The OpenGL API:

Before getting to the topic of how graphics are generated, let us begin with a discussion of the graphics API that we will be using this semester, OpenGL. We will also discuss two related libraries, GLU (the OpenGL utility library) and GLUT (the OpenGL Utility Toolkit). OpenGL is designed to be a machine-independent graphics library, but one that can take advantage of the structure of typical hardware accelerators for computer graphics.

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 operations are not provided. For this reason, a separate library, called GLUT or OpenGL Utility Toolkit, was created to provide these functions. It is the GLUT 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. Throughout, we will assume that programming is done in C++, but most of our examples will compile in C as well. (Do not worry for now if you do not understand the meanings of the various calls. Later we will discuss the various elements in more detail.) 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**

```c
#include <GL/glut.h>  // GLUT, GLU, and OpenGL defs
int main(int argc, char** argv)  // program arguments
{
    glutInit(&argc, argv);  // initialize glut and gl
    // double buffering and RGB
    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
}
```

The call to `glutMainLoop` turns control over to the system. After this, the only return to your program will occur due to various callbacks. (The final "return 0" is only there to keep the compiler from issuing a warning.) Here is an explanation of the first five function calls.

- **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 and the window system and does general...
initialization of GLUT and OpenGL. Any remaining arguments are then left for the user's program to interpret, if desired.

The next procedure, `glutInitDisplayMode`, performs initializations by informing OpenGL how to set up its various buffers. Recall that the frame buffer is a special 2-dimensional array in 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 (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. A partial list of possible arguments is given in Table 1.

<table>
<thead>
<tr>
<th>Display Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLUT_RGB</td>
<td>Use RGB colors</td>
</tr>
<tr>
<td>GLUT_RGBA</td>
<td>Use RGB plus α (recommended)</td>
</tr>
<tr>
<td>GLUT_INDEX</td>
<td>Use colormapped colors (not recommended)</td>
</tr>
<tr>
<td>GLUT_DOUBLE</td>
<td>Use double buffering (recommended)</td>
</tr>
<tr>
<td>GLUT_SINGLE</td>
<td>Use single buffering (not recommended)</td>
</tr>
<tr>
<td>GLUT_DEPTH</td>
<td>Use depth buffer (needed for hidden surface removal)</td>
</tr>
</tbody>
</table>

Color: First off, 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 (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 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 redraw 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.
Fig. 9: General structure of an OpenGL program using GLUT.

- **glutInit**
- **glutInitDisplayMode**
- **glutInitWindowSize/Position**
- **glutCreateWindow**
- **initialize callbacks**
- **your internal initializations**
- **gluMainLoop**

**glutReshapeFunc:**
```plaintext
if (first call) {
  OpenGL initializations
}
(re)set viewport/projection
```

**glutDisplayFunc:**
```plaintext
clear buffers
redraw scene
waiting for next event
```

**glutSwapBuffers**

**other event callbacks:**
```plaintext
update internal state
```
```plaintext
glutPostRedisplay
```
Creation has completed, the system generates a display even. This is how you know that you can now start drawing into the graphics window.

Another type of system event is a reshape event. This happens whenever the window's size is altered. 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 this size. Typically, the first two events that the system will generate for any newly created window are a reshape event (indicating the size of the new window) followed immediately by a display event (indicating that it is now safe to draw graphics in the window).

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.

Various input and system events and their associated callback function prototypes are given in Table 2.

<table>
<thead>
<tr>
<th>Input Event</th>
<th>Callback request</th>
<th>User callback function prototype (return void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse button</td>
<td>glutMouseFunc</td>
<td>myMouse(int b, int s, int x, int y)</td>
</tr>
<tr>
<td>Mouse motion</td>
<td>glutPassiveMotionFunc</td>
<td>myMotion(int x, int y)</td>
</tr>
<tr>
<td>Keyboard key</td>
<td>glutKeyboardFunc</td>
<td>myKeyboard(unsigned char c, int x, int y)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Event</th>
<th>Callback request</th>
<th>User callback function prototype (return void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Re)display</td>
<td>glutDisplayFunc</td>
<td>myDisplay()</td>
</tr>
<tr>
<td>(Re)size window</td>
<td>glutReshapeFunc</td>
<td>myReshape(int w, int h)</td>
</tr>
<tr>
<td>Timer event</td>
<td>glutTimerFunc</td>
<td>myTimer(int id)</td>
</tr>
<tr>
<td>Idle event</td>
<td>glutIdleFunc</td>
<td>myIdle()</td>
</tr>
</tbody>
</table>

Table 2: Common callbacks and the associated registration functions.
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
    ...
}
Typical Callback Setup

```c
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
...
}
```

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.

