openGL

an introduction via simple example

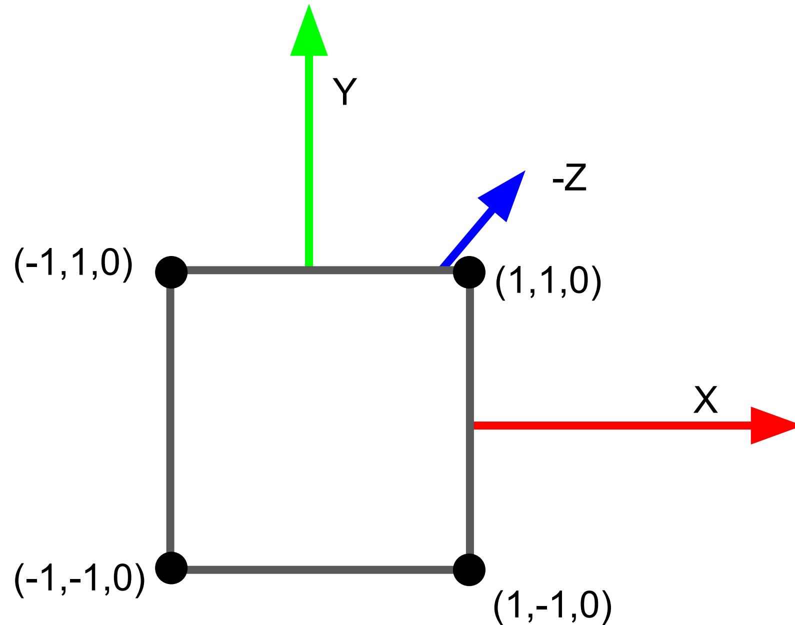
Goals

- Setup a simple scene

- Understand the openGL API from the perspective of shaders

- Understand how this drives openGL application code

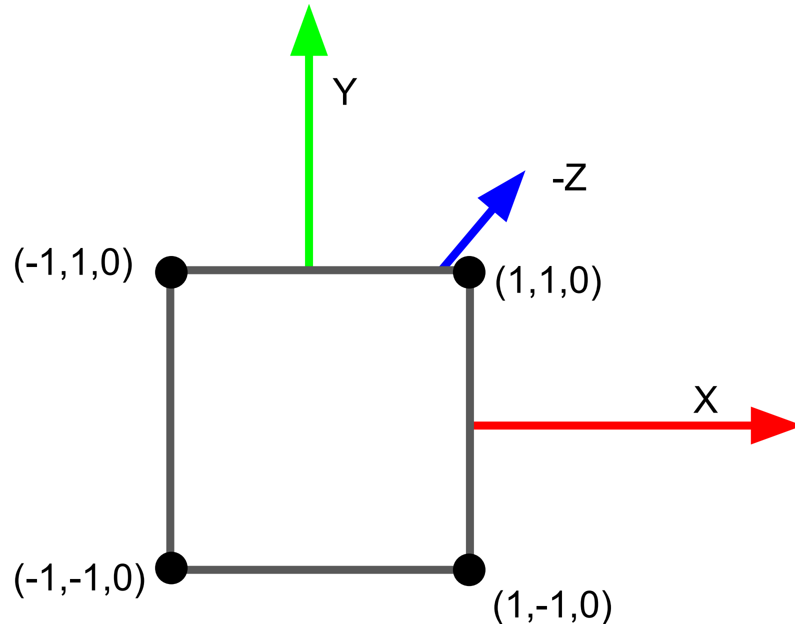
Example: suppose we are drawing this simple scene



