Project1F17

From Immersive Visualization Lab Wiki

Contents

Homework Assignment 1: Rendering Point Clouds

In this project the goal is to read a number of 3D point positions from a file and render them onto the screen.

Besides becoming familiar with the linear algebra involved in rendering 3D scenes, this project will get you familiar with the tools and libraries that will be your friends throughout the quarter. These will include GLFW and GLew, as well as GLm.

1. Getting Your Development Environment Ready (10 Points)

We recommend that you start by getting your development software ready. More information on this is here.

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

A point cloud (http://en.wikipedia.org/wiki/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 (https://photosynth.net/) or Autodesk 123D Catc (http://www.123dapp.com/catch)).

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 (http://en.wikipedia.org/wiki/Wavefront_.obj_file) 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 a any text editor, such as Notepad.

Write a parser for the vertices and normals defined in OBJ files. It should be a simple 'for' loop in which you read each line from the file, for instance with the fscanf (http://www.cplusplus.com/reference/cstdio/fscanf/) command. Your parser does not need to do any error handling - you can assume that the files do not contain errors. Add the parser to the starter code.

Use your parser to load the vertices from the following OBJ files and treat them as point clouds:

- Bunny
- Dragon (http://www.calit2.net/~jschulze/tmp/DragonF14.zip)
- Bear (http://www.calit2.net/~jschulze/tmp/BearF14.zip)

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 (http://www.cplusplus.com/reference/cstdio/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

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 three models in at startup.

Grading:

- -5 for any errors in the parser

3. Rendering the Points with OpenGL (20 Points)

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. Change the Display callback and replace the call to render the cube to render the OBJ object - don't call cube.draw() but call the respective OBJ object's draw command.

Use function keys F1, F2 and F3 to display the three models, respectively. Only one 3D model should be displayed at a time. (4 points per model)

Just like the cube spins during display, make your OBJ models spin as well.

The points should be colored with normal coloring: use the vertex normals you parsed from the file, and map them into the range of zero to one. The x coordinate of the normal should map to red, y to green, z to blue. (4 points)

Use the glPointSize() command to allow the user to adjust the size of the points with the 'p' and 'P' keys (for smaller and larger points, respectively). (4 points)

Grading:

- -3 if data sets load every time a function key is pressed.

4. Manipulating the Points (20 Points)

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

- 'x'/'X': move left/right (along the x axis) by a small amount (3 points)
- 'y'/'Y': move down/up (along the y axis) by a small amount (3 points)
- 'z'/'Z': move into/out of the screen (along the z axis) by a small amount (3 points)
- 's'/'S': scale down/up (about **the model's** center, not the center of the screen) (3 points)
- 'o'/'O': orbit the model about the window's z axis by a small number of degrees per key press, counterclockwise ('o') or clockwise ('O'). The z axis crosses the screen in the center of the window. Orbiting means rotating about the center of the window, as if it it sits on a disk spinning about that point. (4 points)
- 'r': reset position (move object back to center of screen, without changing orientation or scale factor (1 point)
- 'R': reset orientation and scale factor, but leave the object where it is (1 point)

Tweak the values for changes in position and size so that you can manually center each OBJ model in the window and scale it to the size of the window during grading. All translations must be in world space, not object space.

Note: keep the object spinning at all times during manipulation. (2 points) Also, all translations 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 object left should still make the projected image of it be shifted to the left, not to the right.

Grading:

- half the points if manipulation only works in one of the two modes (OpenGL or rasterizer)

5. Rendering the Points through Rasterization (40 Points)

We recommend that you hold off on working on this part of the project until after lecture 3 in order to have all the required math covered in class. The vertex transformation will be covered in lecture 4, and also in discussion. The material is explained on the slides of the second half of the deck for lecture 3.

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. We provide starter code (http://ivl.calit2.net/wiki/images/7/78/GLFW_Rasterizer.cpp) with the functionality to make this memory show up on the screen as pixels.

This is going to require that you evaluate the complete vertex transformation: p' = D * P * C-1 * M * p

As part of the starter code comes a file called rasterizer.cpp which contains code to render a 2D RGB array into your window. Use this code and modify it to render your point models. 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.

Use a background color other than the default of green.

Grading:

- -10 if rasterized model doesn't completely match the position of the OpenGL rendered one
- -10 if model isn't correctly rendered when partially off screen
- -15 if point size can't be changed
- -5 if the program crashes as a result of any manipulation or point size change
- -5 if any bounds aren't checked for correctly
- -3 if background color has not been changed from green
- -5 if not all three models can be loaded
- -5 if normal coloring is not used or not working correctly

6. Extra Credit (10 Points)

For extra credit, you can address one or both deficiencies of the algorithm so far, each will get you 5 points of extra credit.

1) Currently, all points are the same size in image space. However, in reality if the points were little squares, they would get rendered as smaller squares the farther they are from the viewer. Adjust the point size accordingly. Note that you will need to calculate the range of depths the points are mapped into before you render the model.

2) You will notice that the bigger the points the more often they overlap with others the wrong way: points behind others will occlude them if they get rendered later. To fix this, implement a z-buffer and add a z buffer test for every pixel of every point.
