Introduction to OpenGL & GLUT

Christian Miller
CS 354 - Fall 2011
GL APIs

- GL - OpenGL core
- GLU - GL Utilities
- GLUT - GL Utility Toolkit
- GLEW - GL Extension Wrangler
The basic idea
The basic idea
The basic idea

- OpenGL maintains its own state
  - You call a bunch of functions to set it up
  - To draw, submit a list of vertices
    - It will connect them together for you
    - The state determines how they are rendered
// example equilateral triangle
glBegin(GL_LINE_LOOP);
glColor3f(1, 0, 0);
glVertex3f(-.5, -.5, 0);
glColor3f(0, 1, 0);
glVertex3f(.5, -.5, 0);
glColor3f(0, 0, 1);
glVertex3f(0, 0.366, 0);
glEnd();
Basics

- All OpenGL functions are prefixed with gl
- All #defines are prefixed with GL_
- It’s a C API, no classes or even structures provided
Immediate mode drawing

- `glBegin(primitive)` and `glEnd()`
- Will take all vertices submitted between begin/end and connect them together with primitives
- Called immediate mode since the geometry is submitted incrementally as functions are called
PRIMITIVES

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP

GL_POLYGON
GL_QUADS
GL_TRIANGLES

GL_TRIANGLE_STRIP
GL_QUAD_STRIP
GL_TRIANGLE_FAN
Submitting vertices

- `glVertexXT(...)`
  - $X \in \{1, 2, 3, 4\}$ (number of coordinates)
  - $T \in \{i, f, d\}$ (data type of arguments)
- e.g. `glVertex3f( x, y, z )`
- e.g. `glVertex2i( 200, 400 )`
Submitting vertices

• Also pointer mode: glVertex3f(ptr)

• Example:
  
  • float loc[3] = { 1.0f, 2.0f, 3.0f };  
  
  • glVertex3fv(loc);  

Vertex attributes

• Call before submitting a vertex to change attribute
• Normally interpolated across a surface
• glColorXT(...)  
• glNormal3T(...) (normals are only in 3 dimensions)
• glTexCoordXT(...)
INTERPOLATION

// example equilateral triangle
glBegin(GL_TRIANGLES);
    glColor3f(1, 0, 0);
    glVertex3f(-.5, -.5, 0);
    glColor3f(0, 1, 0);
    glVertex3f(.5, -.5, 0);
    glColor3f(0, 0, 1);
    glVertex3f(0, 0.366, 0);
    glEnd();
On units

- OpenGL is unitless
- Use whatever is convenient for your application
- Adjust camera projection to generate correct scale
- Must be consistent!
Default camera

- Camera starts out at \(< 0, 0, 0 >\)
- Looks down the negative Z axis
- OpenGL coordinate system is right-handed:
  - X increases to the right
  - Y increases going up
  - Z increases coming out of the screen
Default camera
Projection

- The default OpenGL camera uses orthographic projection (no perspective)
- Everything in the projection volume is projected onto the framebuffer
  - Everything outside is clipped and not shown
- Default projection volume is a cube extending from $< -1, -1, -1 >$ to $< 1, 1, 1 >$
• Everything in volume projected onto view rectangle
Orthographic setup

// setup basic ortho projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, near, far);
Perspective projection

- Everything in frustum projected onto view rectangle, scaled by distance (farther = smaller)
PERSPECTIVE PROJECTION

- FOVy = vertical field of view (angular, 45° is good)
- Aspect = w / h
Perspective setup

// setup basic perspective projection
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fovy, aspect, near, far);
Near / far planes
Near / far planes

• Why don’t we set near = 0 and far = ∞?
Near / far planes

- Why don’t we set near = 0 and far = ∞?
- Near = 0 means that the near plane is on the point of projection
Near / far planes

- Why don’t we set near = 0 and far = \( \infty \)?
- Near = 0 means that the near plane is on the point of projection
  - Everything gets projected onto a point!
Near / far planes

• Why don’t we set near = 0 and far = ∞?

