CMSC 498M: Chapter 2b
Overview of OpenGL - continued

Reading:
- See the *OpenGL Programming Guide*, (also known as "The Red Book") by the OpenGL Arch. Rev. Board, Shreiner, Woo, Neider, and Davis.

Overview:
- GLUT: Window Manipulation and User Interaction
- OpenGL Transformations
- 3D Viewing in OpenGL

GLUT:
Window Manipulation and User Interaction
Event-Driven Computing

Typical (noninteractive) Program:
- Read Data.
- Process Data.
- Output Results.

Event-Driven Computing: (System’s perspective)
- Check whether an event has occurred.
- if (an event has occurred)
  - call eventHandler(eventInfo).
- Repeat.

Event-Driven Computing: (Programmer’s perspective)
- Register "event-handler" pairs. (For each "event" call a function, called a callback, that performs "handles" this event and returns.)
- Pass control to the operating system.

Event Queue

Operating System/Window Management System: copies all handled events to an Event Queue.

Let EQ be the event queue (first-in, first-out queue).

New event(e):
if (e is handled) { append e to the end of EQ }

Processing Events:
while (true) {
  if (EQ is not empty) {
    call eventHandler(EQ.front( ));
    remove front event from EQ;
  }
}

Event Handlers are functions that your program provides.
OpenGL and GLUT

OpenGL is window-system independent:
- Makes it portable.
- Can be targeted for different platforms: PCs, game consoles, interactive TV set-top boxes.
- Independent of:
  - operating system
  - windows system
  - display size and display properties (only assumes a raster-device).

But: OpenGL still needs to interface with windowing system.
For instance: X-windows, Microsoft’s windows system

GLUT (to the rescue): GLUT (GL Utility Toolkit) is a window-system independent programming interface for OpenGL.

GLUT: OpenGL Utility Toolkit

GLUT:
- is a simple programming interface to the windows system.
- is largely window-system independent.
- supports only pop-up menus (no pull-down menus).
- maintains its own event loop.
- accepts registration of callback functions from user programs.
- has its own (limited) set of fonts.

Some useful sites for GLUT:
- Mark Kilgard’s GLUT page: Full online documentation.
- Nate Robins GLUT for Win32: Has precompiled binaries for Microsoft Windows and installation instructions.
  http://www.xmission.com/~nate/glut.html
GLUT Initialization

\texttt{glutInit} (int* \texttt{argc}, char** \texttt{argv})
- Initialize GLUT library. Must be called first.
- Recognizes and processes GLUT-specific command-line arguments.

\texttt{glutInitWindowSize} (int \texttt{width}, int \texttt{height})
- Specifies the desired window size. (No guarantees that you will get this.)

\texttt{glutInitWindowPosition} (int \texttt{x}, int \texttt{y})
- Requests location (in pixels) of the upper left corner of the window. Note that (0,0) is the upper left corner of the display.

\texttt{glutInitDisplayMode} (unsigned int \texttt{mode})
- Specifies a number of options affecting general operation connected by "boolean or": E.g. GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE, etc...

\texttt{glutCreateWindow} (char* \texttt{window_name})
- Request that the window be created. (This only issues the request and returns. The window is not created immediately.)

GLUT Main Event Loop

\texttt{glutMainLoop} ()
- Starts the GLUT event processing loop.
- Never returns (except through callbacks).
- Calls registered function callbacks (user-defined event handlers) as appropriate.
- Should be called at most once.
- Before calling this, be sure to register your events (or this will be a very uninteresting program). 😊
Sample Program (Part I)

```c
int main ( int argc, char** argv ) { // program arguments
    glutInit( &argc, argv ); // initialize glut and gl
        // double buffering and RGB
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB );
    glutInitWindowSize ( 400, 300 ); // window size
    glutInitWindowPosition ( 0, 0 ); // window position in upper left
    glutCreateWindow ( argv[0] ); // create window
    ...
        // initialize callbacks here (described below) ...
    myInit ( ); // your own initializations
    glutMainLoop ( ); // turn control over to glut
    return 0; // (make the compiler happy)
}
```

GLUT Window Management

- **glutSwapBuffers ()**
  - Swaps front and back buffers.
  - This is part of a process called double-buffering, and is used to produce smooth, flicker-free animations.
- **glutPostRedisplay ()**
  - Mark the current window as "needing to be redrawn".
  - GLUT only redraws the window when you request it.
  - Next iteration of the glutMainLoop will refresh the window using the display-function callback.
  - To achieve a continuous animation, call this function inside a timer loop that wakes up every 1/30 second, say, or in an idle loop.
- **glutFullScreen ()**
  - Resize window to full screen. (See also glutEnterGameMode ( ).)
- **glutReshapeWindow ( int width, int height )**
  - Resize the display window using the parameters.
GLUT Callback Registration

**glutDisplayFunc** (void (*func) ( ) )
- Call the given function whenever the **window needs to be redrawn**.
  - When window is **first created**.
  - When the window is revealed because an overlapping window is removed.
  - When the program called **glutPostRedisplay()**.

