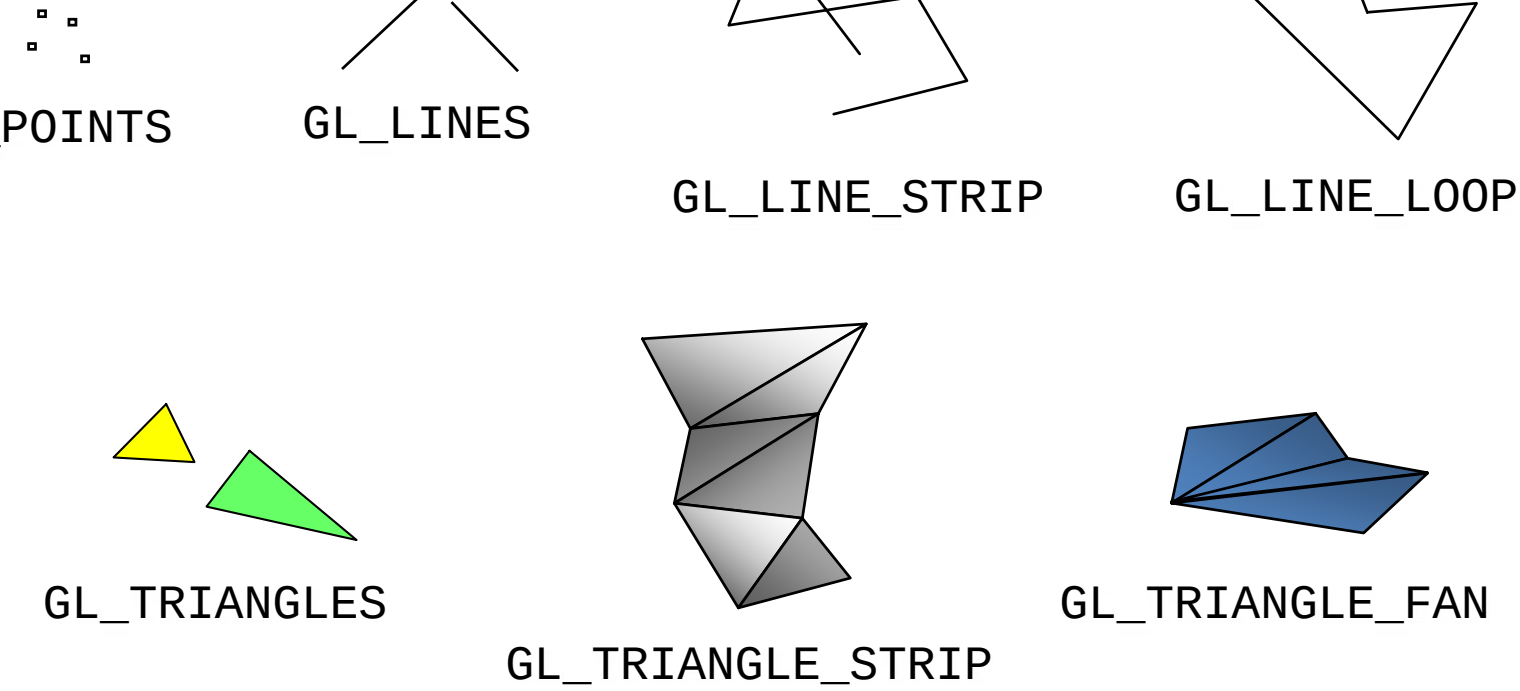
Rendering Basics

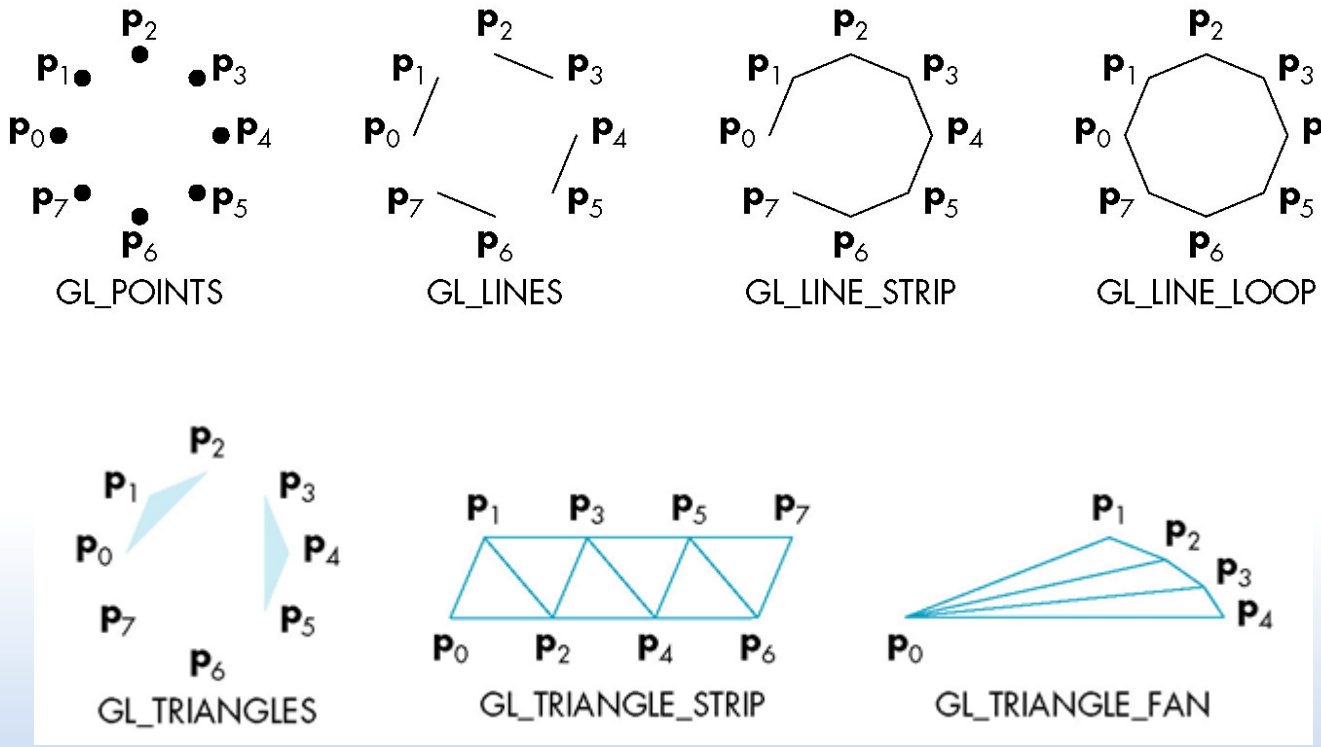
Primitives

- OpenGL Primitives



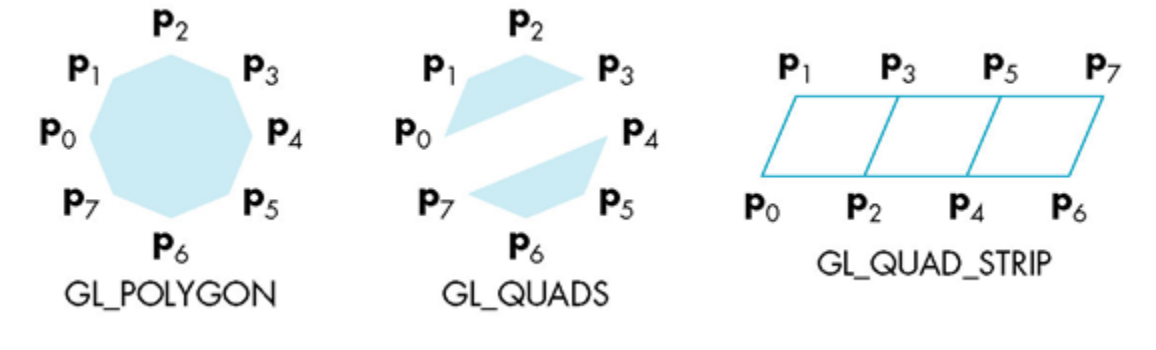
Primitives

- OpenGL Primitives



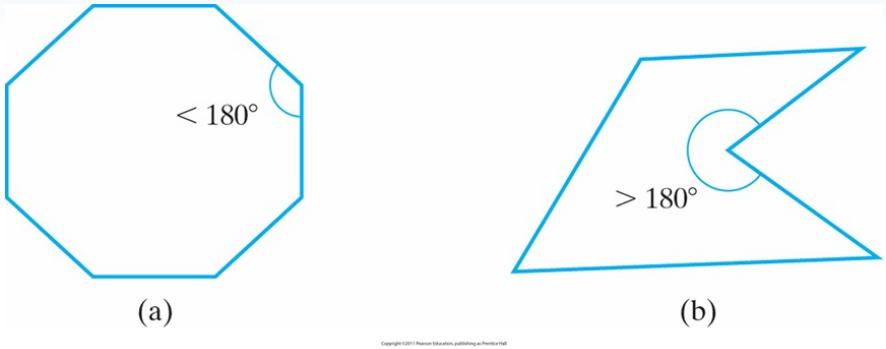
Primitives

- Deprecated OpenGL primitives
  - **Removed** from core OpenGL 3.1 and above
  - DO NOT USE

