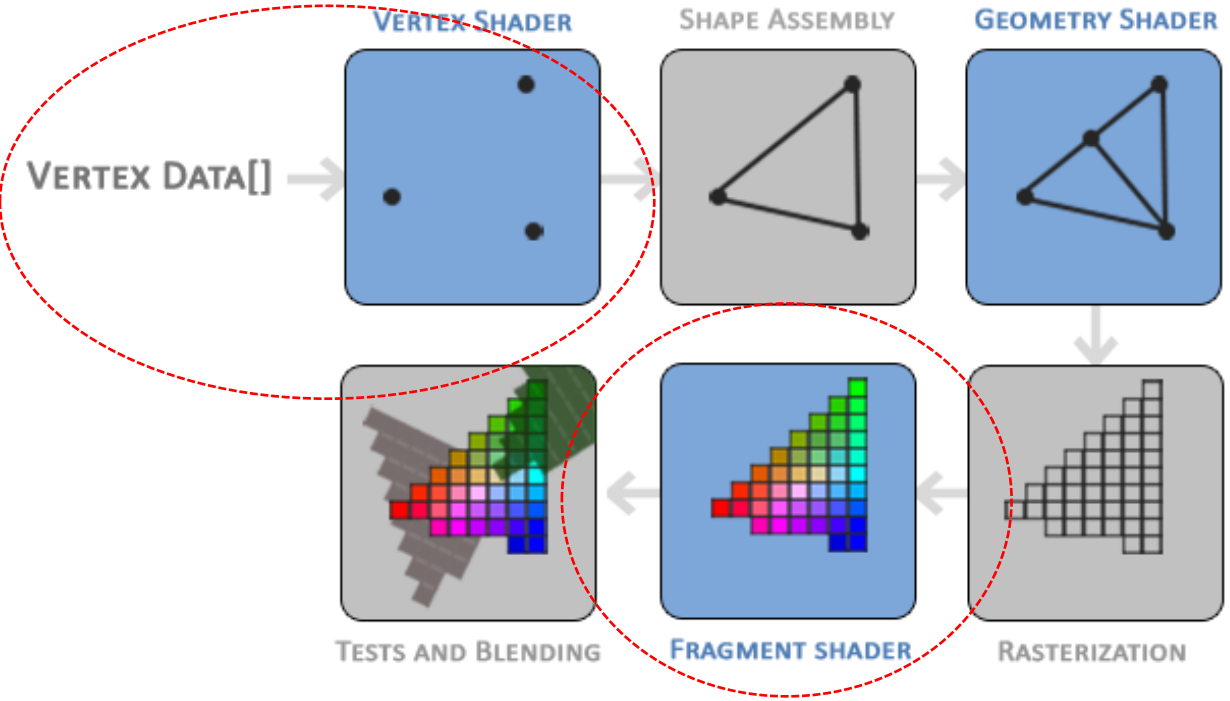
CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

