CSE 167 Assignment 1: Rendering Point Clouds

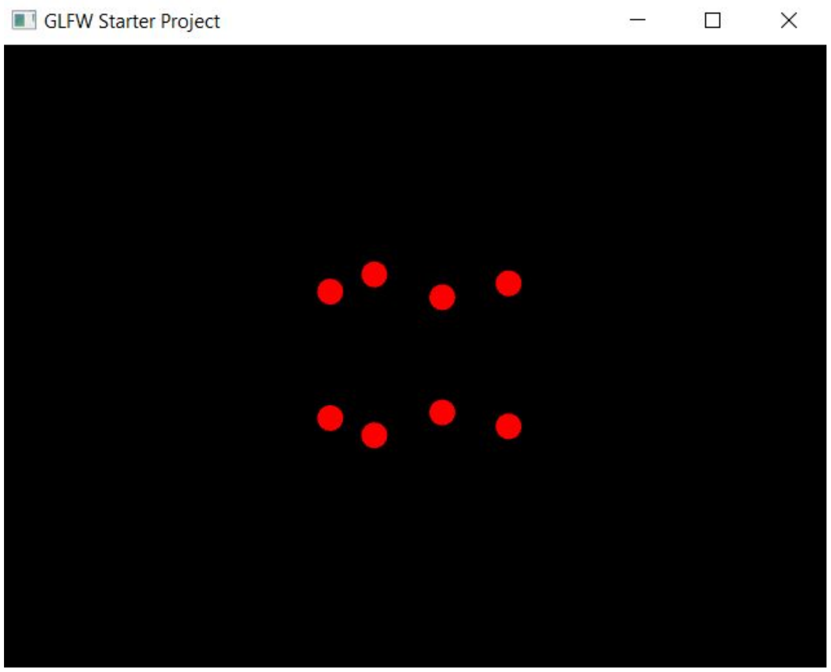
DUE: Friday, Jan 31st at 12:59 PM

The goal of this project is to further understand how we work with vertex data loaded in from a file and render them to the screen. This assignment will get you acquainted with how to work with modern OpenGL, giving you the chance to modify shader programs meant to run on your GPU. It will also build your understanding of how points in space are transformed to both move and scale your points and transform them from 3D points in object space, to points in image space.

Besides getting accustomed with the linear algebra involved in rendering 3D scenes, this project will get you familiar with the tools and libraries we set up in Assignment 0. These are **GLFW** (which handles creating windows and taking input), **GLEW** (which figures out which OpenGL extensions are supported), and **GLM** (your vectors, matrices, linear algebra functions etc).

1. Getting Your Development Environment Ready

You should start this assignment by modifying the starter code from Assignment 0. This part should be done in Assignment 0 already. If not, please catch up. Make sure to download and add the **PointCloud.h/cpp** files and the **RasterizerQuad.h/cpp** files to your solution as well as download the new **Window.h and Window.cpp** files to overwrite the old Window files. Finally, make sure to add **RasterizerQuad.vert/frag** to your project. After making the changes, running the program again should give you the vertices of a cube that is spinning.



2. Reading 3D Points from Files (10 pts)

A point cloud is a 3D collection of points, representing a 3D object. These point clouds are often acquired by laser scanning, but can also be acquired with the Microsoft Kinect and special software, or by processing a large number of photographs of an object and using Structure from Motion techniques (see Microsoft Photosynth or Autodesk 123D Catc).

In this project we're going to render the points defined in OBJ files. Note that OBJ files are normally used to define polygonal models, but for now we're ignoring that and use the vertex definitions to render points, ignoring all connectivity data. OBJ files are 3D model files which store the shape of an object with a number of vertices, associated vertex normals, and connectivity information to form triangles. Wikipedia has excellent information on the OBJ file format. The file format is an ASCII text format, which means that the files can be viewed and edited with any text editor, such as Notepad.

