CS148 Assignment 4

An Introduction to OpenGL

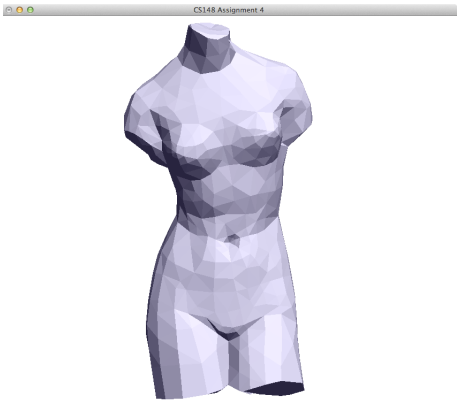


Figure 1: *Flat shaded Venus, an example of built-in lighting in OpenGL (see section 4).*

Goals: *Learn the basics of rendering in OpenGL: rasterizing basic primitives, loading and rendering of 3D geometry using simple wireframes, texture mapping, calculating and using normals for lighting effects.*

This assignment will begin with basic 2D rasterization using OpenGL, and add functionality for camera motion, keyboard input, and the rendering of 3D objects. You will also wrap a texture onto a surface using texture coordinates, and flat shade an object by adding a light to the scene and specifying the normals of each triangle.

In order for you to learn the process of setting up an OpenGL project from scratch, there is no starter code for this assignment.

Hang on to your hats!

1 Rasterizing simple shapes in 2D

Drawing with basic 2D primitives is a great way to test your understanding of the OpenGL setup. Using the example code in the Redbook Ch. 1 as your template, recreate the image in Fig 2.

To facilitate grading, please save the resulting code in a separate file named `main1.cpp`. You will also be asked to turn in a screenshot; if you would like to generate the screenshot directly from your OpenGL code, you may use the example function given in Appendix A.

Figure 2: *Using simple primitives, recreate this 2D image.*

2 Displaying 3D geometry

Now that you have a basic OpenGL program running, it's time to render something a little more interesting. After setting up your camera and implementing some basic callback functions, you'll be ready to load some 3D geometry. We will be importing triangle meshes from OBJ files, a common and simple mesh format.

2.1 3D Camera

Before we can render any 3D objects, we'll have to set up a 3D camera. While you are free to use any camera you prefer, we suggest the perspective camera (see the discussion of `glFrustrum` and `gluPerspective`, Redbook, pp. 153-6).

To test your camera, you should place some simple geometry (for instance, a polygon) within the area you believe to be the viewing volume. When it appears as you expect, you'll be ready to render more complex things!

2.2 Navigation callbacks

Another key part of rendering a 3D environment is providing for user navigation. Fortunately, the OpenGL transformation matrix stack provides a straightforward way to specify rotations, translations, and scalings of various objects in the scene.

Using keyboard and mouse callback functions (Redbook Ch. 1 and p. 734), simple transformations (`glRotatef, glTranslatef`, etc), and your camera transformations (`gluPerspective, glFrustum`, etc) make your camera responsive to user input. At a minimum, the user should be able to rotate around and zoom in and out from an object.

Hint: Think about how to simulate a transformation of the camera by instead transforming the objects being rendered.

2.3 Loading 3D meshes from OBJ files

Next, you will import more interesting geometry from `.obj` mesh files. The OBJ file format is a standard ASCII format that supports both polygonal and free-form 3D geometry. The supported primitives include lines, polygons, and free-form curves and surfaces. Lines and polygons are represented by vertices, while curves and surfaces are usually represented by control points. To read more about the complete grammar rules of the OBJ format, we refer you to Wikipedia[1].

For this assignment, you are only required to parse a small number of these rules. In particular, your viewer will need to be able to parse a triangle mesh, optionally containing texture coordinates. Sample meshes can be found on the course webpage[2].

