CMSC 498M: Chapter 3
Overview of OpenGL

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

Overview:
- Introduction to OpenGL
- GLUT and user interaction
- OpenGL transformations and 3D viewing
- Lighting and texturing

---

OpenGL

**Standard:** Most widely-used/supported 2D/3D graphics API
- Windows NT/95/98/00, UNIX, Linux, MacOS, OS/2, Python, ...
- Bindings for C, C++, Java, Fortran, Ada
- ATI, HP/Compaq, E&S, IBM, Intel, Intergraph, NVIDIA, Microsoft, SGI

**Independent:** of hardware, OS, window system.

**Windowing not included:** Does not include commands for windowing tasks or user interaction.
OpenGL Resources

Online Documentation:
http://www.opengl.org/documentation/
http://msdn.microsoft.com/library/

Tricks/Tips/Examples/Tutorials:
http://nehe.gamedev.net/
http://www.gamedev.net/

Sample Code:
http://www.opengl.org/code/

GLUT: (GL toolkit)
http://www.opengl.org/resources/libraries/glut/
http://www.xmission.com/~nate/glut.html

OpenGL Pipeline Architecture (Simplified)

Image source: Matthew Ward
OpenGL Pipeline Architecture (Simplified)

Display List:
OpenGL drawing commands, pre-compiled for efficiency.

Evaluator:
Vertex preprocessing.

Per-Vertex Operations:
Geometric transformations applied to each vertex.

Primitive Assembly:
Vertices are grouped together to form triangles, polygons, etc.

Rasterization:
Primitives converted into pixels, called fragments, and colored.

Per-Fragment Operations:
Tests (e.g. depth/stencil test) to determine fragment visibility.

Pixel Operations and Texture Memory:
Pixels can be copied, texture mapped, or saved.
OpenGL Buffers

**Color Buffers:** The buffers that are normally drawn into.
- **Double-buffered system:** front and back buffers. (The front buffer is visible, and drawing take place into the back buffer.)
- **Other examples:** stereoscopic buffers (for left and right images).

**Depth Buffer:**
- Used for **hidden surface removal** by storing a depth value for each pixel. Only the **closest** pixel is drawn.

**Stencil Buffer:**
- This allows drawing to be **restricted** to certain portions of the screen.

**Accumulation Buffer:**
- Used for accumulating a **series of images** into a final, composite image. In particular it can be used for **antialiasing** and effects like **motion blur**.

Lighting and Clipping

**Lighting:**
- The color of each vertex is determined base on the object's **material properties** and the relationship to light sources.

**Clipping:**
- Once a primitive has been assembled, it is **clipped** so that it lies within a 3-dimensional region, called the **view frustum**.
- When a polygon is **partially inside**, new vertices are created as needed and vertex attributes are interpolated.
Projection and Rasterization

**Projection:**
Vertices are projected (either perspective or orthogonal).

**Viewport:**
The projected vertices are mapped to the visible portion of the screen (viewport).

**Rasterization:**
Primitives are converted into pixels (fragments).

**Per-Fragment Operations:**
- **Depth Test:** Hidden by a closer fragment?
- **Stencil Test:** Used to restrict drawing to selected portions of the window.

---

Texturing and Fog

**Texture:**
When enabled, a fragment’s texture coordinates are used to index into a texture image, generating a texel. The texel modifies the fragment’s color based on the current texture environment, which may involve blending with the existing color.

**Maxification/Minification:** Different rules can be applied to interpolate values when the texel is smaller (requiring maxification) or larger (requiring minification).

**Fog:**
After texturing, a fog function may be applied to the fragments. This blends a fog color based on the distance of the viewer from the fragment.
Immediate Mode and Display Lists

Immediate Mode:
- By default, OpenGL operates in immediate mode, where the drawing commands are executed immediately.

Display Lists:
- Frequently executed commands that are stored for later execution.
- Display lists reside on the server (the GPU).
- Once created they cannot be modified, but they can be transformed geometrically.
- They store both geometric information (vertices) and other attributes (color, texture coordinates, surface normals).
- Display lists can reference other display lists. This is useful for the display of composite objects (e.g., the wheels of a car).