The PointCloud class has the functionality to render hardcoded cube vertices as points. **Extend its functionality by adding a parser to it that reads in the vertices and normals defined in the OBJ files**. It should be a simple 'for' loop in which you read each line from the file, for instance, with the fscanf command. Your parser does not need to do any error handling - you can assume that the files do not contain errors.

Use your parser to load the vertices from the given Bunny and Dragon obj files.

The above files are already centered about the origin in object space, so you should not worry about centering them when writing your parser for now.
The files provided in this project only use the following three data types (other OBJ files may support more): v for vertex, vn for vertex normal, f for face.

The general format for vertex lines is:
v v_x v_y v_z r g b

where v_x, v_y, v_z are the vertex x, y, and z coordinates and are strictly floats. The values r, g, b define the color of the vertex, and are optional (i.e. they will be missing from most files). Like the vertex coordinates, they are strictly floats, and can only take on values between 0.0 and 1.0.

All values are delimited by a single whitespace character.

The general format for normal is the same as for vertices, minus the color info.

In summary:

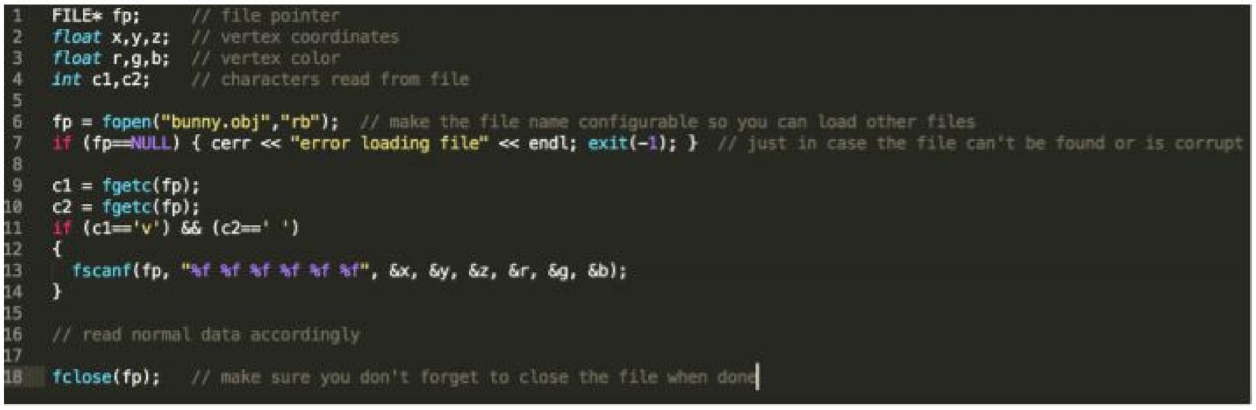
- v: 'vertex': followed by six floating point numbers. The first three are for the vertex position (x, y, z coordinate), the next three are for the vertex color (red, green, blue) ranging from 0 to 1.
  Example: v 0.145852 0.104651 0.517576 0.2 0.8 0.4

- vn: 'vertex normal': three floating point numbers, separated by white space. The numbers are for a vector which is used as the normal for a triangle.
  Example: vn -0.380694 3.839313 4.956321

Lines starting with a '#' sign are comments and should be ignored.

In this homework assignment, you only need to parse for vertices and vertex normals, which are those lines of the file starting with a 'v' and 'vn'.

Write your parser so that it goes through the OBJ file, line by line. It should read in the first character of each line and based on it decide how to proceed, i.e., ignore all lines which do not start with a 'v' or 'vn'. The vertex definitions can be read with the fscanf command. Here is an example:

