OpenGL Drawing Tutorial

ECE 6397 (GPU Programming) – David Mayerich
Simple Drawing – Starting and Ending Primitives

• Drawing commands are encapsulated into groups of primitives

• Begin drawing a primitive:

\texttt{glBegin( GLenum mode )}
  
  – \texttt{GL_POINTS}: each vertex is rendered as a single point
  
  – \texttt{GL_LINES}: every pair of vertices represent a line
  
  – \texttt{GL_TRIANGLES}: every three vertices compose a triangle

• End drawing the primitive:

\texttt{glEnd()}
  
  – Stops drawing the current primitive – use this before starting something new
Draw a Triangle to the Viewport

glVertex3f( float x, float y, float z);
    - Specifies a vertex to be drawn

    glBegin( GL_TRIANGLES );  //start a triangle
    glVertex3f(-0.5f, -0.5f, 0.0f);  //specify each vertex
    glVertex3f(0.5f, -0.5f, 0.0f);
    glVertex3f(0.0f, 0.5f, 0.0f);
    glEnd();  //finish drawing

• Vertex coordinates are specified in terms of the projection:
    - glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)
Vertex Properties

- `glVertex3f(...)` draws geometry but nothing may be displayed.
- Properties can be assigned per-vertex to describe rendering traits.
- The simplest trait to specify is vertex color:

  ```
  glColor3f( float r, float g, float b )
  ```
  - specify the vertex color given a red, green, and blue value

```c
glBegin( GL_TRIANGLES );
    glColor3f(1.0f, 0.0f, 0.0f); //red
    glVertex3f(-0.5f, -0.5f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f); //green
    glVertex3f(0.5f, -0.5f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f); //blue
    glVertex3f(0.0f, 0.5f, 0.0f);
glEnd();
```
Plotting Points in 3D

• 3D coordinates can be specified for objects:

```cpp
void cube()
{
    glBegin( GL_TRIANGLES );
    float s = 0.5;
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(-s, -s, -s); //first vertex of triangle 1
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex3f(-s, -s, s);
    glColor3f(0.0f, 1.0f, 1.0f);
    glVertex3f(-s, s, s); //last vertex of triangle 1
    glColor3f(1.0f, 1.0f, 0.0f);
    glVertex3f(s, s, -s); //first vertex of triangle 2
    ... 
    glEnd();
}
```
Viewing Transformations

• The view transform applies the appropriate transformation to account for the camera
  – world coordinates (where objects are) → camera coordinates (relative viewer)

```c
gluLookAt( double ex, double ey, double ez,
           double cx, double cy, double cz,
           double ux, double uy, double uz)
```

  – \(e\) is the position of the viewer in world space
  – \(c\) is the point that the viewer is looking at
  – \(u\) is the up direction (orthogonal to the view direction)
Camera Function for Arbitrary Directions

• This function will generate a camera with an arbitrary position that is always looking at the origin:

```c
void quick_camera(float x, float y, float z) {
    float sx = z; //calculate a side vector
    float sy = 0;
    float sz = x;
    float ux = y * sz - z * sy; //u is cross(s, v)
    float uy = z * sx - x * sz;
    float uz = x * sy - y * sx;
    float s = sqrt(ux * ux + uy * uy + uz * uz); //normalize u
    //create a view matrix
    gluLookAt(x, y, z, 0, 0, 0, ux/s, uy/s, uz/s);
}
```
Drawing a Colorful Cube

• Render function for the scene:

```c
void render() {
    //initialize the projection matrix
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();

    //orthographic view
    glOrtho(-1, 1, -1, 1, -3, 3); // (with space along z)

    //initialize the modelview matrix (model + view)
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    quick_camera(1, 1, 1); //set up camera
    cube_color(); //draw the cube
}
```
Transformations

• Several functions generate basic transformation matrices and multiply them by the current matrix:

\texttt{glTranslatef( float x, float y, float z )}
– generate a translation matrix and multiply it by the top matrix of the current stack

\texttt{glScalef( float x, float y, float z )}
– generate a scale along each axis