Examples of Callback Functions for System Events

```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
    // ...advance the state of animation incrementally...
    glutPostRedisplay();  // request redisplay
    glutTimerFunc(20, myTimeOut, 0);  // schedule next timer event
}
```

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

Lecture Notes 17 CMSC 427
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: // b indicates the 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
}
Examples of Callback Functions for User Input Events

```c
// 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
  }
}
```

```c
// 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
  }
}
```

Events on a regular basis, then insert a call to `glutTimerFunc` from within the callback function to generate the next one.

<table>
<thead>
<tr>
<th>GLUT Parameter Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLUT_LEFT_BUTTON</td>
<td>left mouse button</td>
</tr>
<tr>
<td>GLUT_MIDDLE_BUTTON</td>
<td>middle mouse button</td>
</tr>
<tr>
<td>GLUT_RIGHT_BUTTON</td>
<td>right mouse button</td>
</tr>
<tr>
<td>GLUT_DOWN</td>
<td>mouse button pressed down</td>
</tr>
<tr>
<td>GLUT_UP</td>
<td>mouse button released</td>
</tr>
</tbody>
</table>

Table 3: GLUT parameter names associated with mouse events. (Constants defined in `glut.h`)
The display callback function for our program is shown below. 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.

Let us present the code, and we will discuss the various elements of the solution in greater detail below.

Sample Display Function

```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
}
```

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;
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. 1.

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. 1: 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.

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 smoothy from one end to the other.

There are a number of drawing attributes other than color. For example, for points it is possible adjust their size (with \texttt{glPointSize()}). For lines, it is possible to adjust their width (with \texttt{glLineWidth()}) and create dashed or dotted lines (with \texttt{glLineStipple()}). It is also possible to pattern or stipple polygons (with \texttt{glPolygonStipple()}). When we discuss 3-dimensional graphics we will discuss many more properties that are used in shading and hidden surface removal.

After drawing the diamond, we change the color to blue, and then invoke \texttt{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 \texttt{GL POLYGON}, but this form is easier to use.

\textbf{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 \textit{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 \texttt{reshape} callback would contain the following call which resizes the viewport, whenever the window is resized.

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

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

// Drawing commands for the right half of window

glViewport(w/2, 0, w/2, h);
// ...drawing commands for the right half of window

glutSwapBuffers(); // swap buffers

Projection Transformation:
In the simple drawing procedure, we said that we were assumin
that the "idealized" drawing area was a unit square over the i
nterval [0, 1] with the origin
in the lower left corner. The transformation that maps the id
ealized 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 coordi
nates whenever the user changes
the size of the graphics window.

However, we need to inform OpenGL of where our "idealized" dr
awing area is so that OpenGL
can map it to our viewport. This mapping is performed by a tran
sformation matrix called
the
projection matrix
, which OpenGL maintains internally. (In future lectures, w
ew i l ld i s c u ss
OpenGL's transformation mechanism in greater detail. In th
e mean time some of this may
seem a bit arcane.)

Since matrices are often cumbersome to work with, OpenGL pro
vides a number of relatively
simple and natural ways of defining this matrix. For our 2-dim
ensional 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

\[
\text{gluOrtho2D}(\text{left}, \text{right}, \text{bottom}, \text{top})
\]

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,
\text{glColor3f()},
).

All arguments are of type
\text{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**

```c

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

---
restoring it later (by popping the stack). We will discuss the entire process of implementing a
ffi and projection transformations later in the semester. For now, we’ll give just basic
information on OpenGL’s approach to handling matrices and transformations.

OpenGL has a number of commands for handling matrices. In order to know which matrix
(Modelview, Projection, or Texture) to which an operation applies, you can set the current
matrix mode. This is done with the following command