**glutReshapeFunc** (void (*func) (int width, int height) )
- Call the given function whenever the **window is resized**.
- Called when the window is **first created**. (This is useful for some graphics **initializations**, such as texture mapping, which can only be done with an extant graphics window.)
- May involve updating the **projection** (gluPerspective) and **viewport** (glViewport) since they depend on the window’s **shape**.

**glutKeyboardFunc** (void (*func) (unsigned char key, int x, int y))
- Call the given function when a **keyboard key is hit**.
- The (x, y) arguments indicate where the mouse was when key was pressed.

**glutSpecialFunc** (void (*func)(int key, int x, int y))
- Call the given function when a special key is hit (function keys, arrows, page-up, page-down, etc.)

**glutMouseFunc** (void (*func) (int button, int state, int x, int y))
- Call the given function when a mouse button is hit or released.
- Button argument: GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON.
- State: GLUT_UP, GLUT_DOWN.
- Position (x, y): Relative to upper left corner.
GLUT Callback Registration (cont)

- **glutMotionFunc**  
  (void (*func) (int x, int y))  
  - Mouse motion while button pressed. (x, y) is current mouse position.

- **glutPassiveMotionFunc**  
  (void (*func) (int width, int height))  
  - Mouse motion without button press. (Warning: This can generate lots of events!)

- **glutIdleFunc**  
  (void (*func) ())  
  - Called whenever no other events are on the event queue.  
  - Passing NULL disables this.

- **glutTimerFunc**  
  (unsigned int msecs, void (*func) (int value), value)  
  - Callback every msecs milliseconds (or more): Best effort.  
  - The value parameter is used for setting multiple alarms.  
  - Function func called with the specified value parameter.

Sample Program (Part II)

```c
int main ( int argc, char** argv ) { // program arguments
    ... (given in Part I) ...
    glutDisplayFunc ( myDraw ); // set up callbacks
    glutReshapeFunc ( myReshape );
    glutMouseFunc ( myMouse );
    glutKeyboardFunc ( myKeyboard );
    glutTimerFunc (20, myTimeOut, 0 ); // (see below)
    ... (given in Part I) ...
}
```
Sample Program (Part III)

```c
void myDraw () { // called to display window
    // ...insert your drawing code here ...
}
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
    // ...advances the state of animation incrementally...
    glutPostRedisplay (); // request redisplay
    glutTimerFunc (20, myTimeOut, 0); // request next timer event
}
```

Sample Program (Part IV)

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

```c
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 ( ); // swap buffers (make visible)
}
```

DEMO
OpenGL provides support for transformations. There are a number of contexts in which transformations are used.

**Modelview Mode (GL_MODELVIEW):** Used for
- transforming objects in the scene and
- changing the coordinates into a form that is easier for OpenGL to deal with.

**Projection Mode (GL_PROJECTION):** Used for projecting objects onto the 2d image plane.

**Texture Mode (GL_TEXTURE):** Used for transforming (wrapping) textures onto surfaces of your objects.
Transformations in OpenGL

OpenGL has three matrix stacks. Matrix operations apply to the current stack. To specify which stack you want to manipulate, use:

```c
glMatrixMode (mode);
```

where (mode) is either:
- `GL_MODELVIEW`
- `GL_PROJECTION`
- `GL_TEXTURE`.

Modelview mode is the most common (and the default), so it is common to switch back to Modelview mode.

```c
glMatrixMode (GL_PROJECTION);
// ... do something in Projection mode ... 
glMatrixMode (GL_MODELVIEW);
```

---

OpenGL Matrix Stack Operations

**Matrix Stack**: Each matrix mode has a stack of matrices. All operations apply to the active matrix at the top of the stack.

- `glLoadIdentity ( )`
  - Set the active matrix to identity.
- `glLoadMatrixf (GLfloat m )`
- `glLoadMatrixd (GLdouble* m )`
  - Set the 16 values of the active matrix to those specified by m.
- `glMultMatrixf (GLfloat m )`
- `glMultMatrixd (GLdouble* m )`
  - Post-multiplies the active matrix by m.

**Beware**: OpenGL assumes column-major order, whereas C++ assumes row-major order.
OpenGL Matrix Stack Operations

Stack Operations: It is possible to save the current matrix state by pushing and popping.

`glPushMatrix` ( )
- Make a copy of the active matrix and push it on the stack.

`glPopMatrix` ( )
- Pop the active matrix off the stack.