OpenGL Drawing Primitives
OpenGL Naming Conventions (for C/C++)

Functions: begin with gl (example: glBegin - draw an object)

Constants: begin with GL_ (example: GL_POLYGON - a polygon)

Types: begin with GL (example: GLfloat - single-precision float)

OpenGL Data Types

<table>
<thead>
<tr>
<th>Suffix</th>
<th>OpenGL Datatype</th>
<th>C/C++ Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>GLbyte</td>
<td>signed char</td>
</tr>
<tr>
<td>s</td>
<td>GLshort</td>
<td>short</td>
</tr>
<tr>
<td>i</td>
<td>GLint</td>
<td>int</td>
</tr>
<tr>
<td>ab</td>
<td>GLubyte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>us</td>
<td>GLushort</td>
<td>unsigned short</td>
</tr>
<tr>
<td>ui</td>
<td>GLuint</td>
<td>unsigned int</td>
</tr>
<tr>
<td>f</td>
<td>GLfloat</td>
<td>float</td>
</tr>
<tr>
<td>d</td>
<td>GLdouble</td>
<td>double</td>
</tr>
</tbody>
</table>

OpenGL Vertices and Primitives

OpenGL supports a number of different drawing primitives. Each primitive is specified by enumerating the vertices that define it.

General Form:
```
glBegin ((object type));
glVertex... (...);
glVertex... (...);
glVertex... (...);
glEnd ();
```

Note: There are a number of other attributes that can be placed within the glBegin...glEnd pair. These affect things like color, texture, and surface normals. We will discuss these later.

Vertices: May be 2D (x,y), 3D (x,y,z), or 4D (x,y,z,w), where the w coordinate is usually 1. Called homogeneous coordinates.
Specifying Vertices

Vertex Arguments: All objects in OpenGL are constructed from convex polygons, which are represented by their vertex coordinates. The argument type is specified by the suffix to the OpenGL function name:

\[
\text{<func}_\text{name}> \; \text{<dim>} \; \text{<type>} \; (\text{<argument list>})
\]

Examples:
- 2D point in GLint (int) coordinates:
  \[
  \text{glVertex2i} \; (200, \text{-}150);
  \]
- 3D point in GLfloat (float) coordinates:
  \[
  \text{glVertex3f} \; (200.3f, \text{-}150f, 40.75f);
  \]

Vector (array) Arguments: Add suffix "v" to the function name
- 3D point in GLdouble (double) coordinates given as a vector:
  \[
  \text{GLdouble pt}[3] = \{200.3, \text{-}150, 40.75\};
  \text{glVertex3dv} \; (\text{pt});
  \]

Object Types - Isolated Points

\text{GL POINTS}: Draws a set of isolated points.

\[
\begin{align*}
\text{glBegin} & \; (\text{GL POINTS}) ; & \text{(0,1)} & \text{(1,1)} \\
\text{ glVertex2i} & \; (0, 0) ; & \text{ } & \text{ } \\
\text{ glVertex2i} & \; (0, 1) ; & \text{ } & \text{ } \\
\text{ glVertex2i} & \; (1, 0) ; & \text{ } & \text{ } \\
\text{ glVertex2i} & \; (1, 1) ; & \text{ } & \text{ } \\
\text{ glEnd} & \; ( ) ; & \text{(0,0)} & \text{(1,0)}
\end{align*}
\]
Object Types - Line Loop (Polyline)

GL_LINE_LOOP: Draws a closed polygonal line (segments joined end to end).

```c
glBegin ( GL_LINE_LOOP );
    glVertex2i ( 0, 0 );
    glVertex2i ( 0, 1 );
    glVertex2i ( 1, 1 );
    glVertex2i ( 1, 0 );
glEnd ( );
```

Variants:
- GL_LINE_STRIP: Polygonal line, but not closed off to form a loop.
- GL_LINES: A sequence of line segments, not connected to each other.