```
glMatrixMode(mode);
```

where `mode` is either `GL_MODELVIEW`, `GL_PROJECTION`, or `GL_TEXTURE`. The default mode is `GL_MODELVIEW`.

GL_MODELVIEW is by far the most common mode, the convention in OpenGL programs is
to assume that you are always in this mode. If you want to modify the mode for some
reason, you first change the mode to the desired mode (`GL_PROJECTION` or `GL_TEXTURE`),
perform whatever operations you want, and then immediately change the mode back to
`GL_MODELVIEW`.

Once the matrix mode is set, you can perform various operations to the stack. OpenGL has an
unintuitive way of handling the stack. Note that most operations below (except `glPushMatrix()`) alter the contents of the matrix at the top of the stack.

- **glLoadIdentity()**: Sets the current matrix to the identity matrix.
- **glLoadMatrix*(M)**: Loads (copies) a given matrix over the current matrix. (The ‘*’ can be either ‘f’ or ‘d’ depending on whether the elements of `M` are `GLfloat` or `GLdouble`, respectively.)
- **glMultMatrix*(M)**: Post-multiplies the current matrix by a given matrix and replaces the current matrix with this result. Thus, if `C` is the current matrix on top of the stack, it will be replaced with the matrix product `C · M`. (As above, the ‘*’ can be either ‘f’ or ‘d’ depending on `M`.)
- **glPushMatrix()**: Pushes a copy of the current matrix on top the stack. (Thus the stack now has two copies of the top matrix.)
- **glPopMatrix()**: Pops the current matrix off the stack.

Warning: OpenGL assumes that all matrices are 4×4 homogeneous matrices, stored
in column-major order. (In contrast, most modern programming languages linearize 2-
dimensional arrays by storing them in row-major order.) That is, a matrix is presented
as an array of 16 values, where the first four values give column 0 (for `x`), then column 1
(for `y`), then column 2 (for `z`), and finally column 3 (for the homogeneous coordinate, usually
called `w`). For example, given a matrix `M` and vector `v`, OpenGL assumes the following

Fig. 22: Matrix stack operations.

Fig. 23: Transformation pipeline.
Since $\mathbf{M}$ is on the top of the stack, we need to first apply translation ($\mathbf{T}$) to $\mathbf{M}$, and then apply rotation ($\mathbf{R}$) to the result, and then do the drawing ($\mathbf{v}$). Note that the order of application is the exact reverse from the conceptual order. This may seem confusing (and it is), so remember the following rule.

**Drawing/Transformation Order in OpenGL’s**

First, conceptualize your intent by drawing about the origin and then applying the appropriate transformations to map your object to your desired location. Then implement this by applying transformations in reverse order, and do your drawing. It is always a good idea to enclose everything in a push-matrix and pop-matrix pair.

Although this may seem backwards, it is the way in which almost all object transformations are performed in OpenGL:

1. Push the matrix stack,
2. Apply (i.e., multiply) all the desired transformation matrices with the current matrix, but *in the reverse order* from which you would like them to be applied to your object,
3. Draw your object (the transformations will be applied automatically), and
4. Pop the matrix stack.

---

**Projection Revisited:** Last time we discussed the use of `gluOrtho2D()` for defining simple 2-dimensional projection. This call does not really do any projection. Rather, it computes the desired projection transformation and multiplies it times whatever is on top of the current matrix stack. So, to use this we need to do a few things. First, set the matrix mode to `GL_PROJECTION`, load an identity matrix (just for safety), and the call `gluOrtho2D()`. Because of the convention that the Modelview mode is the default, we will set the mode back when we are done.

If you only set the projection once, then initializing the matrix to the identity is typically redundant (since this is the default value), but it is a good idea to make a habit of loading the identity for safety. If the projection does not change throughout the execution of our program, and so we include this code in our initializations. It might be put in the reshape callback if reshaping the window alters the projection.
Since $M$ is on the top of the stack, we need to first apply translation ($T$) to $M$, and then apply rotation ($R$) to the result, and then do the drawing ($\vec{v}$). Note that the order of application is the exact reverse from the conceptual order. This may seem confusing (and it is), so remember the following rule.

Drawing/Transformation Order in OpenGL's

First, conceptualize your intent by drawing about the origin and then applying the appropriate transformations to map your object to its desired location. Then implement this by applying transformations in reverse order, and do your drawing. It is always a good idea to enclose everything in a push-matrix and pop-matrix pair.

Although this may seem backwards, it is the way in which almost all object transformations are performed in OpenGL:

1. Push the matrix stack,
2. Apply (i.e., multiply) all the desired transformation matrices with the current matrix, but in the reverse order from which you would like them to be applied to your object,
3. Draw your object (the transformations will be applied automatically), and
4. Pop the matrix stack.

The final and correct fragment of code for the rotation is shown in the code block below.