• Near = 0 means that the near plane is on the point of projection
  • Everything gets projected onto a point!

• Far = ∞ ruins numerical precision in the Z coordinate, makes depth information unreliable
Moving things around

- Once you’ve set up the camera’s projection volume, you probably want to move the camera around.
- Conceptually, this is the same as moving the objects around in the opposite direction.
MOVING THINGS AROUND
Model / View Transformations

- These are how you move objects and the camera
- Specified as if you’re moving the objects relative to their previous position / orientation
- Can do multiple transforms in a row

```
// initialize modelview transform
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```
Translation

```
glTranslateT(x, y, z);  // T = \{ f, d \}
```
glRotateT(angle, x, y, z);
// $T = \{ f, d \}$, angle is in degrees CCW
// $< x, y, z >$ is axis of rotation
// Happens around origin!
Rotating around an arbitrary point

\[ T(-p) \]
\[ R_x(\pi/4) \]
\[ T(p) \]
**Scaling**

```c
glScaleT(x, y, z);  // T = { f, d }
// < x, y, z > is per-axis scale
// Also around origin!
```
Order matters
Some notes

- Transforms are all represented as $4 \times 4$ matrices.
- There are a couple ways to stack transforms together, we’ll go over the intuition behind this in detail soon.
- For now, every time transforms don’t seem to be doing what you expect them to, try reversing the order.
User interface

• GLUT is an event-driven windowing system
  • Basically all UIs are built like this
• Programming this way is probably not what you’re used to...
Event-driven programs

• Most programs you’ve written until now have probably been procedural

• Event-driven programs enter a loop, then react with callbacks
  • A callback is just a function that’s called whenever a particular type of event happens
GLUT main function

```c
int main(int argc, char *argv[]) {
    // Initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA); // configure framebuffer
    glutInitWindowSize(window_width, window_height); // init window size
    glutInitWindowPosition(100, 100); // and position (upper-left corner)
    glutCreateWindow("CS 354"); // and name

    glutDisplayFunc(display);
    glutReshapeFunc(resize);
    glutMouseFunc(mouseButton); // these 5 lines register event callbacks
    glutMotionFunc(mouseMotion); // each one is a function pointer
    glutKeyboardFunc(keyboard);

    initGL(); // do OpenGL state initialization

    // enter main loop
    glutMainLoop();

    return 0;
}
```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear last frame

    // load the identity matrix (in modelview mode)
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // PERFORM ALL DRAWING HERE

    // example equilateral triangle
    glBegin(GL_LINE_LOOP);
    glColor3f(1, 0, 0);
    glVertex3f(-.5, -.5, 0);
    glColor3f(0, 1, 0);
    glVertex3f(.5, -.5, 0);
    glColor3f(0, 0, 1);
    glVertex3f(0, .366, 0);
    glEnd();

    glFlush(); // finish the drawing commands
    glutSwapBuffers(); // and update the screen
    glutPostRedisplay(); // ask GLUT to call display again
}
Reshape callback

```c
void resize(int width, int height)
{
    // This reshape function is called whenever the user
    // resizes the display window.
    window_width = width;
    window_height = height;
    window_aspect = (float)width / (float)height;

    // resize the viewport
    glViewport(0, 0, window_width, window_height);

    // setup basic orthographic projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-window_aspect, window_aspect, -1.0, 1.0, 1.0, -1.0);

    // switch over to modelview
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glutPostRedisplay(); // let GLUT know to redraw the screen
}
```
Some other callbacks

void mouseButton(int button, int state, int x, int y);
void mouseMotion(int x, int y);
void keyboard(unsigned char key, int x, int y);
Other GLUT Functionality

- Pop-up menus
- Fullscreen
- Several other types of events and callbacks
void initGL()
{
    // Perform any necessary GL initialization in this function
    // enable depth testing, and set clear color to white
    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    // resize the window
    resize(window_width, window_height);
}