CMSC 425: Lecture 3
Basic Elements of OpenGL and GLUT

Thursday, Jan 31, 2013

Reading: See any standard reference on OpenGL or GLUT.

Graphics Libraries: In this lecture we will discuss programming a 3-dimensional interactive graphics system. In particular, we will discuss OpenGL and GLUT. The essence of producing interactive graphics involves generating a description of the scene being rendered, and repeating this process at a rate of roughly 30 frames per second. We call each such redrawing a *display cycle* or a *refresh cycle*, since your program is refresh the current contents of the image.

In modern graphics systems, the image is generated by the graphics processing unit (GPU), and the job of your program is to describe the scene to be displayed to the GPU so it can be rendered. The commands sent to the GPU take the form of procedure calls made by your program to a graphics library. The procedures of the library are defined according to an application programmer's interface (API). There are a few common graphics APIs. Generally, they come in two different types:

Retained Mode: The system maintains the state of the computation in its own internal data structures. With each refresh cycle, this data is transmitted to the GPU for rendering. A *scene graph* is typically employed as the method for storing this information. Examples of retained-mode systems include *Java3d*, *Ogre3D*, and *Open Scenegraph*. The principal advantage is that global optimizations can be performed, since all the scene information is known to the system. This approach is less well suited to time-varying data sets, since the internal representation of the data set needs to be updated frequently.

Immediate Mode: In this sort of system, your program provides all the information needed to draw each scene with each display cycle. In other words, your program transmits commands directly to the GPU for execution. Examples of immediate-mode systems include *OpenGL* and *DirectX*. Immediate-mode systems tend to be more efficient, since they provide closer control of the hardware. The principal disadvantage is that it not possible for the system to perform the sort of global optimizations (such as culling non-visible entities), which is possible with retained-mode approaches.

OpenGL: OpenGL is a widely used industry standard graphics API. It has been ported to virtually all major systems, and can be accessed from a number of different programming languages (C, C++, Java, Python, ...). Because it works across many different platforms, it is very general. This is in contrast to the principal alternative, DirectX, which has been designed to work primarily on Microsoft systems.

For the most part, OpenGL operates in *immediate mode*, which means that each function call results in a command being sent directly to the GPU. There are some retained elements, however. For example, transformations, lighting, and texturing need to be set up, so that they can be applied later in the computation.

Because of the design goal of being independent of the window system and operating system, OpenGL does *not* provide capabilities for windowing tasks or user input and output. For example, there are no commands in OpenGL to create a window, to resize a window, to determine the current mouse coordinates, or to detect whether a keyboard key has been hit. Everything is focused just on the process of generating an image. In order to achieve these other goals, it is necessary to use an additional *toolkit*. There are a number of different toolkits, which provide various capabilities. We will cover a very simple one in this class, called *GLUT*, which stands for the *GL Utility Toolkit*. GLUT has the virtue of being very simple, but it does not have a lot of features. To get these features, you will need to use a more sophisticated toolkit.

There are many, many tasks needed in a typical large graphics system. As a result, there are a number of software systems available that provide utility functions. For example, suppose that you want to draw a sphere. OpenGL does not have a command for drawing spheres, but it can draw triangles. What you would like is a utility function which, given the center and radius of a sphere, will produce a collection of triangles that approximate the sphere's shape. OpenGL provides a simple collection of utilities, called the *GL Utility Library* or *GLU* for short.

Since we will be discussing a number of the library functions for OpenGL, GLU, and GLUT during the next few lectures, let me mention that it is possible to determine which library a function comes from by its prefix. Functions from the OpenGL library begin with "gl" (as in "glTriangle"), functions from GLU begin with "glu" (as in "gluLookAt"), and functions from GLUT begin with "glut" (as in "glutCreateWindow").

The Main Program: Before discussing how to draw shapes, we will begin with the basic elements of how to create a window. OpenGL was intentionally designed to be independent of any specific window system. Consequently, a number of the basic window and system operations are not provided. This is the principal reason for *GLUT*. This toolkit which provides the necessary tools for requesting that windows be created and providing interaction with I/O devices.