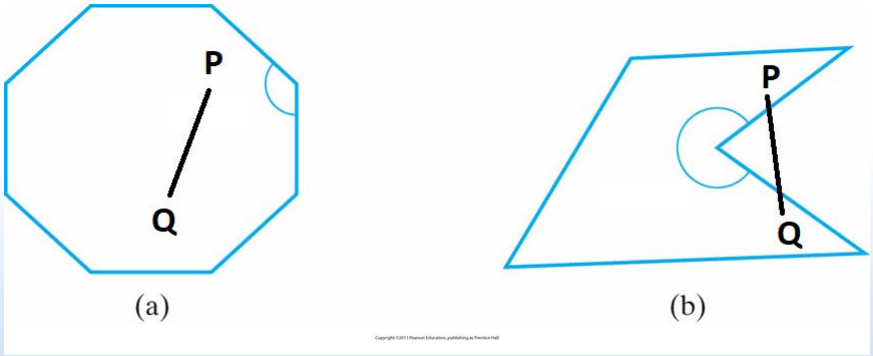


Polygons

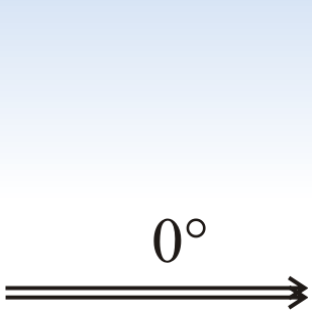
- Convexity
  - Convex if *all* interior angles are <180º
    - Whereas concave if *at least one* angle is >180º
  - Convex if and only if
    - For any two points in the object, all points on the line segment between these points are also in the object
      - Line segment does not cross any edges

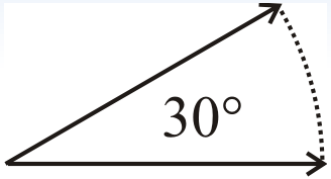


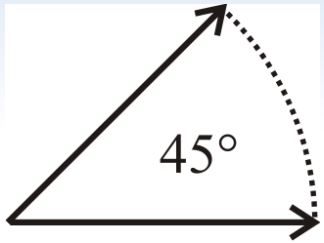
Convex Concave

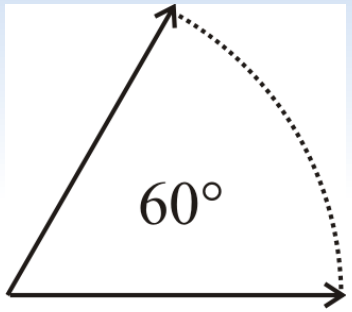


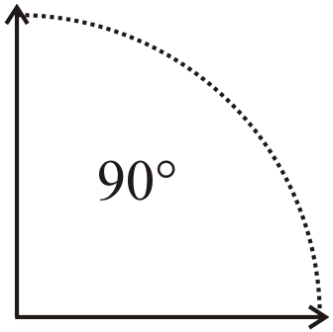
Angles

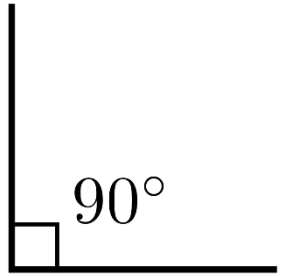




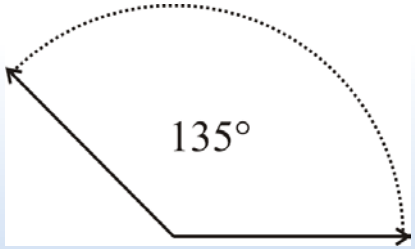


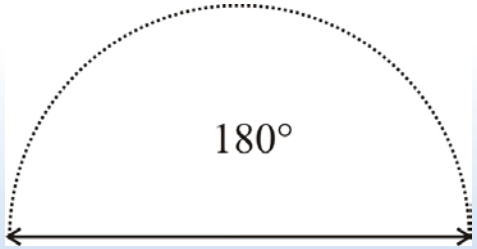


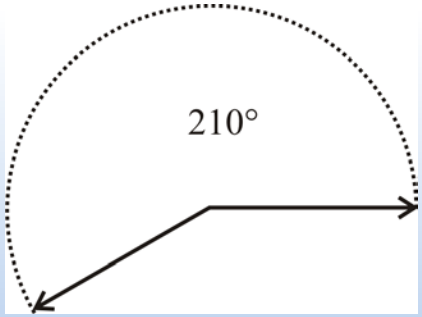


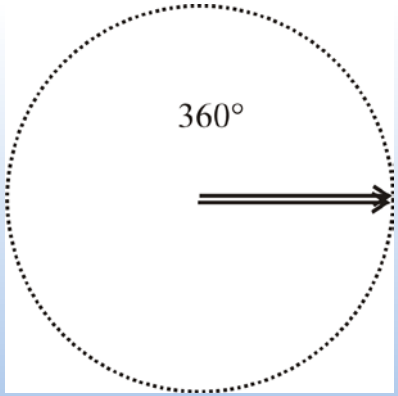


Right angle: The vertical is perpendicular to the horizontal base line





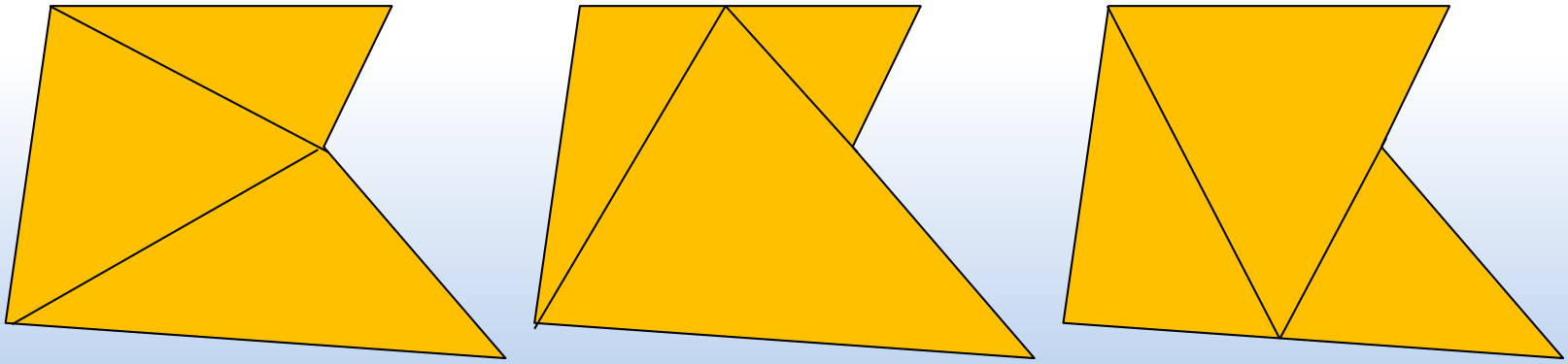




http://www.homeschoolmath.net/teaching/g/measure_angles.php

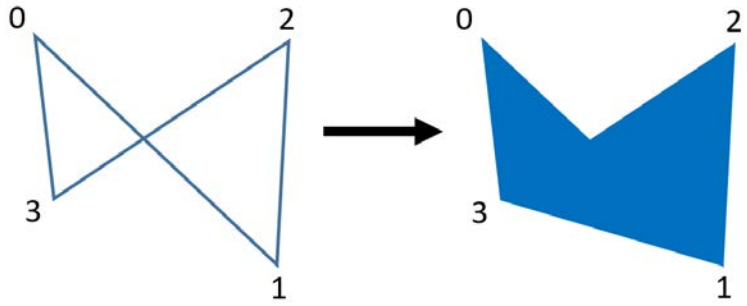
Polygons

- Convexity
  - Concave polygons may cause problems in certain algorithms
  - Can break concave polygons into triangles (always convex)
    - Angles within a triangle sum to 180^o
    - Polygon triangulation, however involves storing more polygons



