6.837 Computer Graphics Fall 2019
Programming Assignment 0: OpenGL Mesh Viewer
Due Wednesday, September 11 by 8:00 pm

Please note that this assignment is not at all included in your grade (even the extra credit will not count). This assignment is meant to get you familiar with problem sets in this class, and you do not have to submit anything for it.

1 Getting Started

Let's start off by looking at `main.cpp`. It contains a fully functional application that displays a teapot. Other than that, it's not very interesting.

To build the starter code, use cmake to generate your preferred build system (`make, Visual Studio, Xcode`) and then execute the build. On the Athena systems, we recommend you use make to do an out-of-source build:

```
$ cd <path_to_zero>
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Once you've successfully built the executable, run it by typing "`./a0`" at the terminal. It should display a teapot, and that's all it really does...It's now your job to make this application a bit more interesting by modifying the code.

2 Requirements

2.1 Color Changes

Add the ability to change the color of the displayed model. Right now, the color is set to [0.5, 0.5, 0.9] (RGB), which is a boring light blue. Your task is to wire the `c` key to toggle through several other colors (feel free to choose

which colors you want). How do you handle keyboard events? Notice that, when you press keys while the application is running, the console says something like this:

```
Unhandled key press 67.
Unhandled key press 76.
Unhandled key press 76.
Unhandled key press 79.
```

The code that prints these messages is in the `keyCallback` function. Modify the code to handle the `C` key appropriately. A reasonable way to do this might be to have the `c` key increment some sort of global counter variable and then use that variable to select a color in the `updateMaterialUniforms` function.

Note that OpenGL will not immediately redraw the scene after it has called the code in `keyCallback`. Instead, the while-loop in `main` constantly calls a number of functions. These functions include `updateMaterialUniforms` function, which configures a new material color, `drawScene`, which draws the scene to an off-screen buffer, and `glfwSwapBuffers`, which make the off-screen buffer visible.

2.2 Light Position Changes

Add the ability to change the position of the light. In the code, the light is placed at [2.0,3.0,5.0]. Wire the arrow keys to change the position of the light. More specifically, the left/right arrow keys should increment/decrement the first value of the position by 0.5, and the up/down arrow keys should do the same for the second value. This can be done quite similarly to the suggested method for the previous requirement, but note that light updating happens in `updateLightUniforms`.

2.3 Mesh Loading and Display

Once you have completed the above requirements, we can move on to the tough part: loading new objects. In the sample code, we have provided several 3D meshes in OBJ format. It is your job to write the code to load and display these files. OBJ files are a fairly standard format that can describe

all sorts of shapes, and you'll be handling a subset of their functionality.

Let's look at `sphere.obj`. It's a big file, but it can be summarized as follows:

```
#This file uses ...
 ...
v 0.148778 -0.987688 -0.048341
v 0.126558 -0.987688 -0.091950
 ...
vn 0.252280 -0.951063 -0.178420
vn 0.295068 -0.951063 -0.091728
...
f 22/23/1 21/22/2 2/2/3
f 1/1/4 2/2/3 21/22/2
...
```

Each line of this file starts with a token followed by some arguments. The lines that start with v define *vertices*, the lines that start with `vn` define *normals*, and the lines that start with `f` define *faces*. There are other types of lines, and your code should ignore these.

First, you should read in all of the vertices ("v") into an array (`vecv`) (or any other data structure that allows you to quickly reference the ith element). Then, do the same for the normals ("`vn`"), loading them into another array (`vecn`).

Understanding the faces ("`f`") is a little more difficult. Each face is defined using nine numbers in the following format: $a/b/c\ d/e/f\ g/h/i$. This defines a face with three vertices with indices a, d, g and respective normals c, f, i (you can ignore b, e, h for this assignment). The general OBJ format allows faces with an arbitrary number of vertices; you'll just have to handle triangles. The starter code provided uses `rec.record(Vector3f pos, Vector3f normal)` to add a vertex's position and its normal. For a given face, you must record their vertices consecutively. At the end, you'll call `rec.draw` which iterates over the recorded vertices 3 at a time to display the triangles.

So let's say you have the vertices and normals stored in `vecv` and `vecn`. Then you'd the aforementioned face as follows:

```
rec.record(vecv[a-1], vecn[c-1]);
rec.record(vecv[d-1], vecn[f-1]);
rec.record(vecv[g-1], vecn[i-1]);
```

You may be wondering why there are all those minus-ones. It's because the faces index vertices and normals from 1, and C/C++ indexes from 0. If you have this implemented for one face, the rest is fairly straightforward: you simply loop over all the faces to record the complete mesh, then call `rec.draw`. This should happen in `drawObjMesh`.

Near the top of `main.cpp`, `vecv` is defined as an `std::vector` of `Vectof3f`s. An `std::vector` is simply an array of objects of the same type. In this case, it is an array of Vector3f objects, `std::vector<Vector3f> vecv`. Since you're going to use several C++ standard library types in this assignment, we add `using namespace std` to the top of the file, which allows you to drop the `std::` and use the shorter name `vector<Vector3f>`.

If you're interested in learning more about the standard container classes, check out the reference pages at `http://en.cppreference.com` and the STL documentation at `http://www.sgi.com/tech/stl/`.

To add a new entry to this array, use `vecv.push_back(Vector3f(0,0,0));` As for iterating over entries in an `std::vector`, here is an example method using indices:

