Team: J.J. Abrahams

Members: Mitchell Marshe and Benjamin Johnson

**Manual**

Compiling:

cd into project directory

mkdir build

cd build

cmake ..

make -j8

Running:

cd build

./bin/lens

Controls:

ESC - exit.

W - move or zoom forward.

A - strafe or rotate towards the left.

S - move or zoom backward.

D - strafe or rotate towards the right.

Up Arrow - move camera upward.

Left Arrow - roll camera towards the left.

Down Arrow - move camera downward.

Right Arrow - roll camera towards the right.

Left Mouse - dynamic local or center camera rotation.

Right Mouse - dynamic camera zoom.

C - toggle between orbital or first person camera mode.

Q - decrease exposure setting.

E - increase exposure setting.

Space - Take a photo in the game. Take photos to gain points.

J - snap a screenshot. Kept in screenshots folder.

1 - Menger cube first level fractile.

2 - Menger cube second level fractile.

3 - toggle to show mesh objects.

4 - toggle to show lens effects.

Bonus:

Inside main.cc, there are the following variable that one can play around with:

drunkMode - enable a blurry vision of drunkenness.

light\_rays\_for\_bokeh - increase number for depth of field. Doesn’t really work well though.

showGui - toggles the gui on/off.

Game Technology Final Report

Our final project for Game Technology is to develop a simple game engine in OpenGL to handle our Photography Simulator. Photography Simulator is a basic game where a user controls a camera to snap photos of specific objects to gain points. This game features many cool lens effect, such as flare and blur. We decided to write our code in OpenGL since we really disliked Ogre3D and felt more comfortable in OpenGL due to our experience in Computer Graphics. Using three Computer Graphics projects, we had enough code for the logic of camera and user controls, including a rendering pipeline. From here, we needed to implement what a (game) object would be to complete the core of our game engine. Then we needed specific features like saving a screenshot, scoring, and having a basic GUI for our user in this game demo. Meanwhile, we wanted to make sure we supported Windows 10 and Linux.

Sounds like a relatively simple project, but while developing the game engine, there were many problems to be solved. For instance, loading in 3D objects from .obj files to be used as assets. In OpenGL, there are no native objects like the cube or sphere as we see in Ogre3D. Although, one could make C++ code to pipe the data needed to render out a cube, doing so is cumbersome. This isn’t an ideal situation as a game developer. We want complicated objects that are built from 3D software. Therefore, we needed a way to import 3D information like Wavefront’s .obj file.

Using resources from the tutorial websites learnopengl.com and opengl-tutorials.com, it was suggested that we create a wrapper class around Assimp to handle parsing in the .obj files. Assimp is a very robust 3D file parser that can handle pretty much anything, including an entire scene in multiple formats. We decided to just support single trimesh objects as it is the basis to most objects. All of the object’s information is stored in a struct called Mesh, which stores vectors of vertices, uvs, normals, faces, and then a single material ID. This struct is then passed onto a shader for rendering out the object.

An interesting problem with loading in a file with Assimp, is that we couldn’t find files locally. This is because we didn’t know how to include different files other than source code in the Cmake. The solution was to implement a simple script to capture the .exe path as a current working directory. This script is called Filesystem, it finds a working path for both Windows 10 and Linux.

Now that we have an object loader, we needed a class called Object to abstract most of the setup process for an object into the pipeline (or rather scene). In other words, we don’t want to manually code up 100 lines for one object, but rather the fewest lines of code to add something in. Ideally, Object should be instantiated in one line of code, however, there there is a problem which will be explained later. Currently, it takes about 10 lines of code to setup every Object in the scene.

These 10 or so lines of code for adding an Object is as follows. Create a new Object. Pass in a file path to load in a .obj. Bind a 4 by 4 model matrix (a graphics thing). Translate, rototate, and/or scale the object. Load in the shaders, lights, and other properties. Finally, call setup, which is really a graphics thing. Woah! This sucks…

The Object class didn’t really fully abstract the process of adding an object into the scene. Well, allow us to explain an anomaly that occurred. When we added code needed to bind data into the CPU for processing, our objects would be stuck in screen space and be exploded. The image on the screen looked as if we were looking into a colliscope. It’s a terrible anomaly! We couldn’t figure it out as we didn’t understand this graphic problem, nor had the time.