An OBJ format file containing a triangle mesh usually includes vertex locations (v), texture coordinates (vt), normals (vn), and faces (f). Here is an example:

```
# 'v' indicates that a line contains a vertex:
v .0 .0 .0
v .0 .0 1.5
v 1.5 1.5 1.5

# 'vt' indicates texture coordinates:
vt .0 .0
vt .8 .0
vt .8 .5

# 'vn' is used to specify normals:
vn .707 .707 .0

# 'f' is used for faces.
f 1/1/1 2/2/1 3/3/1
```

The above is a very simple OBJ file; it contains a single triangle. Notice:

- Comment lines begin with a hash mark (#)
- Blank space and blank lines can be freely added to the file to aid in formatting and readability.
- Vertices, normals, etc. are specified in faces using indices that start at 1.

This file contains two types of data: vertex data (vertex locations, texture coordinates, and normals) and face data (triangles). Formally, the syntax for vertex data is:

```
v x y z
vt u v (or, vt u v w)
vn dx dy dz
```

Here, the character `v` indicates that the floating point values `x y z` define the location of a vertex. Similarly, `vt` defines a texture coordinate `u v` or `u v w`, and `vn` a normal. Please notice that in an OBJ file, the number of elements for different vertex data components need not be the same. This means that the number of vertices and the number of vertex normals (or texture coordinates) may differ. For example, this file contains three vertices yet only one normal (which is shared by each of the three vertices).

The face (`f`) data entries specify which vertex elements are combined in order to form faces (for this assignment, we will only be using triangular faces.) As you can see, there are a few different ways to specify a face:

```
f v1 v2 v3
f v1/t1 v2/t2 v3/t3
f v1//n1 v2//n2 v3//n3
f v1/t1/n1 v2/t2/n2 v3/t3/n3
```

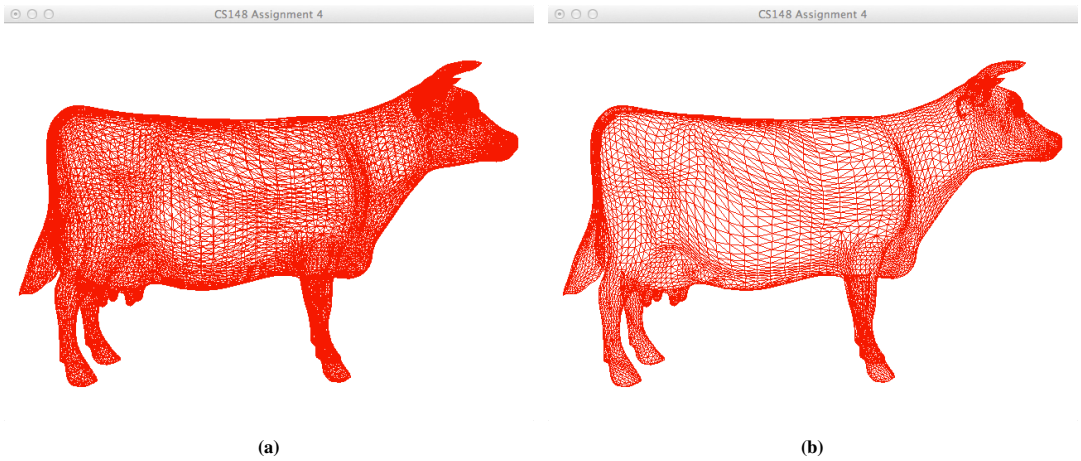
Here, the first line shows the simplest way to specify a face: by providing the indices of three vertices (`v1 v2 v3`). Optionally, you might provide texture coordinates (`vt`) via `f v1/t1 ...`) or normals (via `f v1//n1 ...`) The extra slash is not a typo; it indicates that we've skipped the texture coordinate. Finally, we could specify both texture coordinates and normals using `f v1/t1/n1...`, etc. No spaces are permitted before or after the slash.

Your task: Write a simple parser routine to import vertices, texture coordinates, and face information from an OBJ file (vertex normals may be discarded.) You will need to devise your own data structure(s) to store this information; think about how to do so in a way that will make your calls to `GL_TRIANGLES` (part 2.4) easier!

Hint: The colored triangle (`.tri`) format used in HW3 was based on the OBJ file format.

