INTRODUCTION TO OPENGL

Pramook Khungurn
CS4621, Fall 2011

What is OpenGL?

- Open Graphics Library
- Low level API for 2D/3D rendering with GPU
  - For interactive applications
- Developed by SGI in 1992
  - 2009: OpenGL 3.2
  - 2010: OpenGL 4.0
  - 2011: OpenGL 4.2
- Competitor: Direct3D

Where does it work?

What OpenGL Does

- Control GPU to draw simple polygons.
- Utilize graphics pipeline.
  - Algorithm GPU uses to create 3D images
  - Input: Polygons & other information
  - Output: Image on framebuffer
  - How: Rasterization
  - More details in class next week.

What OpenGL Doesn’t Do

- Manage UI
- Manage windows
- Decide where on screen to draw.
  - It’s your job to tell OpenGL that.
- Draw curves or curved surfaces
  - Although can approximate them by fine polygons

Jargons

- Bitplane
  - Memory that holds 1 bit of info for each pixel
- Buffer
  - Group of bitplanes holding some info for each pixel
- Framebuffer
  - A buffer that hold the image that is displayed on the monitor.
**Java OpenGL (JOGL)**

- OpenGL originally written for C.
- JOGL = OpenGL binding for Java

- Give Java interface to C OpenGL commands.
- Manage framebuffer.

**Demo 1**

**GLCanvas**

- UI component
- Can display images created by OpenGL.
- OpenGL "context"
  - Store OpenGL states.
  - Provides a default framebuffer to draw.

- Todo:
  - Create and stick it to a window.

**Events**

- GUI is often done by **event-driven programming**.
- Certain UI components acts as sensors.
- Something happen to them (i.e., the user click them)
- UI components broadcast a message, called events.
- Objects interested in such events can sign up as listeners.
- Listeners run appropriate code when receiving a event.
  - Updating the display, update database, send info over the net, etc.
- GLCanvas broadcast 4 events related to OpenGL.

**GLEventListener**

- Handle events generated by GLCanvas.
- Implement to use OpenGL commands.

- Todo:
  - Create.
  - Implement 4 methods.
  - Call GLCanvas.addGLListener.
GLEventListener Methods

- init
  - Called once when OpenGL context is created.
- display
  - Called every time GLCanvas repaints itself.
- resize
  - Called every time GLCanvas resizes.
- dispose
  - Called before OpenGL context is destroyed.

OpenGL Commands

- gl.glClearColor(0, 0, 0, 0)
  - Change the color used to clear screen to black.
- gl.glClear(GL2.GL_COLOR_BUFFER_BIT)
  - Clear the buffers that store color with the color specified by glClearColor.
- glColor3d(1.0, 1.0, 1.0)
  - Change the current color to white.
  - All vertices drawn afterwards are white until told otherwise.

GL Objects

- Store OpenGL states, commands, and constants
- Many classes based on OpenGL version.
  - GL2, GL3, GL4, GL2ES1, GL2ES2, etc.
- We use GL2.
  - Backward compatibility.
- Get instance from GLAutoDrawable.
  - Passed into every GLEventListener method.
  - final GL2 gl = new drawable.getGL().getGL2();

Demo 1’s display method

@override
def display(GLAutoDrawable drawable):
    
    final GL2 gl = drawable.getGL().getGL2();
    gl.glClearColor(0, 0, 0, 0);
    gl.glClear(GL2.GL_COLOR_BUFFER_BIT);
    glColor3d(1.0, 1.0, 1.0);
    gl.glBegin(GL2.GL_POLYGON);
    
    gl.glVertex3d(-0.5, -0.5, 0.0);
    gl.glVertex3d(0.5, -0.5, 0.0);
    gl.glVertex3d(0.5, 0.5, 0.0);
    gl.glVertex3d(-0.5, 0.5, 0.0);
    
    glEnd();
One Command at a Time

- `glBegin(GL_POLYGON)`
  - Tell OpenGL we will draw a polygon.
- `glVertex3f(x, y, z)`
  - Specify the polygon’s vertices.
- `glEnd()`
  - Tell OpenGL that we’re done specifying the geometry.

Command Naming

- In C,
  - commands = functions
  - No two functions can have the same name.
  - Some commands take different arguments but do the same thing.