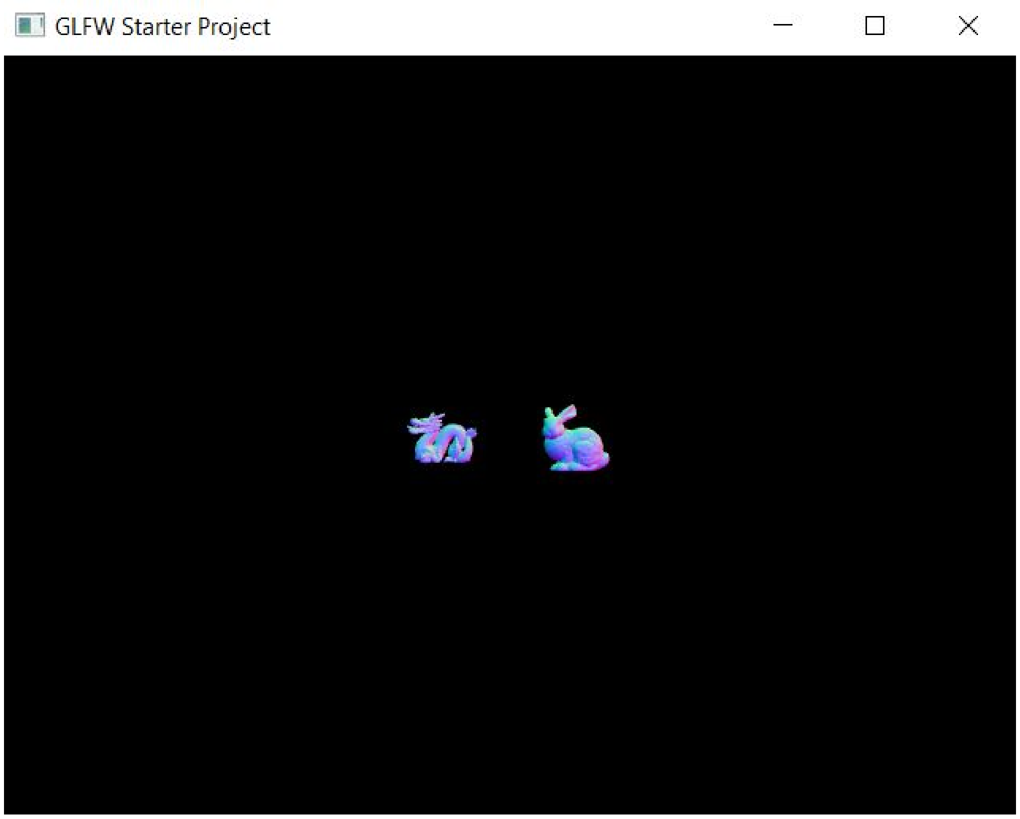


```
FILE* fp;     // file pointer
float x,y,z;  // vertex coordinates
float r,g,b;  // vertex color
int c1,c2;    // characters read from file

fp = fopen("bunny.obj","rb");  // make the file name configurable so you can load other files
if (fp==NULL) { cerr << "error loading file" << endl; exit(-1); }  // just in case the file can't be found or is corrupt

c1 = fgetc(fp);
c2 = fgetc(fp);
if (c1=='v') && (c2==' ')
{
  fscanf(fp, "%f %f %f %f %f %f", &x, &y, &z, &r, &g, &b);
}

// read normal data accordingly

fclose(fp);   // make sure you don't forget to close the file when done
```

Load all two models in at startup.

3. Rendering the Points with OpenGL (20 pts)

To display the vertices you loaded, use the provided starter code. It contains hooks for rendering points with OpenGL, which are ready for you to use.



Please render two meshes on the same screen (similar to the image shown above).

Just like the cube spins during display, make your OBJ models spin as well. **THE BUNNY AND DRAGON SHOULD ROTATE IN DIFFERENT DIRECTIONS (i.e. one about the y-axis, the other about the z-axis)**

The points should be colored with normal coloring: use the vertex normals you parsed from the file, and map them into the range of 0 to 1. The x coordinate of the normal should map x to red, y to green, z to blue. To assign the color, modify shader.vert to read in vertex normals, pass the vertex normal to the fragment shader, and assign fragColor the value of the normal. If you want more information on OpenGL shaders, learnopengl has an article on them: https://learnopengl.com/Getting-started/Shaders

4. Manipulating the Points (20 pts)

Modify and use the updatePointSize() command in PointCloud to allow the user to adjust the size of the points with the 'p' and 'P' keys (for smaller and larger points, respectively).

