CMSC 425: Lecture 4
More about OpenGL and GLUT
Tuesday, Feb 5, 2013

Reading: See any standard reference on OpenGL or GLUT.

Basic Drawing: In the previous lecture, we showed how to create a window in GLUT, how to get user input, but we have not discussed how to get graphics to appear in the window. Here, we begin discussion of how to use OpenGL to draw objects.

Before being able to draw a scene, OpenGL needs to know the following information: what are the *objects* to be drawn, how is the image to be *projected* onto the window, and how *lighting* and *shading* are to be performed. To begin with, we will consider a very the simple case. There are only 2-dimensional objects, no lighting or shading. Also we will consider only relatively little user interaction.

Idealized Drawing Window: Because we generally do not have complete control over the window size, it is a good idea to think in terms of drawing on a rectangular *idealized drawing region*, whose size and shape are completely under our control. Then we will scale this region to fit within the actual graphics window on the display. There are many reasons for doing this. For example, if you design your program to work specifically for 400×400 window, but then the user resizes the window, what do you do? It is a much better idea to assume a window of arbitrary size. For example, you could define two variables w and h, that indicate the width and height of the window, respectively. The values w and h could be measured in whatever units are most natural to your application: pixels, inches, light years, microns, furlongs or fathoms.

OpenGL allows for the grahics window to be broken up into smaller rectangular subwindows, called *viewports*. We will then have OpenGL scale the image drawn in the idealized drawing region to fit within the viewport. The main advantage of this approach is that it is very easy to deal with changes in the window size.

We will consider a simple drawing routine for the picture shown in the figure. We assume that our idealized drawing region is a unit square over the real interval $[0, 1] \times [0, 1]$. (Throughout the course we will use the notation $[a, b]$ to denote the interval of real values z such that $a \leq z \leq b$. Hence, $[0, 1] \times [0, 1]$ is a unit square whose lower left corner is the origin.) This is illustrated in Fig. 1.

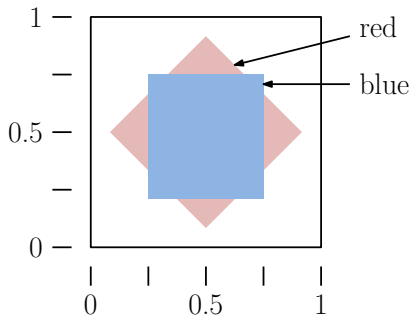

Fig. 1: Drawing produced by the simple display function.

GLUT uses the convention that the origin is in the upper left corner and coordinates are given as integers. This makes sense for GLUT, because its principal job is to communicate with the window system, and most window systems use this convention. On the other hand, OpenGL uses the convention that coordinates are (generally) floating point values and the origin is in the lower left corner. Recalling the OpenGL goal is to provide us with an idealized drawing surface, this convention is mathematically more elegant.

The Display Callback: Recall that the *display callback function* is the function that is called whenever it is necessary to redraw the image, which arises for example:

- the initial creation of the window
- (possibly) whenever the window is uncovered by the removal of some overlapping window,
- whenever your program explicitly requests that it be redrawn, through the use of the function glutPostRedisplay

The display callback function for our program is shown in the following code block. We first erase the contents of the image window, then do our drawing, and finally swap buffers so that what we have drawn becomes visible. (Recall double buffering from the previous lecture.) This function first draws a red diamond and then (on top of this) it draws a blue rectangle. Let us assume double buffering is being performed, and so the last thing to do is invoke glutSwapBuffers() to make everything visible.