All commands of the form

```c
gl <name> {1234} {b s i f d ub us ui} {v}
```

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Data Type</th>
<th>Typical Corresponding C-Language Type</th>
<th>OpenGL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8-bit integer</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>s</td>
<td>16-bit integer</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>long</td>
<td>GLint, GLsize</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point</td>
<td>float</td>
<td>GLfloat, GLclampf</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point</td>
<td>double</td>
<td>GLdouble, GLclampd</td>
</tr>
<tr>
<td>ub</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
<td>GLubyte, GLboolean</td>
</tr>
<tr>
<td>us</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>u</td>
<td>32-bit unsigned integer</td>
<td>unsigned long</td>
<td>GLuint, GLenum, GLbitfield</td>
</tr>
</tbody>
</table>

OpenGL is a State Machine

- OpenGL remembers values the user specified.
- Use these values until user changes it.
- Examples of stored values:
  - Color uses to clear screen
  - Color of vertices
  - Transformation matrices

- After calling `glColor3f(1,1,1)`
  - All vertices specified by `glVertex` afterwards are white.
  - Until the user calls `glColor` again with a different color.

Demo 2

```c
glBegin(GL2.GL_TRIANGLES);
{
  glClearColor(1.0f, 0.5f, 0.5f);
  glVertex3f(0.25f, 0.0f, 0.0f);
  glVertex3f(-0.5f, -0.5f, 0.0f);
  glVertex3f(0.5f, -0.5f, 0.0f);
  glColor3f(0.5f, 1.0f, 0.5f);
  glVertex3f(-0.25f, 0.0f, 0.0f);
  glVertex3f(0.25f, 0.0f, 0.0f);
  glVertex3f(0.0f, -0.5f, 0.0f);
  glVertex3f(-0.5f, 0.0f, 0.0f);
  glVertex3f(0.5f, 0.0f, 0.0f);
}
glEnd();
```
Geometric Primitives in OpenGL

- 3 types of geometry OpenGL can draw:
  - Points
  - Line segments
  - Polygons
- No curves or curved surfaces.
- Can approximate curves by short line segments.

Vertices

- OpenGL specifies shapes by specifying its vertices.
  - Point: 1 vertex
  - Lines: 2 vertices
  - N-gon: n vertices
- Vertices can have associated information.
  - Color
  - Specified by glColor
  - Normal
  - Specified by glNormal
  - Texture coordinate
  - Specified by glTexCoord
- More on this later…

glVertex

- glVertex specifies one such vertex.
- Must be called between glBegin(…) and glEnd()
- Up to 4 arguments
  - Position (x, y, z, w) in homogeneous coordinate.
  - For 2 arguments
    - z = 0, w = 1
  - For 3 arguments
    - w = 1

Specifying Shapes

- Start with glBegin(<< shape type >>).
- Use glVertex to specify each vertex.
  - Do it in counterclockwise order.
  - The shape might not display otherwise.
- Stop with glEnd()

Avaiable Shapes

Demo 3

```c
glBegin(GL2.GL Line LOOP);
for(int i = 0; i < 256; i++)
{
  double theta = 2 * Math.PI / 256;
  double y = 0.75 * Math.sin(theta);
  double x = 0.75 * Math.cos(theta);
  glVertex2d(x, y);
}
glEnd();
```
Polygons OpenGL Can Draw

- All the points on the polygon must lie on the same plane.
- Otherwise, correctness of drawing is not guaranteed.
- **Notice:** All points on a triangle are coplanar.
- Not true for other polygons.

Polygons OpenGL Can Draw

- No edges must intersect.
- Must be convex.
- Must not have holes.

Drawing Arbitrary Polygons

- Decompose them into convex polygons.
  - Preferably triangles.

Default Viewing

- Eye at (0,0,1).
- Look in negative z direction.
- Orthographic projection
  - No foreshortening.
  - No sense of depth.
  - 2D coordinate system if don’t care about z-axis.

Default Coordinate System
GLU

- OpenGL Utility Library
- C library usually ships with OpenGL
- Many useful functions:
  - Drawing of some curved surfaces.
  - Interpretation of OpenGL errors.
  - Simple camera setup.

- In JOGL,
  - Encapsulated by GLU class.
  - Create one when needed.
  - GLU glu = new GLU();


gluOrtho2D

- Usage: Issue these three commands together.
  - glMatrixMode(GL2.GL_PROJECTION);
  - glLoadIdentity();
  - glu.gluOrtho2D(left, right, bottom, top);

- Will become clear afterwards why by next week.

- Only primitives drawn afterwards are affected.
  - Coordinate system is “defined” by projection matrix.
  - OpenGL remembers the matrix until it changes.

When Resizing Window

- But we probably want this…

 gluOrtho2D

- glu.gluOrtho2D(double left, double right, double bottom, double top)

- Change the projection matrix to orthographic projection.
  - Details in CS 4620 lecture next week.