We need to do two things

1. Define the geometry
2. Define how the geometry should be visualized

Geometry is defined in terms of vertices



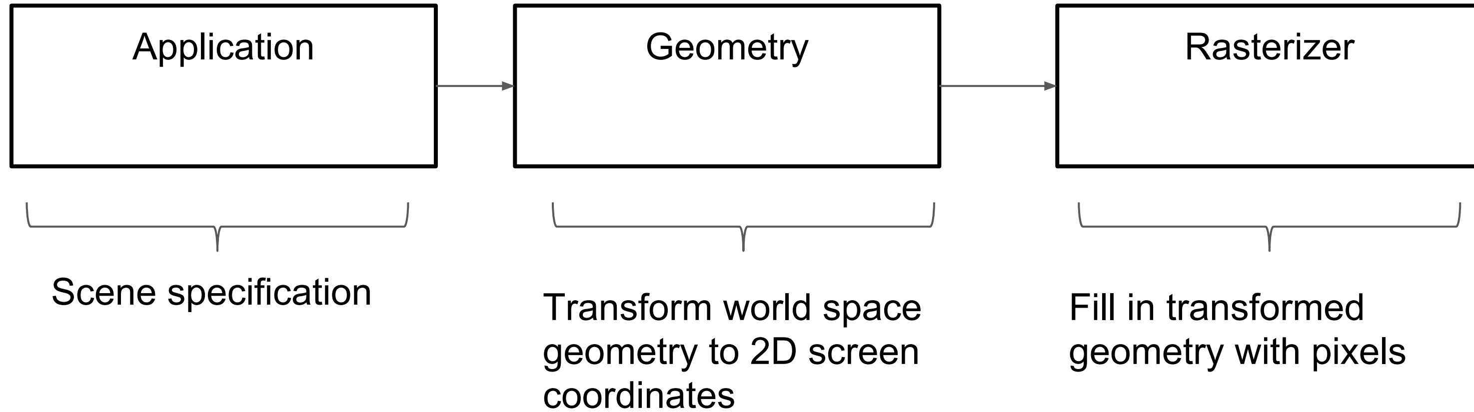
World Coordinate System

- Right-handed
- Y coord is UP

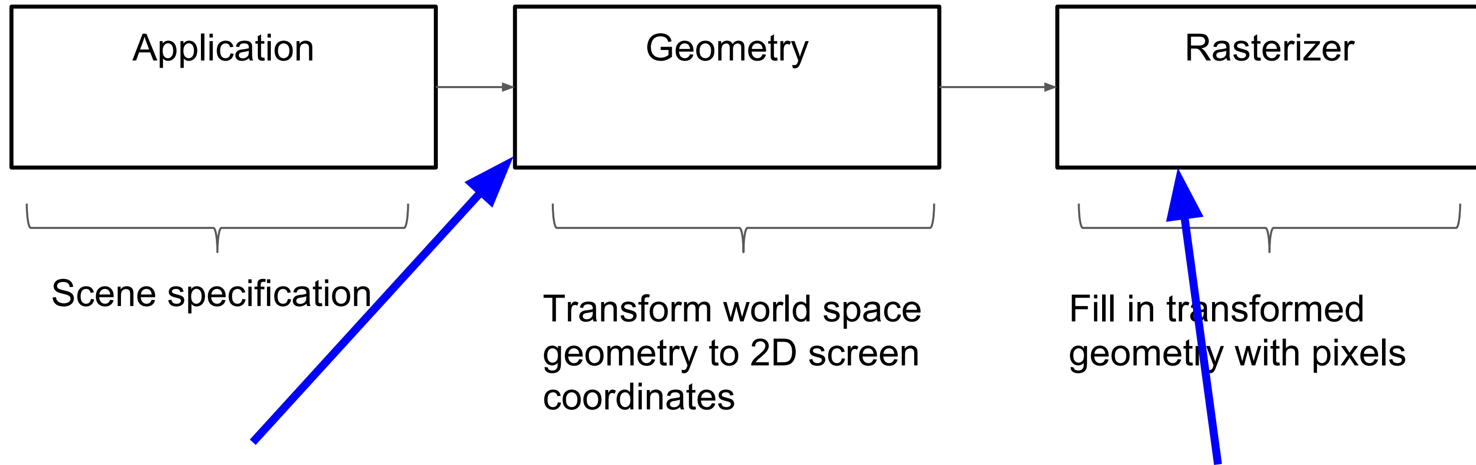
Vertices

- Position, color, normals, etc

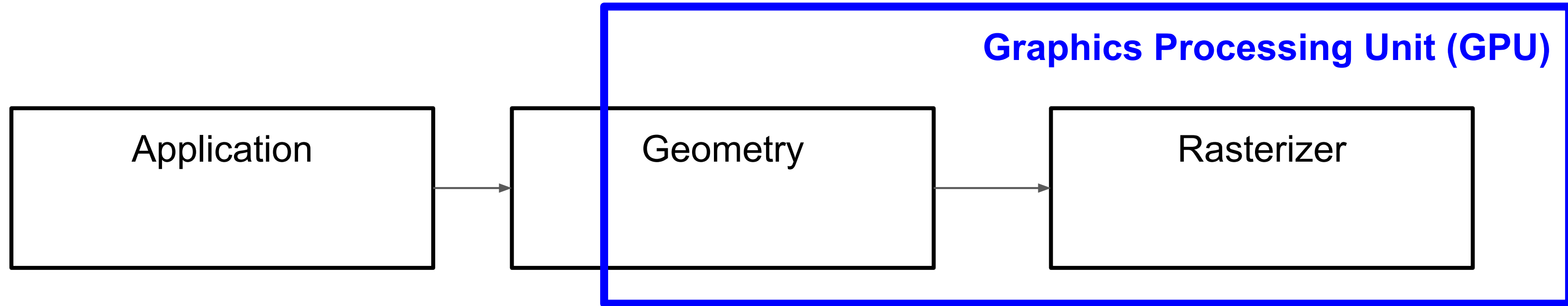
Rendering vertices



3. Defining how vertices are rendered



3. Defining how vertices are rendered



GPU is a SPMD architecture (Single Program, Multiple Data)
- Vertices processed in parallel
- Pixels processed in parallel