Sample Display Function

```
void myDisplay() {                              // display function
    glClear(GL_COLOR_BUFFER_BIT);               // clear the window

    glColor3f(1.0, 0.0, 0.0);                   // set color to red
    glBegin(GL_POLYGON);                        // draw a diamond
        glVertex2f(0.90, 0.50);
        glVertex2f(0.50, 0.90);
        glVertex2f(0.10, 0.50);
        glVertex2f(0.50, 0.10);
    glEnd();

    glColor3f(0.0, 0.0, 1.0);                   // set color to blue
    glRectf(0.25, 0.25, 0.75, 0.75);            // draw a rectangle

    glutSwapBuffers();                          // make it all visible
}
```

Clearing the Window: The command glClear() clears the window, by overwriting it with the background color. The background color is black by default, but generally it may be set by the call:

glClearColor(GLfloat Red, GLfloat Green, GLfloat Blue, GLfloat Alpha).

The type GLfloat is OpenGL's redefinition of the standard float. To be correct, you should use the approved OpenGL types (e.g. GLfloat, GLdouble, GLint) rather than the obvious counterparts (float, double, and int). Typically the GL types are the same as the corresponding native types, but not always.

Colors components are given as floats in the range from 0 to 1, from dark to light. Recall that the A (or α) value is used to control transparency. For opaque colors A is set to 1. Thus to set the background color to black, we would use glClearColor(0.0, 0.0, 0.0, 1.0), and to set it to blue use glClearColor(0.0, 0.0, 1.0, 1.0). (**Tip:** When debugging your program, it is often a good idea to use an uncommon background color, like a random shade of pink, since black can arise as the result of many different bugs.) Since the background color is usually independent of drawing, the function glClearColor() is typically set in one of your initialization procedures, rather than in the drawing callback function.

Clearing the window involves resetting information within the drawing buffer. As we mentioned before, the drawing buffer may store different types of information. This includes color information, of course, but depth or distance information is used for hidden surface removal. Typically when the window is cleared, we want to clear everything. (Occasionally it is useful to achieve certain special effects by clearing only some of the buffers.) The glClear() command allows the user to select which buffers are to be cleared. In this case we only have color in the depth buffer, which is selected by the option GL_COLOR_BUFFER_BIT. If we had a depth buffer to be cleared it as well we could do this by combining these using a "bitwise or" operation:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

Drawing Attributes: The OpenGL drawing commands describe the geometry of the object that you want to draw. More specifically, all OpenGL is based on drawing *convex polygons*, so it suffices to specify the *vertices* of the object to be drawn. The manner in which the object is displayed is determined by various *drawing attributes* (color, point size, line width, etc.).

The command glColor3f() sets the drawing color. The arguments are three GLfloat's, giving the R, G, and B components of the color. In this case, RGB $= (1, 0, 0)$ means pure red. Once set, the attribute applies to all subsequently defined objects, until it is set to some other value. Thus, we could set the color, draw three polygons with the color, then change it, and draw five polygons with the new color.

This call illustrates a common feature of many OpenGL commands, namely flexibility in argument types. The suffix "3f" means that three floating point arguments (actually GLfloat's) will be given. For example, glColor3d() takes three double (or GLdouble) arguments, glColor3ui() takes three unsigned int arguments, and so on. For floats and doubles, the arguments range from 0 (no intensity) to 1 (full intensity). For integer types (byte, short, int, long) the input is assumed to be in the range from 0 (no intensity) to its maximum possible positive value (full intensity).

But that is not all! The three argument versions assume RGB color. If we were using RGBA color instead, we would use glColor4d() variant instead. Here "4" signifies four arguments. (Recall that the A or alpha value is used for various effects, such an transparency. For standard (opaque) color we set $A = 1.0$.)

In some cases it is more convenient to store your colors in an array with three elements. The suffix "v" means that the argument is a vector. For example glColor3dv() expects a single argument, a vector containing three GLdouble's. (Note that this is a standard C/C++ style array, not the class vector from the C++ Standard Template Library.) Using C's convention that a vector is represented as a pointer to its first element, the corresponding argument type would be "const GLdouble$*$".

Whenever you look up the prototypes for OpenGL commands, you often see a long list with various suffixes to indicate the argument types. Some examples for glColor are shown in the following code block.