```c
Drawing an Rotated Rectangle (Correct)

glPushMatrix(); // save the current matrix (M)
    glTranslatef(x, y, 0); // apply translation (T)
    glRotatef(20, 0, 0, 1); // apply rotation (R)
    glRectf(-2, -2, 2, 2); // draw rectangle at the origin
    glPopMatrix(); // restore the old matrix (M)
```

Projection Revisited:

Last time we discussed the use of `gluOrtho2D()` for defining simple 2-dimensional projection. This call does not really do any projection. Rather, it computes the desired projection transformation and multiplies it times whatever is on top of the current matrix stack. So, to use this we need to do a few things. First, set the matrix mode to GL_PROJECTION, load an identity matrix (just for safety), and the call `gluOrtho2D()`. Because of the convention that the Modelview mode is the default, we will set the mode back when we are done.

If you only set the projection once, then initializing the matrix to the identity is typically redundant (since this is the default value), but it is a good idea to make a habit of loading the identity for safety. If the projection does not change throughout the execution of our program, and so we include this code in our initializations. It might be put in the reshape callback if reshaping the window alters the projection.
transformation at the top of the current transformation stack. (Recall OpenGL's transformation structure from the previous lecture on OpenGL transformations.) This should be done in Modelview mode.

Conceptually, this change of coordinates is performed last, after all other Modelview transformations are performed, and immediately before the projection. By the "reverse rule" of OpenGL transformations, this implies that this change of coordinates transformation should be the first transformation on the Modelview transformation matrix stack. Thus, it is almost always preceded by loading the identity matrix. Here is the typical calling sequence.

This should be called when the camera position is set initially, and whenever the camera is (conceptually) repositioned in space.

### Typical Structure of Redisplay Callback

```c
void myDisplay() {
    // clear the buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity(); // start fresh
    // set up camera frame
    gluLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
    myWorld.draw(); // draw your scene
    glutSwapBuffers(); // make it all appear
}
```

The arguments are all of type GLdouble. The arguments consist of the coordinates of two points and vector, in the standard coordinate system. The point eye = (ex, ey, ez)T is the viewpoint, that is the location of the viewer (or the camera). To indicate the direction that the camera is pointed, a central point at which the camera is directed is given by at = (ax, ay, az)T. The "at" point is significant only in that it defines the viewing vector, which indicates the direction that the viewer is facing. It is defined to be at − eye (see Fig. 31).

These points define the position and direction of the camera, but the camera is still free to rotate about the viewing direction vector. To fix last degree of freedom, the vector →up = (ux, uy, uz)T provides the direction that is "up" relative to the camera. Under typical circumstances, this would just be a vector pointing straight up (which might be (0, 0, 1)T in your world coordinate system). In some cases (e.g. in a flight simulator, when the plane banks to one side) you might want to have this vector pointing in some other direction (e.g., up relative to the pilot's orientation). This vector need not be perpendicular to the viewing vector. However, it cannot be parallel to the viewing direction vector.
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt( ... ); // set up camera frame
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity(); // set up projection
    gluPerspective(fovy, aspect, near, far); // or glFrustum(...)
    glMatrixMode(GL_MODELVIEW);
    myWorld.draw(); // draw everything
    glutSwapBuffers();
}
Each light source may either be enabled (turned on) or disabled (turned off). By default they are all disabled. Again, this is done using `glEnable()` (and `glDisable()`). The properties of each light source is set by the command `glLight*()`.

Let us consider a light source 0, whose position is \((2, 4, 5, 1)\) in homogeneous coordinates, and which has a red ambient intensity, given as the RGB triple \((0.9, 0.0, 0.0)\), and white diffuse and specular intensities, given as the RGB triple \((1.5, 1.5, 1.5)\). (Normally all the intensities will be of the same color, albeit of different strengths. We have made them different just to emphasize that it is possible.) There are no real units of measurement involved here. Usually the values are adjusted manually by a designer until the image "looks good."

Light intensities are actually expressed in OpenGL as RGBA, rather than just RGB triples. The 'A' component can be used for various special effects, but for now, let us just assume the default situation by setting 'A' to 1. Here is an example of how to set up such a light in OpenGL. The procedure `glLight*()` can also be used for setting other light properties, such as attenuation.