Object Types - Polygon

GL_POLYGON: Draws a filled convex polygon.

```c
glBegin ( GL_POLYGON );
    glVertex2i ( 0, 0 );
    glVertex2i ( 0, 1 );
    glVertex2i ( 1, 1 );
    glVertex2i ( 1, 0 );
glEnd( );
```

Note: OpenGL assumes that all polygons are convex, meaning that all interior angles are at most 180 degrees. OpenGL is largely "silent" about errors, so be careful.
**Object Types - Triangles**

**GL_TRIANGLES:** Draws a series of filled triangles. Each sequence of three vertices defines a separate triangle.

```c
 glBegin ( GL_TRIANGLES );
  glVertex2i ( 0, 1 );
  glVertex2i ( 0, 0 );
  glVertex2i ( 1, 1 ); // abc
  glVertex2i ( 1, 1 );
  glVertex2i ( 0, 0 );
  glVertex2i ( 1, 0 ); // cbd
  glVertex2i ( 1, 1 );
  glVertex2i ( 0, 0 );
  glVertex2i ( 1, 0 ); // cde
 glEnd ( );
```

**Note:** List vertices in a consistent order, say, **counterclockwise**. This is used to distinguish **front** (CCW) and **back** (CW) sides.

**Object Types - Triangle Strip**

**GL_TRIANGLE_STRIP:** Draws a series of triangles by joining the next vertex to the previous two.

```c
 glBegin ( GL_TRIANGLE_STRIP );
  glVertex2i ( 0, 1 ); // a
  glVertex2i ( 0, 0 ); // b
  glVertex2i ( 1, 1 ); // c→abc
  glVertex2i ( 1, 0 ); // d→bcd
  glVertex2i ( 2, 0 ); // e→cde
 glEnd ( );
```

**Orientation:** The first triangle is oriented **counterclockwise**. OpenGL will orient all remaining triangles in the strip in the same way.
**Drawing Attributes**

**Attributes:** affect the manner in which objects are drawn.
- These can be placed within each glBegin...glEnd pair.
- Once set, they affect subsequent objects, until changed again.

**Points:**
- Point size: glPointSize (2.0);
- Point color: glColor3f (0.0, 0.0, 1.0);
  (Sets RGB color components: Red, Green, Blue.)

**Lines:**
- Line width: glLineWidth (2.0);
- Line color: glColor3f (0.0, 0.0, 1.0);

**Polygons:**
- Front and/or back: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK.
  - The front is the side where vertices appear in counter-clockwise order.
- Face color: glColor3f (0.0, 0.0, 1.0);
- Lighting material properties: glMaterialf (…)

---

**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 \texttt{eventHandler(eventInfo)}.
- Repeat.

Event-Driven Computing: (Programmer’s perspective)
- Register "event-handler" pairs. (For each "event" call a function, called a \texttt{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 \texttt{EQ} be the event queue (first-in, first-out queue).

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

Processing Events:
\[
\text{while (true)} \{
  \text{if } (\text{EQ is not empty}) \{
    \text{call } \texttt{eventHandler(EQ.front())};
    \text{remove front event from EQ};
  \}
\}
\]
**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
  - window 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.
GLUT Initialization

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

```c
void glutInitWindowSize ( int width, int height )
{
    // Specifies the desired window size. (No guarantees that you will get this.)
}
```

```c
void glutInitWindowPosition ( int x, int 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.
}
```

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

```c
void glutCreateWindow ( char* 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

```c
void 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
    // ...advance 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) {
        case GLUT_LEFT_BUTTON: // b indicates button
            if ( s == GLUT_DOWN ) ... // left button pressed
                else if ( s == GLUT_UP ) ... // left button released
                    break;
            // ...
        }
    }
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;
            // ...
    }
    }
```
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
    glVertex2f(0.25, 0.25, 0.75, 0.75);  // draw a rectangle
    glutSwapBuffers();  // swap buffers (make visible)
}
```

DEMO
Transformations in OpenGL

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:

```
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.

```
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.

**WARNING!** OpenGL assumes column-major order, whereas C/C++ assume 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)`
- Post-multiply the active matrix by a translation matrix that translates by (x, y, z).