Polygons

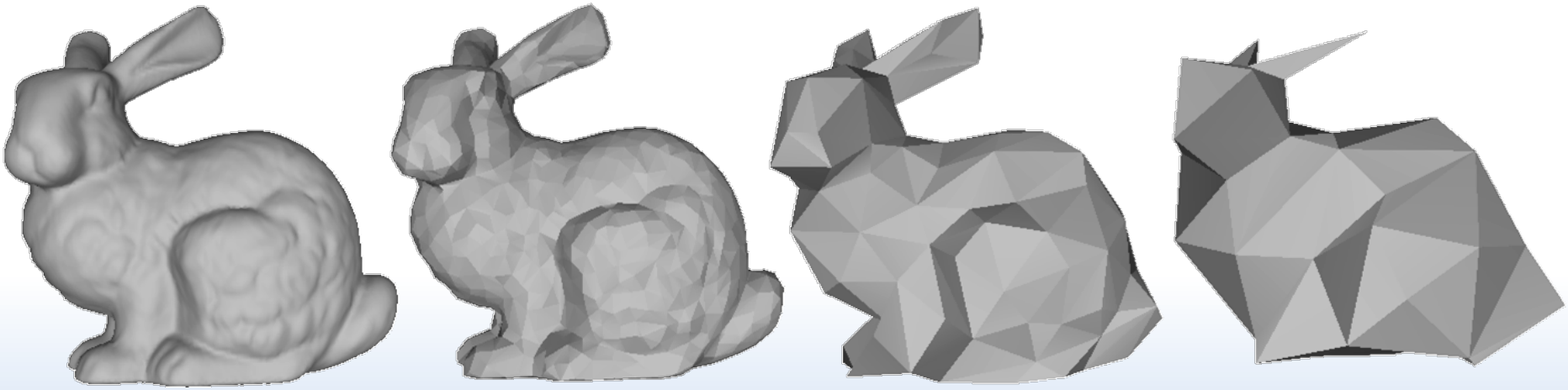
- Polygon issues
  - OpenGL will only display polygons correctly if they are
    - Simple
      - Edges cannot cross
    - Convex
    - Flat
      - All vertices are on the same plane



  - If these conditions are violated
    - OpenGL will still produce output, but it may not be what is desired
  - Triangles satisfy all these conditions
    - By removing GL_POLYGON, GL_QUADS, GL_QUAD_STRIP, these issues no longer of concern

Polygons

- Any surface can be tessellated into triangles
  - The more triangles used, the more accurate a surface can be modeled

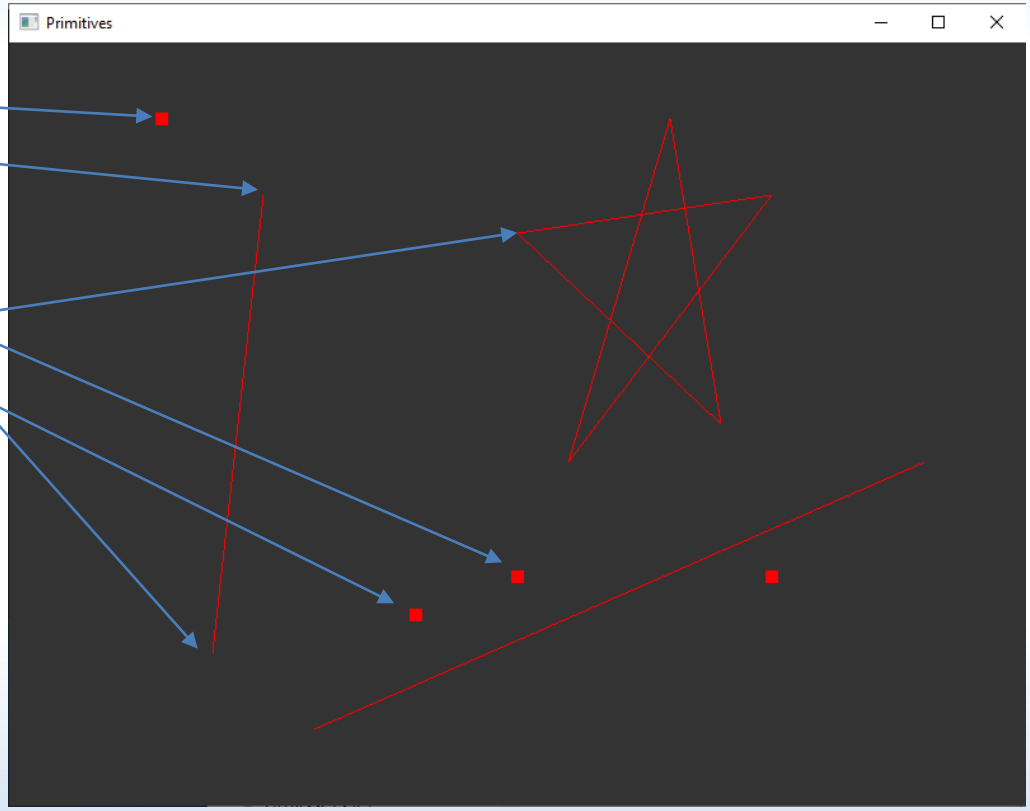


69,451 polygons 2,502 polygons 251 polygons 76 polygons

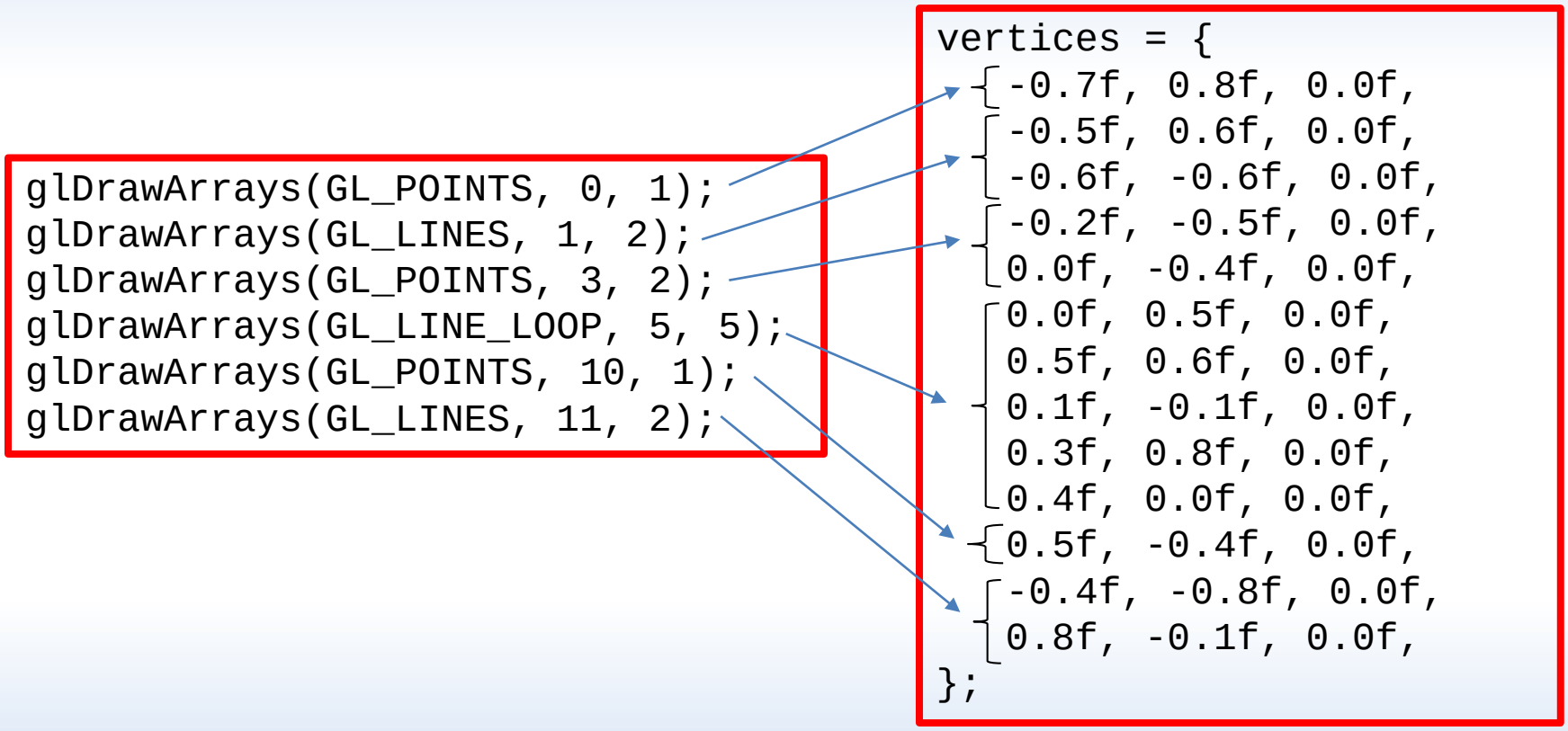
http://www.cs.virginia.edu/~luebke/bunny.html