```c
// Setting up a simple lighting situation

GLfloat ambientIntensity[4] = {0.9, 0.0, 0.0, 1.0};
gLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientIntensity);

GLfloat lt0Intensity[4] = {1.5, 1.5, 1.5, 1.0}; // white
gLightfv(GL_LIGHT0, GL_DIFFUSE, lt0Intensity);
gLightfv(GL_LIGHT0, GL_SPECULAR, lt0Intensity);

GLfloat lt0Position[4] = {2.0, 4.0, 5.0, 1.0}; // location
gLightfv(GL_LIGHT0, GL_POSITION, lt0Position);

gLightf (GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);  // attenuation params (a,b,c)
gLightf (GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.0);
gLightf (GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.1);
gEnable(GL_LIGHT0);
```

Defining Surface Materials (Colors): When lighting is in effect, rather than specifying colors using `glColor()` you do so by setting the material properties of the objects to be rendered. OpenGL computes the color based on the lights and these properties. Surface properties are assigned to vertices (and not assigned to faces as you might think). In smooth shading, this vertex information (for colors and normals) are interpolated across the entire face. In flat shading the information for the first vertex determines the color of the entire face. Every object in OpenGL is a polygon, and in general every face can be colored in two different...
In most graphic scenes, polygons are used to bound the faces of solid polyhedra objects and hence are only to be seen from one side, called the front face. This is the side from which the vertices are given in counterclockwise order. By default OpenGL only applies lighting equations to the front side of each polygon and the back side is drawn in exactly the same way. If in your application you want to be able to view polygon faces from both sides, it is possible to change this default (using `glLightModel()` so that each side of each face is colored and shaded independently of the other. We will assume the default situation.

Surface material properties are specified by `glMaterialf()` and `glMaterialfv()`.

```c
GLfloat color[] = {0.0, 0.0, 1.0, 1.0}; // blue
GLfloat white[] = {1.0, 1.0, 1.0, 1.0}; // white

// set object colors
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100);

glPushMatrix();
    glTranslatef(...); // your transformations
    glRotatef(...);
    glBegin(GL_POLYGON); // draw your shape
        glNormal3f(...); glVertex(...); // remember to add normals
        glNormal3f(...); glVertex(...);
        glNormal3f(...); glVertex(...);
    glEnd();

glPopMatrix();
```

Recall from the Phong model that each surface is associated with a single color and various coefficients are provided to determine the strength of each type of reflection: emission, ambient, diffuse, and specular. In OpenGL, these two elements are combined into a single vector given as an RGB or RGBA value. For example, in the traditional Phong model, a red object might have a RGB color of (1, 0, 0) and a diffuse coefficient of 0.5. In OpenGL, you would just set the diffuse material to (0.5, 0, 0). This allows objects to reflect different colors of ambient and diffuse light (although I know of no physical situation in which this arises).

Other options: You may want to enable a number of GL options using `glEnable()`. This procedure takes a single argument, which is the name of the option. To turn each option off, you can use `glDisable()`. These optional include:
Figure 9: Top: Colored Phong-shaded spheres with edge lines and highlights. Bottom: Colored spheres shaded with hue and luminance shift, including edge lines and highlights. Note: In the first Phong shaded sphere (violet), the edge lines disappear, but are visible in the corresponding hue and luminance shaded violet sphere. In the last Phong shaded sphere (white), the highlight vanishes, but is noticed in the corresponding hue and luminance shaded white sphere below it. The spheres in the second row also retain their "color name".

Figure 10: Left to Right: a) Phong shaded object. b) New metal-shaded object without edge lines. c) New metal-shaded object with edge lines. d) New metal-shaded object with a cool-to-warm shift.

Figure 11: Left to Right: a) Phong model for colored object. b) New shading model with highlights, cool-to-warm hue shift, and without edge lines. c) New model using edge lines, highlights, and cool-to-warm hue shift. d) Approximation using conventional Phong shading, two colored lights, and edge lines.
Figure 9: Top: Colored Phong-shaded spheres with edge lines and highlights. Bottom: Colored spheres shaded with hue and luminance shift, including edge lines and highlights. Note: In the first Phong shaded sphere (violet), the edge lines disappear, but are visible in the corresponding hue and luminance shaded violet sphere. In the last Phong shaded sphere (white), the highlight vanishes, but is noticed in the corresponding hue and luminance shaded white sphere below it. The spheres in the second row also retain their "color name".

Figure 10: Left to Right: a) Phong shaded object. b) New metal-shaded object without edge lines. c) New metal-shaded object with edge lines. d) New metal-shaded object with a cool-to-warm shift.

Figure 11: Left to Right: a) Phong model for colored object. b) New shading model with highlights, cool-to-warm hue shift, and without edge lines. c) New model using edge lines, highlights, and cool-to-warm hue shift. d) Approximation using conventional Phong shading, two colored lights, and edge lines.