Example:
1. `glLoadIdentity` ( );
2. `glLoadMatrixf` ( A );
3. `glPushMatrix` ( );
4. `glMultMatrixf` ( B );
5. `glPushMatrix` ( );
6. `glPopMatrix` ( );

OpenGL Transformation Operations

Standard Transformations: Rather than specifying your own matrix, it is more common to use one of the standard transformations. Below "GLtype" is either "GLfloat" or "GLdouble".

`glTranslate{fd}` ( GLtype x, GLtype y, GLtype z )
- Multiply the active matrix by the translation matrix that translates by (x, y, z).

`glRotate{fd}` ( GLtype angle, GLtype x, GLtype y, GLtype z )
- Multiply the active matrix by the rotation matrix that rotates CCW by angle degrees about the vector (x, y, z).

`glScale{fd}` ( GLtype sx, GLtype sy, GLtype sz )
- Multiply the active matrix by the scale matrix that scales x by sx, y by sy and z by sz.
Composing Transformations

Transformations can be composed through matrix multiplication.

**Efficient**: allows us to perform a series of transformations with a single matrix/vector multiplication.

**Order**: Because we post-multiply, the order of evaluation is from right to left:

\[ M_3 \cdot M_2 \cdot M_1 \cdot p \rightarrow (M_3 \cdot (M_2 \cdot (M_1 \cdot p))) \]

**Note**: The order in which you request transformations is the reverse of the order in which they are applied.

**Remember**: Matrix multiplication is **associative** \(((AB)C = A(BC))\), but not **commutative** \((AB \neq BA)\). So the order in which matrices are listed does matter.

---

**Example**: Rotate the plane counterclockwise by angle \(\theta\) about the point \(t = (t_x, t_y)\).

1. Translate to origin \(T_1\)
2. Rotate about origin \(R_\theta\)
3. Translate origin back to \(t\) \(T_2\)

\[ T_2 R_\theta T_1 \]
Transformations and Drawing

The active transformation matrix is automatically applied to all drawing. The typical order is:
- save the current matrix state (push)
- apply the desired transformation matrix to active matrix
- draw your object(s)
- restore the matrix state (pop)

Example: Suppose that myRect() draws a 1×1 rectangle centered at the origin. We want to draw a 2×1 rectangle centered at (px,py) and rotated (about the z-axis) by 20 degrees CCW.

Conceptual Solution:
- Call myRect() to draw original rectangle.
- Scale it to 2×1.
- Rotate it 20 degrees CCW about z-axis.
- Translate it to (px,py).

OpenGL Order: Is the reverse of this [Remember this!].

```
glPushMatrix();             // save the state
    glTranslatef(px, py, 0);   // translate
    glRotatef(20, 0, 0, 1);    // rotate
    glScalef(2, 1, 1);         // scale
    myRect();                 // draw
glPopMatrix();             // restore the state
```
Composite Objects and Compound Motion

**Composite Object:** is formed from multiple parts that are linked together (e.g., the arms, legs, head, torso of a person).

**Compound Motion:** involves coordinating the movement of the individual parts to form a motion of the composite object.

**Example:** Consider a bicycle consisting of three parts: frame, front wheel, and back wheel. The wheels are identical.

- `drawFrame( )`: Draws the frame with its center at the origin.
- `drawWheel( )`: Draws one wheel centered at the origin.

**Other variables:**
- `b` = location of bike origin relative to world origin.
- `wf/wb` = vector offset of front/back wheel relative to bike origin.
- `rf/rb` = rotation (in degrees) of the front and back wheel.

---

**Drawing the Bike:** Conceptual order:
- Draw front wheel at the origin.
- Rotate front wheel by `rf` degrees.
- Translate it to `wf`.
- Draw the back wheel at the origin.
- Rotate it by `rb` degrees.
- Translate it to `wb`.
- Draw frame.
- Translate everything to `b`.

For simplicity we assume the bike is in the x,y-plane so rotations are about the z-axis.

**OpenGL order:** We reverse this process by setting up the transformations first and then drawing.
Composite Objects and Compound Motion

OpenGL order:

```
glPushMatrix();
glTranslatef(b_x, b_y, b_z);
drawFrame();
```

```
Draw back wheel
```

```
Draw front wheel
```

```
glPushMatrix();
glTranslatef(wb_x, wb_y, wb_z);
glRotatef(rb, 0, 0, 1);
drawWheel();
glPopMatrix();
```

```
glPushMatrix();
glTranslatef(wf_x, wf_y, wf_z);
glRotatef(rf, 0, 0, 1);
drawWheel();
glPopMatrix();
```

```
glPopMatrix();
```

2D Projection and Viewport Transformation

**Projection Transformation**: Maps points from your idealized drawing area to a rectangular *viewport*.

**Viewport Transformation**: Maps points from the viewport a region of your graphics window. (Usually all of it, but you can specify any portion you like.)
2D Projection Transformation