Primitives

```
vertices = {
    -0.7f, 0.8f, 0.0f,
    -0.5f, 0.6f, 0.0f,
    -0.6f, -0.6f, 0.0f,
    -0.2f, -0.5f, 0.0f,
    0.0f, -0.4f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.5f, 0.6f, 0.0f,
    0.1f, -0.1f, 0.0f,
    0.3f, 0.8f, 0.0f,
    0.4f, 0.0f, 0.0f,
    0.5f, -0.4f, 0.0f,
    -0.4f, -0.8f, 0.0f,
    0.8f, -0.1f, 0.0f,
};
```

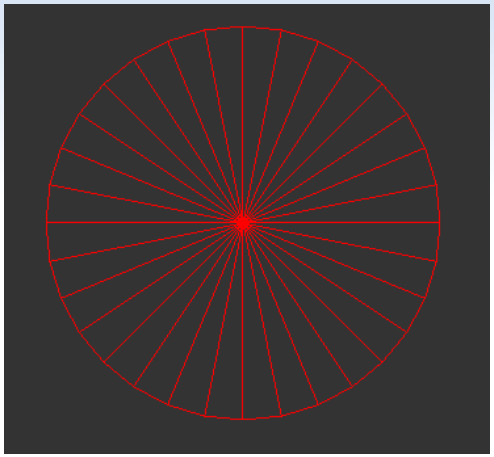


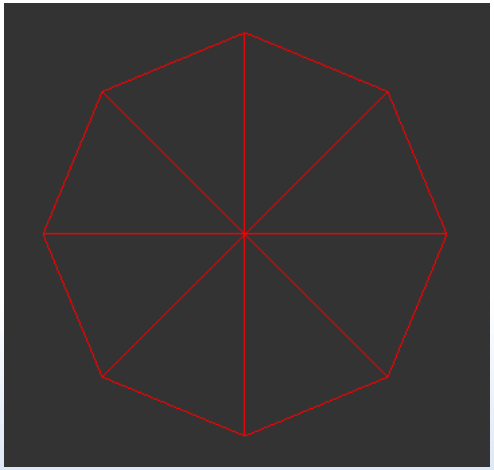
Primitives

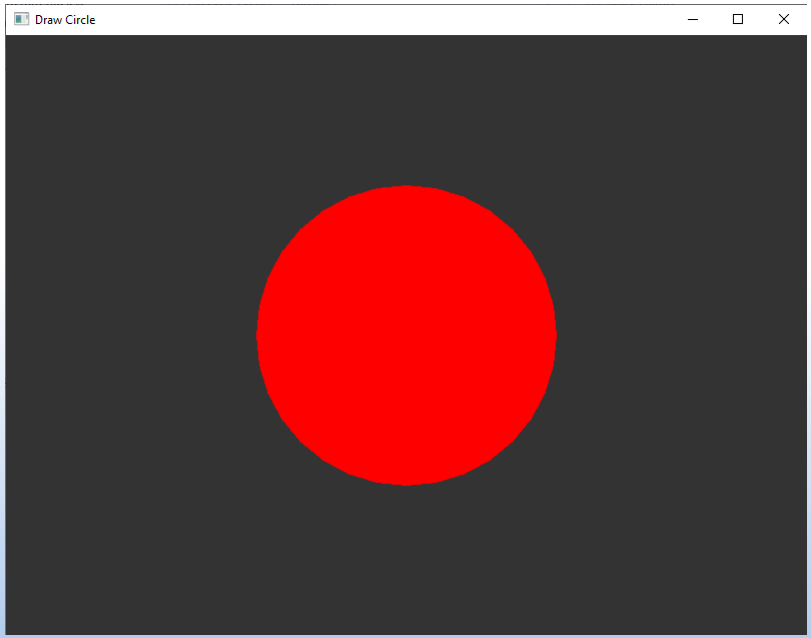

glDrawArrays(GL_POINTS, 0, 1);
glDrawArrays(GL_LINES, 1, 2);
glDrawArrays(GL_POINTS, 3, 2);
glDrawArrays(GL_LINE_LOOP, 5, 5);
glDrawArrays(GL_POINTS, 10, 1);
glDrawArrays(GL_LINES, 11, 2);
vertices = {
-0.7f, 0.8f, 0.0f,
-0.5f, 0.6f, 0.0f,
-0.6f, -0.6f, 0.0f,
-0.2f, -0.5f, 0.0f,
0.0f, -0.4f, 0.0f,
0.0f, 0.5f, 0.0f,
0.5f, 0.6f, 0.0f,
0.1f, -0.1f, 0.0f,
0.3f, 0.8f, 0.0f,
0.4f, 0.0f, 0.0f,
0.5f, -0.4f, 0.0f,
-0.4f, -0.8f, 0.0f,
0.8f, -0.1f, 0.0f,
};

Draw Circle

```
#define _USE_MATH_DEFINES
#include <cmath>

std::vector<GLfloat> gVertices;
#define MAX_SLICES 32
#define MIN_SLICES 8
unsigned int gSlices = MIN_SLICES;
```



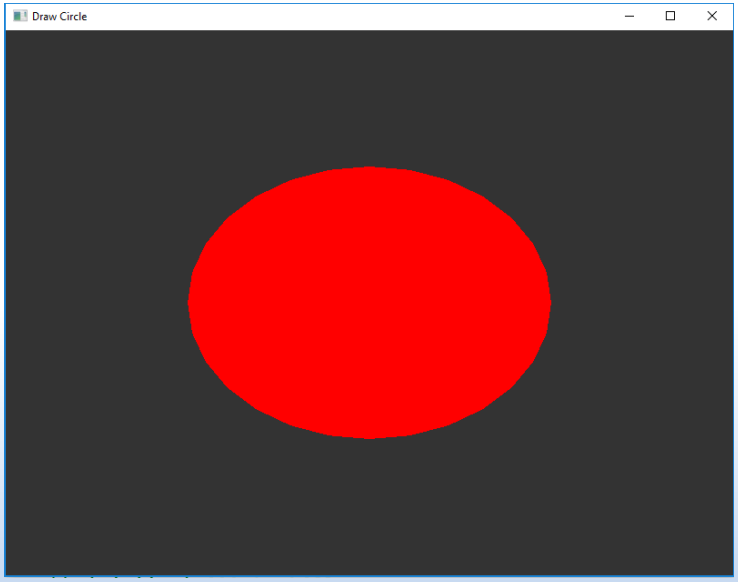




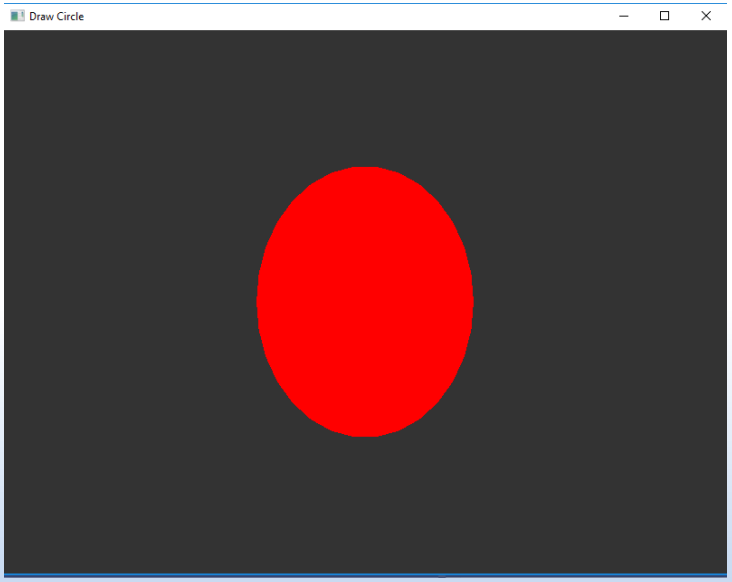
Draw Circle

```
// controls whether circle or elipse
float gScaleFactor = static_cast<float>(gWindowHeight) /
        gWindowWidth;
```

