CSE 167 Assignment 2

Due: Feb 14 12:59 PM

From the two previous assignments we have got somewhat familiar with an OpenGL program. Now it's the time to really get to the fun part of lighting and shading!

From this point we're no longer using the rasterizer code you wrote for assignment 1. We will be using OpenGL for all our rendering in later homeworks. Welcome to the shader world!

In this homework you will learn how to:

1. Modify parser to support triangle meshes (10 pts)
2. Center and scale the 3D models (10 pts)
3. Render polygons (triangle meshes) in modern OpenGL (10 pts)
4. Control objects using the mouse (30 pts)
    a. rotating (20/30 pts)
    b. translating (10/30 pts)
5. Set material properties (10 pts)
6. Set up light sources (20 pts)
    a. directional light (5/20 pts)
    b. point light (15/20 pts)
7. Enable Light Interaction Controls (15 pts)
    PURE FUN: Spot lights

A video demo of this assignment will be posted later this week. Refer to Piazza for more information!

Part One Complete the Model Loader

You already know how to load point clouds from assignment one. Now it is time to modify the parser you've written to support the triangle meshes by loading the connectivities of vertices.
In the OBJ file format, the connectivity is defined by lines starting with an 'f' for face. Just as how you parsed for vertices and normals, now parse for face and store properly.
Example line of face:

```
f 31514//31514 31465//31465 31464//31464
```

In this example, the 31514^{th} vertex, 31465^{th} vertex, and the 31464^{th} vertex forms a triangle in a counterclockwise manner.
Notice: Make sure the stored faces values are 0-based because they are 1-based in the OBJ file.

What to render:

- All 3 objects (dragon, bunny, bear) individually to the screen
- Switch among three models by pressing F1, F2, and F3 keys

Part Two Scaling and Centering

In order to make mouse control work well, we want those models to be centered in the window and have similar scale on the screen. To achieve this goal, you will have to process the vertices of each of the models, translate and scale them.
An easy way to do this is to process the vertices value, find such scale factor and translation once when parsing the values, and store transmation matrix in the memory (see below).

What to render:

- Center the model such that the geometric center of the model resides in the center of the rendering window. This can be done by looking at each dimension (x, y, z) independently, calculate the maximum and minimum coordinates along each dimension, and use the midpoint between them as the centering point of the model.

- Scale all the models to the same size: fit them within an imaginary 2x2x2 cube centered around the origin. In other words: the vertices position values should be in range of [-1,1].

Part Three Rendering in Modern OpenGL

In the starter code, you have been switching between your own rasterizer an with modern OpenGL. Now it's time to unleash the power of OpenGL: use VAO (Vertex Array Object), VBOs (Vertex Buffer Object), and EBOs (element buffer object). Refer to here if you want to learn more about these OpenGL objects: https://learnopengl.com/Getting-started/Hello-Triangle
Rasterizer mode is no longer needed for this and later homeworks, so feel free to deprecate it :)

What to render:

- properly generate such buffer like you did for assignment one
- bind them to the model vao and properly pass to OpenGL
- Render the model with triangle mesh faces

Part Four Mouse Control

Now everything looks nice and good on the screen. It's time to add some interaction function to your program. You are going to disable the spin function in your code, and substitute it with some real update function.

What to render:

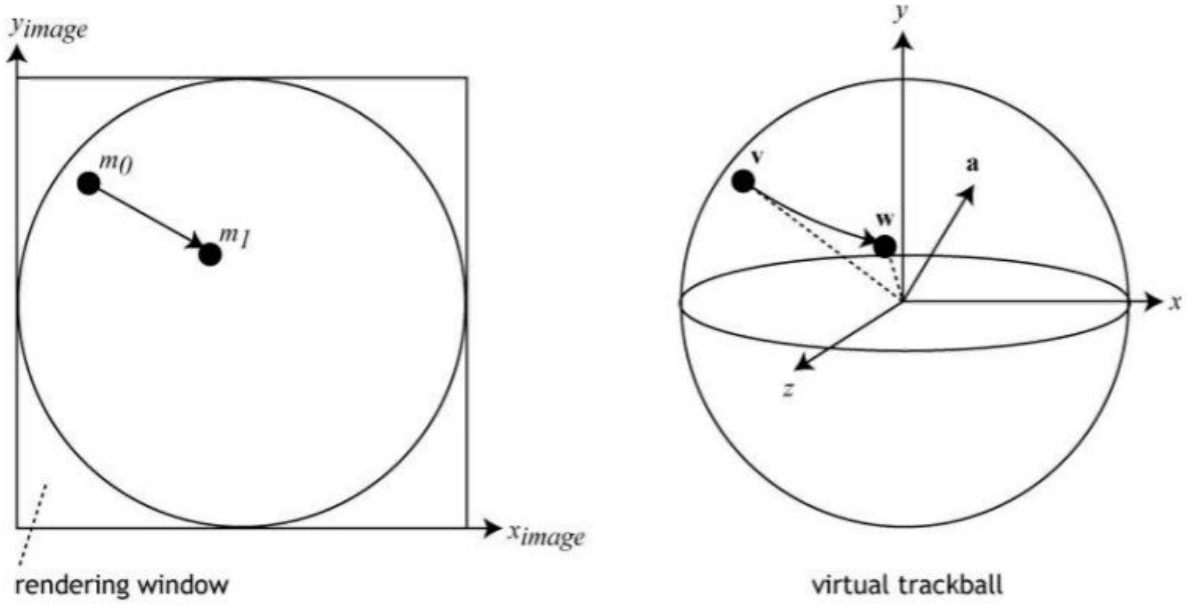
- **Rotation**:
  While the left mouse button is pressed and the mouse is moved, rotate the model about the center of your graphics window. We will refer to this operation as "trackball rotation". A video will be posted on Piazza about how it should work.
- **Translation in x-y plane**:
  When the right mouse button is pressed and the mouse is moved, move the model in the plane of the screen (similar to translation by keyboard operation in assignment 1). Scale this operation so that when the model is in the screen space z=0 plane, the model follows your mouse pointer as closely as possible. If you don't have a right mouse

button, use your mouse button along with a function key (Shift, Control, Alt, etc.) to enter this mode.

- **Translation along z-axis**:
  Use the mouse wheel to move the model along the screen space z axis (i.e., in and out of the screen = push back and pull closer).
- **Scale**:
  Retain the functionality of the 's'/'S' keys to scale the model about its object space origin. The other keyboard functions for the control of the 3D models are no longer needed, but it does not hurt to keep them supported.

Details on the Trackball Rotation

The figure below illustrates how to translate mouse movement into a rotation axis and angle. m_0 and m_1 are consecutive 2D mouse positions. These positions define two locations v and w on an invisible 3D sphere that fills the rendering window. Use their cross product as the rotation axis `a = cross(v,w)` and the angle between v and w as the rotation angle.



Horizontal mouse movement exactly in the middle of the window should result in a rotation just around the y-axis. Vertical mouse movement exactly in the middle of the window should result in a rotation just around the x-axis. Mouse movements in other areas and directions should result

in rotations about an axis a which is not parallel to any single coordinate axis, and is determined by the direction the mouse is moved in.

Once you have calculated the trackball rotation matrix for a mouse drag, you will need to multiply it with the object-to-world transformation matrix of the object you are rotating. For step by step instructions, take a look at this tutorial. Note that the tutorial was written for Windows messages, instead of GLFW mouse events. This means that you'll need to replace the "CSierpinskiSolidsView::OnLButtonDown", "CSierpinskiSolidsView::OnMouseMove", etc. with an appropriate GLFW equivalent.

