Lab 3 OpenGL I

CS123 Fall 2010

1 Introduction

From now on, our weekly labs will focus on the use of OpenGL. While the lectures and projects will let you get a deeper understanding of the inner workings of graphics, our labs will now focus on the more general use of OpenGL. The culmination of these next 8 labs will leave you with a working knowledge of OpenGL, which you may want to take advantage of in your final project.

This week we present an overview of some of the features OpenGL offers.

2 OpenGL Basics

2.1 The CPU and the GPU

The CPU is short for Core Processing Unit and the GPU is short for Graphics Processing Unit. Traditionally, the CPU has been a mostly MIMD (Multiple Instruction Multiple Data) architecture, whereas the GPU uses SIMD (Single Instruction Multiple Data) instructions. GPUs incorporate hardware instructions for vector operations (like dot products and matrix multiplication) that require many instructions on traditional CPUs. GPUs offer specialized data primitives (like vectors and matrices) as well. Although the boundaries between the CPU and GPU are becoming blurry (Intel now offers certain SIMD instructions and vector primitives on its latest processors), they are still significantly distinct platforms which are most suitable for different applications.

Many graphics algorithms are *embarrassingly parallel*. For example, the lighting equation (which we'll learn about in detail later in the semester) can be evaluated independently on each pixel in a given image. Since GPUs are designed with tens, hundreds, or even thousands of processing cores (known as *stream processors*), they tend to work really well for computer graphics applications. There are many competing GPU programming APIs, such as OpenGL and DirectX, as well as hardware vendor-specific APIs such as NVIDIA's CUDA. This semester, we'll be focusing on OpenGL, because we believe it offers the best cross-platform and cross-hardware functionality available today.

2.2 The Rendering Pipeline

OpenGL has what we call a *rendering pipeline*: a systematic series of steps which are performed on vertex and pixel data. There are various buffers to store data at each step in the pipeline. A diagram of OpenGL's rendering pipeline is shown in Figure 1.

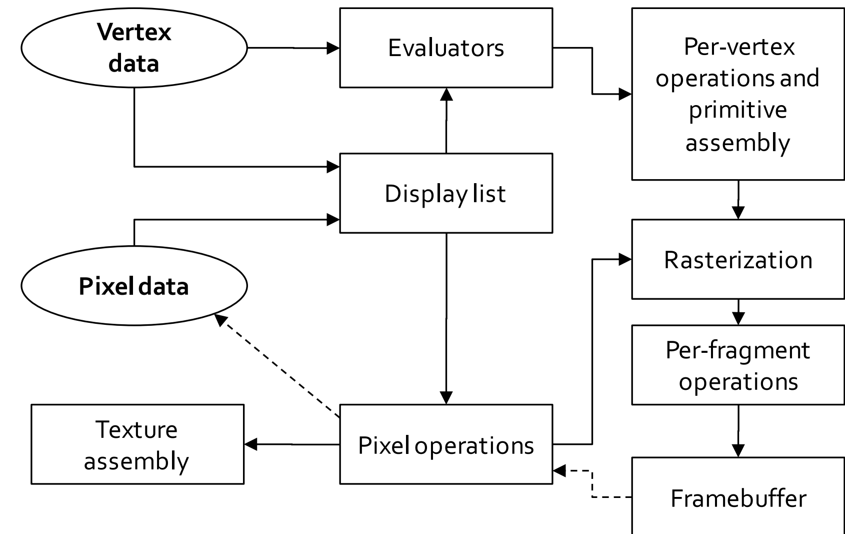

Figure 1. OpenGL's rendering pipeline

We'll learn about the pieces of this pipeline as the class progresses. For now, just be aware that the pipeline exists. The pipeline covers much of OpenGL's high-level architecture.

2.3 Basic Syntax of OpenGL commands

Most OpenGL commands are defined in the <GL/gl.h> header file. OpenGL commands adhere to the C standard (not the C++ standard). You will notice that many of them take pointers, and none of them accept references. OpenGL does not offer classes or inheritance. Although wrappers like GLT exist, we will not be using them as they add unnecessary overhead. An OpenGL command is usually surrounded by a *prefix* and a *suffix*, which explain part of how the method works.

