Introduction to OpenGL

Jian Huang
CS 456 Computer Graphics
OpenGL and GLUT Overview

• What is OpenGL & what can it do for me?
• OpenGL in windowing systems
• Why GLUT
• GLUT program template
What Is OpenGL?

• Graphics rendering API
  – high-quality color images composed of geometric and image primitives
  – window system independent
  – operating system independent
OpenGL Architecture
OpenGL as a Renderer

• Geometric primitives
  – points, lines and polygons
  – Image Primitives
  – images and bitmaps
• separate pipeline for images and geometry
  – linked through texture mapping
• Rendering depends on state
  – colors, materials, light sources, etc.
Relate APIs

• AGL, GLX, WGL
  – glue between OpenGL and windowing systems
• GLU (OpenGL Utility Library)
  – part of OpenGL
  – NURBS, tessellators, quadric shapes, etc
• GLUT (OpenGL Utility Toolkit)
  – portable windowing API
  – not officially part of OpenGL
OpenGL and Related APIs

application program

OpenGL Motif widget or similar

GLX, AGL or WGL

X, Win32, Mac O/S

GLUT

GLU

GL

software and/or hardware
Preliminaries

• Header Files
  – #include <GL gl.h>
  – #include <GL glu.h>
  – #include <GL glut.h>

• Libraries

• Enumerated types

• OpenGL defines numerous types for compatibility
  – GLfloat, GLint, GLenum, etc.
GLUT Basics

- Application Structure: event driven
- Configure and open window
- Initialize OpenGL state
- Register input callback functions
  - render
  - resize
  - input: keyboard, mouse, etc.
- Enter event processing loop
Sample Program

```c
void main( int argc, char** argv )
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutCreateWindow( argv[0] );
    init(); // initiate OpenGL states, program variables
    glutDisplayFunc( display );  // register callback routines
    glutReshapeFunc( resize );
    glutKeyboardFunc( key );
    glutIdleFunc( idle );
    glutMainLoop(); // enter the event-driven loop
}
```
OpenGL Initialization

- Set up whatever state you’re going to use

```c
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glClearDepth( 1.0 );
    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
    glEnable( GL_DEPTH_TEST );
}
```
GLUT Callback Functions

• Routine to call when something happens
  – window resize or redraw
  – user input
  – animation

• “Register” callbacks with GLU
  – glutDisplayFunc( display );
  – glutIdleFunc( idle );
  – glutKeyboardFunc( keyboard );
Rendering Callback

- Do all of our drawing here

```c
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glBegin( GL_TRIANGLE );
        glVertex3fv( v[0] );
        glVertex3fv( v[1] );
        glVertex3fv( v[2] );
    glEnd();
    glutSwapBuffers();
}
```
Idle Callbacks

- Use for animation and continuous update

```c
void idle( void )
{
    t += dt;
    glutPostRedisplay();
}
```

```c
glutIdleFunc( idle );
```
User Input Callbacks

- Process user input

```c
glutKeyboardFunc( keyboard );
void keyboard( char key, int x, int y )
{
    switch( key ) {
        case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;
        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
    }
}
```
Elementary Rendering

• Geometric Primitives
• Managing OpenGL State
• OpenGL Buffers
OpenGL Geometric Primitives

- All geometric primitives are specified by vertices
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
OpenGL Command Formats

```
oglVertex3fv( v )
```

**Number of components**
- 2 - (x,y)
- 3 - (x,y,z)
- 4 - (x,y,z,w)

**Data Type**
- b - byte
- ub - unsigned byte
- s - short
- us - unsigned short
- i - int
- ui - unsigned int
- f - float
- d - double

**Vector**
- Omit "v" for scalar form
- glVertex2f( x, y )
Specifying Geometric Primitives

- Primitives are specified using
  
  ```
  glBegin( primType );
  glEnd();
  ```

- `primType` determines how vertices are combined
  
  ```
  GLfloat red, greed, blue;
  GLfloat coords[3];
  glBegin( primType );
  for (i =0;i <nVerts; ++i ) {
    glColor3f( red, green, blue );
    glVertex3fv( coords );
  }
  glEnd();
  ```
OpenGL Color Model

• Both RGBA (true color) and Color Index
Controlling Rendering

- Appearance
- From Wireframe to Texture mapped
OpenGL’s State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping
Manipulating OpenGL State