[1] http://en.wikipedia.org/wiki/Wavefront_.obj_file

[2] https://web.stanford.edu/class/cs148/assets/hws/hw4_meshes.zip



(a) (b)

Figure 3: *Wireframe renderings. (a) Geometry can be difficult to resolve when drawing the outline of every triangle; (b) removing backface triangles greatly improves legibility.*

Hint: If you are having trouble parsing the face data, you may find the `sscanf` function handy[3].

2.4 Displaying 3D Wireframes with backface culling

One simple way to see if your geometry has been loaded properly is by drawing just the outline of each triangle. This is known as a *wireframe* rendering, and allows you to see exactly how triangles adjoin. To draw only the outlines of each triangle, you can change the way OpenGL renders polygons via `glPolygonMode` (Redbook, p.61). For instance,

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

will instruct OpenGL to draw both front and back facing polygons with a filled interior. Similarly,

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

will instruct OpenGL to draw only the outline of the polygon. Of course, if you are drawing both front and back facing polygons, you will be able to see the outlines of back-facing triangles through the interiors of the front-facing polygons. The clarity of a wireframe rendering can be improved by removing or *culling* these backwards-facing triangles.

To instruct OpenGL to start culling, issue the command:

```
glEnable(GL_CULL_FACE);
```

To specify which faces to cull, you will have to call `glCullFace(GLenum mode)` with the appropriate mode. You can read more about culling in the Redbook, pp. 61-2.

Your task: Render an OBJ mesh of your choice with backface culling. Please submit the code to do so in a separate file named `main2.cpp`, along with a screenshot. Before taking the screenshot, you should use your mouse and keyboard callbacks from Section 2.2 to transform the mesh into a nice orientation.

3 Texture Mapping

Now that you can import, display, and interact with 3D geometry, you're ready to experiment and have fun with basic shading and lighting effects. In this section, you will implement texture mapping.

If you are unclear about the texture mapping process, you might want to review Ch. 11 in Shirley before proceeding further. The Redbook (Ch. 9) provides lots of information about advanced techniques; however, only the first 15 pages or so are needed to understand and perform simple 2D texture mapping.

Loading the proper image data into your texture (i.e., loading the bytes representing colored pixels) can be tricky. Here are a few lines of code that will use the `SimpleImage` class from HW3 to do so:

```
SimpleImage texPNG(``cs148-cube.png'');
int w = texPNG.GetWidth();
int h = texPNG.GetHeight();
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_FLOAT, texPNG.GetData());
```

[3] http://www.cplusplus.com/reference/cstdio/sscanf/

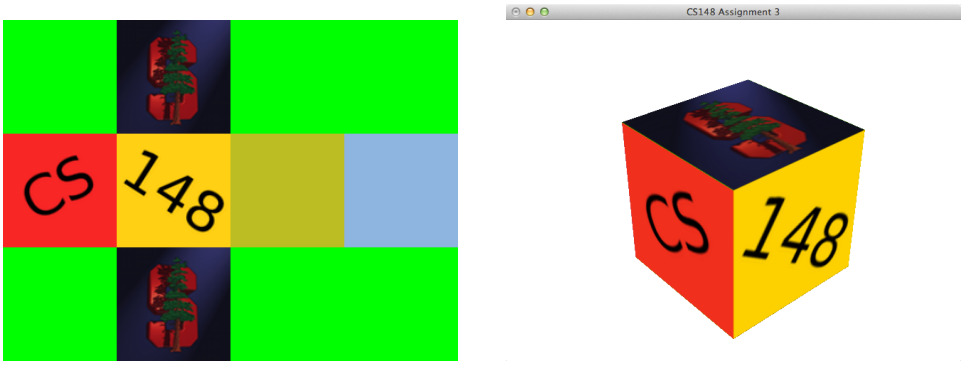


Figure 4: *A simple 2D texture is wrapped around a cube, using the geometry and texture coordinates specified in the provided* `tex-cube.obj`.

However, you are not required to use SimpleImage to load your texture.