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

`glScale(fd) (GLtype sx, GLtype sy, GLtype sz)`
- Post-multiply the active matrix by a 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 conceptually 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** \(t\) to origin \(T_1\).
2. **Rotate** about origin \(R_\theta\).
3. **Translate** origin back to \(t\) \(T_2\).

![Diagram](image_url)
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 1x1 rectangle centered at the origin. We want to draw a 2x1 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 2x1.
- Rotate it 20 degrees CCW about z-axis.
- Translate it to (px,py).

OpenGL Order: Is the reverse of this.

```c
glPushMatrix();  // save the state
glTranslatef(px, py, 0);  // translate
glRotatef(20, 0, 0, 1);   // rotate
glScalef(2, 1, 1);        // scale
myRect();
glPopMatrix();
```

WARNING!
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.

---

**Composite Objects and Compound Motion**

**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.

---

Chapter 2, Slide 49
Copyright © David Mount and Amitabh Varshney

Chapter 2, Slide 50
Copyright © David Mount and Amitabh Varshney
Composite Objects and Compound Motion

OpenGL order:

1. `glPushMatrix()`: Begin pushing the current matrix to the viewpoint transformation stack.
2. `glTranslatef(bx, by, bz);`: Translate by BX, BY, BZ.
3. `drawFrame();`: Draw the frame.
4. `glPushMatrix()`: Begin pushing the current matrix to the viewpoint transformation stack.
5. `glTranslatef(wbx, wby, wbz);`: Translate by WBX, WBY, WBZ.
6. `glRotatef(rb, 0, 0, 1);`: Rotate by RB degrees around the X-axis.
7. `drawWheel();`: Draw the wheel.
8. `glPopMatrix()`: Pop the current viewpoint transformation matrix.
9. `glPushMatrix()`: Begin pushing the current matrix to the viewpoint transformation stack.
10. `glTranslatef(wfx, wfy, wfz);`: Translate by WFX, WFY, WFZ.
11. `glRotatef(rf, 0, 0, 1);`: Rotate by RF degrees around the X-axis.
12. `drawWheel();`: Draw the wheel.
13. `glPopMatrix()`: Pop the current viewpoint transformation matrix.
14. `glPopMatrix()`: Pop the current viewpoint transformation matrix.

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.)

Your drawing area (world coordinates) | Clipped | Viewport (screen coordinates)
2D Projection Transformation

Projection Transformation: is set up in Projection mode.

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

Viewport Transformation

Viewport Transformation: is set up using glViewport.

- 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.
3D Viewing in OpenGL

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:

```c
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:

- `eye = (eye_x, eye_y, eye_z)` is the **location of the eye**.
- `at = (at_x, at_y, at_z)` is a point that the viewer is looking at. The vector `at - eye` is the **viewing direction**.
- `up = (up_x, up_y, 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.
  ```gl
  glLoadIdentity();
  gluLookAt( ... );    // V
  glPushMatrix();
  glTranslatef( ... );  // T
  glRotatef( ... );    // R
  // ... do some drawing ...
  glPopMatrix();
  ```
- Each vertex given in the drawing process will be rotated, then translated, 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.

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

- 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:

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

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

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

Example:

\begin{verbatim}
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();
}
\end{verbatim}

Assuming depth buffering is used.

Lighting in OpenGL
Illumination

Illumination Models:
- Light is a very complex physical phenomenon.
- Most illumination models in graphics are based on simple geometric optics, as opposed to more complex (but realistic) wave optics.

Local Illumination Models: (OpenGL does this)
- Point light sources and direct interactions with light.
- No shadows, no indirect reflection
- Easy to implement and very efficient.

Shading:
- Involves determining the intensity of illumination (and its color) incident at a surface point.
- Can be computationally intensive, so it is common to compute it accurately at a few points (e.g., vertices) and interpolate in between.

Light/Surface Interaction

Light Reflection:
- Ambient: A background glow that illuminates all objects, irrespective of light source location.
- Diffuse: A uniform scattering of light, characterized by matte (non-shiny) objects, like cloth or foam rubber.
- Specular: Shiny (metallic-like) reflection, like a polished wood table.
- Reflective (Pure): No light scattering, like a mirror.
- Transparent/Translucent: Light passes through material.
Lighting in OpenGL

Lighting in OpenGL:
- Ambient, diffuse, and specular illuminations are supported.
- Attenuation is supported: illumination grows dimmer as the distance from the light source increases.
- Users define light sources: position, type, color.
- Front and back sides of polygons may be given different colors.

There is a bewildering number of options and parameters that need to be set up in OpenGL in order to use lighting:

Lighting/Shading model: Global parameters that affect how illumination and shading are computed.

Light properties: Options that define the location, colors, and intensities of the lights.

Object material properties: The color of the object and degree of ambient, diffuse, specular reflection.

Enabling: Lighting can be enabled or disabled.