Vertex shader

Inputs: raw geometry vertices, user state

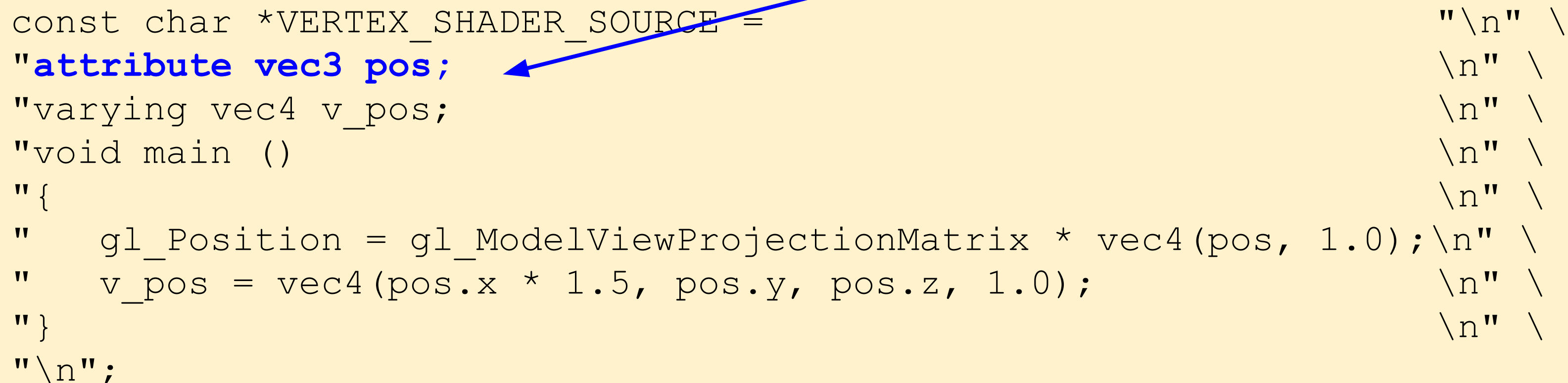
Output: transformed, projected vertices

```
const char *VERTEX_SHADER_SOURCE =                              "\n" \
"attribute vec3 pos;                                            \n" \
"varying vec4 v_pos;                                            \n" \
"void main ()                                                   \n" \
"{                                                              \n" \
"   gl_Position = gl_ModelViewProjectionMatrix * vec4(pos, 1.0);\n" \
"   v_pos = vec4(pos.x * 1.5, pos.y, pos.z, 1.0);               \n" \
"}                                                              \n" \
"\n";
```

Vertex shader

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices



Vertex shader

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices

Matrices which convert from input coordinates to clip space coordinates

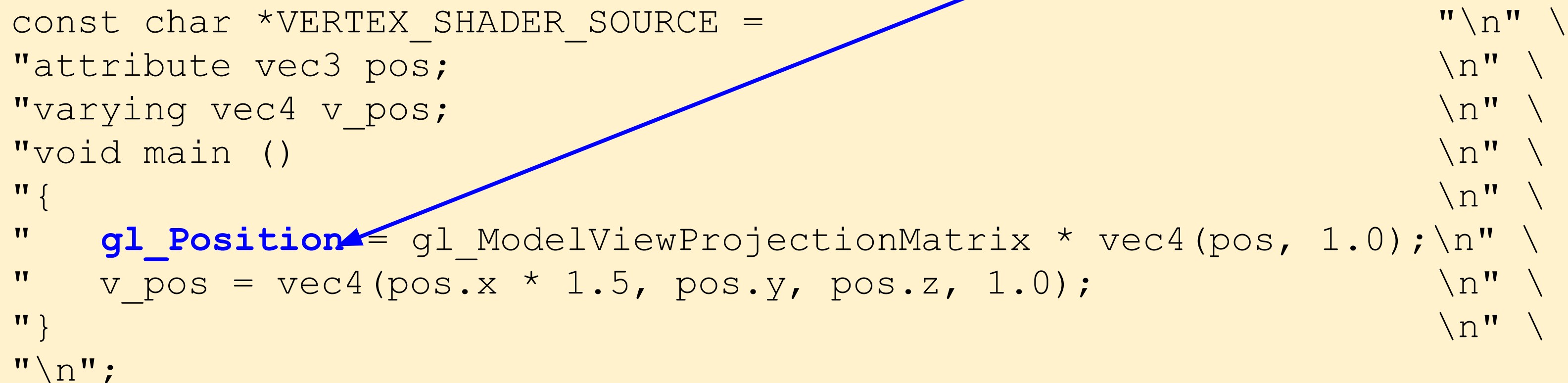
```
const char *VERTEX_SHADER_SOURCE =                                  "\n" \
"attribute vec3 pos;                                                \n" \
"varying vec4 v_pos;                                                \n" \
"void main ()                                                       \n" \
"{                                                                  \n" \
"   gl_Position = gl_ModelViewProjectionMatrix * vec4(pos, 1.0);\n" \
"   v_pos = vec4(pos.x * 1.5, pos.y, pos.z, 1.0);                   \n" \
"}                                                                  \n" \
"\n";
```