Prefix Everything with begins with `gl`, as you may have noticed. We also use `glu` when using the GLU library commands (more on these later).

Suffix A number and a letter. The number represents how many arguments the method takes in. The letter represents the type of the arguments.

For example, glVertex3f is an OpenGL command that takes in three floating point arguments.

3 OpenGL as a state machine

OpenGL is a state machine. States are set by the user before drawing a scene. The resulting rendering reflects these changes in state.

Two important states are GL_PROJECTION and GL_MODELVIEW, which enable editing of the *projection* or *modelview* matrices. The projection matrix defines how the camera's view volume has been transformed. The modelview matrix defines how the camera and objects have been rotated, scaled or translated in the scene. Every time a vertex is drawn in the scene, its final location in the scene reflects the transformations currently represented by the modelview matrix. The way these vertices get projected to the screen depends on the current projection matrix. Thus, before we draw any vertices we should be in GL_MODELVIEW mode. Before we modify the camera we should be in GL_PROJECTION mode. We can change the mode between GL_MODELVIEW and GL_PROJECTION with a call to *glMatrixMode*. You don't need to worry too much about the projection matrix and view volumes in detail yet; we'll discuss these in class.

There are also binary states such as DEPTH_TEST, LIGHT# and GL_LINE/POINT/FILL that can be turned on and off, typically with the use of *glDisable* and *glEnable*.

There are also settings that have multiple options. One example of such a setting is the shading model, for which you can specify an option to set as a parameter (i.e glStateOption(GL_ANY_OPTION)). We will go over these and a few others throughout the course of this lab.

4 Buffers in OpenGL

OpenGL stores information about a scene in *buffers*, each of which contains a rectangular array of data. The size of the rectangular array is the same size as the image to be displayed. The buffers available to you in OpenGL include the color buffer, depth buffer and frame buffer.

Color Buffer The color buffer stores the color at each of the pixels in the scene.

Depth Buffer A scene is composed of objects that get rendered to the screen. Each of these objects has a certain magnitude of depth in the scene. The depth buffer keeps track of what the smallest depth is in a scene for every pixel. This is used to know which objects should or shouldn't be rendered based on which obscures/occludes which.

Frame Buffer The frame buffer is composed of all other buffers (such as the depth buffer), except for the color buffer. All of the buffers included in the frame buffer represent non-visual information about the scene and they can be used to perform tasks such as hidden surface elimination, antialiasing, stenciling, drawing smooth motion.

You should probably clear your buffer between frames by calling glClear. If you don't clear a buffer before rendering a frame, any information from the previous frame that is not overwritten will be preserved.

Tip: You can use glClear to clear multiple buffers at once by using the bitwise

Suffix	Data type	Corresponding C++ type	OpenGL type definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

OR operator to combine the *bit masks*. For example, glClear(DEPTH_BUFFER_BIT | COLOR_BUFFER_BIT) will clear both the depth buffer and the color buffer at once.

There are also methods to set the *clear values*, which are the default values that OpenGL will use when you call *glClear*. For example, if you use *glClearColor*(red, green, blue) you can specify what the default color of all the pixels will be when the color buffer is cleared.

5 Support Libraries

OpenGL is powerful, but it is also a low-level language. There are higher level libraries that use OpenGL's capabilities to make OpenGL's capabilities easier to use. Examples of these libraries include GLX, GLUT, and GLU. Other graphics platforms have their own support libraries as well.

Although we will make use of support libraries during lab, you may not use them in your cs123 projects (except for the final project).

6 Useful References

Some useful resources can be found through the cs123 web site. The Blue Book, which you will find to be extremely useful, provides documentation of GL and GLU methods that you will need for this lab.

7 Let's Get Started

7.1 Setting Up

We recommend you use Qt Creator to do this lab, which you can access by running /course/cs123/bin/qtcreator.