Changing arguments types in OpenGL

```
void glColor3d(GLdouble red, GLdouble green, GLdouble blue)
void glColor3f(GLfloat red, GLfloat green, GLfloat blue)
void glColor3i(GLint red, GLint green, GLint blue)
... (and forms for byte, short, unsigned byte and unsigned short) ...

void glColor4d(GLdouble red, GLdouble green, GLdouble blue, GLdouble alpha)
... (and 4-argument forms for all the other types) ...

void glColor3dv(const GLdouble *v)
... (and other 3- and 4-argument forms for all the other types) ...
```

Drawing commands: OpenGL supports drawing of a number of different types of objects. The simplest is glRectf(), which draws a filled rectangle. All the others are complex objects consisting of a (generally) unpredictable number of elements. This is handled in OpenGL by the constructs glBegin(*mode*) and glEnd(). Between these two commands a list of vertices is given, which defines the object. The sort of object to be defined is determined by the *mode* argument of the glBegin() command. Some of the possible modes are illustrated in Fig. 2. For details on the semantics of the drawing methods, see the reference manuals.

Note that in the case of GL_POLYGON only *convex polygons* (internal angles less than 180 degrees) are supported. You must subdivide nonconvex polygons into convex pieces, and draw each convex piece separately.

```
glBegin(mode);
    glVertex(v0); glVertex(v1); ...
glEnd();
```

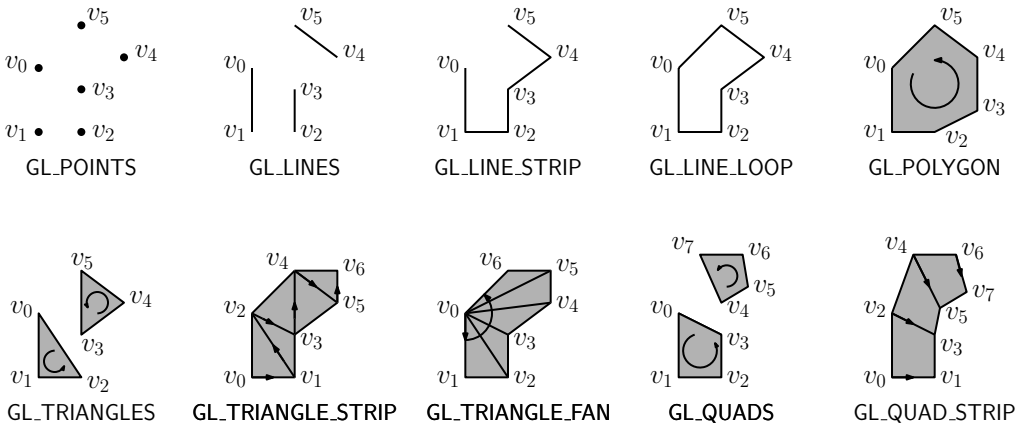

Fig. 2: Some OpenGL object definition modes. (It is a good idea to draw primitives using a consistent direction, say counterclockwise.)

In the example above we only defined the x- and y-coordinates of the vertices. How does OpenGL know whether our object is 2-dimensional or 3-dimensional? The answer is that it does not know.

OpenGL represents all vertices as 3-dimensional coordinates internally. This may seem wasteful, but remember that OpenGL is designed primarily for 3-d graphics. If you do not specify the z-coordinate, then it simply sets the z-coordinate to 0.0. By the way, glRectf() always draws its rectangle on the $z = 0$ plane.

Between any glBegin()...glEnd() pair, there is a restricted set of OpenGL commands that may be given. This includes glVertex() and also other command attribute commands, such as glColor3f(). At first it may seem a bit strange that you can assign different colors to the different vertices of an object, but this is a very useful feature. Depending on the shading model, it allows you to produce shapes whose color *blends* smoothly from one end to the other.

There are a number of drawing attributes other than color, but until we start discussing lighting and 3-dimensional drawing, color is by far the most common attribute. (See the OpenGL documentation for other examples.)