Vertex shader

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices

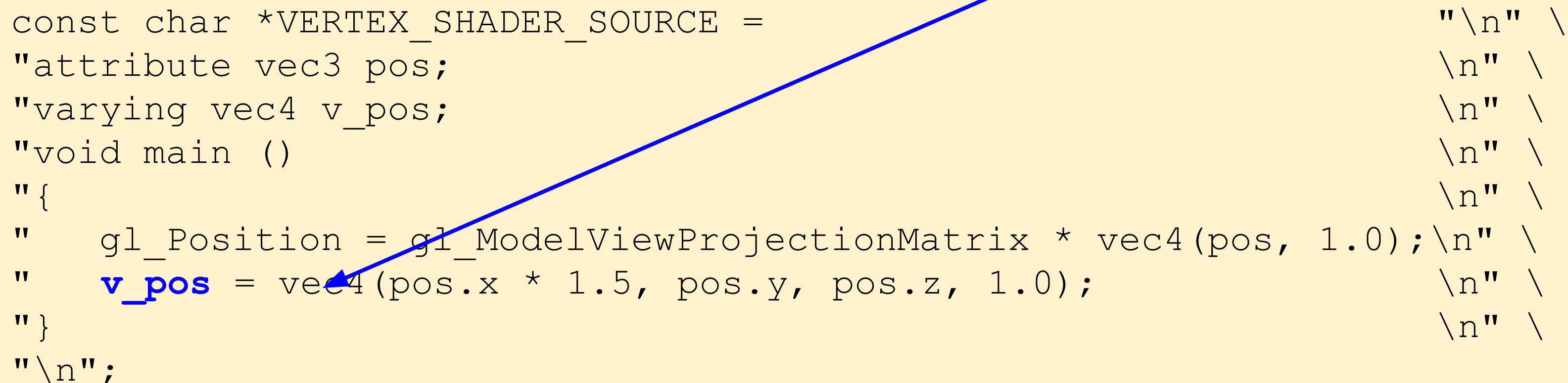
Output position



Vertex shader

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices



Vertex shader

attribute -> vertex property

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices

```
const char *VERTEX_SHADER_SOURCE =                                   "\n" \
"attribute vec3 pos;                                                 \n" \
"varying vec4 v_pos;                                                 \n" \
"void main ()                                                        \n" \
"{                                                                   \n" \
"   gl_Position = gl_ModelViewProjectionMatrix * vec4(pos, 1.0);\n" \
"   v_pos = vec4(pos.x * 1.5, pos.y, pos.z, 1.0);                    \n" \
"}                                                                   \n" \
"\n";
```

Vertex shader

Inputs: raw geometry vertices, user state

Output: transformed, projected vertices

varying -> vertex property will be interpolated in fragment shader

```
const char *VERTEX_SHADER_SOURCE =                              "\n" \
"attribute vec3 pos;                                            \n" \
"varying vec4 v_pos;                                            \n" \
"void main ()                                                   \n" \
"{                                                              \n" \
"   gl_Position = gl_ModelViewProjectionMatrix * vec4(pos, 1.0);\n" \
"   v_pos = vec4(pos.x * 1.5, pos.y, pos.z, 1.0);               \n" \
"}                                                              \n" \
"\n";
```

Fragment shader

Inputs: Per-pixel, interpolated vertex state and user parameters

Output: pixel color

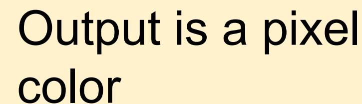
```
const char * FRAGMENT_SHADER_SOURCE =
"\n" \
"uniform vec4 u_color;                                          \n" \
"void main ()                                                   \n" \
"{                                                              \n" \
"   gl_FragColor = vec4(u_color.r, u_color.g, u_color.b, u_color.a); \n" \
"}                                                              \n" \
"\n";
```

Fragment shader

Inputs: Per-pixel, interpolated vertex state and user parameters

Output: pixel color

```
const char * FRAGMENT_SHADER_SOURCE =
"\n" \
"uniform vec4 u_color;                                              \n" \
"void main ()                                                       \n" \
"{                                                                  \n" \
"    gl_FragColor = vec4(u_color.r, u_color.g, u_color.b, u_color.a); \n" \
"}                                                                  \n" \
"\n";
```



Fragment shader

Inputs: Per-pixel, interpolated vertex state and user parameters

Output: pixel color

```
const char * FRAGMENT_SHADER_SOURCE =
"\n" \
"uniform vec4 u_color;                                              \n" \
"void main ()                                                       \n" \
"{                                                                  \n" \
"   gl_FragColor = vec4(u_color.r, u_color.g, u_color.b, u_color.a); \n" \
"}                                                                  \n" \
"\n";
```

uniform -> variable is the same for every fragment/vertex

So, our remaining work...

How do we define and send vertex information to the card?

How do we load and compile our shaders?

How do we invoke the shader to draw?

Defining vertices

Vertices have multiple geometric information associated with them, e.g. position, texture coordinate, normal, color, ..

Let’s focus on positions for now

Best practice is to store values in a flat array

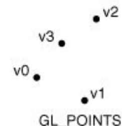
x1| y1|z1|x2|y2|z2| … |xn|yn|zn

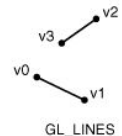
How vertices are interpreted depends on the primitive type

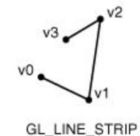
E.g. points, lines, triangles, etc.

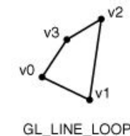
Defining vertices

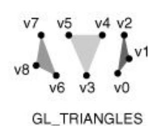
```
GLfloat vertices[12] =
{
  -1.0f, -1.0f, 0.0f,
  -1.0f,  1.0f, 0.0f,
   1.0f,  1.0f, 0.0f,
   1.0f, -1.0f, 0.0f,
};

glGenBuffers(1, &theQuadId);
glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
     vertices, GL_STATIC_DRAW);
```