The support code for lab 3 is built into the main support code tree. Use *git pull* to get the latest version of the support code. Resolve any conflicts and use *git add* to commit your fixed files to your local repository. Finally, use *git commit* to save everything. If you type *git pull* again, you should see a message like "Everything is up-to-date."

- In general, anything that won't be changed between frames will be put in the *initializeGL* method, which is called once when the application is started.

- Features of the scene that lend themselves to be modified from frame to frame (camera specifications and vertex/object drawing and transforming) should be put in the *paintGL* method, which will be called every time OpenGL draws a frame.

- The *updateCamera* method will be called when the window is resized or the camera is changed.

- The *applyRenderSettings* method will be called when you need to change render settings, such as after a user selection or a change in environment.

7.2 Make a Sphere

Let's start by creating a sphere. For basic shapes like spheres, GLU can help. Simply call *gluSphere* to draw a sphere. There are certain parameters you can specify; take a look at the blue book for these. Once you have filled in your parameters to a reasonable extent, run your program. (For the GLUquadric, use the provided m_quadric).

Whenever you draw a shape make sure you're in GL_MODELVIEW mode. You can call *glLoadIdentity* to reset your current matrix to the identity matrix. By loading the identity matrix, you clear out any old transformations that might already be in the buffer.

7.3 The Camera

By default, the camera is located at the origin. Thus, you're actually viewing the *inside* of the sphere. Notice that gluSphere tessellates both the inside and outside of the sphere. When you tessellate your own sphere in the Shapes assignment, you'll only draw one side of each triangle so only the *outside* of each shape will be visible.

We need to translate the camera and then rotate it to look at the origin where our sphere is. A good place to do all of this in the updateCamera method.

First, we will set up a perspective view volume. Call gluPerspective, specifying the camera parameters that can be found in the m_camera member variable. Don't worry too much about what this function does, as you'll be implementing the transformations yourself in the Camtrans assignment.

Once you have set up the projection matric with gluPerspective, you can transform the camera. GLU has a nice method called gluLookAt that will point your

camera in a specified direction, making your camera "look towards" a point. This method takes in several parameters; again, use the corresponding values stored in the provided m_camera member variable. You'll be implementing this function in Camtrans as well, so enjoy the magic for now ☺

Note: Make sure that you are in the right matrix mode and that you load the identity matrix before calling the gluLookAt and gluPerspective functions. Both of these functions modify the *current* OpenGL matrix. If you update the wrong matrix, you won't get an error, but you might get incorrect results.

Although we say we are changing the position and orientation of the object, OpenGL's internal camera is *always* situated at (0,0,0), facing down the negative z axis. It is the position of the objects that change that make it seem that the camera is in a different position and orientation. Hence, we group the camera transformations with the modelview matrix.

7.4 Line, Fill and Point

There are three different modes we can view our shapes in: wireframe, solid and vertices only. These modes are represented by GL_LINE, GL_FILL and GL_POINT, respectively. We have included buttons for you that allow you to choose between the three. You will need to fill in the methods for each of these buttons. Fill in the appropriate part of the applyRenderSettings method and call it in the correct place. Play around with each of the views. Switch to GL_FILL mode for the next part of the lab.

7.5 Color

Hopefully at this point you can see the sphere in its entirety. Right now it's white; let's change that. Before we draw the sphere we can specify the color of the object. Every vertex will be given that color. We can do this using glColor3f(red,green,blue) before our gluSphere call. Use the values in the local variable m_color (or choose your own color) and color in your sphere.

Note: If you use the glColor3f, color values range between 0 and 1. Pure blue would be glColor3f(0,0,1). If we used glColor3i(r,g,b) instead, the values range from 0 to 255.

Each triangle in OpenGL is shaded (colored in) based on the colors at each of the vertices that compose it. In our case, since we are specifying every vertex to be the same color, every triangle should be shaded in with that color. If the vertices were different colors, OpenGL would perform interpolation automatically.