```c
 glEnable ( GL_LIGHTING );
 glDisable ( GL_LIGHTING );
```
**Lights**

**Lights and Lighting:**
- At least 8 light sources: `GL_LIGHT0`, ..., `GL_LIGHT7`.
- `glLight*()` used to define individual light properties.
- `glLightModel*()` used to define global lighting properties.
- To determine the maximum number of lights supported in your implementation use:
  ```
  glGetIntegerv ( GL_MAX_LIGHTS, GLint* num_lights )
  ```
- You need to enable (turn on) each light that you plan to use.
  ```
  glEnable ( GL_LIGHT0 );
  glEnable ( GL_LIGHT1 ); ...
  ```

---

**glLight*()**

For **scalar-valued** parameters:
```
glLight{if} ( GLenum light, GLenum pname, (TYPE) param )
```

For **vector-valued** parameters:
```
glLight{if}v ( GLenum light, GLenum pname, (TYPE)* param )
```

Where:
- `(light)` can be: `GL_LIGHT0`, ..., `GL_LIGHT7`.
- `(pname)` can be:
  ```
  GL_POSITION:
  Light position.
  GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR:
  Light colors: (RGBA vector)
  GL_SPOT_DIRECTION, GL_SPOT_EXPONENT, GL_SPOT_CUTOFF:
  Spotlight parameters.
  GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION:
  Parameters for attenuation: (scalar)
  ```
Light Position

Lights at infinity:
- The light position is given as a homogenous \([x, y, z, w]\) vector. When \(w = 0\), this defines a light source at infinity.
- Provides additional efficiency, since light vector is the same for all points in the scene.

When to set light position:
- The \([x, y, z, w]\) vector is given in world coordinates. OpenGL needs to convert these into view-frame coordinates. This is done by multiplying this vector times the Modelview transformation.
- But, with each redraw cycle, the viewer typically moves and we alter the Modelview transformation (by calling \texttt{gluLookAt()}).
- The upshot is that light positions need to be set after issuing the \texttt{gluLookAt()} call.
- Other lighting settings can be done only once.

Sample Lighting Setup

```c
void setUpMyLighting () {
    // light intensity and location
    GLfloat ambientIntensity [4] = { 0.9, 0.0, 0.0, 1.0 }; // red
    GLfloat diffSpecIntensity [4] = { 1.2, 1.2, 1.2, 1.0 }; // white
    GLfloat position [4] = { 2.0, 4.0, 5.0, 1.0 }; // hit