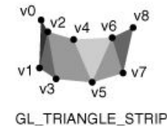


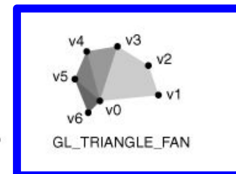


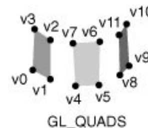


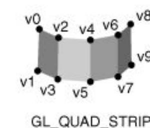


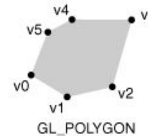






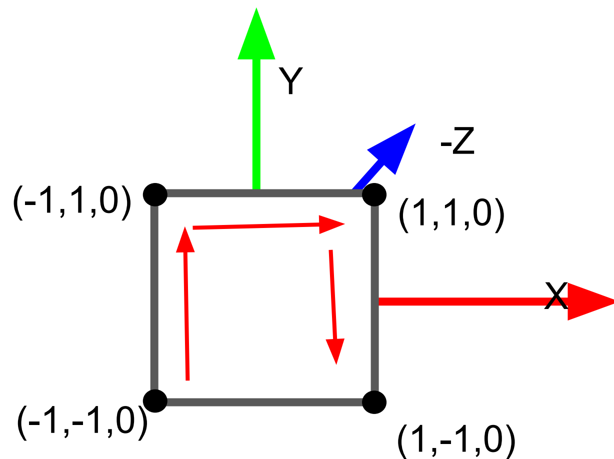






Defining vertices

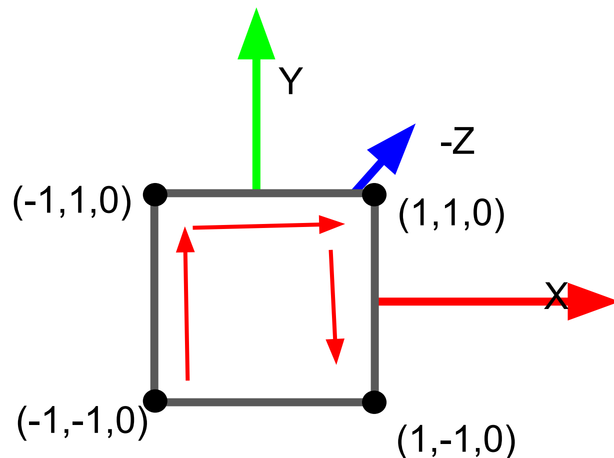
```
GLfloat vertices[12] =
{
  -1.0f, -1.0f, 0.0f,
  -1.0f,  1.0f, 0.0f,
   1.0f,  1.0f, 0.0f,
   1.0f, -1.0f, 0.0f,
};

glGenBuffers(1, &theQuadId);
glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
     vertices, GL_STATIC_DRAW);
```



Defining vertices

Allocate memory on the GPU for our vertices and copy the data over. theQuadId will refer to it.

```
GLfloat vertices[12] =
{
  -1.0f, -1.0f, 0.0f,
  -1.0f,  1.0f, 0.0f,
   1.0f,  1.0f, 0.0f,
   1.0f, -1.0f, 0.0f,
};

glGenBuffers(1, &theQuadId);
glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
     vertices, GL_STATIC_DRAW);
```

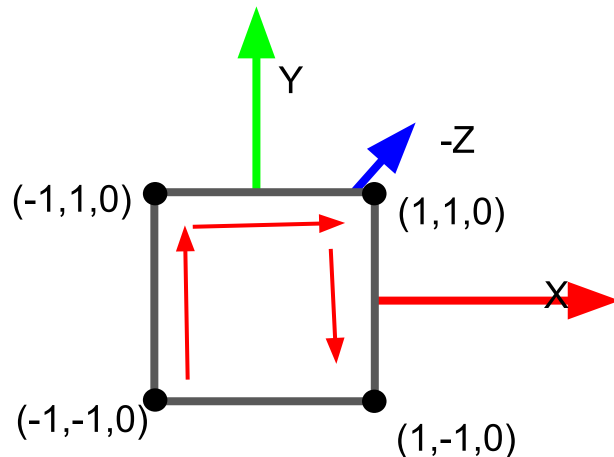


Triangle strip!

Defining vertices

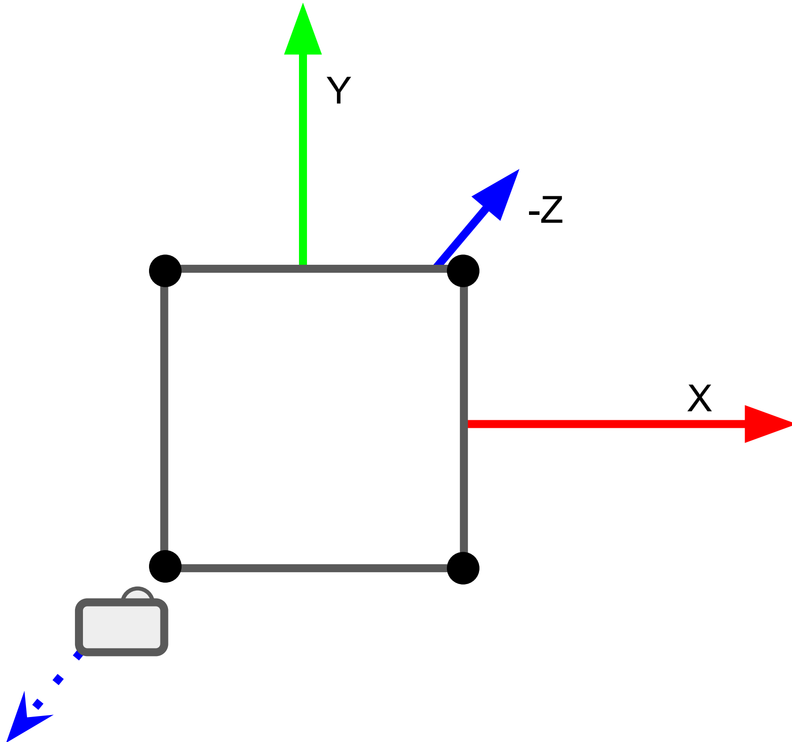
Indicates that the vertex positions don’t change

```
GLfloat vertices[12] =
{
  -1.0f, -1.0f, 0.0f,
  -1.0f,  1.0f, 0.0f,
   1.0f,  1.0f, 0.0f,
   1.0f, -1.0f, 0.0f,
};

glGenBuffers(1, &theQuadId);
glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
     vertices, GL_STATIC_DRAW);
```



Triangle strip!