Let us begin by considering a typical main program (see the following code fragment). Throughout, we will assume that programming is done in C/C++, but our examples can be readily adapted to other languages. (Do not worry for now if you do not understand the meanings of the various calls. We will discuss them in greater detail below.) This program creates a window that is 400 pixels wide and 300 pixels high, located in the upper left corner of the display.

Typical OpenGL/GLUT Main Program

```
#include <GL/glut.h>                          // GLUT, GLU, and OpenGL defs
int main(int argc, char** argv)               // program arguments
{
    glutInit(&argc, argv);                    // initialize glut and openGL
                                              // double buffering and RGBA
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(400, 300);             // initial window size
    glutInitWindowPosition(0, 0);             // initial window position
    glutCreateWindow(argv[0]);                // create window

    ...initialize callbacks here (described below)...

    myInit();                                 // your own initializations
    glutMainLoop();                           // turn control over to glut
    return 0; // we never return here; this just keeps the compiler happy
}
```

Note that the call to glutMainLoop turns control over to the system. After this, the only return to your program will occur due to various events, called *callbacks*. (The final "return 0" is only there to keep the compiler from issuing a warning.) Here is an explanation of these functions.

glutInit: The arguments given to the main program (argc and argv) are the command-line arguments supplied to the program. This assumes a typical Unix environment, in which the program is invoked from a command line. We pass these into the main initialization procedure, glutInit. This procedure must be called before any others. It processes (and removes) command-line arguments that may be of interest to GLUT (e.g., allowing the user to override the default window size) and the window system and does general initializations. Any remaining arguments are then left for the user's program to interpret, if desired.

glutInitDisplayMode: This performs initializations informing OpenGL how to set up its frame buffer. Recall that the frame buffer is a special 2-dimensional array in the GPU's memory where the graphical image is stored. OpenGL maintains an enhanced version of the frame buffer with additional information. For example, this includes depth information for hidden surface removal. The system needs to know how we are representing colors of our general needs in order to determine the *depth* (that is, the number of bits) to assign for each pixel in the frame buffer. The argument to glutInitDisplayMode is a logical-or (using the operator "|") of a number of possible options, which are given in Table 1.

Display Mode	Meaning
GLUT_RGB	Use RGB colors
GLUT_RGBA	Use RGB plus α (recommended)
GLUT_INDEX	Use colormapped colors (not recommended)
GLUT_DOUBLE	Use double buffering (recommended)
GLUT_SINGLE	Use single buffering (not recommended)
GLUT_DEPTH	Use depth buffer (needed for hidden surface removal)

Table 1: Arguments to glutInitDisplayMode. (Constants defined in glut.h)

.

Let us discuss each of these elements in greater detail.

Color: We need to tell the system how colors will be represented. There are three methods, of which two are fairly commonly used: GLUT_RGB or GLUT_RGBA. The first uses standard RGB colors (24-bit color, consisting of 8 bits of red, green, and blue), and is the default. The second requests RGBA coloring. In this color system there is a fourth component (designated "A" or α), which indicates the opaqueness of the color (1 = fully opaque, 0 = fully transparent). This is useful in creating transparent effects. We will discuss how this is applied later this semester. (It turns out that there is no advantage in trying to save space using GLUT_RGB over GLUT_RGBA, since according to the GLUT documentation, both are treated the same.)

Single or Double Buffering: The next option specifies whether single or double buffering is to be used, GLUT_SINGLE or GLUT_DOUBLE, respectively. To explain the difference, we need to understand a bit more about how the frame buffer works. In raster graphics systems, whatever is written to the frame buffer is immediately transferred to the display. This process is repeated frequently, say 30–60 times a second. To do this, the typical approach is to first erase the old contents by setting all the pixels to some background color, say black. After this, the new contents are drawn. However, even though it might happen very fast, the process of setting the image to black and then redrawing everything produces a noticeable flicker in the image.

Double buffering is a method to eliminate this flicker. In double buffering, the system maintains two separate frame buffers. The *front buffer* is the one which is displayed, and the *back buffer* is the other one. Drawing is always done to the back buffer. Then to update the image, the system simply swaps the two buffers (see Fig. 1). The swapping process is very fast, and appears to happen instantaneously (with no flicker). Double buffering requires twice the buffer space as single buffering, but since memory is relatively cheap these days, it is the preferred method for interactive graphics.