Scale factor = 1.0f

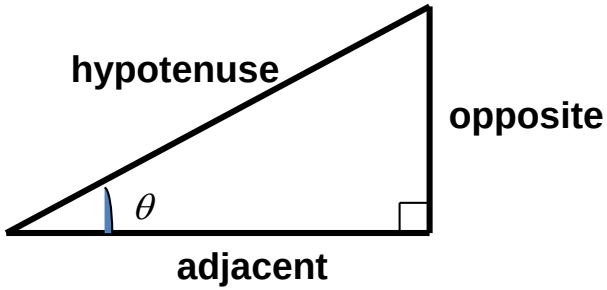


Scale factor = 0.6f



Trigonometry

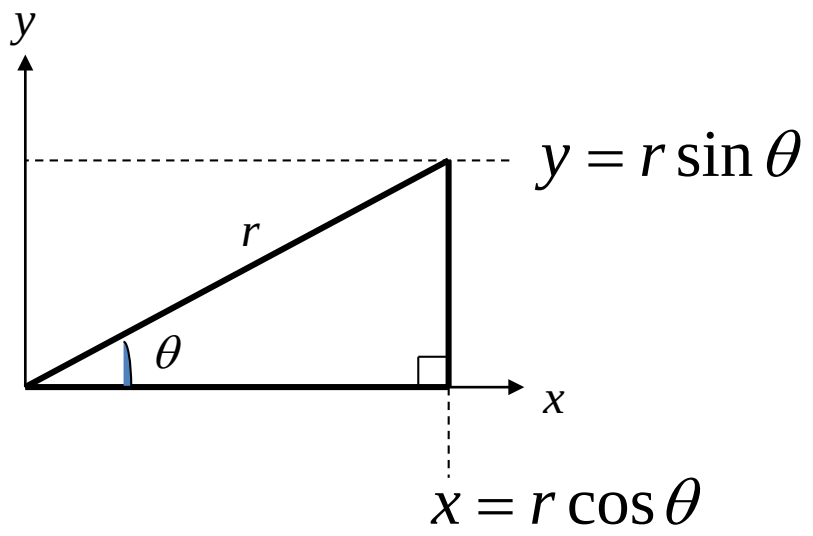
- Trigonometry
  - Right-angled triangle
    - Hypotenuse is the longest side



$$\sin\theta = \frac{\textbf{opposite}}{\textbf{hypotenuse}}$$

$$\cos\theta = \frac{\textbf{adjacent}}{\textbf{hypotenuse}}$$

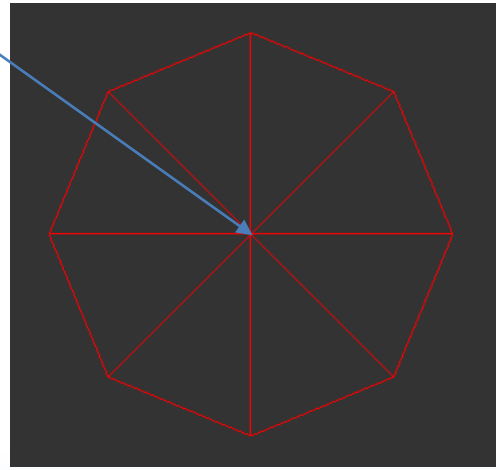
$$\tan\theta = \frac{\textbf{opposite}}{\textbf{adjacent}}$$



Draw Circle

- In init()

```
// initialise centre, i.e. (0.0f, 0.0f, 0.0f)
gVertices.clear();
gVertices.push_back(0.0f); // x
gVertices.push_back(0.0f); // y
gVertices.push_back(0.0f); // z
```

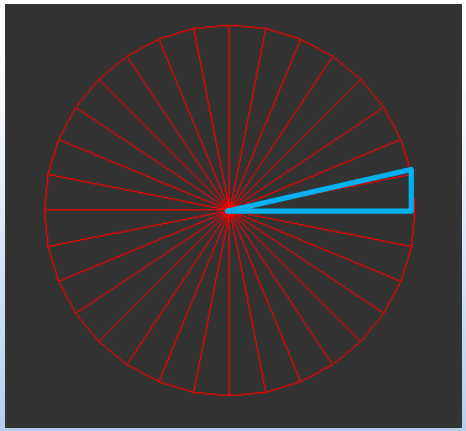


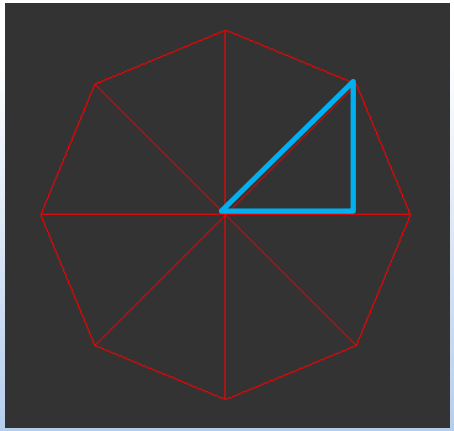
- In generate_circle()
  - $\pi = 180^{o}$ ($2 \times \pi = 360^{o}$)

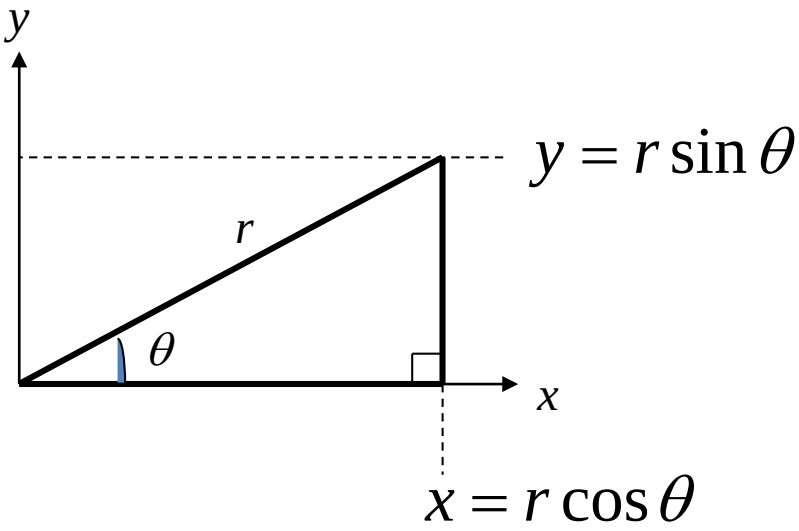
```
float slice_angle = M_PI * 2.0f / slices;
float angle = 0;
float x, y, z = 0.0f;
```

Draw Circle

```
// generate vertex coordinates for circle
for (int i = 0; i < slices + 1; i++)
{
	x = radius * cos(angle) * scale_factor;
	y = radius * sin(angle);

	vertices.push_back(x);
	vertices.push_back(y);
	vertices.push_back(z);

	angle += slice_angle;
}
```