```
for(int i=0; i < vecv.size(); i++) {
   Vector3f &v = vecv[i];
   //do something with v[0], v[1], v[2]
}
```

Please also keep in mind that you'll need another array to store the faces (perhaps `vecf`). It may be tempting to try to draw them as they are read from the OBJ, but OpenGL requires you to redraw the model whenever the window is obstructed or resized (and also when you change the color or lightning).

Your final executable should take the OBJ files via standard input:
`./a0 < ../data/sphere.obj`

The "<" operator will put the contents of sphere.obj into the "standard input" stream. This stream can be accessed using the cin object. For example, to read a single line of data from the stream (all characters up to the next newline):

```
const int MAX_BUFFER_SIZE = 4096;
char buffer[MAX_BUFFER_SIZE];}
cin.getline(buffer, MAX_BUFFER_SIZE);}
```

cin.getline will return zero at the end of the file. You can use this fact to step through each line in the file. Once you have an array of characters (the text from a single line of the file), you can parse it using a stringstream object. Create a stringstream object from an array of chacters (buffer) as follows:

```
stringstream ss(buffer);
```

Now that you have a stringstream object, you can read tokens (separated by spaces) from the buffer in order by using the ">>" operator. For example, given the input string "v 1.0 1.1 1.2", in the following code:

```
Vector3f v;
string s;
ss >> s;
ss >> v[0] >> v[1] >> v[2];
```

will put the value "v" into s, and load the values 1.0, 1.1, and 1.2 into v[0], v[1], and v[2]. Note that you can compare the string objects to constant strings using the regular "==" operator.

```
if (s== "v") {
      //do something
}
```

You should be reading from the buffer and using that information to populate the position, normal, and face arrays in `loadInput`.

Make sure that you're able to load and view the three provided files without crashing; these are the only three files we'll test your program on.

You may want to run the provided sample solution `a0soln` to get an idea of how your application should work (run `.a0soln < data/garg.obj` and read the console output for usage instructions).

3 Extra Credit

Here are some ideas (sorted roughly by increasing level of difficulty) that might spice up your project. The amount of extra credit given will depend on the difficultly of the task and the quality of your implementation. In addition, feel free to suggest your own extra credit ideas! Just because it's not on this list doesn't mean we won't give you some extra points (although if it's a big addition, make sure you run it by the course staff first just to make sure).

Easy

- Program the `r`-key to spin the model as a rotating display would show.
- Modify the code so that the `c` key smoothly transitions between different colors (rather than just toggling it).
- The current lighting model has creates images that have very high contrast. You can lighten the shadows by adding a constant offset "ambient" lighting term to the shader code in `starter0_util.h`. The ambient term approximates soft interreflections and is a common "hack" in real-time computer graphics.

Medium

- Implement a mouse-based camera control to allow the user to rotate and zoom in on the object. Credit will vary depending on the quality of the implementation.
- The starter code computes lighthing for each vertex, and lets the rasterizer interpolate shading values over adjacent triangles (see `https://en.wikipedia.org/wiki/Gouraud_shading`). You will get higher-quality lighting if you instead interpolate the vertex normals, and compute lighting on the interpolated normals (this is known as *Phong shading*). Modify the vertex and fragment shader code in `starter0_util.h` to implement Phong shading.

Hard

- Large meshes are quite difficult to draw and process. For interactive applications, such as video games, it's often desirable to simplify meshes as much as possible without sacrificing too much quality. Implement a mesh simplification method, such as the one described in Surface Simplification Using Quadric Error Metrics (Garland and Heckbert, SIGGRAPH 97).

4 Submission

As a final step, write a `README.txt` that answers the following questions:

- How do you compile and run your code? **Provide instructions for Athena Linux.**
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. *This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.*
- Did you do any extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what and how you did it.
- Got any comments about this assignment that you'd like to share? Was it too long? Too hard? were the requirements unclear? Did you have fun, or did you hate it? Did you learn something, or was it a total waste of your time? Feel free to be brutally honest; we promise we won't take it personally.

Submit the following on Stellar

- Your code (probably just `main.cpp` for this assignment).
- A compiled executable built from your code name `a0`.
- The aforementioned `README.txt` file.
- Any additional files necessary to run your program.