    // global lighting options
    glShadeModel ( GL_SMOOTH ); // (or GL_FLAT)
    glEnable ( GL_LIGHTING ); // enable lighting
    glEnable ( GL_LIGHT0 ); // enable light 0
    // set up light 0 properties
    glLightfv ( GL_LIGHT0, GL_AMBIENT, ambientIntensity );
    glLightfv ( GL_LIGHT0, GL_DIFFUSE, diffSpecIntensity );
    glLightfv ( GL_LIGHT0, GL_SPECULAR, diffSpecIntensity );
    glLightfv ( GL_LIGHT0, GL_POSITION, position );
}
```
Drawing Objects with Lighting

\texttt{glMaterial}( )\texttt*: with lighting use \texttt{glMaterial}( ) to specify colors:

- Object colors under illumination are computed as a component-wise multiplication of the light colors and material colors.
- Material properties can be specified differently for ambient, diffuse, and specular reflection.
- In addition to this emission (glowing) can be defined:
  - Lights do not influence emission.
  - Emissive objects do not illuminate other objects.

\texttt{glNormal}( ):

- Used to specify vertex surface normals for shading.

---

\texttt{glMaterial*( )}

For scalar-valued parameters:

\texttt{glMaterial(if) ( GLenum (face), GLenum (pname), (TYPE) (param) )}

For vector-valued parameters:

\texttt{glMaterial(if)v ( GLenum (face), GLenum (pname), (TYPE)* (param) )}

where:

- (face) can be:
  \texttt{GL_FRONT, GL_BACK, GL_FRONT\_AND\_BACK}:
  Indicates which side of the polygon is being colored. (Recall that front face is the side from which vertices are listed in CCW order.)

- (pname) can be:
  \texttt{GL\_AMBIENT, GL\_DIFFUSE, GL\_SPECULAR, GL\_EMISSION}:
  Material colors (RGBA vectors). (You can set both ambient and diffuse at once using \texttt{GL\_AMBIENT\_AND\_DIFFUSE}.)

\texttt{GL\_SHININESS}:
  Specular illumination exponent (called $\alpha$ above).
void doMyDrawing() {
    GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGBA object color (red)
    // set material color
    glMaterialfv(GL_FRONT_AND_BACK,
                 GL_AMBIENT_AND_DIFFUSE, red);

    glBegin(GL_POLYGON); // draw polygon
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glEnd();
}

In flat shading only
one normal is needed.

void doMyDrawing() {
    GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGBA object color (red)
    // set material color
    glMaterialfv(GL_FRONT_AND_BACK,
                 GL_AMBIENT_AND_DIFFUSE, red);

    glBegin(GL_POLYGON); // draw polygon
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glEnd();
}

You can assign different
colors to different
vertices. OpenGL blends.

In flat shading only
one normal is needed.

void doMyDrawing() {
    GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGBA object color (red)
    // set material color
    glMaterialfv(GL_FRONT_AND_BACK,
                 GL_AMBIENT_AND_DIFFUSE, red);

    glBegin(GL_POLYGON); // draw polygon
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glEnd();
}

You can assign different
colors to different
vertices. OpenGL blends.

In flat shading only
one normal is needed.

void doMyDrawing() {
    GLfloat red[4] = {1.0, 0.0, 0.0, 1.0}; // RGBA object color (red)
    // set material color
    glMaterialfv(GL_FRONT_AND_BACK,
                 GL_AMBIENT_AND_DIFFUSE, red);

    glBegin(GL_POLYGON); // draw polygon
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glNormal3f(...); glVertex3f(...);
    glEnd();
}

You can assign different
colors to different
vertices. OpenGL blends.

In flat shading only
one normal is needed.
Texture Maping in OpenGL

Texture and Surface Detail

We have seen how to provide color to objects using:

- **Solid colors**: through rasterization.
- **Lighting and shading**: through various lighting and shading models.

Next we consider how to add realism through **surface detail**.

Examples:

- **Natural surfaces**: stone, wood, gravel, grass.
- **Printing and painting**: printed labels, billboards, newspapers.
- **Clothing and fabric**: woven and printed patterns, upholstery.
**Image Texturing**

![Image of a room with a lamp and a teapot](image)

*Image courtesy, Foley, van Dam, Feiner, Hughes*

---

**Texture Mapping Process**

- **Texture Space** $(s, t)$
- **Object Space** $(x, y, z)$
- **Screen Space (+depth)** $(x_s, y_s, z_s)$

**Texture mapping function:** Maps from **texture** to **object** space.

**Inverse mapping function:** Maps the other way. This is actually what we need in texture mapping—which texel corresponds to a given **surface point**.
Texturing in OpenGL: Basic Steps

One-Time Initialization: (After window is created.)
- Create and specify a texture object:
  • Create a new texture object. (OpenGL provides identifier.)
  • Provide the associated texture image to OpenGL.

For each Redrawing:
- Enable texture mapping.
- Draw the textured polygons:
  • Identify which texture is to be used.
  • Specify texture coordinates with vertices.
- Disable texture mapping:
  • when returning to normal drawing mode.

Creating Texture Object(s)