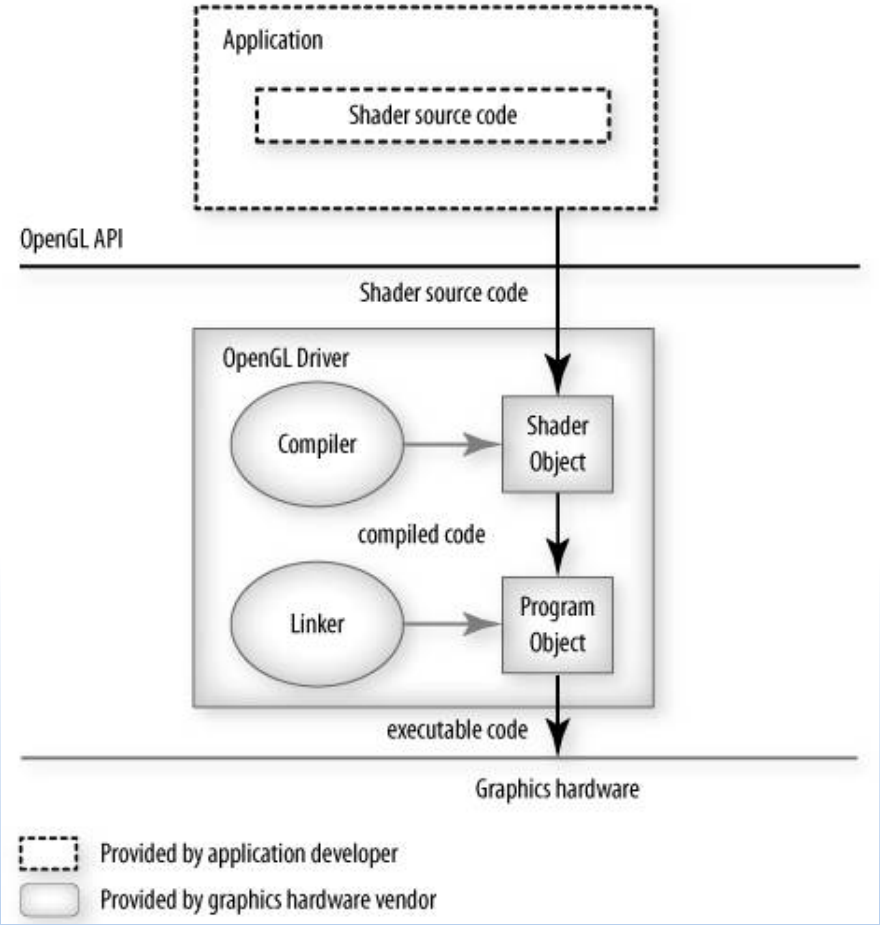






OpenGL Shaders

- Execution model for OpenGL shaders



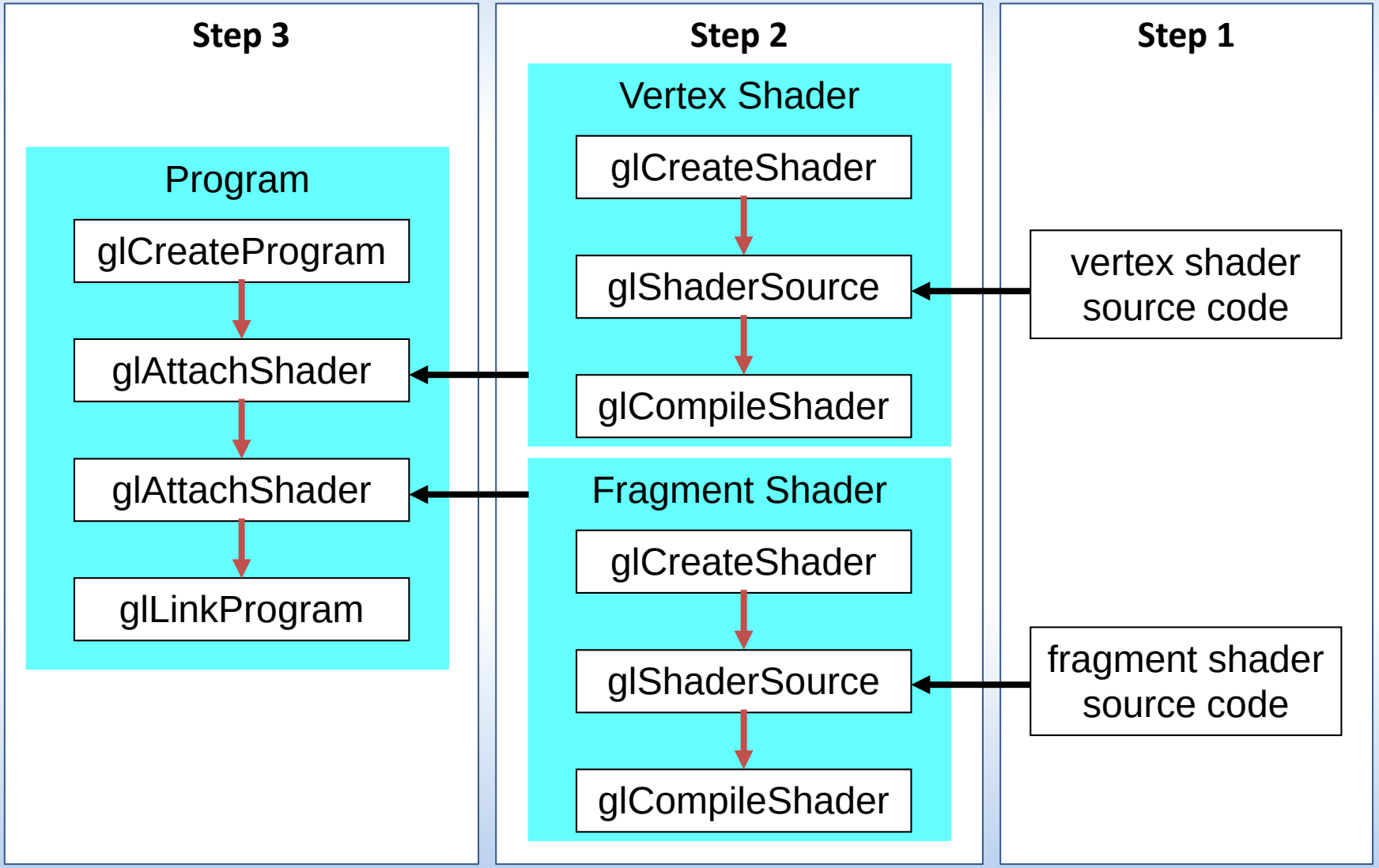
OpenGL Shaders

- Load shaders from plain-text file
  - Step 1: Read shader source code from file
- Compile shaders
  - Step 2: Create and compile shader objects
    - Create one or more (empty) shader objects using
      - `glCreateShader()`
    - Provide source code for the created shader objects using
      - `glShaderSource()`
    - Compile each shader using
      - `glCompileShader()`

OpenGL Shaders

- Link shaders into a program
  - Step 3: Attach shaders to a program object and link
    - Create a program object using
      - `glCreateProgram()`
    - Attach all the shader objects to the program object using
      - `glAttachShader()`
    - Link the program object using
      - `glLinkProgram()`
  - Using the shader program
    - Install executable shader program as part of OpenGL current state
      - `glUseProgram()`

OpenGL Shaders



OpenGL Shaders

- ShaderProgram class
  - Contains
    - Public member function to compile and link a pair of vertex and fragment shaders

```
void compileAndLink(const std::string vShaderFilename,
      const std::string fShaderFilename);
```

    - Shader program identifier stored in private member variable

```
GLuint mProgramID;
```