Transforming vertices so we can see them

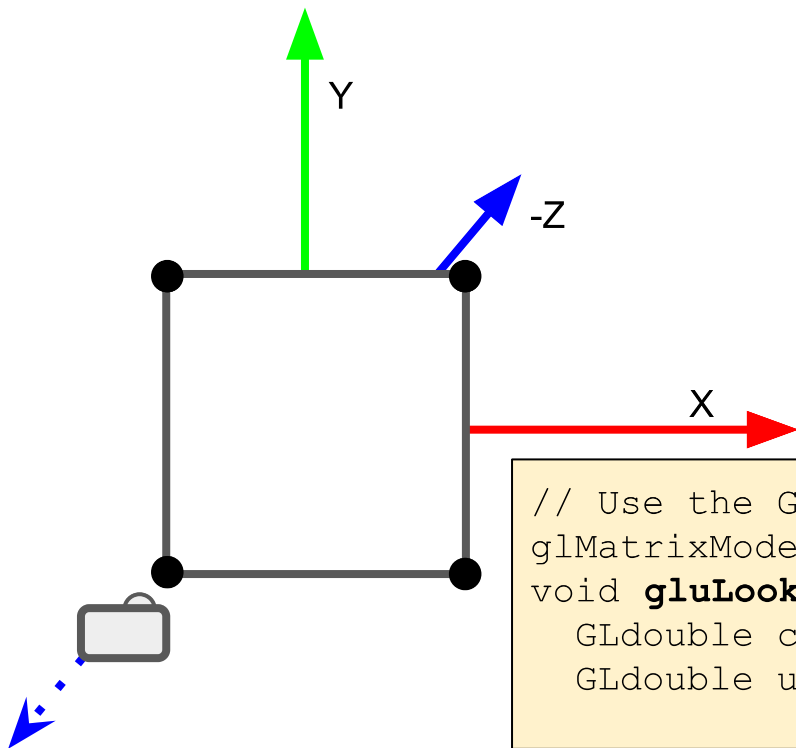


Camera concept

- Specifies which parts of the scene you want to draw
- A helpful concept; doesn't exist in the scene per se
- Default position is (0,0,0) facing the negative Z axis

```
// Use default eye/camera
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Transforming vertices so we can see them

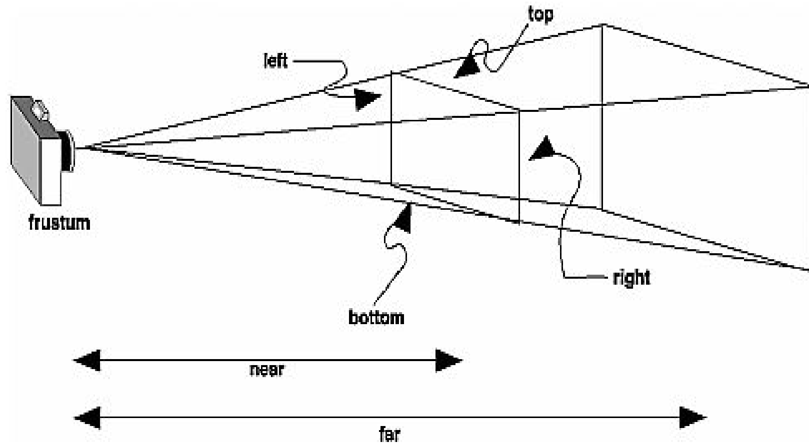


Camera concept

- Specifies which parts of the scene you want to draw
- A helpful concept; doesn’t exist in the scene per se

```
// Use the GLU helper function
glMatrixMode(GL_MODELVIEW);
void gluLookAt(GLdouble eyeX,GLdouble eyeY,GLdouble eyeZ,
  GLdouble centerX, GLdouble centerY, GLdouble centerZ,
  GLdouble upX, GLdouble upY, GLdouble upZ);
```

Transforming vertices so we can see them



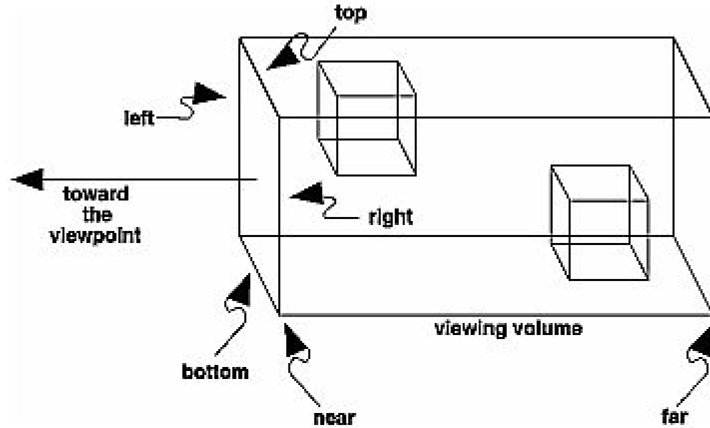
glprogramming.com

Camera concept

- Specifies clipping volume
- How to project 3D points onto the 2D screen
  - Orthographic
  - **Perspective**

void ***gluPerspective****(GLdouble* fovy*, GLdouble* aspect*,*
GLdouble near*, GLdouble* far*);*

Transforming vertices so we can see them



glprogramming.com

Camera concept

- Specifies clipping volume
- How to project 3D points onto the 2D screen
  - **Orthographic**
  - Perspective

void ***glOrtho****(GLdouble* left*, GLdouble* right*, GLdouble* bottom*,*
GLdouble top*, GLdouble* near*, GLdouble* far*);*

Specifying the viewport

Viewport

- Amount of window to draw on
- Should be reset when the window changes size
- Values are in pixels

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Clearing the screen

Setting the clear color

```
void glClearColor(GLdouble r, GLdouble g, GLdouble b, GLdouble a);
```

Clearing buffers

```
void glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

How do we load and compile our shaders?