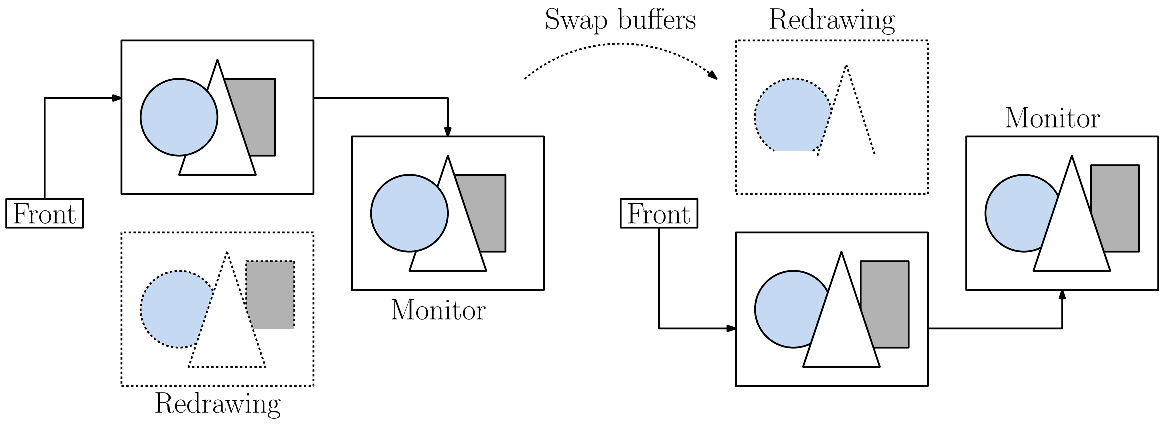

Fig. 1: Double buffering.

Depth Buffer: One other option that we will need later with 3-dimensional graphics will be hidden surface removal. Virtually all raster-based interactive graphics systems perform hidden surface removal by an approach called the *depth-buffer* or *z-buffer*. In such a system, each fragment stores its distance from the eye. When fragments are rendered as pixels only the closest is actually drawn. The depth buffer is enabled with the option GLUT_DEPTH. For this program it is not needed, and so has been omitted. When there is no depth buffer, the last pixel to be drawn is the one that you see.

glutInitWindowSize: This command specifies the desired width and height of the graphics window. The general form is glutInitWindowSize(int width, int height). The values are given in numbers of pixels.

glutInitPosition: This command specifies the location of the upper left corner of the graphics window. The form is glutInitWindowPosition(int x, int y) where the (x, y) coordinates are given relative to the upper left corner of the display. Thus, the arguments $(0, 0)$ places the window in the upper left corner of the display.

Note that glutInitWindowSize and glutInitWindowPosition are both considered to be only *suggestions* to the system as to how to where to place the graphics window. Depending on the window system's policies, and the size of the display, it may not honor these requests.

glutCreateWindow: This command actually creates the graphics window. The general form of the command is glutCreateWindow(char* title), where title is a character string. Each window has a title, and the argument is a string which specifies the window's title. We pass in argv[0]. In Unix argv[0] is the name of the program (the executable file name) so our graphics window's name is the same as the name of our program.

Asynchonous Creation: Note that the call glutCreateWindow does not really create the window, but rather merely sends a request to the system that the window be created. Why do you care? In some OpenGL implementations, certain operations cannot be performed unless the window (and the associated graphics context) exists. Such operations should be performed only *after* your program has received notification that the window really exists. Your program is informed of this event through an event callback, either *reshape* or *display*. We will discuss them below.

The general structure of an OpenGL program using GLUT is shown in Fig. 2. (Don't worry if some elements are unfamiliar. We will discuss them below.)