We had to dedicate some of this time for mechanics like taking a screenshot, and getting a score for taking pictures, and then showing that score onto a GUI. This is the Photography Simulation part we planned to demo out of our game engine. Saving a screenshot wasn’t too bad to implement as we have done so in a previous graphics project. We used stb\_image\_write, a single header library to support saving .jpgs. This library works both for Linux and Windows 10.

Getting a score for taking a specific picture of an object was an interesting feat. Instead of implementing raycasting, we decided to do color picking to determine if our user had targeted an object. This method is inherently slow, but really simple to implement. Anyways, if our user is told to take a picture of a dog, then that user must be directly aiming at the dog and press J to snap a photo.

When a photo is taken on a specific object, a score will be hosted on a GUI somewhere in the screenspace. The GUI is built around the ImGUI library, which is really easy to use. We hope by the end of the project, we’ll have some features where the user can toggle some of the game’s settings. Moreover, if you look down below, there are descriptions of pipeline and the various classes that were implemented in this project.

Main

Pipeline

…

<<<OpenGl Setup>>>

<<<Lights Setup>>>

<<<Objects Setup>>>

…

Rendering Loop

<<<Objects Render>>>

<<<Post Processing>>>

...

Camera

Camera is a class to handle orbital and first person camera movement and screen view from world space.

Controller

Controller is a class to handle user input. See Manual for usage!

Filesystem

The Filesystem is a simple script to handle file paths in our project for Windows 10 and Linux systems. We needed this script because we couldn’t find certain files since we didn’t know how to make Cmake link them. Here is the inspiration for this code: <http://www.codebind.com/cpp-tutorial/c-get-current-directory-linuxwindows/>

GUI

A simple wrapper around ImGUI to project our player’s score.

Lights

Lights is a very simple set of classes for Directional, Point, and Spot Lights. Each class has setters and getter for light properties such as position, direction, ambient, diffuse, specular, etc. Each type of light has a default constructor with “ideal” values.

Within Main, one can set up to 10 of each light type. If none are instantiated, there is a default directional light down the pipeline in the render\_pass class. Theses lights are to be held inside their respected arrays and sent off to the render\_pass to be loaded into default or object shaders. These shaders require that loadLights be called. See the section Main for more details.

Most of the Lights pipeline logic was inspired by: <https://learnopengl.com/Lighting/Multiple-lights>

Loader

Loader is a simple class to handle extracting a single trimesh object from a wavefront .obj file. One can easily extend this wrapper to support more .obj features such as multimeshes and materials. This class wraps around Assimp, which is a parsing library for most supported 3D files. This code was inspired by these two tutorials:

<https://learnopengl.com/Model-Loading/Model> <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/>

Material

Material is a simple struct that holds arrays of unsigned int ids to OpenGl 2D Diffuse and Specular Textures. Currently, this code is unused due to the lack of support of loading in materials from the Loader class.

Mesh

Mesh is a struct that holds the vertices, uvs, normals, faces, and material\_id to an object.

Object

Object is a class to handle loading, setting, and rendering an object in our game engine. One can load in an single trimesh .obj by giving a path. Then set that object’s position, rotation, and scale, as well as what diffuse and specular materials are to be used. One must also pass in the shaders and lights to be used. Then one can render the object.

Saver

Saver is a simple script to save .jpg images. This code is a wrapper around stb\_image\_write.h, a robust single header library for saving multiple image types.

Floor

Floor is a class that holds code to create a native OpenGL object which is the checkerboard in the world.

Menger

Menger is a class that holds code to create a native OpenGL object which is the fractal cube on top of the Christmas tree in the scene. Toggle 3 and move the camera towards the top of the tree to see!

Assets

This is a folder inside src which contains all of our sourced wavefront .obj and texture files. See Resources.txt for links!