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &VERTEX_SHADER_SOURCE, NULL);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &FRAGMENT_SHADER_SOURCE, NULL);
glCompileShader(fragmentShader);

theShaderProgram = glCreateProgram();
glAttachShader(theShaderProgram, vertexShader);
glAttachShader(theShaderProgram, fragmentShader);
glLinkProgram(theShaderProgram);

glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

How do we load and compile our shaders?

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1
glCompileShader(vertexShader);

GLuint fragmentShader = glCrea
glShaderSource(fragmentShader,
glCompileShader(fragmentShader

theShaderProgram = glCreatePro
glAttachShader(theShaderProgra
glAttachShader(theShaderProgra
glLinkProgram(theShaderProgram

glDeleteShader(vertexShader);
glDeleteShader(fragmentShader)
```

```
// not shown, but actual code checks for
// compilation and link errors after these
// calls, e.g.
GLint success;
GLchar infoLog[512];
glGetShaderiv(vertexShader,
    GL_COMPILE_STATUS, &success);

if (!success)
{
  glGetShaderInfoLog(vertexShader,
        512, NULL, infoLog);
  std::cout << "Vertex shader did not
        compile\n" << infoLog <<std::endl;
}
```

How do we load and compile our shaders?

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &VERTEX_SHADER_SOURCE, NULL);
glCompileShader(vertexShader);

GLuint fragmentShader = glCreateShader(GL_
glShaderSource(fragmentShader, 1, &FRAGMEN
glCompileShader(fragmentShader);

theShaderProgram = glCreateProgram();
glAttachShader(theShaderProgram, vertexShader);
glAttachShader(theShaderProgram, fragmentShader);
glLinkProgram(theShaderProgram);

glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

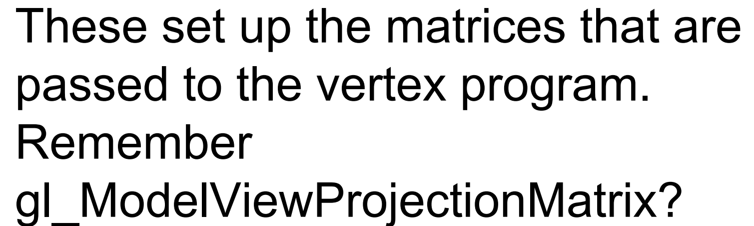
- Create program on GPU and return ID
- Attach vertex and fragment shaders
- Link

How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```

How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLflo
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```



How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLf
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```

All subsequent geometry will use this shader!

How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```

How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```

How do we invoke the shader to draw?

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -10, 10);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // doing matrix math
glUseProgram(theShaderProgram);

glBindBuffer(GL_ARRAY_BUFFER, theQuadId);
GLuint posAttrLoc = glGetAttribLocation(theShaderProgram, "pos");
glVertexAttribPointer(posAttrLoc, 3,
    GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(posAttrLoc);

glBindVertexArray(theQuadId);
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glutSwapBuffers();
```

Swap buffers (for double buffered input); otherwise, we would call glFlush()

Scaling it up: Multiple objects

- Only need to define the geometry once (called **Instancing**)
-
- Transformation matrices to move/rotate/scale the geometry for each instance
  - Scale, Translate, Rotate
  - Best practice is to transform each vertex using the vertex shader (rather than application)
  - **Matrix stack:** push/pop matrix stack used to handle relative coordinate systems

```
glPushMatrix();
        glTranslatef(0, 1, 0);
        glRotatef(45.0, 0, 0, 1);
        glScalef(1.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glPopMatrix();
```

Core profile vs Immediate mode

- Core profile: the modern OpenGL API
  - Better performance
  - Requires a lot of boilerplate code to get setup
  - Complete flexibility via **shaders**
  -
- Immediate mode: the old, deprecated API
  - Simple state machine for all drawing
  - Much easier to use
  - Slower
  -
- You’ll see both used in the basecode for assignments

```
// Immediate mode draw line
glBegin(GL_LINES);
	glColor4f(1,0,0,1);
	glVertex3f(0,0,0);
	glVertex3f(1,0,0);
glEnd();
```

Helper APIs

- Lots of helper APIs for openGL applications
    - GLEW (C/C++), helpers for accessing 2.0+ features
    - Glut (C/C++), supports simple primitive, windowing and user input
    - FLTK (C/C++), AntTweakBar, simple GUIs
    - Euler (C/C++), matrix and linear algebra
    - glMatrix.js (JS), matrix routines
    - webgl-utils.js (JS), window/canvas helpers
    - FBX, OBJ, **BVH**, exs of open geometry and modeling formats

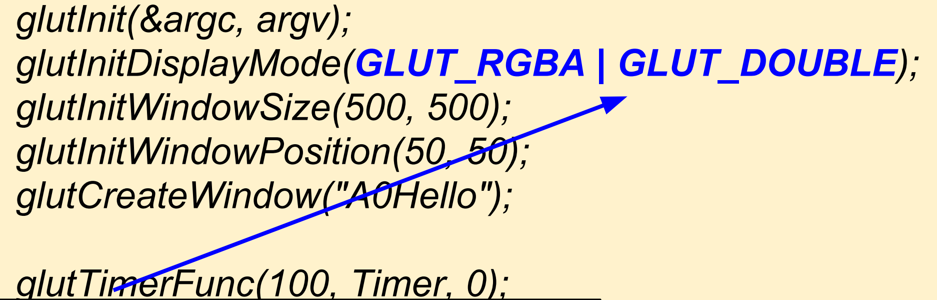
GLUT - simple window and input API

- Window management
  - Resize
  - Creation
  - Draw
  -
- User input
  - Keyboard
  - Mouse
  - Right click menu
  -
- Timer
  - Note it’s better to keep application update and drawing separate. Why?