    - Public member function to use the compiled and linked shader program

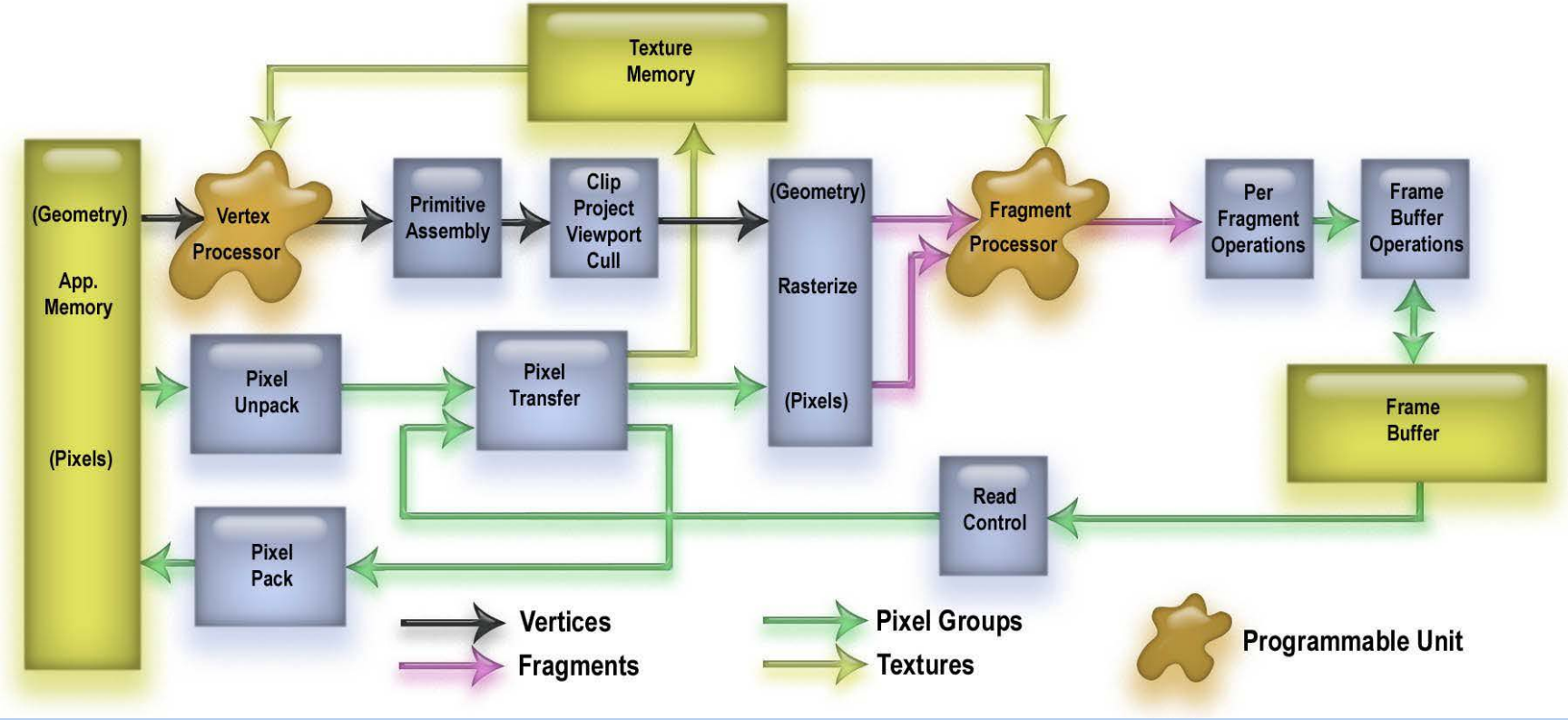
```
void use();
```

OpenGL Shaders

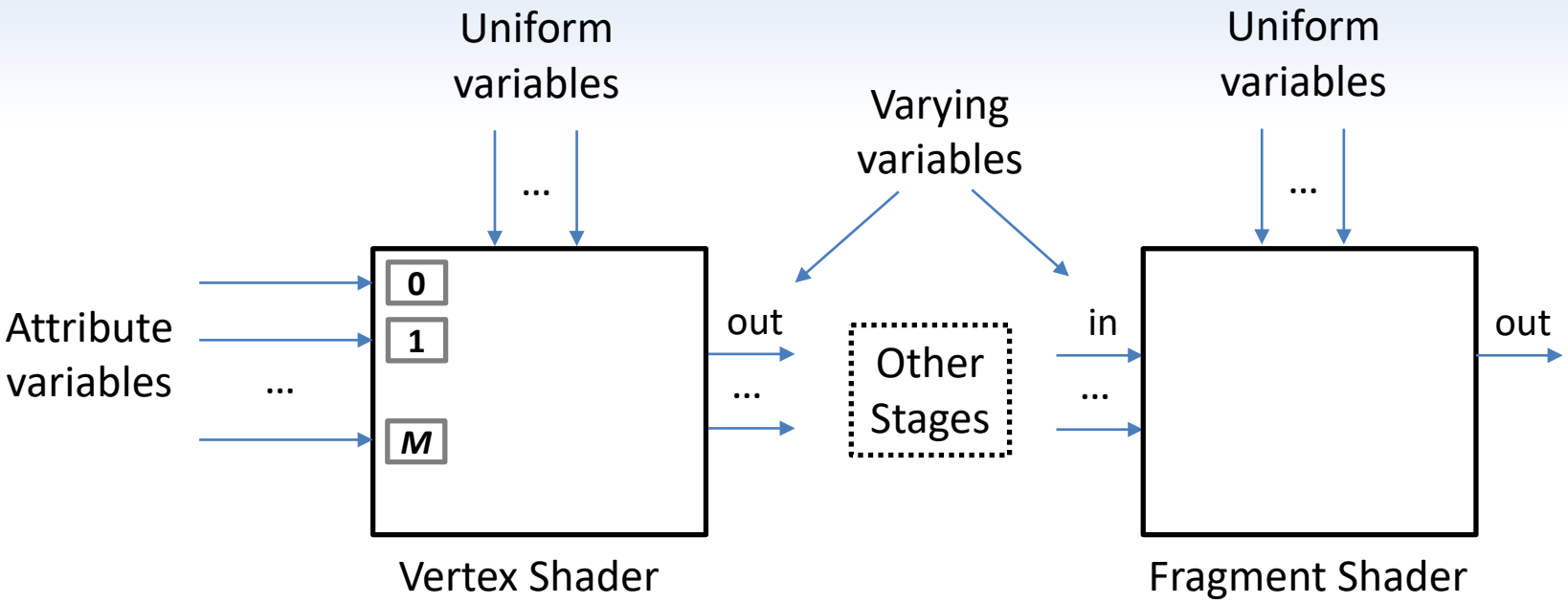
- ShaderProgram class
  - Contains multiple overloaded
    - `setUniform()` functions
      - Specifies the value of a shader's uniform variable
    - OpenGL provides a number of such functions of the form: `glUniform{dim}{type}{v}`
      - For example
        - `void glUniform1f(GLint location, GLfloat v0);`
        - `void glUniform3fv(GLint location, GLsizei count, const GLfloat *value);`
        - `void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat *value);`

OpenGL

- Rendering pipeline
  - OpenGL programmable pipeline



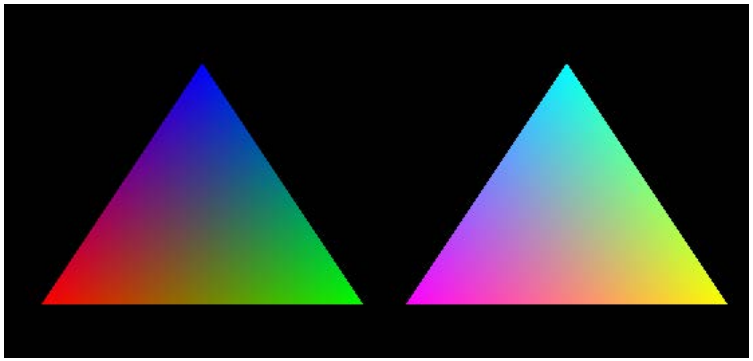
OpenGL Shaders



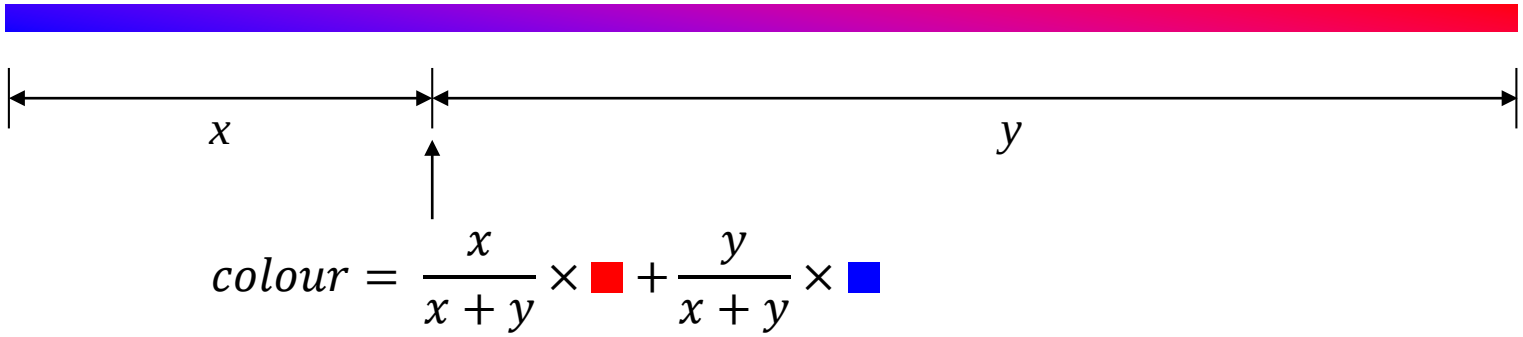
- Unless the **flat** keyword is used
  - Vertex shader outputs **interpolated** → fragment shader inputs
- Built-in variables, e.g.
  - **gl_Position** (must be assigned a value)
  - **gl_PointSize**

OpenGL Shaders

- Interpolation



  - Linear interpolation
    - Default (smooth) interpolates in perspective-correct fashion