Your task: Using the texture coordinates you parsed in Section 2.3, texture map `tex-cube.obj` using the image `cs148-cube.png` to create a result very similar to that in Figure 4. Both mesh and texture can be found in the collection of meshes linked on the course page. Save your code in a separate file called `main3.cpp` and take a screenshot.

4 Flat shading

Flat shading is the simplest way to approximate 3D lighting effects. When applying shading to a surface, OpenGL will compute the illumination of that surface using a provided normal. (Remember, the normal tells OpenGL which way the triangle is facing, and therefore how much light is striking it and whether or not that light will be visible to the camera.)

In order to shade your geometry, you will first need to place at least one light in your scene. The Redbook Ch. 5 reviews how to enable and place lights. Remember to check that your light is outside of your geometry!

You will also need to compute normals for each face (many OBJ meshes, including most of those provided in the course webpage, do not include normals.) In fact, flat shading is called "flat" because a single normal is specified for each face. For instance, when a normal is given via `glNormal3f` immediately before the calls to `glVertex3f` that specify the three vertices of a triangle, that normal will be applied identically to each vertex. Provided that lighting is enabled, OpenGL will then automatically use the lights in the scene to compute the amount of illumination present on that face.

In order to compute the correct normal for each face of your triangle mesh, you can use the cross product of the vectors defining the two sides of the triangle (see Figure 5). Use care to ensure that the normal is facing the correct way, and that it is normalized. Remember, faces specified as (v_0, v_1, v_2) are oriented in an anticlockwise manner by convention. Shirley Ch. 2 contains a review of the vector operations you will need in order to compute the normal of a face.

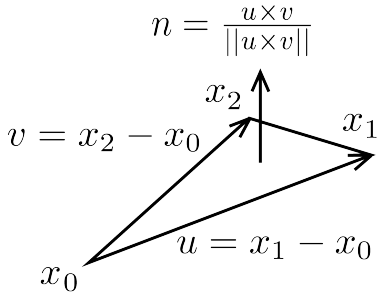


Figure 5: *For any triangle, the normal can be calculated by taking the cross product between the vectors defining two of the triangle's edges.*

Your task: Create a flat shaded model by computing and specifying normals for each mesh face. You will need to enable lighting and place at least one light in your scene, and the result should look similar to the teaser image (`venus.obj`). Save your code in a separate file called `main4.cpp` and take a screenshot.

5 What to turn in

To submit your assignment, please bundle the following into a `.zip` file:

1. A PDF or .rtf file containing:
   - Part 1: A screenshot of your recreation of figure 2.
   - Part 2.4: A screenshot of your wireframe mesh.
   - Part 3: A screenshot of your texture mapped cube.
   - Part 4: A screenshot of your flat shaded mesh.
2. `main1.cpp, main2.cpp, main3.cpp, main4.cpp`
3. Any "accidental art"– weird images, mistakes, &c.– demonstrating challenges you faced. Include a brief explanation of what went wrong and how you fixed it.

4. A text file README containing anything else we should know: names of classmates you worked with, help received, references, etc.

Extra credit will be awarded at the grader's discretion for additional renderings:

1. More advanced texture mappings.
2. Advanced user interactions
3. ... any other creative rendering you can think of!

Please include screenshots and the code necessary to recreate your images.

Your `.zip` file can be submitted using the form on the students' section of the course webpage.

6 Appendix A: Taking a screenshot

Here is a method you may find useful if you would like to automatically generate a screenshot of your current window buffer:

```
void screenshot(){
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);

    float out[3 * w * h];
    RGBColor BG(0,0,0);
    SimpleImage shot(w, h, BG);

    glReadPixels(0, 0, w, h, GL_RGB, GL_FLOAT, &out[0]);
    for (int y = 0; y < h; ++y){
        for(int x = 0; x < w; ++x){
            int index = (3 * w * y) + 3*x;
            float red   = out[index];
            float green = out[index + 1];
            float blue  = out[index + 2];
            shot.setPixel(x,y,RGBColor(red, green, blue));
        }
    }
    shot.WritePNG("screenshot.png");
}
```