\texttt{glRotateref( float angle, float x, float y, float z )}
– rotate by a specified number of degrees along the axis \((x, y, z)\)
Problems with Transformations

• The order that triangles are drawn matters
  – At some angles, rear faces will overdraw front faces
  – Back-face culling:
    • triangles facing away from the viewer aren’t rendered
    • glEnable( GL_CULL_FACE )
  – Depth buffering:
    • pixels are only rendered if they are closer than previous pixels
    • glEnable( GL_DEPTH_TEST )

• The order of transformations matters
Lighting

• Specify lighting parameters:
  
  `glLightfv( GLenum light, GLenum pname, const float* params)`
  
  - specify a light using an identifier (GL_LIGHT0, GL_LIGHT1, etc.)
  - set a parameter (GL_AMBIENT, GL_DIFFUSE, GL_POSITION, etc.)
  - provide the appropriate parameters as a pointer to a list of floats

  • example (directional light, diffuse white):
    
    ```
    float diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv( GL_LIGHT0, GL_DIFFUSE, diffuse);
    float position[] = {1.0f, 1.0f, -1.0f, 0.0f};
    glLightfv( GL_LIGHT0, GL_POSITION, position);
    ```

  • Enable lighting:
    
    ```
    glEnable( GL_LIGHTING );
    glEnable( GL_LIGHT0 );
    ```
Helpful Lighting Notes

• Surface normals have to be specified for each vertex:
  ...
g1Normal3f(-1.0f, 0.0f, 0.0f);
g1Color3f(0.0f, 0.0f, 0.0f);
g1Vertex3f(-s, -s, -s);
  ...

• It is generally convenient to rely on the vertex color as the diffuse and ambient lighting color:
  glColorMaterial( GL_FRONT, GL_AMBIENT_AND_DIFFUSE );
g1Enable( GL_COLOR_MATERIAL );

• It is also helpful to have OpenGL automatically normalize normals (it may not by default):
  glEnable( GL_NORMALIZE );
Display Lists and Speed

• Calls to glBegin( ) and glEnd( ) specify vertices (and their properties) that will be sent to the GPU for rendering
• When you start working with thousands of vertices, this data transfer is a bottleneck
• Generate a display list, that stores all of the vertices and their properties on the GPU:

  glGenLists( unsigned int n )
  – generate n display lists to store geometry

  glNewList( unsigned int list, GLenum mode)
  – generate a new list at ID list
  – mode just specifies if the commands are just compiled (GL_COMPILE) or simultaneously run (GL_COMPILE_AND_EXECUTE)
Building and Using Display Lists

• Generate a display list as an initialization step:

```c
lists = glGenLists(2); //generate two display list IDs starting at index list
glNewList(list, GL_COMPILE); //create a new display list
render_object(); //use standard commands to specify the geometry
glEndList(); //end the display list
glNewList(list+1, GL_COMPILE); //create another display list for a different object
render_another_object(); //render the object using normal commands
glEndList(); //end the display list
...
```

• Execute a display list during render-time:

```c
void render(){ //render function called in event loop
... //do any setup and transformations
glCallList(list); //call the list (run the stored geometry commands)
... //change any state variables for the next object
glCallList(list+1); //run the second list
}
```
Notes on Display Lists

• Display lists are not used commercially – they have technically been deprecated from the OpenGL standard

• The current standards are vertex buffer objects:

```c
unsigned int triangle;
float data[] = {1.0, 0.0, 1.0, 0.0, 0.0, -1.0, -1.0, 0.0, 1.0};
glGenBuffers(1, &triangle);
glBindBuffer( GL_ARRAY_BUFFER, triangle);
glBufferData( GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW );
glBindBuffer( GL_ARRAY_BUFFER, triangle);
glVertexPointer(3, GL_FLOAT, 0, NULL);
glEnableClientState( GL_VERTEX_ARRAY );
glDrawArrays( GL_TRIANGLES, 0, sizeof(data) / sizeof(float) / 3 );
```

• VBOs are more efficient and in line with what DirectX and Vulkan use
Shading Concepts

• Vertex Shader
  – Program executed by the GPU for each vertex

• Fragment Shader
  – Program executed by the GPU for each filled pixel (fragment)