To help you better picture what to do, here is a line by line commented version of the trackBallMapping function:

```
/* The CPoint class is a specific Windows class. Either use separate x
and y values for the mouse location, or use a Vector3 in which you ignore the z coordinate. */
Vec3f CSierpinskiSolidsView::trackBallMapping(CPoint point)
{
    // Vector v is the synthesized 3D position of the mouse location on the trackball
    Vec3f v;
    // this is the depth of the mouse location: the delta between the plane through
    // the center of the trackball and the z position of the mouse
    float d;
    // this calculates the mouse X position in trackball coordinates, which range from -1 to +1
    v.x = (2.0*point.x - windowSize.x) / windowSize.x;
    // this does the equivalent to the above for the mouse Y position
    v.y = (windowSize.y - 2.0*point.y) / windowSize.y;
    // initially the mouse z position is set to zero, but this will change below
    v.z = 0.0;
    // this is the distance from the trackball's origin to the mouse location,
    // without considering depth (=in the plane of the trackball's origin)
    d = v.Length();
    // this limits d to values of 1.0 or less to avoid square roots of negative
    //values in the following line
    d = (d<1.0) ? d : 1.0;
    // this calculates the Z coordinate of the mouse position on the trackball,
    // based on Pythagoras: v.z*v.z + d*d = 1*1
    v.z = sqrtf(1.001 - d*d);
    // Still need to normalize, since we only capped d, not v.
    v.Normalize();
    return v;  // return the mouse location on the surface of the trackball
}
```

Part Five Define Materials

In order to render the 3D models as realistically as possible, write shaders to render them with the Phong illumination model and per pixel lighting. Per pixel lighting means that you have to do all of your light reflection calculations in the fragment shader.
For the shader to work, you will have to define light sources as well as different materials for each of your 3D models. Write vertex and fragment shaders to support the features listed below. In this part of the project, you should use fixed positions for the lights and all other lighting parameters. But know that later you're going to have to make some of these parameters user modifiable, so you may want to already introduce variables instead of using constant values.

Assign each of the 3D model files different material properties, following the instructions below. Each object should have a different color of your choice.

What to define:

- One model with very shiny, no diffuse reflection
- One model with only diffuse reflection, with no shininess
- One model with significant diffuse **and** specular reflection component
- Enable switch between normal coloring and Phong illumination model with key 'n'

Rendering Mode Switch

Support keyboard key 'n' to switch between the rendering modes of normal coloring and your new Phong illumination model.

- Normal coloring is useful to keep around so that you can check if your surface normals have been calculated correctly. Normal coloring should work just as you implemented it in assignment one, except now you render the entire 3D model (all triangles, not just the vertices) with normal shading.
- In Phong render mode, you render your 3D models more realistically, determined by their materials and the type of light source shining on them.

Part Six Define Lights

You will need to create two separate light sources: a directional light and a point light. Use two C++ classes to define the parameters for the two types of light sources. Your class implementation is very flexible, but should at least contain these variables:

What to define:

Directional Light Properties

- color (vec3)
- direction (vec3)

Point Light Properties

- color (vec3)
- position (vec3): world coordinate of the light source
- Notice: use **linear** attenuation when calculating the light intensity

Part Seven Interactive Light Controls

To test out the effect of your light sources, make the light sources movable with the mouse, and add keyboard commands for certain illumination parameters as described below.

You will have the following components implemented:

- Light Scene Selection
- Light Representation (indicate the location of point light)
- Light Movements

Light Selection

Make the lights selectable with the number keys.

What to render:

- Keys ‘1’, ‘2’ and ‘3’ are used to switch among 3 scenes.
- Key '1': Directional light(rotating) only. Mouse movement does not apply to anything.
- Key '2': Directional light(rotating) + point light. Mouse movement applies to point light.

- Key '4': Toggling directional light on/off on all scenes.
- Key '0': switch the control between the 3D model and spot light.

Light Representation

What to render:

- **Directional light**: No representation needed.
- **Point light**: Draw the sphere model provided in the color of that point light.