Projection Transformation: is set up in Projection mode.

```c
glMatrixMode ( GL_PROJECTION );
glLoadIdentity ( );
gluOrtho2D ( left, right, bottom, top );
glMatrixMode ( GL_MODELVIEW );
```

Viewport Transformation

Viewport Transformation: is set up using glViewport.

```c
glViewport ( x, y, wid, hgt );
```
- `(x, y)` are the lower left corner of the viewport (in pixels).
- `(wid, hgt)` are the width and height of the viewport (in pixels).
- Use `glViewport(0, 0, winWid, winHgt)` to use full window.
- A good place to put this is in your reshape callback.
Transformations for 3D Viewing

**Modelview**: Map 3D world coordinates to 3D view coordinates.

**Projection**: Projects 3D objects onto the 2D view plane.

**Viewport**: Map the view plane to the graphics viewport.

Overview of the Viewing Process:
3D Viewing Pipeline: An Expanded View

Modeling Transformation → World Coordinates

Viewing Transformation → View (Camera) Coordinates

Projection Transformation → Projection Coordinates

Normalization and Clipping → Normalized Device Coordinates

Viewport Transformation → Device Coordinates

Specifying the Camera Position

World frame: The frame in which you represent your points.

View Frame: (also called camera or eye frame) Is specified by the following quantities, relative to world coordinates:

Eye Location: The center of projection.

Viewing Direction: A unit vector that is normal to the view plane, called the view-plane normal.

Camera twist: The camera’s rotation about the viewing axis, specified by indicating the “up” direction for the camera.
Specifying the View Frame: gluLookAt( )

In OpenGL the view frame is specified by calling:

```
    gluLookAt ( eye_x, eye_y, eye_z, at_x, at_y, at_z, up_x, up_y, up_z )
```

All arguments are of type GLdouble. Where the following are given in world coordinates:

- \( \text{eye} = (\text{eye}_x, \text{eye}_y, \text{eye}_z) \) is the location of the eye.
- \( \text{at} = (\text{at}_x, \text{at}_y, \text{at}_z) \) is a point that the viewer is looking at. The vector \( \text{at} - \text{eye} \) is the viewing direction.
- \( \text{up} = (\text{up}_x, \text{up}_y, \text{up}_z) \) is a vector indicating which direction is up relative to the camera. It is used to encode the camera's twist about the viewing direction. (It need not be orthogonal to the view direction, but it cannot be parallel to the view direction vector.)
Specifying the View Frame: *gluLookAt*( )

- *gluLookAt*( ) constructs a matrix that converts from world coordinates to view coordinates and multiplies it times the top of the modelview matrix stack.
- Normally this is the first matrix on the modelview stack.

```c
#include <GL/gl.h>

void draw() {
  glLoadIdentity();
  gluLookAt(...);  // V
  glPushMatrix();
  glRotatef(...);  // R
  glTranslatef(...);  // T
  // ... do some drawing ...
  glPopMatrix();
}
```

- Each point given in the drawing process will be transformed first by T, then by R, and finally converted into view coordinates by V.

---

Perspective Projection

**Perspective Projection:** Points are projected towards the center of projection (eye).
**Perspective Projection in OpenGL**

**glFrustum:** To specify the projection, you give a frustum (truncated pyramid) centered at the camera location.

\[ \text{glFrustum}( \text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) \]

- All arguments are of type `GLdouble`.
- This is typically done in `GL_PROJECTION` mode.
- All objects outside this frustum are clipped.

**Symmetric Viewing:** There is a simpler form of `glFrustum()` for when the viewing situation is symmetric about the z-axis:

\[ \text{gluPerspective}( \text{fovy}, \text{aspect}, \text{near}, \text{far}) \]

- `fovy`: y-field of view (angle given in degrees)
- `aspect`: window's aspect ratio \( w/h = (\text{right-left})/(\text{top-bottom}) \).
**gluPerspective( ) Operation**

*glFrustum() and gluPerspective():* Generate a transformation matrix and multiply it times the top of the current matrix stack (usually the projection stack).

**Example:**

```c
void myDisplay () {
    glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glLoadIdentity ();
    gluLookAt ( ... ); // set up view frame
    glMatrixMode ( GL_PROJECTION ); // set up projection
    glLoadIdentity ();
    gluPerspective ( fovy, aspect, near, far ); // or glFrustum( )
    glMatrixMode ( GL_MODELVIEW );
    myWorld.draw ( ); // draw everything
    glutSwapBuffers ( );
}
```

Assuming depth buffering is used.

---

**Summary**

**Topics Covered:**
- GLUT: Window Manipulation and User Interaction
- OpenGL Transformations
- 3D Viewing in OpenGL

Please see [OpenGL documentation](https://www.opengl.org/resources/documentation/) and tutorials on web pages (like NeHe) for further information.