```c
void glGenTextures ( GLsizei n, GLuint* textureIDs );
- Returns (n) currently unused texture IDs in (textureIDs).
- Each texture ID is an integer greater than 0.

void glBindTexture ( GLenum target, GLuint textureID );
where (target) is GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D.
- if (textureID) is being used for the first time a new texture object is created and assigned the ID = (textureID). This is now the active texture.
- if (textureID) has been used before, the texture object with ID = (textureID) becomes active.
```
Specifying a 2-d Texture Object

```c
glTexImage2D ( GLenum target, GLint level,
   GLint internalFormat, GLsizei width, GLsizei height,
   GLint border, GLenum format, GLenum type,
   const GLVoid* texels );
```

Example:

```c
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0,
   GL_RGBA, GL_UNSIGNED_BYTE, myImage );
```

• (format) and (type) are used to specify the way in which the
texels are stored in your image array.
• (internalFormat) specifies how OpenGL should store the data
  internally.
• (width) and (height) give the image size.
• (level) and (border) have other uses (see documentation).

Specifying How Texture is Applied

How is the color of the texture pixel combined with the existing
pixel?

The main issue to do with whether the texture color is
combined with existing object color after lighting (modulation)
or is just painted on (replacement).

```c
glTexEnv{if} ( GLenum {target}, GLenum {pname}, {TYPE} {value} );
```

where (target) is: GL_TEXTURE_ENV.

```c
<pname>
   GL_TEXTURE_ENV_MODE
<value> (some common choices)
   GL_MODULATE (mix with lighting) or,
   GL_REPLACE (just paint this color).
```
Specifying how Texture is Applied

There are also parameters that specify how the texture is to be mapped. These involve issues such whether the texture should wrap around (repeat) and how to magnify/shrink it.

```
glTexParameter{if} ( GLenum (target), GLenum (pname),
    (TYPE) (value) );
```

where (target) can be: GL_TEXTURE_1D, GL_TEXTURE_2D, ...

<table>
<thead>
<tr>
<th>(pname)</th>
<th>(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_TEXTURE_WRAP_S</td>
<td>GL_CLAMP, GL_REPEAT</td>
</tr>
<tr>
<td>GL_TEXTURE_WRAP_T</td>
<td>GL_CLAMP, GL_REPEAT</td>
</tr>
<tr>
<td>GL_TEXTURE_MAG_FILTER</td>
<td>GL_NEAREST, GL_LINEAR, ...</td>
</tr>
<tr>
<td>GL_TEXTURE_MIN_FILTER</td>
<td>GL_NEAREST, GL_LINEAR, ...</td>
</tr>
</tbody>
</table>

OpenGL’s default value for GL_TEXTURE_MIN_FILTER is very strange. Always specify a value for this parameter.

WARNING!

Enable the Texture and Draw

```
glEnable ( GL_TEXTURE_2D );
```

- Enable 2-d texturing.

```
glTexCoord2f ( GL_FLOAT s, GL_FLOAT t );
```

- Specify texture coordinates for the next vertex. (As with glNormal() and glColor(), this applies to all subsequent vertices until changed.)

- Irrespective of the image size, texture coordinates s,t always vary from 0 to 1.

```
glDisable ( GL_TEXTURE_2D );
```

- Disable 2-d texturing, to return to simple coloring.
Example

**Texture Initialization:**

```plaintext
glGenTextures(...); // create new texture objects
glBindTexture(...);  // make this texture active
glTexParameteri(...); // define texture properties
// ... input texture array from file or generate ...
glTexImage2D(...);   // provide the texture to OpenGL
```

**Displaying a Textured Object:**