After drawing the diamond, we change the color to blue, and then invoke glRectf() to draw a rectangle. This procedure takes four arguments, the (x, y) coordinates of any two opposite corners of the rectangle, in this case $(0.25, 0.25)$ and $(0.75, 0.75)$. (There are also versions of this command that takes double or int arguments, and vector arguments as well.) We could have drawn the rectangle by drawing a GL_POLYGON, but this form is easier to use.

Viewports: OpenGL does not assume that you are mapping your graphics to the entire window. Often it is desirable to subdivide the graphics window into a set of smaller subwindows and then draw separate pictures in each window. The subwindow into which the current graphics are being drawn is called a *viewport*. The viewport is typically the entire display window, but it may generally be any rectangular subregion.

The size of the viewport depends on the dimensions of our window. Thus, every time the window is resized (and this includes when the window is created originally) we need to readjust the viewport to ensure proper transformation of the graphics. For example, in the typical case, where the graphics are drawn to the entire window, the reshape callback would contain the following call which resizes the viewport, whenever the window is resized.

Setting the Viewport in the Reshape Callback

```
void myReshape(int winWidth, int winHeight)       // reshape window
{
    ...
    glViewport (0, 0, winWidth, winHeight);       // reset the viewport
    ...
}
```

The other thing that might typically go in the myReshape() function would be a call to glutPostRedisplay(), since you will need to redraw your image after the window changes size.

The general form of the command is

glViewport(GLint x, GLint y, GLsizei width, GLsizei height),

where (x, y) are the pixel coordinates of the *lower-left corner* of the viewport, as defined relative to the lower-left corner of the window, and *width* and *height* are the width and height of the viewport in pixels.

For example, suppose you had a $w \times h$ window, which you wanted to split in half by a vertical line to produce two different drawings. This is presented in the following code block.

Setting up viewports for a split screen

```
glClear(GL_COLOR_BUFFER_BIT);       // clear the window
glViewport (0, 0, w/2, h);          // set viewport to left half
  // ...drawing commands for the left half of window
glViewport (w/2, 0, w/2, h);        // set viewport to right half
  // ...drawing commands for the right half of window
glutSwapBuffers();                  // swap buffers
```

Projection Transformation: In the simple drawing procedure, we said that we were assuming that the "idealized" drawing area was a unit square over the interval $[0, 1]$ with the origin in the lower left corner. The transformation that maps the idealized drawing region (in 2- or 3-dimensions) to the window is called the *projection*. We did this for convenience, since otherwise we would need to explicitly scale all of our coordinates whenever the user changes the size of the graphics window.

However, we need to inform OpenGL of where our "idealized" drawing area is so that OpenGL can map it to our viewport. This mapping is performed by a transformation matrix called the *projection matrix*, which OpenGL maintains internally. (In future lectures, we will discuss OpenGL's transformation mechanism in greater detail. In the mean time some of this may seem a bit arcane.)

Since matrices are often cumbersome to work with, OpenGL provides a number of relatively simple and natural ways of defining this matrix. For our 2-dimensional example, we will do this by simply informing OpenGL of the rectangular region of two dimensional space that makes up our idealized drawing region. This is handled by the command

gluOrtho2D(left, right, bottom, top).

First note that the prefix is "glu" and not "gl", because this procedure is provided by the GLU library. Also, note that the "2D" designator in this case stands for "2-dimensional." (In particular, it does *not* indicate the argument types, as with, say, glColor3f()).

All arguments are of type GLdouble. The arguments specify the x-coordinates (*left* and *right*) and the y-coordinates (*bottom* and *top*) of the rectangle into which we will be drawing. Any drawing that we do outside of this region will automatically be clipped away by OpenGL. The code to set the projection is given below.