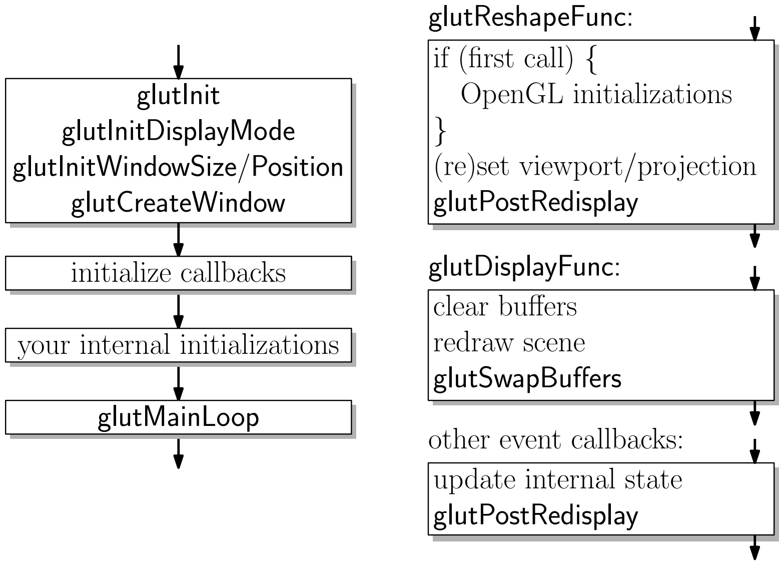

Fig. 2: General structure of an OpenGL program using GLUT.

Event-driven Programming and Callbacks: Virtually all interactive graphics programs are *event driven*. Unlike traditional programs that read from a standard input file, a graphics program must be prepared at any time for input from any number of sources, including the mouse, or keyboard, or other graphics devises such as trackballs and joysticks.

In OpenGL this is done through the use of *callbacks*. The graphics program instructs the system to invoke a particular procedure whenever an event of interest occurs, say, the mouse button is clicked. The graphics program indicates its interest, or *registers*, for various events. This involves telling the window system which event type you are interested in, and passing it the name of a procedure you have written to handle the event.

Note: If you program in C++, note that the Glut callback functions you define must be "standard" procedures; they cannot be class member functions.

Types of Callbacks: Callbacks are used for two purposes, *user input events* and *system events*. User input events include things such as mouse clicks, the motion of the mouse (without clicking) also called *passive motion*, keyboard hits. Note that your program is only signaled about events that happen to your window. For example, entering text into another window's dialogue box will not generate a keyboard event for your program. Here are some common examples. A summary is given in Table 2. There are a number of others as well (special keys, status of shift/control/alt, detecting key releases.)

System Event	*Callback request*	*User callback function prototype* (return void)
(Re)display	glutDisplayFunc	myDisplay()
(Re)size window	glutReshapeFunc	myReshape(int w, int h)
Timer event	glutTimerFunc	myTimer(int id)
Idle event	glutIdleFunc	myIdle()
Input Event	*Callback request*	*User callback function prototype* (return void)
Mouse button	glutMouseFunc	myMouse(int b, int s, int x, int y)
Mouse motion	glutPassiveMotionFunc	myMotion(int x, int y)
Keyboard key	glutKeyboardFunc	myKeyboard(unsigned char c, int x, int y)

Table 2: Common callbacks and the associated registration functions.

Display Event: At a minimum, every OpenGL program must handle this event. It is invoked when the system senses that the contents of the graphics window need to be redrawn, for example:

- the window is created initially or it has been resized,
- the window system has determined that the graphics window has been "damaged" (which might occur, for example, if an obscuring window has been moved away, thus revealing all or part of the graphics window),
- the program explicitly requests redrawing, for example, because the internal state has changed in a way that affects the scene, by calling glutPostRedisplay.

Recall from above that the command glutCreateWindow does not actually create the window, but merely requests that creation be started. In order to inform your program that the creation has completed, the system generates a display event. This is how you know that you can now start drawing into the graphics window.

Reshape Event: This happens whenever the window's size is altered, including its initial creation. The callback provides information on the new size of the window. Recall that your initial call to glutInitWindowSize is only taken as a suggestion of the actual window size. When the system determines the actual size of your window, it generates such a callback to inform you of the actual size.

Idle/Timer Events: Often in an interactive graphics program, the user may not be providing any input at all, but it may still be necessary to update the image. For example, in a flight simulator the plane keeps moving forward, even without user input. To do this, the program goes to sleep and requests that it be awakened in order to draw the next image. There are two ways to do this, a *timer event* and an *idle event*. An idle event is generated every time the system has nothing better to do. This is often fine, since it means that your program wastes no cycles.

