Overview

- Basic primitives and rendering in OpenGL
- Transformations and viewing
- GLUT and the interaction / display loop
- More primitives and rendering
- Development tips

Getting Started…

- OpenGL is a cross platform 3D graphics library that takes advantage of specialized graphics hardware.

Two scenes rendered with a shading language developed at Stanford.

- Read the Red Book! It’s a great resource and is very readable.

- OpenGL is a state machine: polygons are affected by the current color, transformation, drawing mode, etc.
Specifying Object Vertices (Ch.2 p.42)

- **Every object is specified by vertices**
  
  ```
  glVertex3f (2.0, 4.1, 6.0);  // specifies a vertex at the x, y, z coordinate (2.0, 4.1, 6.0).
  // The “3f” means 3 floating point coordinates.
  
  Other examples:
  
  glVertex2i (4, 5);  // 2 integers for x and y. z = 0.
  glVertex3fv (vector);  // float vector[3] = {5.0, 3.2, 5.0};
  ```

- **Current color affects any vertices**

  ```
  glColor3f (0.0, 0.5, 1.0);  // no Red, half-intensity Green, full-intensity Blue
  ```

- **Vertices are specified only between `glBegin(mode)` and `glEnd()`**, usually in a counter-clockwise order for polygons.

  ```
  glBegin (GL_TRIANGLES);
  glVertex2i (0, 0);
  glVertex2i (2, 0);
  glVertex2i (1, 1);
  glEnd();
  ```
### Primitive Types in `glBegin` (Ch.2, p.44)

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td><code>GL_POINTS</code></td>
</tr>
<tr>
<td>Lines</td>
<td><code>GL_LINES</code>, <code>GL_LINE_STRIP</code>, <code>GL_LINE_LOOP</code></td>
</tr>
<tr>
<td>Triangles</td>
<td><code>GL_TRIANGLES</code>, <code>GL_TRIANGLE_STRIP</code>, <code>GL_TRIANGLE_FAN</code></td>
</tr>
<tr>
<td>Quads</td>
<td><code>GL_QUADS</code>, <code>GL_QUAD_STRIP</code></td>
</tr>
<tr>
<td>Polygons</td>
<td><code>GL_POLYGON</code></td>
</tr>
</tbody>
</table>

```c
glBegin(GL_LINES);
    [lots of glVertex calls];
glEnd();

glBegin(GL_QUADS);
    [lots of glVertex calls];
glEnd();
```
Transformations and Viewing (Ch.3)

OpenGL has 3 different matrix modes:

- GL_MODELVIEW
- GL_PROJECTION
- GL_TEXTURE

For example, choose to act on the projection matrix with:

```c
glMatrixMode(GL_PROJECTION);
```

- The *Modelview* matrix is used for your object transformations.
- The *Projection* matrix sets up the perspective transformation. It is usually set once at the beginning of your program.
- The *Texture* matrix can be used to warp textures (not commonly used).
OpenGL: Modelview matrix

- Transforms the viewpoint and objects within the scene.
- Example:

```c
glMatrixMode(GL_MODELVIEW); // set the current matrix
glLoadIdentity(); // load the identity matrix
glTranslatef(10.5, 0, 0); // translate 10.5 units along x-axis
glRotatef(45, 0, 0, 1); // rotate 45 degrees CCW around z-axis
DrawCube(); // cube is defined centered around origin
```

- Where will this end up?
- Answer: on the x-axis, rotated 45 degrees CCW. First image on page 107, fig 3-4.

Remember that the operations are right multiplied, so the transformation just before `DrawCube()` takes effect first.

- You can use `gluLookAt(...) (page 119)` in addition to rotations and translations to affect the viewpoint.
OpenGL: Projection Matrix

- **Sets up a perspective projection.** (page 123)

- **A few available options:**
  - `glfrustrum (...)`; // sets up a user defined viewing frustum
  - `gluPerspective (fovy, aspect, near, far);`
    // calculates viewing frustum for you, given field-of-view in degrees, aspect ratio, and near and far clipping planes.
  - `glortho (...)`; // creates orthographic (parallel) projection. Useful for 2D rendering.

- **Example:**

  ```c
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(64, (float)windowWidth / (float)windowHeight, 4, 4096);
  ```
GLUT is a library that handles system events and windowing across multiple platforms, and also provides some nice utilities. We *strongly* suggest you use it. Find it from the proj3 web page.