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
glutInitWindowSize(500, 500);
glutInitWindowPosition(50, 50);
glutCreateWindow("A0Hello");

glutTimerFunc(100, Timer, 0);
glutDisplayFunc(Draw);
glutMainLoop();
```

GLUT - simple window and input API

- Window management
  - Resize
  - Creation
  - Draw
  -
- User input
  - Keyboard
  - Mouse
  - Right click menu
  -
- Timer
  - Note it's better to keep application update and drawing separate. Why?



Window supports RGBA colors and double buffering

GLUT - simple window and input API

- Window management
  - Resize
  - Creation
  - Draw
  -
- User input
  - Keyboard
  - Mouse
  - Right click menu
  -
- Timer
  - Note it’s better to

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
glutInitWindowSize(500, 500);
glutInitWindowPosition(50, 50);
glutCreateWindow("A0Hello");

glutTimerFunc(100, Timer, 0);
glutDisplayFunc(Draw);
glutMainLoop();
```

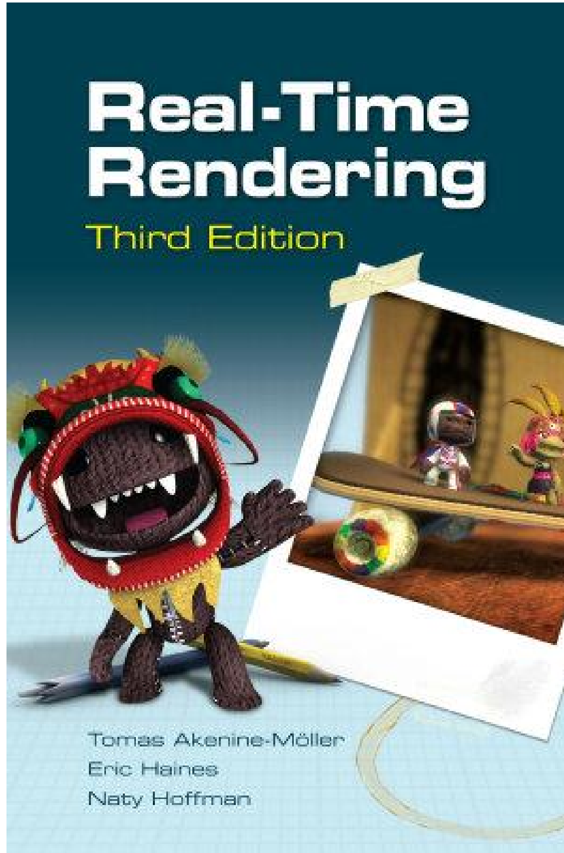
Callbacks: pass function pointers; these functions will be invoked on timer timeout and when we need to redraw

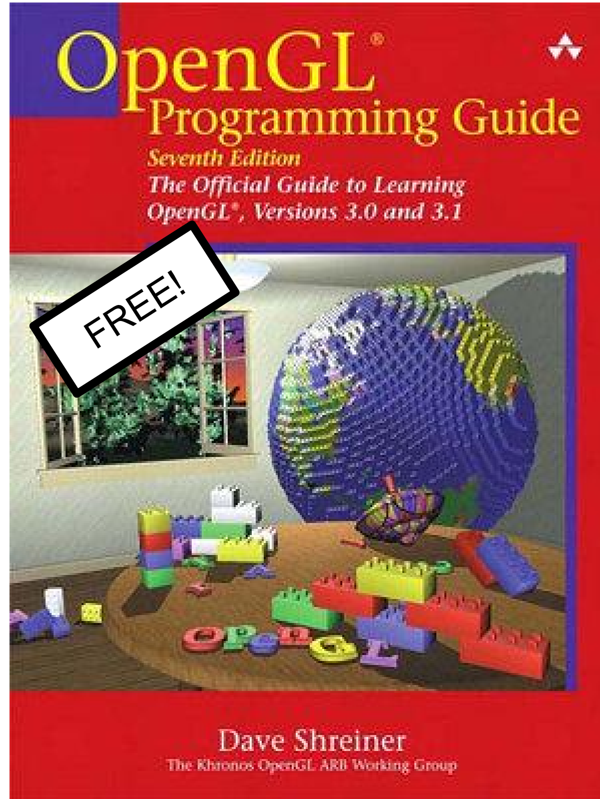
Sidebar: Other (high level) ways of drawing in 3D

- Scene graph APIs
  - Organizes geometry, contains default primitives and shaders
  - OpenSceneGraph (C/C++)
  - OGRE (C/C++)
  - Panda 3D (Python)
  - Three.js (JS)

- Game engines
  - Unity
  - Unreal Engine 4

Books





http://www.glprogramming.com/red/

Online resources

Khronos

- Open standards committee for graphics, parallel computing, etc
- OpenGL 4.x API Quick Reference Card, www.opengl.org/documentation/glsl/

Tutorial resources

C/C++	webGL	openGL ES
https://learnopengl.com/ NeHe tutorials (old API, but concepts are sound)	http://learningwebgl.com/ http://webglfundamentals.com	Android: developer.android.com Apple: developer.apple.com/