Often, you want to have more precise control of timing (e.g., when trying to manage parallel threads such as artificial intelligence and physics modeling). If so, an alternate approach is to request a timer event. In a timer event you request that your program go to sleep for some period of time and that it be "awakened" by an event some time later, say 1/50 of a second later. In glutTimerFunc the first argument gives the sleep time as an integer in milliseconds and the last argument is an integer identifier, which is passed into the callback function.

Input Events: These events include keyboard key presses and mouse key presses and mouse motion. When any such event occurs, the system will inform you which key has been pressed (or released), and where the cursor is at the instant of the event. Here are the principal input events your program can listen for.

For example, the function calls shown in the code fragment below shows how to register for the following events: display events, reshape events, mouse clicks, keyboard strikes, and timer events. The functions like myDraw and myReshape are supplied by the user, and will be described later.

Typical Callback Setup

```
int main(int argc, char** argv)
{
    ...
    glutDisplayFunc(myDraw);                // set up the callbacks
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(20, myTimeOut, 0);        // timer in 20/1000 seconds
    ...
}
```

Most of these callback registrations simply pass the name of the desired user function to be called for the corresponding event. The one exception is glutTimeFunc whose arguments are the number of milliseconds to wait (an unsigned int), the user's callback function, and an integer identifier. The identifier is useful if there are multiple timer callbacks requested (for different times in the future), so the user can determine which one caused this particular event.

Callback Functions: What does a typical callback function do? This depends entirely on the application that you are designing. Some examples of general form of callback functions is shown below.

Note that the timer callback and the reshape callback both invoke the function glutPostRedisplay. This procedure informs OpenGL that the state of the scene has changed and should be redrawn (by calling your drawing procedure). This might be requested in other callbacks as well.

Note that each callback function is provided with information associated with the event. For example, a reshape event callback passes in the new window width and height. A mouse click callback passes in four arguments, which button was hit (b: left, middle, right), what the buttons new state is (s: up or down), the (x, y) coordinates of the mouse when it was clicked (in pixels). The various parameters used for b and s are described in Table 3. A keyboard event callback passes in the character that was hit and the current coordinates of the mouse. The timer event callback passes in the integer identifier, of the timer event which caused the callback. Note that each call to glutTimerFunc creates only one request for a timer event. (That is, you do not get automatic repetition of timer events.) If you want to generate events on a regular basis, then insert a call to glutTimerFunc from within the callback function to generate the next one.

Examples of Callback Functions for System Events

```
void myDraw() {                         // called to display window
   // ...insert your drawing code here ...
   glutSwapBuffers();                   // make the new stuff visible
}
void myReshape(int w, int h) {          // called if reshaped
    windowWidth = w;                    // save new window size
    windowHeight = h;
    // ...may need to update the projection ...
    glutPostRedisplay();                // request window redisplay
}
void myTimeOut(int id) {                // called if timer event
    // ...advance the state of animation incrementally...
    glutTimerFunc(20, myTimeOut, 0);    // schedule next timer event
}
```

Examples of Callback Functions for User Input Events

```
                                        // called if mouse click
void myMouse(int b, int s, int x, int y) {
    switch (b) {                        // b indicates the button
        case GLUT_LEFT_BUTTON:
            if (s == GLUT_DOWN)         // button pressed
                // ...
            else if (s == GLUT_UP)      // button released
                // ...
            break;
        // ...                         // other button events
    }
}
                                        // called if keyboard key hit
void myKeyboard(unsigned char c, int x, int y) {
   switch (c) {                         // c is the key that is hit
      case ’q’:                         // ’q’ means quit
          exit(0);
          break;
      // ...                           // other keyboard events
   }
}
```

GLUT Parameter Name	Meaning
GLUT_LEFT_BUTTON	left mouse button
GLUT_MIDDLE_BUTTON	middle mouse button
GLUT_RIGHT_BUTTON	right mouse button
GLUT_DOWN	mouse button pressed down
GLUT_UP	mouse button released

Table 3: GLUT parameter names associated with mouse events. (Constants defined in glut.h)