### Starting up:

```c
int main (int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize (windowWidth, windowHeight);
    glutInitWindowPosition (0, 0);
    glutCreateWindow ("248 Video Game!");

    SetStates();  // Initialize any rendering states (your code).
    RegisterCallbacks();  // Set up event callbacks (your code, coming up).

    glutMainLoop();  // Transfer control to GLUT. Doesn’t return.
    return 0;
}
```
Setting Up Rendering States

- OpenGL is a \textit{state} machine: polygons are affected by the current color, transformation, drawing mode, etc.

- Enable and disable features such as lighting, texturing, and alpha blending.
  - \texttt{glEnable (GL\_LIGHTING);} // enable lighting (disabled by default)

- Forgetting to enable something is a common source of bugs! Make sure you enable any features that you need (list of defaults is in Appendix B).
GLUT Event Callbacks

- Register functions that are called when certain events occur.

Examples:

```c
    glutDisplayFunc( Display );       // called when its time to draw
    glutKeyboardFunc( Keyboard );     // receives key input
    glutReshapeFunc( Reshape );       // called when window reshapes
    glutMouseFunc( Mouse );           // called when button changes
    glutPassiveMotionFunc( PassiveFunc ); // mouse moves, no buttons
    glutMotionFunc( MouseDraggedFunc ); // mouse moves, some buttons
    glutIdleFunc( Idle );             // called whenever idle
```
OpenGL – Depth Buffer, Double Buffer

• **Buffers store color and depth**
  - Allows Hidden Surface Removal, so there is proper ordering of objects in 3D space. This will be discussed later in the course.

• **Double buffering:**
  - Draw on *back* buffer while *front* buffer is being displayed.
  - When finished drawing, swap the two, and begin work on the new back buffer.
  - `glutSwapBuffers();`  // called at the end of rendering

• **Clearing the buffers:**

  ```c
  // Clear to this color when screen is cleared.
  glColor3f(0.0, 0.0, 0.0);
  // Clear color and depth buffers.
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  ```
OpenGL: Normals and Lighting

- OpenGL can simulate lighting for you, given some information on the geometry. Specify vertex normals as you specify geometry.
- Normal vectors should be of unit length (normalized) in most cases.

```c
// each vertex has a different normal here
glColor3f (0.8, 1.0, 0.5);
.glbegin(GL_TRIANGLES);
  glNormal3fv (n0);
  glVertex3fv (v0);
  glVertex3fv (v1);
  glVertex3fv (v2);
.glend();

