s, --scene <1 or 2>      Scene to render.
                            Only scene 2 supports raytracing
                            Default=1
  -v, --verbose             Print debug messages
  -h, --help                Print this help message and exit

Report bugs to <rory.rudolph@gatech.edu>
```

For example, to run scene 1's physics simulation (raytracing not supported), execute one of these:

```
$ ./ece-6122-final-project
$ ./ece-6122-final-project -s 1
```

To limit the amount of frames to render before automatically exiting, execute with the -f option:

```
$ ./ece-6122-final-project -f 120
```

When running scene 2, it is recommended to first run without raytracing, as raytracing is CPU intensive:

```
$ ./ece-6122-final-project -s 2
$ ./ece-6122-final-project -s 2 -f 0
```

When running with raytracing enabled, an image will be created and written to disk for every frame rendered. Therefore, it is recommended to use a small frame size (less than 600):

```
$ ./ece-6122-final-project -s 2 -f 300 -r
```

For raytracing on scene 2, if you have ffmpeg installed a movie will automatically be created from the static images and stored to “movie.mpeg”. You can watch the raytraced images with your favorite mpeg1 player:

```
$ vlc movie.mpeg
$ mpv movie.mpeg
$ mplayer movie.mpeg
```

If an error occurs or you do not have ffmpeg installed, you can always run the command later:

```
$ ffmpeg -framerate 60 -i image%04d.ppm movie.mpeg
```

# III. Code Description

## Overall Execution

The program begins by parsing command-line arguments and initializing the needed OpenGL, physics, shaders, and raytracing objects. Depending on the scene selected, certain meshes (.obj and .mtl files) will be loaded in via the external assimp library. These meshes contain color and vertices information corresponding to a particular shape. After this, the meshes are sent to various other program entities, like the OpenGL rendering engine, the Physics engine, and the ray tracer. These modules are described in a bit more detail as follows.

## Rendering Engine

The OpenGL Rendering Engine is composed of many classes and primarily handles all the calculations involved with vertex array objects, vertex buffer objects, meshes, cameras, lights, shaders, and rendering to a GLFW window. It does not run physics simulations nor update the vertices thereof, but does coordinate the vertices and their colors from model space to world space, complete with whatever transformation may be needed.

The Physics Engine wraps the Bullet3 physics library and handles all collisions between object in the scene, including objects loaded from file and static plane objects like the ground and walls. Once the Physics Engine is finished, the objects’ vertice positions are updated and given to the OpenGL engine for rendering to the screen.

The Text Writer class loads, coordinates, and calculates text to the GLFW window. It uses the FreeType library under the hood for displaying various fonts. This is primarily used for writing the performance statistics to the screen.

Simple in theory, the argument parser plays a critical role to parse the contents of the command-line arguments in to their respective configuration variables.

## Raytracing

Ray tracing is the idea that you can model reflection and refraction by recursively following the path that light takes as it bounces through a given scene. From the perspective of the window, we are looking at a 2D rendering of a 3D scene where light will bounce off objects at a specified depth of bounces. Each time the light bounces on an object, we update that pixel until the depth of bounces has been reached for all of the pixels in the window view. We pass in the objects specified at a given position dictated by the scene and physics engine. For each frame, we run the ray tracing on the objects in the scene and iterate until all of the scene frames have been processed.

In terms of the function itself, we have to initialize the objects with surface color, emission color, how transparent the object is, and how reflective the object is. We then render the image at each pixel to the depth of the ray specified. We have to then transform our scene from world view to our camera view.

## Objects

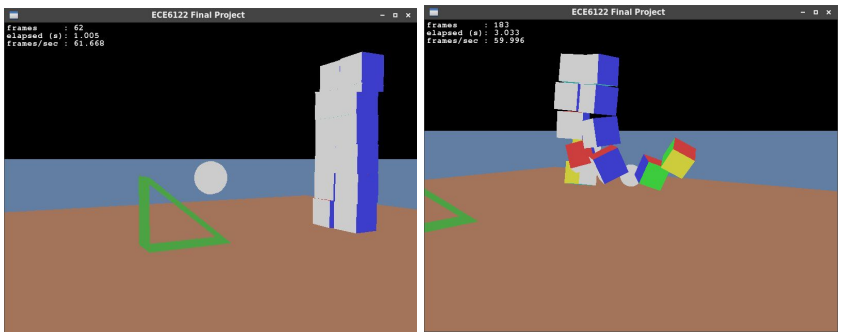
The objects in the scene are spheres falling into the screen. The objects are first generated in OpenGL and correspond to the effects of gravity and collision of other objects done by the physics engine. The objects are initialize with a position in space, an initial color (which is later replaced with the output of the ray tracing), and additional properties such as adding mesh to the objects.

# IV. Performance and Results

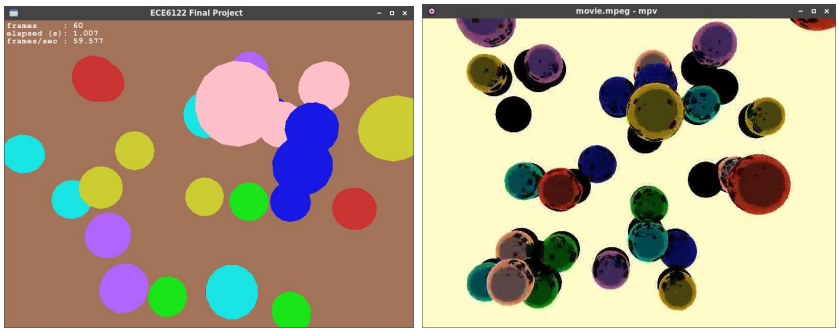
For a performance measure, we decided to calculate the frames per second for how long it took to render 1, 5, and 20 “somewhat complex” objects falling into frame with raytracing enabled.

| | Frames per Second |
|---|---|
| Any number without raytracing | ~60 fps |
| 1 Spheres | ~15 fps |
| 5 Spheres | ~4 fps |
| 20 Spheres | ~1 fps |

**Scene 1: Knocking over Blocks**




**Scene 2: Falling Balls with Raytracing**




# V. Lessons Learned

This was a long project but one we hope you will like. The project was mostly combining two projects into one since we had five members of our team. The ray tracing is computationally exhausting. We developed a very naive way to iterate through the marching (which we know that other optimization methods have been applied to this, however for the time of the project we decided it wasn't necessary and chose to do the more simple method). We found ways to help with parallelization but it is The geometry of the camera view to the world view transformation took a while to code up. One of the main issues we were having is to move the camera while preserving the geometry of the angle of the rays into the camera view. If we had more time, this would be something we've develop further.

The Physics rendering and OpenGL rendering takes a lot of time to get right, and documentation is often times limited. It is best to start projects with these pieces as early as possible. Both can be tricky to get right.

# VI. Contributions

- Vaibhav Dedhia - rendering optimization
- Dan Kilanga - movie stitching, raytracing optimization
- Theodore LaGrow - helped develop the ray tracing part of the project, helped write report
- Chidiebere Okoli - rendering optimization, helped write the report
- Rory Rudolph - OpenGL 3D render engine, argument parsing, Bullet physics implementation, image stitching, helped organize the group, helped write report, and design the scene.