7.6 Lighting

You should now see a nice colored sphere. But there is another problem: it looks more like a circle than it does a sphere! That's because everything is the same color. Without any interaction with lighting there's no way to perceive the 3-dimensionality of the shape. To fix this problem, we need to turn on the lights!

The first thing to do is enable OpenGL's lighting capability. Enable GL_LIGHTING and fill in the appropriate part of the applyRenderSettings method now. OpenGL can support up to eight lights in a scene. Each light corresponds to a constant {LIGHT0, LIGHT1, LIGHT2, LIGHT3, etc.}. We can turn on any number of these lights and then reposition them or define their lighting characteristics.

For now just enable LIGHT0. Looks more 3-dimensional now, doesn't it? The lighting is interacting with the vertices and their positions on the sphere. While we won't go into how exactly this works just yet, the end result is that parts of the sphere facing the light are brighter than parts that are facing away from the light. Thus, we perceive the shape as 3-dimensional.

7.7 Shading

Looks sort of blocky, doesn't it? That's because we're using **flat shading** right now. Recall that the shading of each triangle depends on color of the vertices that compose it.

Self-test: Recall our discussion in lecture about shared vs. repeated vertices. What is the major difference between the two vertex specifications? What type of vertex specification do you think GLU is using? Why?

Right now we are using a shading mode called flat shading (GL_FLAT). This mode defines that each triangle[1] will be of constant color with no transitioning between vertices. The resulting object looks blocky and not very smooth. So how do we make it look smooth? Use GL_SMOOTH!

[1] You may have noticed that gluSphere actually uses rectangles as its basic unit (instead of triangles) but we explain concepts in terms of triangles as they are more widely used.

Fill in the applyRender settings method to support viewing your sphere in either GL_FLAT or GL_SMOOTH mode. You may use the OpenGL documentation to help you figure out how to accomplish this task.

7.8 More Shapes

Using GLU, make a cylinder with radius 0.5 and height 2. Draw it before you draw the sphere. Notice that the sphere will be obstructing the cylinder. That's because by default, OpenGL paints its shapes on top of shapes rendered first, similar to how your paint strokes in Brush are drawn on top of existing strokes.

We can fix this default behavior by enabling *depth testing* (GL_DEPTH_TEST) in OpenGL. Do it. Now what gets rendered to the screen will be determined by the actual depth/distance of the object relative to the camera.

7.9 A Glance at Transformations

Both objects are at (0,0,0) right now, so they intersect each other. Let's separate them. We need to call glTranslatef(x,y,z) to specify another translation. Note that the modelview matrix affects every vertex drawn, which means we will end up translating all future objects (like the sphere) as well! Thus, we have two options.

1. Completely re-specify the transformations for each object just before each object is drawn to the screen
2. Save the transformation matrix before we do the translation, and then restore it when we're done with it

OpenGL provides two functions: *glPushMatrix* and *glPopMatrix*. There are four basic steps we need to follow to use these:

1. Calling *glPushMatrix* puts a copy of your original modelview matrix on top of a stack (separate from your application's execution stack)
2. Modify your matrix using the *glTranslatef* method to modify the current matrix.
3. Draw your object.
4. Call *glPopMatrix* to get rid of the matrix at the top of the stack and revert your current matrix back to the original.

Try translating your cylinder to (3,0,0), using the *glPushMatrix* and *glPopMatrix* functions.

8 Final Thoughts

When you're finished, show a TA your program to get checked off. If you have time you can play around with OpenGL's *triangle fan* and *triangle strip* constructs. These can be used to actually tessellate shapes of your own, which you will be doing in Shapes. There are also a number of ways you can tessellate shapes including immediate mode, using vertex arrays. You can also make tessellating more efficient by using display lists and indexing arrays. As you will learn this semester, efficiency is key in graphics, and optimizing can be a lot of fun!

© 2010 Ben Herila and Roger Fong,
released under the Creative Commons Share-Alike 3.0 Unported License.

DirectX is a registered trademark of Microsoft Corporation. OpenGL is a registered trademark of SGI, and other registered trademarks are the properties of their respective owners.