Setting a Two-Dimensional Projection

```
glMatrixMode(GL_PROJECTION);            // set projection matrix mode
glLoadIdentity();                       // initialize to identity
gluOrtho2D(0.0, 1.0, 0.0, 1.0);         // map unit square to viewport
glMatrixMode(GL_MODELVIEW);             // set matrix mode back to default
```

The first command tells OpenGL that we are modifying the projection transformation. (OpenGL maintains three different types of transformations, as we will see later.) Most of the commands that manipulate these matrices do so by multiplying some matrix times the current matrix. Thus, we

initialize the current matrix to the identity, which is done by glLoadIdentity(). This code usually appears in some initialization procedure or possibly in the reshape callback.

Where does this code fragment go? It depends on whether the projection will change or not. If we make the simple assumption that are drawing will always be done relative to the $[0, 1]^2$ unit square, then this code can go in some initialization procedure. If our program decides to change the drawing area (for example, growing the drawing area when the window is increased in size) then we would need to repeat the call whenever the projection changes.

At first viewports and projections may seem confusing. Remember that the viewport is a rectangle within the actual graphics window on your display, where you graphics will appear. The projection defined by gluOrtho2D() simply defines a rectangle in some "ideal" coordinate system, which you will use to specify the coordinates of your objects. It is the job of OpenGL to map everything that is drawn in your ideal window to the actual viewport on your screen. This is illustrated in Fig. 3.

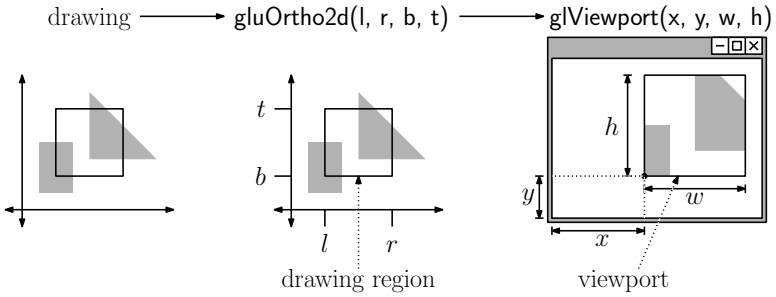

Fig. 3: Projection and viewport transformations.

The complete program is shown in the code block below.

Sample OpenGL Program

```
#include <cstdlib>                      // standard definitions
#include <iostream>                     // C++ I/O

#include <GL/glut.h>                    // GLUT (also loads gl.h and glu.h)

void myReshape(int w, int h) {                  // window is reshaped
    glViewport (0, 0, w, h);                    // update the viewport
    glMatrixMode(GL_PROJECTION);                // update projection
    glLoadIdentity();
    gluOrtho2D(0.0, 1.0, 0.0, 1.0);             // map unit square to viewport
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();                        // request redisplay
}

void myDisplay(void) {                          // (re)display callback
    glClearColor(0.5, 0.5, 0.5, 1.0);           // background is gray
    glClear(GL_COLOR_BUFFER_BIT);               // clear the window
    glColor3f(1.0, 0.0, 0.0);                   // set color to red
    glBegin(GL_POLYGON);                        // draw the diamond
        glVertex2f(0.90, 0.50);
        glVertex2f(0.50, 0.90);
        glVertex2f(0.10, 0.50);
        glVertex2f(0.50, 0.10);
    glEnd();
    glColor3f(0.0, 0.0, 1.0);                   // set color to blue
    glRectf(0.25, 0.25, 0.75, 0.75);            // draw the rectangle
    glutSwapBuffers();                          // swap buffers
}

int main(int argc, char** argv) {               // main program
    glutInit(&argc, argv);                      // OpenGL initializations
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);// double buffering and RGB
    glutInitWindowSize(400, 400);               // create a 400x400 window
    glutInitWindowPosition(0, 0);               // ...in the upper left
    glutCreateWindow(argv[0]);                  // create the window

    glutDisplayFunc(myDisplay);                 // setup callbacks
    glutReshapeFunc(myReshape);
    glutMainLoop();                             // start it running
    return 0;                                   // ANSI C expects this
}
```