Lecture I
OPENGL LECTURE NOTES

The `OpenGL` API was originally developed by Silicon Graphics, Inc (SGI). It is now an open standard that is widely available on most platforms. As an "API", it hides platform dependence from the user.

§1. The OpenGL API

OpenGL is a graphic API. As such, it is basically platform independent, and available in all major platforms. It was originally developed by SGI.

The pritive number types are denoted `GLint`, `GLfloat` and `GLdouble`, corresponding the similar types in C/C++. Note the prefix "`GL`" in these names indicating a OpenGL primitive type.

In OpenGL, the most basic geometric object is a **vertex**. This is basically a point, but it can be 2 or 3 dimensional. The types of the coordinates might be `int` or `float`. The obvious way to specify such points might be

```
  Vertex(int x, int y); // 2-dimensional
  Vertex(float x, float y, float z); // 3-dimensional
```

The actual OpenGl syntax is similar:

```
  glVertex2i(GLint x, GLint y);
  glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

The name `glVertex2i` can be parsed to yield insights into the data type. First, the prefix `gl` indicate OpenGL functions, just as the prefix `GL` indicate OpenGL types. The suffixes `2i` and `3f` indicate the dimension and number types: "2-dimensional integer" and "3-dimensional float" respectively. If `vec2` is an array of two double values, and `vec3` is an array of three int values, we could also define vertices in the following two ways:

```
glVertex2dv( vec2 );
glVertex3iv( vec3 );
```

In general, the type of a `glVertex` has a suffix of two characters "`nt`" or three characters "`ntv`" where `n` is 2, 3 or 4, and `t` is `i` (int), `f` (float) or `d` (double).

Next, we can group vertices to form more complex geometric shapes:

```
glBegin(GL_LINES); // GL_LINES is a defined constant
  glVertex2f(x1, y1);
  glVertex2f(x2, y2);
```

```
  ...
  glVertex2f(x7, y7);
  glVertex4f(x8, y8);
glEnd();
```

The above construct actually defines 4 lines. Thus `GL_LINES` takes a list of $2n$ vertices, and pairs them up to form n lines.

In general, the vertex grouping construct is as follows:

```
glBegin(<GLtype>); // GLtype is some defined constant
  glVertex*(x1, y1); // the first vertex
  glVertex*(x2, y2); // the second vertex
  ...
  glVertex*(xn, yn); // the n-th vertex
glEnd();
```

where `GLtype` tells OpenGL how to interpret the list of vertices. Here are some choices for `GLtype`:

```
GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
```

The first is basically a set of unrelated points. The last two types corresponds to a polygonal line and a closed polygonal loop. But there are also true "polygon" types:

```
GL_POLYGON, GL_TRIANGLES, GL_QUADS,
GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN
```

The difference between the closed polygonal loops and polygons is that the latter has an interior (provided it is defined properly) while closed loops cannot refer to an interior (even when it is well-defined). We can choose to fill the interior with some color or pattern, and we can choose to display or not display the edges of the polygon.

Attributes. Each of our geometric objects have a fixed set of attributes depending on their types. Each attribute is bound to some value which the user can set. These attributes determine the display properties of the object. Thus a point has the color attribute and a size attribute. Thus, we might say `glPointSize(2.0)` and this means that each point will be rendered as 2 pixels across. Lines have color, thickness and type (solid, dashed, dotted) attributes. Polygons have even more attributes.

Display Modes. When we construct geometric objects, we can asked the objects to be displayed immediately. This is called the **immediate-mode**. A more sophisticated model assumes that there is a special **display processor** whose job is to display data on the screen, and has its own special **display memory**. Once a geometric object has sent to the display membory, the display processor can store the data in some **display list** stored in this display memory. The graphics program can then issue display instructions to the display processor, without needing to resend the details about the display list. This is called the **retained-mode**, and can be advantageous in a thinwire situation.

In OpenGL, we can define and manipulate display lists. They are defined similar to geometric objects, but enclosed in `glNewList` and `glEndList` instead of `glBegin` and `glEnd`. Each list must have a unique identifier (an integer). It also has a compilation flag. Suppose we want to define a box. The format is therefore

```
glNewList( BOX, GL_COMPILE ); //BOX is unique int identifier
  glBegin(GL_POLYGON);
    glColor3f(1.0, 0.0, 0.2);
    glVertex2f(-1.0, -1.0);
    ...
    glVertex2f(-1.0, 1.0);
  glEnd();
glEndList();
```

Instead of `GL_COMPILE`, we could have `GL_COMPILE_AND_EXECUTE`. To use the above, we execute the function `glCallList(BOX);` Note that as usual, the present state determines what other transformations will apply to the BOX. If these transformations change, the BOX will appear to move.

§2. Event Model Programming

In OpenGL, the GUI aspects and the windowing support is found in additional support libraries. The first of these libraries is GLU (graphics utility library). The latter is found in GLUT (GL Utility Toolkit).

Traditional interactions between a computer program and the outside world (or the user) is essentially predetermined by the program – the program prints an output, or requests an input. Of course the user's input can change the course of events in the program, but the order of interaction is predictable.

With the development of the GUI interfaces, we are faced with a completely different interaction model. The program interacts with a set of **logical input devices** – these may be the traditional keyboard, but it could also be the mouse and various window widgets. Moreover, the order of the interactions is quite unpredictable – the user can request a window to be closed anytime, and every mouse motion is potentially a request for interaction. This readiness to serve is characteristic of GUI-based programs. In broad outline, this is also typical of how operation systems provide services. This model of interaction is often called an event-based model or event-driven interaction. The idea is that

Let us now see how to construct programs based on an event-driven interaction. We will be specifically discussing OpenGL's version of this model. Assume that the mouse generates various events such as **move event** (when a mouse is moved with some depressed mouse button), **passive move event** (when the mouse is moved without any depressed mouse buttons), **mouse event** when a mouse button is depressed or released. Note that in this model, when we first depress a mouse button, we generate an event, but maintaining the depressed mouse button generates no event. The next mouse event thing that can happen is one of two things: the mouse moves or the mouse button is released.

When a mouse event is generated, we return its position (among other information that depends on the event) – corresponding to some position on the screen. Of course this "position" is a logical concept, since the mouse is not literally on the screen. But as feedback to the use, we display this "position" on the screen using some **mouse cursor**. But who is supposed to handle this event (i.e., the information such as "position")? The answer lies in **call back functions**, In the GLUT model, we need to write a Mouse Event function whose prototype is:

```
void mouse_callback( int button, int state, int x, int y)
{
  if (button == GLUT_LEFT_BUTTON & state == GLUT_DOWN)
    exit();
}
```

All that this call back does is to exit the window program in case the depressed mouse button is the "GLUT_LEFT_BUTTON". This function must then be **registered** for handling mouse events:

```
glutMouseFunc( mouse_callback );
```

What other minimal callback functions do we need to construct and register?

```
int main(int argc, char **argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB );
  glutCreateWindow("square");
  myinit();
  glutReshapeFunc( myReshape ) ;  // called when window is resized
  glutMouseFunc(mouse ) ;
  glutMainLoop();
}//main
```