Once a model has been loaded, support the following keyboard commands to manipulate it:

- ‘p'/’P': scale down/up the size of each point as mentioned
- ‘a'/’d': move left/right (along the x axis) by a small amount
- 'w'/'s': move down/up (along the y axis) by a small amount
- 'z'/'Z': move into/out of the screen (along the z axis) by a small amount
- 'c'/'C': scale down/up (about the model's center, not the center of the screen)
- 'r': reset position (move objects back to center of screen, without changing orientation or scale factor
- 'R': reset orientation and scale factor, but leave the objects where they are

Tweak the values for changes in position and size so that you can manually center the OBJ models in the window and scale it to the size of the window during grading.

All transformations must be in world space, not object space.

Note: keep the objects spinning at all times during manipulation (Note this rotation is in object’s local space, not about the center of the world)!. Also, all transformations should be relative to the viewer, not to the objects' local space. In other words, if an object has spun 180 degrees, then pressing the key to move the objects left should still make the projected image of it be shifted to the left, not to the right. Also note that the operations have to be cumulative, which means that every time you move the objects they move from where it was moved to before.

Note that we will allow the objects to overlap when you scale them. **HOWEVER, be wary of scaling the object down too much.** Scaling the object down too much will cause you to zero out your model matrix, making your objects disappear!

4. Rendering Points through Rasterization (50 pts)

In this part of the assignment you need to write code to display the point cloud using your own rasterization code. This is important: in this part of the project we're going to render into a block of memory located in a RasterizerQuad object. The provided starter code (specifically, RasterizerQuad.h/cpp) has 2 functions of interest that you need to fill in order to rasterize your point cloud and display it to the screen.

- **clearBuffers()** for clearing your **pixelBuffer** and **zBuffer**
- **rasterize()** for transforming your point clouds and filling in your pixel buffer

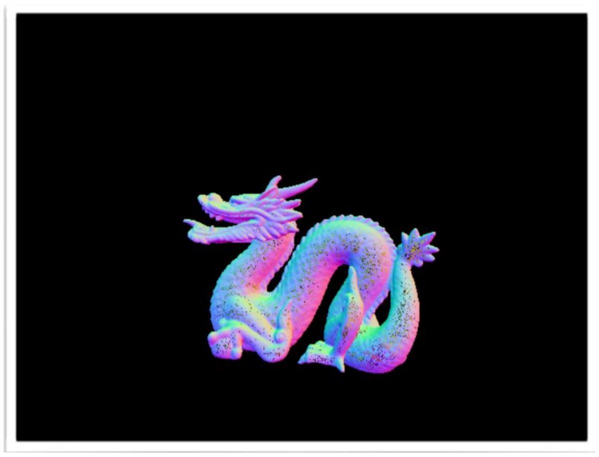
We provide you with a function called **drawPoints** to draw your point into the pixel buffer when you transform them. Getting the x, y, and z coordinates to pass into drawPoints is going to require that you evaluate the complete vertex transformation:

$$\mathbf{p' = D * P * C^{-1} * M * p}$$

Part of the starter code (RasterizerQuad.h/cpp) contains code to render a 2D RGB array represented as a 1D array of floats into your window. Use this code and modify it to render your point models. **You should not change anything else in the RasterizerQuad code but the two functions mentioned above.** The models have to be rendered in the same place as when you render them with OpenGL. Add support for the 'm' key to switch between the two rendering modes. Use normal coloring again for the points, just like in OpenGL mode. Also, just like in OpenGL mode, support the 'p'/'P' keys to change the point size. For bigger points, draw squares of integer widths, such as 2x2, 3x3, 4x4,…,nxn。

YOU ONLY NEED TO RENDER ONE OF THE OBJs IN YOUR RASTERIZER.

Example



6. Submission

Upload your code in a zip file to Gradescope by January 31st 12:59 PM

ZIP file should include all the h/cpp and .frag/.vert files. DO NOT ZIP THE OBJ FILES!

Grading Session: January 31st 1:00 pm - 3:00 pm