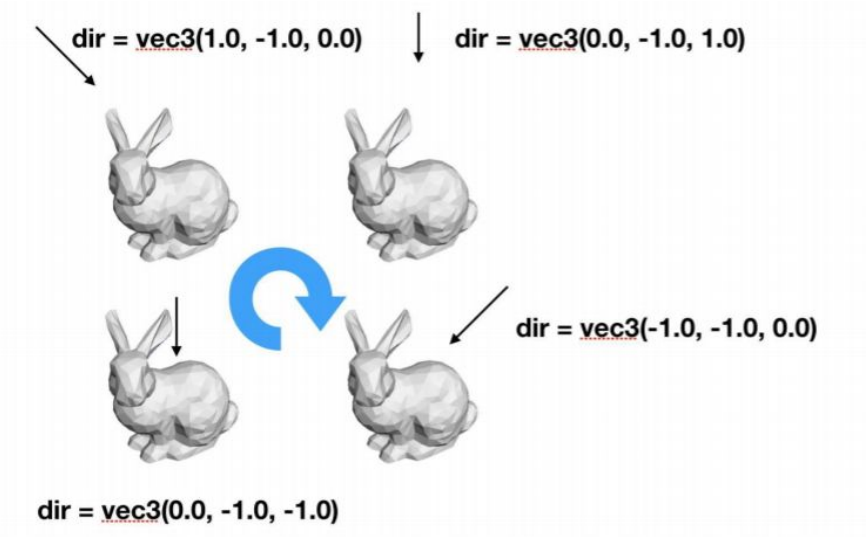
Representation Notices:

- The sphere needs to be scaled down on screen, just big enough to be clearly visible. Use a uniform scale matrix to scale it down as part of the model matrix transformation.
- To draw sphere in the color of the light source. You should use the light color as the light's **ambient** reflection value, and **DO NOT** use diffuse or specular reflection (they wouldn't work properly because the light source is inside of the light source geometry). Or somehow tell the shader you're rendering light source representation, and output that color directly.

Light Movements

What to render:

- Directional Light
    - The mouse does not affect this light source. The directional light should be “hovering” over the object, i.e rotating around positive y-axis.



- Point Light
    - Mouse control: same as **rotation** in **Part four**. For your convenience, here is a copy: **Rotation**: While the left mouse button is pressed and the mouse is moved, rotate the light about the center of your graphics window (origin of world coordinate). We will refer to this operation as "trackball rotation".
    - Notice that light intensity should be visibly different on the model as the light is moved closer and farther from the model surface.

PURE FUN Spot light

Notice: this part is only provided to those who want to work a little bit more with lighting. Completing **this part of work will NOT give you any extra credit, and you will NOT be graded** on this part.
Try to enable key interactions for a spot light as well. This time you get to choose whatever key you want to work with!

Spotlight

- Should do everything the point light does, plus:
- Somekey should make the spot wider/narrower.
- Some other key should make the spot edge sharper/blurrier.

Notice: The light intensity should be visibly different on the model as the light is moved closer and farther from the model surface, with an even more visible effect than with the point light (it uses linear vs. quadratic attenuation).

Notes and Tips

Learn OpenGL provides vertex and fragment shader code, as well as the corresponding C++ code for different lighting parameters. The shader does almost exactly what you need. Find out how it differs from the equations given on the discussion slides for this homework project and make necessary modifications.

Lighthouse 3D also provides excellent tutorials for the necessary shaders: for directional lights, point lights and spot lights.

This tutorial provides very useful information on light parameters in chapters 6 and 7. An additional tutorial on different light types is provided in chapter 8.

This table of material properties may inspire you to select interesting material parameters. But note that there are specific requirements for the materials you use, which make it necessary to eliminate one or more reflection components (ambient, diffuse, specular) when you define your own materials.

Submission

DUE DATE: Feb 14 12:59 pm

Please zip all your source code files and submit the zip file on Gradescope before the due date. Source code files are all .h files and .cpp files, plus the vertex shader and fragment shader. Please **do not** include any OBJ files, .xcodeproj, or .sln file.

Friendly reminder, the due date is **before** the grading session from 1:00 pm to 3:00 pm on the same day. Otherwise there will be a 30% penalty for late submission. Notice that if you submit the code on time, but did not attend the grading session on Feb 14, there will be that 30% penalty as well. Please attend the grading session on time.