// all vertices have the same normal here
.glbegin(GL_TRIANGLES);
  glNormal3fv (n0);
  glVertex3fv (v0);
  glVertex3fv (v1);
  glVertex3fv (v2);
.glend();
```
OpenGL: Lighting  (Ch.5 p.173)

- `glEnable (GL_LIGHTING);`
- **OpenGL supports a minimum of 8 lights.**
  - `glEnable (GL_LIGHT0);`
  - ...
  - `glEnable (GL_LIGHT7);`
- Lights have a position, type, and color, among other things.
- **Position:**
  - `float light0Position[4] = {1.0, 0.0, 4.0, 1.0};`
  - `glLightfv (GL_LIGHT0, GL_POSITION, light0Position);`
- Types of lights are point light, directional light, and spotlight. The fourth component of position (1.0 above) determines the type. 0 is for directional lights, 1 is for point/spot lights.  (page 187)
- **Color has a few components:** Ambient, Diffuse, Specular. Read about them in the text.
OpenGL: Lighting (cont.)

- OpenGL supports 2 basic shading models: flat and smooth.
  - `glShadeModel(GL_FLAT);`
  - `glShadeModel(GL_SMOOTH);`

- Lighting calculations can be expensive, so investigate other options (ie lightmaps) if needed.
OpenGL: Material Properties (Ch.5)

- You can specify different material properties for different polygons, changing the effect of lights.
  - Use `glMaterial*(GLenum face, GLenum pname, TYPE param);`

- Some properties (pname), page 202:
  - GL_AMBIENT: Ambient color of material
  - GL_DIFFUSE: Diffuse color of material
  - GL_SPECULAR: Specular component (for highlights)
  - GL_SHININESS: Specular exponent (intensity of highlight)

- Color plate 17 in the book shows a few examples.
OpenGL: Texturing
OpenGL: Texturing
OpenGL: Texturing
OpenGL: Texturing

- **Loading your data**
  - this can come from an image: ppm, tiff
  - create at run time
  - final result is always an array

- **Setting texture state**
  - creating texture names, scaling the image/data, building Mipmaps, setting filters, etc.

- **Mapping the texture to the polygon**
  - specify s,t coordinates for polygon vertices
OpenGL: Texturing

• **Loading your data**
  - this can come from an image: ppm, tiff
    - libtiff, libppm, etc.
    - remember the ordering of color channels and bits per channel! ie: RGBA, or AGBR, 32 bits or 8 bits?
  - You can tell OpenGL how to read your data by setting certain texture state (see next slide)

• create at run time
  - procedural textures, 3D textures, adding specular highlights

• final result is always an array
OpenGL: Texturing

- Setting texture state
  - create texture names
    - `glGenTextures(int num, int* texNames)`
    - `glBindTexture(GL_TEXTURE_2D, texName);`
  - Tell OpenGL how to read your array
    - `glPixelStorei(GL_UNPACK_SWAP_BYTES, int num);`
    - `glPixelStorei(GL_UNPACK_ALIGNMENT, int num);`
  - Scale your array to be $2^n+2(b)$, $b = \{0,1\}$ if you have a border or not
    - `gluScaleImage(GL_RGBA, w0, h0, GL_UNSIGNED_BYTE, img, w1, h1, GL_UNSIGNED_BYTE, imgScaled)`
    - `gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, w0, h0, GL_RGBA, GL_UNSIGNED_BYTE, img);`
OpenGL: Texturing

• **Setting texture state (cont)**
  • Tell OpenGL what to do when the s,t values are not within \([0,1] \times [0,1]\) range.
    • `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);`
    • `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);`
    • **GL_CLAMP**: any values larger than 1.0 are clamped to 1.0
    • **GL_REPEAT**: wrap larger values to the beginning of the texture (see OpenGL book, pg 411)
  • Set the filters for minification/magnification
    • `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);`
    • other parameters: **GL_LINEAR**, other mipmap options
OpenGL: Texturing

- **Setting texture state (cont)**
  - Tell OpenGL about your data array (image, etc.)
    - `glTexImage2D(GL_TEXTURE_2D, int lod, int num_components, width, height, border, format_of_data_pixel, size_of_each_channel, img_array)`
    - If you used to `gluBuild2DMipmaps` scale your image and create a multi-resolution pyramid of textures, then you do NOT need to use `glTexImage2D`. The `gluBuild2DMipmaps` command will already tell OpenGL about your array.
OpenGL: Texturing

- **Mapping the texture to the polygon**
  - specify \((s,t)\) texture coordinates for \((x,y,z)\) polygon vertices
  - texture coordinates \((s,t)\) are from \(0,1:\)

\[
\begin{align*}
\text{glTexCoord2f}(s,t); & \\
(x_0, y_0, z_0) & \rightarrow \quad (x_1, y_1, z_1) \\
(x_2, y_2, z_2) & \rightarrow \quad (x_3, y_3, z_3)
\end{align*}
\]
OpenGL: Texturing

• Let’s look at code!
OpenGL: Texturing

- Advanced Texture techniques
  - Multitextures
  - automatic texture generation
    - Let OpenGL determine texture coordinates for you
  - Environment Mapping
  - Texture matrix stack
**OpenGL: Alpha Blending**

- When enabled, OpenGL uses the alpha channel to blend a new fragment’s color value with a color in the framebuffer.

\[
\text{New color} \quad (r_1,g_1,b_1,a_1) \quad \text{"source"} \quad + \quad \text{Color in framebuffer} \quad (r_0,g_0,b_0,a_0) \quad \text{"destination"} \quad = \quad (r',g',b',a')
\]

\[
r' = a_1 \times r_1 + (1-a_1) \times r_0
\]

```c
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ZERO);
...draw green square ...
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
...draw brown square with alpha = 0.5...
```
Alpha blending with multiple textures

- one way to do multi-pass rendering
- number of “texture passes” over a polygon is independent of the maximum number of multi-texture units on the graphics card
  - GeForce 2 has only 2 texture units!
- slower because geometry is sent $n$ times to the card for $n$ texture passes
- demo and code if you want to see it
Development

- **On Windows:**
  - Download the GLUT libraries (linked off the proj3 webpage).
  - You want to link your project with: opengl32.lib, glut32.lib, and glu32.lib. This is under Project->Settings->Link in MS Visual Studio.

- **On Linux:**
  - GLUT is already installed on the graphics lab PCs.
  - In your Makefile, compile with flags: -L/usr/lib -lGL -lGLU -lglut

- Call glutReportErrors() once each display loop for debugging.
  - This will report any errors that may have occurred during rendering, such as an illegal operation in a glBegin/glEnd pair.
Questions?