- For now, it sets the 2D coordinate system so that:
  - Bottom-left corner is (left, bottom)
  - Bottom-right corner (right, bottom)
  - Top-left corner is (left, top)
  - Top-right corner is (right, top)
What We Want

• When resizing window, aspect ratio of drawn pictures remain the same.
• Keep the same drawing code.
• No change to display method.
• Can do so by changing the coordinate system.
• When to do this?
  • Each time the window size change.
  • Implement reshape method.

reshape

• public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height);
  • x, y  
    • Coordinate of top-left corner of GLCanvas in pixels.
  • width, height  
    • Size of GLCanvas in pixels.

Demo 4

@Override
public void reshape(GLAutoDrawable drawable, int x, int y, int w, int h) {
    final GL2 gl = drawable.getGL().getGL2();
    final GLU glu = new GLU();
    if (w == 0) w = 1;
    if (h == 0) h = 1;
    double aspect = w * 1.0 / h;
    gl.glMatrixMode(GL2.GL_PROJECTION);
    gl.glLoadIdentity();
    if (w > h) glu.gluOrtho2D(-aspect, aspect, -1, 1);
    else glu.gluOrtho2D(-1, 1, -1/aspect, 1/aspect);
}

When Width > Height

When Width < Height
OpenGL Vertex Transformations

- Coordinates specified by glVertex are transformed.
- End result: window coordinate (in pixels)

Modeling Transform

- Object space
  - Coordinate system used by modeler.
  - Local to the 3D geometric model being created.
  - What is specified by glVertex is in this space.
- World space
  - Coordinate system of the scene.
  - Used to position models relative to one another.
- Modeling transform
  - Object space \(\rightarrow\) world space.
  - Can also
  - Change positions of parts relative to one another.

View Transform

- Used to set the camera.
- Eye space is a coordinate system where:
  - Eye is at \((0,0,0)\).
  - Look in negative z direction.
  - Y-axis is "up."
- Much like camera setup of PA1.
- View transform
  - world space \(\rightarrow\) eye space
**Modelview Matrix**

- OpenGL combines modeling transform and view transform into one transform.
- Modelview transform
  - object space $\rightarrow$ eye space

\[
\begin{bmatrix}
\text{Modelview}
\end{bmatrix} = \begin{bmatrix}
\text{View} \\
\text{Model}
\end{bmatrix}
\]

**Manipulating Modelview Matrix**

- OpenGL keeps a number of 4x4 matrices as states.
  - Modelview
  - Projection
  - Texture

- `glMatrixMode`  
  - Specify which matrix to manipulate  
  - For modelview matrix:  
    \[
    \text{glMatrixMode}(GL2.GL_MODELVIEW);
    \]
  - Always manipulate outside `glBegin(…)` `glEnd()` block.  
  - Invalid operation otherwise.

**Commands for Manipulating Matrices**

- Say OpenGL keeps the current matrix as M.

- `glLoadIdentity()`  
  - Set M = I.

- `glLoadMatrixd(double[] a, int s)`  
  - Variant: `glLoadMatrixf`  
  - a = array of at least 16 doubles.  
  - a has elements of the matrix in column major order.

\[
M = \begin{bmatrix}
\end{bmatrix}
\]

- `glTranslated(double x, double y, double z)`  
  - Set M = MT where T is a 3D translation matrix.

- `glScaled(double a, double b, double c)`  
  - Set M = MS where S is a 3D scaling matrix.

\[
T = \begin{bmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
S = \begin{bmatrix}
a & 0 & 0 & 0 \\
0 & b & 0 & 0 \\
0 & 0 & c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
M = \begin{bmatrix}
a_{0} & a_{1} & a_{2} & 0 \\
a_{3} & a_{4} & a_{5} & 0 \\
a_{6} & a_{7} & a_{8} & 0 \\
a_{9} & a_{10} & a_{11} & 1
\end{bmatrix}
\]

\[
M = \begin{bmatrix}
\end{bmatrix}
\]
Commands for Manipulating Matrices

- `glRotated(double angle, double x, double y, double z)`
  - Set `M = MR` where `R` is a 3D translation matrix.
  - `angle` = angle of rotation in degrees.
  - `(x,y,z)` is the axis of rotation.
  - For 2D rotation, rotate around the z-axis. For example:
    ```
    gl.glRotated(30, 0, 0, 1);
    ```

Example 1
```
gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();

gl.glTranslatef(1,2,3);

M = | 1 0 0 |
    | 0 1 0 |
    | 0 0 1 |

gl.glBegin(GL_POINTS);
    { gl.glVertex3f(1,1,1); }
gl.glEnd();
```

Example 1
```
M = | 1 0 0 |
    | 0 1 0 |
    | 0 0 1 |

v_world = M * v_object = | 1 0 0 | | 1 1 1 | = | 2 |
                           | 0 1 0 | | 0 1 1 |   | 4 |
                           | 0 0 1 | | 0 0 1 |   | 4 |
```