Vertex Colours

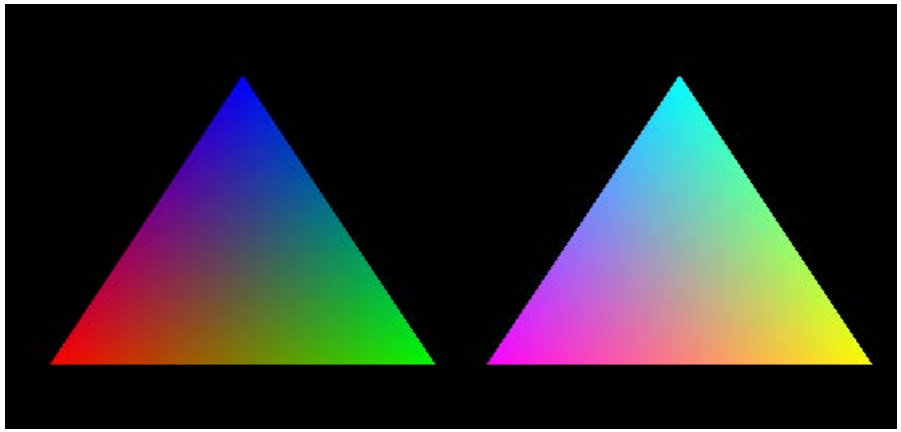
- In vertex shader

```
layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec3 aColor;

// output data
out vec3 vColor;

void main()
{
   gl_Position = vec4(aPosition, 1.0f);

   vColor = aColor;
}
```

Vertex Colours

- In fragment shader

```
in vec3 vColor;

// output data
out vec3 fColor;

void main()
{
    fColor = vColor;
}
```



Vertex Colours

- Using separate arrays
  - Array for positions

```
std::vector<GLfloat> vertexPositions = {
   -0.8f, -0.3f, 0.0f,     // vertex 0
   -0.2f, -0.3f, 0.0f,     // vertex 1
   ...
};
```

  - Array for colours

```
std::vector<GLfloat> vertexColors = {
   1.0f, 0.0f, 0.0f,       // vertex 0
   0.0f, 1.0f, 0.0f,       // vertex 1
   ...
};
```

Vertex Colours

- Separate VBOs

```
std::vector<GLuint> gVBO;
...
gVBO.resize(2, 0);
glGenBuffers(2, &gVBO[0]);

glBindBuffer(GL_ARRAY_BUFFER, gVBO[0]);
glBufferData(GL_ARRAY_BUFFER,
      sizeof(float) * vertexPositions.size(),
      &vertexPositions[0], GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, gVBO[1]);
glBufferData(GL_ARRAY_BUFFER,
      sizeof(float) * vertexColors.size(),
      &vertexColours[0], GL_STATIC_DRAW);
```

Vertex Colours

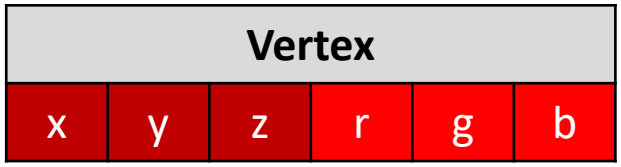
- Associate with VAO
  - Vertex attribute streams 0 and 1

```
glBindVertexArray(gVAO);

glBindBuffer(GL_ARRAY_BUFFER, gVBO[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, gVBO[1]);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
```

Interleaved Vertex Attributes

- Interleaved vertices

```
struct VertexColor
{
   GLfloat position[3];
   GLfloat color[3];
};
std::vector<GLfloat> vertices =
{
   0.0f, 0.5f, 0.0f,   // vertex 0: x, y, z
   1.0f, 0.0f, 0.0f,   // vertex 0: r, g, b
   0.5f, -0.5f, 0.0f,  // vertex 1: x, y, z
   0.0f, 1.0f, 0.0f,   // vertex 1: r, g, b
   ...
};
```



Interleaved Vertex Attributes

- VBO

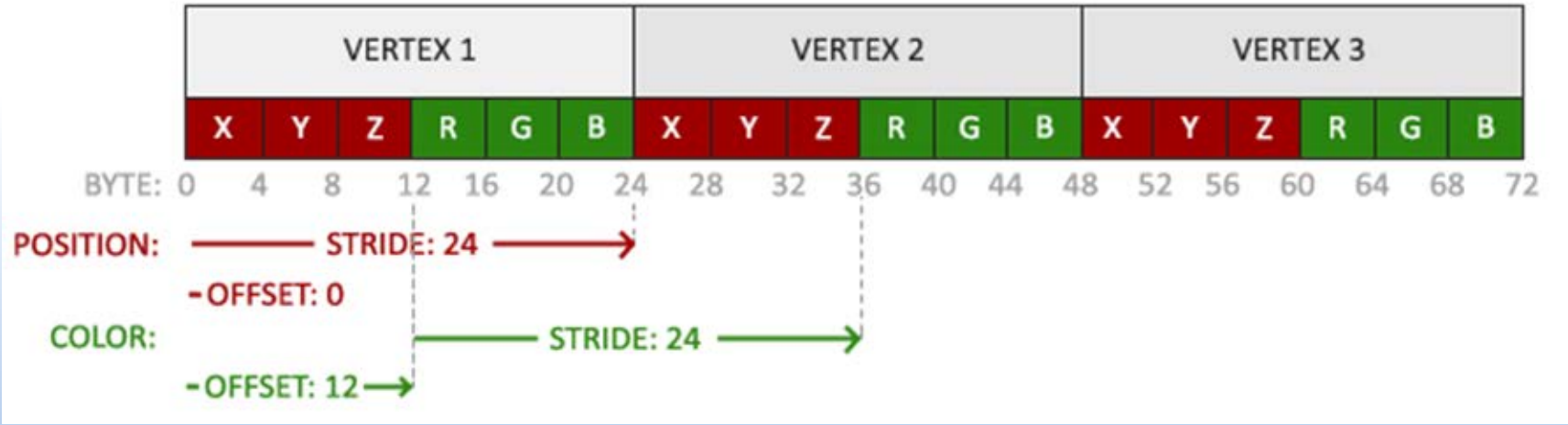
```
glGenBuffers(1, &g_VBO);
glBindBuffer(GL_ARRAY_BUFFER, g_VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(float) *
vertices.size(), &vertices[0], GL_STATIC_DRAW);
```

```
std::vector<GLfloat> vertices =
{
   0.0f, 0.5f, 0.0f,   // vertex 0: x, y, z
   1.0f, 0.0f, 0.0f,   // vertex 0: r, g, b
   0.5f, -0.5f, 0.0f,  // vertex 1: x, y, z
   0.0f, 1.0f, 0.0f,   // vertex 1: r, g, b
   ...
};
```

gVBO													
	0.0	0.5	0.0	1.0	0.0	0.0	0.5	-0.5	0.0	0.0	1.0	0.0	…	…

Interleaved Vertex Attributes

- VAO
  - ➢ Vertex attribute configuration
    - Stride - byte offset between consecutive vertex attributes
      - `sizeof(VertexColor)` or `sizeof(float)*6`
    - Offset - first component of the first vertex attribute
      - Position: `offsetof(VertexColor, position)` or `0`
      - Color: `offsetof(VertexColor, color)` or `sizeof(float)*3`



https://learnopengl.com/Getting-started/Shaders

Interleaved Vertex Attributes

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    sizeof(VertexColor),
    reinterpret_cast<void*>(offsetof(VertexColor, position)));
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    sizeof(VertexColor),
    reinterpret_cast<void*>(offsetof(VertexColor, color)));
```

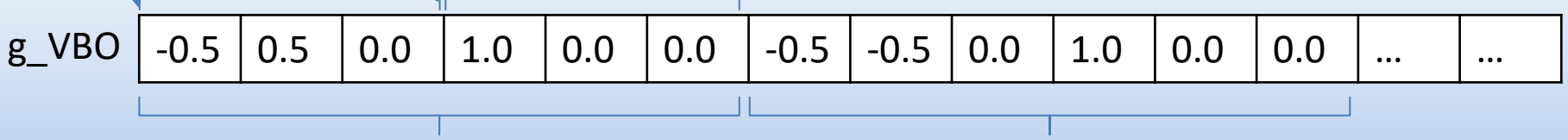
Offset

Stride

```
struct VertexColor {
    GLfloat position[3];
    GLfloat color[3];
};
```

Offset for position

Offset for colour



Stride = sizeof(VertexColor)

References

- Among others, material sourced from
  - Hearn, Baker & Carithers, “Computer Graphics with OpenGL”, Prentice-Hall
  - Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley
  - Bailey & Cunningham, “Graphics Shaders: Theory and Practice,” CRC Press
  - Randi Rost, “OpenGL Shading Language,” Addison-Wesley
  - Joey de Vries, “Learn OpenGL,” https://learnopengl.com/
  - https://www.khronos.org/opengl/wiki/
  - http://en.wikipedia.org/wiki/