• Appearance is controlled by current state
  for each (primitive to render) {
    update OpenGL state
    render primitive
  }

• manipulating vertex attributes is most common way to manipulate state
  – glColor*() / glIndex*()
  – glNormal*()
  – glTexCoord*()
Controlling current state

- **Setting State**
  
  ```c
  glPointSize( size );
  glLineStipple( repeat, pattern );
  glShadeModel( GL_SMOOTH );
  ```

- **Enabling Features**
  
  ```c
  glEnable( GL_LIGHTING );
  glDisable( GL_TEXTURE_2D );
  ```
Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen
Coordinate Systems and Transformations

• Steps in Forming an Image
  – specify geometry (world coordinates)
  – specify camera (camera coordinates)
  – project (window coordinates)
  – map to viewport (screen coordinates)

• Each step uses transformations
• Every transformation is equivalent to a change in coordinate systems
3D Transformations

- A vertex is transformed by 4 x 4 matrices
- All affine operations are matrix multiplication
- Matrices are stored column-major in OGL
- Matrices are always post-multiplied

Except in perspective projection, the fourth row = (0,0,0,1), w left unchanged.

OpenGL uses stacks of matrices, the programmer must remember that the last matrix specified is the first applied.
Specifying Transformations

• Programmer has two styles of specifying transformations
  – specify matrices \texttt{glLoadMatrix, glMultMatrix}
  – specify operation \texttt{glRotate, glOrtho}
• Programmer does not have to remember the exact matrices
• check appendix of Red Book
Programming Transformations

- Prior to rendering, view, locate, and orient:
  - eye/camera position
  - 3D geometry
- Manage the matrices
  - including matrix stack
- Combine (composite) transformations
- Transformation matrices are part of the state, they must be defined prior to any vertices to which they are to apply.
- OpenGL provides matrix stacks for each type of supported matrix (ModelView, projection, texture) to store matrices.
Transformation Pipeline

- object
- eye
- clip
- normalized device
- window

1. Modelview Matrix
2. Projection Matrix
3. Perspective Division
4. Viewport Transform

**Vertex:**
- Modelview
- Projection

**Diagram:**
- Input: vertex
- Process: Modelview, Projection, Perspective Division, Viewport Transform
- Output: window
Matrix Operations

- **Specify Current Matrix Stack**
  - `glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`

- **Other Matrix or Stack Operation**
  - `glLoadIdentity()` `glPushMatrix()` `glPopMatrix()`

- **Viewport**
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted
  - `glViewport( x, y, width, height )`
Projection Transformation

• Perspective projection
  – `gluPerspective( fovy, aspect, zNear, zFar )`
  – `glFrustum( left, right, bottom, top, zNear, zFar )` (very rarely used)

• Orthographic parallel projection
  – `glOrtho( left, right, bottom, top, zNear, zFar )`
  – `gluOrtho2D( left, right, bottom, top )`
  – calls `glOrtho` with z values near zero

• Warning: for `gluPerspective()` or `glFrustum()`, don’t use zero for `zNear`!
Applying Projection

• Transformations
• Typical use (orthographic projection)

```c
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glOrtho( left, right, bottom, top, zNear, zFar );
```
Viewing Transformations

• Position the camera/eye in the scene
• To “fly through” a scene
• change viewing transformation and redraw scene

    gluLookAt( eye x ,eye y ,eye z ,
              aim x ,aim y ,aim z ,
              up x ,up y ,up z )

• up vector determines unique orientation
• careful of degenerate positions
Modeling Transformations

• Move object
  – glTranslate(fd)( x, y, z )

• Rotate object around arbitrary axis
  – glRotate(fd)( angle, x, y, z )
  – angle is in degrees

• Dilate (stretch or shrink) object
  – glScale(fd)( x, y, z )
Projection is left handed

- Projection transformation (gluPerspective, glOrtho) are left handed
  - think of $z_{Near}$ and $z_{Far}$ as distance from viewpoint
- Everything else is right handed, including the vertexes to be rendered
Common Transformation Usage

• Example of `resize()` routine
  – restate projection & viewing transformations

• Usually called when window resized

• Registered a callback for `glutReshapeFunc()`
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w / h, 1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );
}