```plaintext
glEnable(GL_TEXTURE_2D);  // enable texturing
glBindTexture(...);      // activate the desired texture
glBegin(GL_TRIANGLES);  // draw the object
    glTexCoord2f(...);
glNormal3f(...);
glVertex3f(...);
// ... (draw other vertices in the same way)
glEnd();
```

Minimization Filtering and MIP-mapping

What if one screen-space pixel overlaps many texture pixels?

Ideally we should average these pixels, but this takes time. So OpenGL just takes one.

**Result:** A jagged appearance, aliasing.

**MIP-mapping:** Precompute averages and build hierarchy based on powers of 2.

To render: OpenGL gets appropriate level in the MIP-map, and use this pixel. This smooths out the jagged lines.
MIP-mapping in OpenGL

How to Set Up a MIP Mapped Texture:

```c
// request MIP-map for min filter
glTexParameteri ( GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_NEAREST_MIPMAP_LINEAR );

// present image to OpenGL
glTexImage2D ( /* …as before… */ , myImage );

// compute MIP-maps
gluBuild2DMipmaps ( GL_TEXTURE_2D,  // (always the same)
    GL_RGB,  // internal format
    imgWidth, imgHeight,  // image size
    GL_RGB,  // image format
    GL_UNSIGNED_BYTE,  // image type
    myImage );  // the image
```

Pixel Buffers and Operations
Pixel Buffers and Operations

Pixel Buffers: OpenGL maintains from one to many pixel buffers. These buffers store different types of information and have different functions. We will discuss:

- Buffer concepts: bitmaps, pixmaps, depth.
- OpenGL Buffers: color-, depth-, accumulation-, and stencil buffers.
- Transfer: reading and writing pixel buffers.
- Imaging Operations: user operations, bitblt, blend.

Bitmaps and Pixmaps

Pixel: A picture element.

Bitmap: A 2D array of single-bit pixels (0/1 or black/white).

Pixmap: Stack of bitmaps. The number of bits per pixel is called its depth.

To represent full RGB color, it is sufficient to have 24-bit depth, 8 bits each for red, green, and blue.
**Bitmaps and Pixmaps**

- **Bitmap (depth = 1)**
- **Pixmap (depth = 8)**
- **Pixmap (depth = 24)**

**OpenGL Buffers**

- **Color Buffer**: Stores image color information.
  - RGB: Red, green, blue
  - RGBA or RGBA: Alpha-channel used for blending operations, such as transparency.
- **Depth Buffer**: Stores distance to object pixel.
  - Used for hidden surface removal - the closest pixel survives.
  - Also called the Z-buffer (z-coordinate stores distance).
- **Accumulation Buffer**: Used for composing and blending images.
  - Useful for achieving affects such as motion blur.
- **Stencil Buffer**:
  - Useful for masking operations.
Buffer Creation/Specification in GLUT

```c
void glutInitDisplayMode ( unsigned int mode ): where mode is the bitwise OR of GLUT display mode bit masks:
    GLUT_RGBA: Select an RGBA (direct) color. (Default)
    GLUT_RGB: (Same as GLUT_RGBA).
    GLUT_INDEX: Select indexed color.
    GLUT_SINGLE: Use single buffering. (Default)
    GLUT_DOUBLE: Use double buffering.
    GLUT_ACCUM: Allocate space for accumulation buffer.
    GLUT_ALPHA: Allocate space for a color blending.
    GLUT_DEPTH: Allocate space for depth buffer.
    GLUT_STENCIL: Allocate space for stencil buffer.
```

Example:
```c
    glutInitDisplayMode ( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
```

Reading and Writing Buffers

```c
    glReadPixels ( x, y, width, height, format, type, *pixels )
    glRasterPos2i ( x, y )
    glDrawPixels ( width, height, format, type, *pixels )
    glCopyPixels ( x, y, width, height, format, type, buffer )
```

where:

format:
```c
    GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA,
    GL_COLOR_INDEX, GL_DEPTH_COMPONENT, ...
```

type:
```c
    GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_FLOAT, ...
    GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_SHORT_5_6_5, ...
```

buffer:
```c
    GL_COLOR, GL_DEPTH, GL_STENCIL
```
Summary

Topics Covered:
- Introduction to OpenGL
- GLUT and user interaction
- OpenGL transformations and 3D viewing
- Lighting and texturing
- Buffers