Example 2
```
gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();

M = | 1 0 0 |
    | 0 1 0 |
    | 0 0 1 |

gl.glTranslatef(1,2,3);

M = | 1 0 0 |
    | 0 1 0 |
    | 0 0 1 |

v_world = M * v_object = | 1 0 0 | | 2 |
                           | 0 1 0 | | 4 |
                           | 0 0 1 | | 4 |
```

Example 1
```
M = | 1 0 0 |
    | 0 1 0 |
    | 0 0 1 |

gl.glMatrixMode(GL2.GL_MODELVIEW);
gl.glLoadIdentity();

gl.glTranslatef(1,2,3);

gl.glBegin(GL_POINTS);
    { gl.glVertex3f(1,1,1); }
}
gl.glEnd();
```
Example 2

```cpp
glMatrixMode(GL2.GL_MODELVIEW);
glLoadIdentity();

M = 
  1 0 0 0
  0 1 0 0
  0 0 1 0
  0 0 0 1

glTranslatef(1,2,3);
glScalef(10,10,10);

glBegin(GL_POINTS);
{
    glVertex3f(1,1,1);
}
glEnd();
```

Interpreting Matrix Commands

- They transform the coordinate system.

```cpp
glMatrixMode(GL2.GL_MODELVIEW);
glLoadIdentity();

Origin at (0,0,0)

--

glTranslatef(1,2,3);

Origin at (1,2,3)
```

Example 2

```cpp
glMatrixMode(GL2.GL_MODELVIEW);
glLoadIdentity();

M = 
  1 0 0 0
  0 1 0 0
  0 0 1 0
  0 0 0 1

glTranslatef(1,2,3);
glScalef(10,10,10);

glBegin(GL_POINTS);
{
    glVertex3f(1,1,1);
}
glEnd();
```

Demo 5, 6, 7, 8
Matrix Stack
- OpenGL maintains a stack of 4x4 matrices.
- Useful when doing hierarchical transformations.
  - Rendering scene graphs.
  - Rendering fractals.
- Stack manipulation commands
  - `glPushMatrix()`
  - `glPopMatrix()`

Scene
- Suppose we wrote two methods:
  - `circle()`: draw a unit sphere centered at (0,0)
  - `square()`: draw a square of size length 2 centered at (0,0)
- Want to use the above methods to draw the scene.

Scene Hierarchy

Scene Graph
Scene Graph → Code

```c
void scene(GL2 gl)
{
    gl.glPushMatrix();
    gl.glTranslated(-1.7, -1.0, 0.0);
    tableAndTray(gl);
    gl.glPopMatrix();
    floor(gl);
    gl.glPushMatrix();
    gl.glTranslated(0.9, -1.0, 0.0);
    chair(gl);
    gl.glPopMatrix();
}
```

Scene Graph → Code

```c
void chair(GL2 gl)
{
    gl.glPushMatrix();
    gl.glTranslated(0.1, 0.35, 0.0);
    gl.glScaled(0.1, 0.35, 1.0);
    square(gl, 0.0, 1.0, 1.0);
    gl.glPopMatrix();
    gl.glPushMatrix();
    gl.glTranslated(0.3, 0.8, 0.0);
    gl.glScaled(0.3, 0.1, 1.0);
    square(gl, 0.0, 1.0, 1.0);
    gl.glPopMatrix();
    gl.glPushMatrix();
    gl.glTranslated(0.7, 1.1, 0.0);
    gl.glScaled(0.1, 1.1, 1.0);
    square(gl, 0.0, 1.0, 1.0);
    gl.glPopMatrix();
}
```

Demo 9

Drawing Fractals

- Sierpinski Triangle

![Sierpinski Triangle](image)

Drawing Fractals

- We can define “levels” of Sierpinski triangles.
- Recursive definition
  - Level k+1 is made from arranging 3 copies of Level k.
  - The one on last slide is Level 8.

`void sierpinski(GL2 gl, int k)
{
    if (k == 0)
        triangle(gl);
    else
    {
        gl.glPushMatrix();
        gl.glTranslated(0.1, 0.15 / Math.sqrt(3.0), 0.0);
        sierpinski(gl, k-1);
        gl.glPopMatrix();
        gl.glPushMatrix();
        gl.glTranslated(0.25, -0.25 / Math.sqrt(3.0), 0.0);
        sierpinski(gl, k-1);
        gl.glPopMatrix();
        gl.glPushMatrix();
        gl.glTranslated(0.25, -0.25 / Math.sqrt(3.0), 0.0);
        sierpinski(gl, k-1);
        gl.glPopMatrix();
    }
}`
Demo 10