2.1.3 OpenGL Pipeline – First Triangle

OpenGL – First Triangle

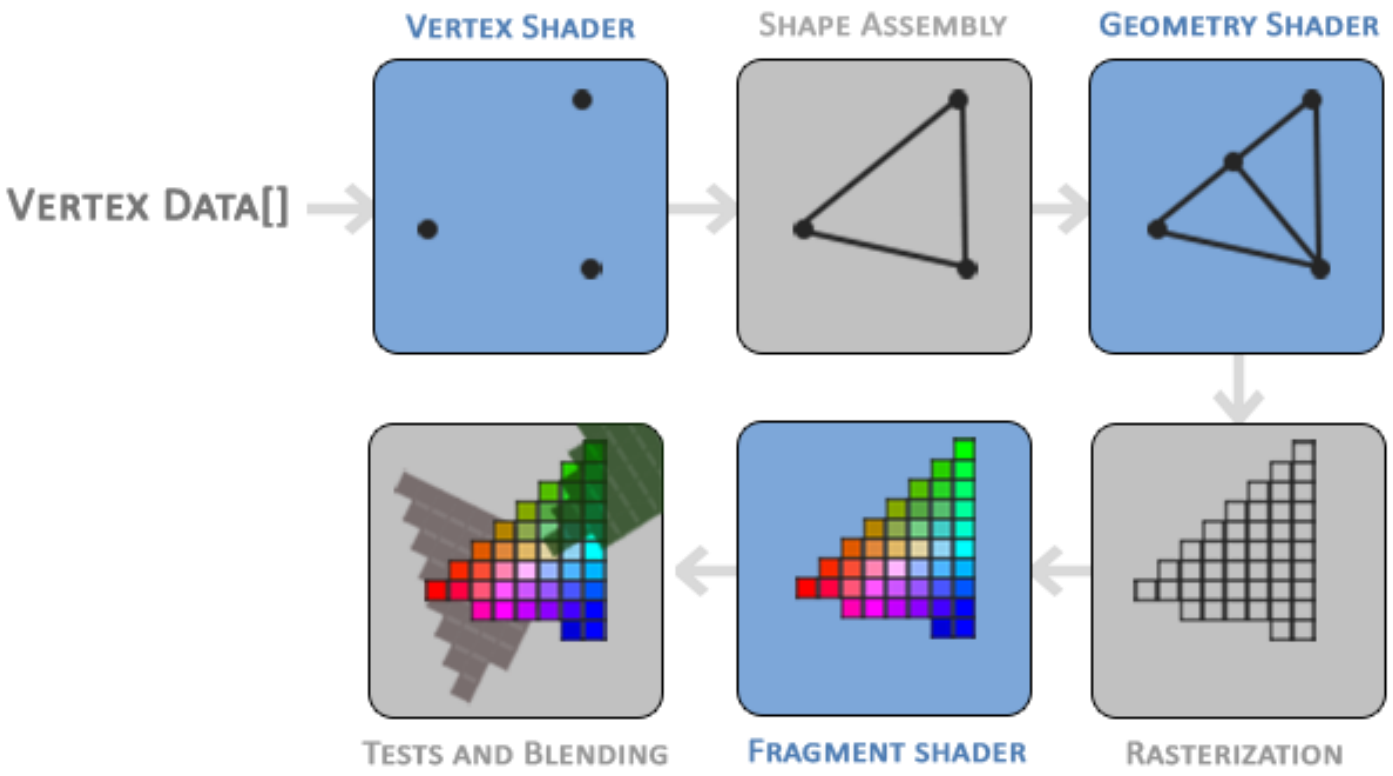


As you can see, the graphics pipeline is quite a complex whole and contains many configurable parts.

However, for almost all the cases we only have to work with the vertex and fragment shader.

The geometry shader is optional and usually left to its default shader. There is also the tesselation stage and transform feedback loop that we haven't depicted here, but that's something for later.

OpenGL – First Triangle

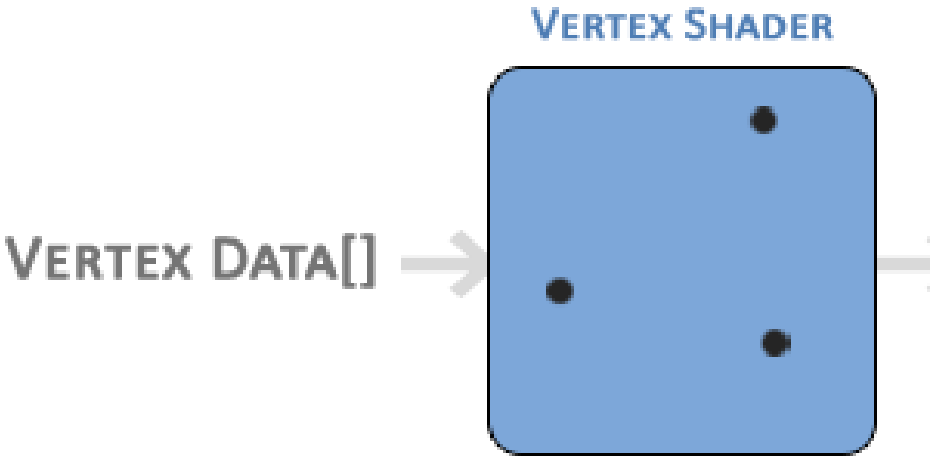


In modern OpenGL we are required to define at least a vertex and fragment shader of our own (there are no default vertex/fragment shaders on the GPU).

For this reason it is often quite difficult to start learning modern OpenGL since a great deal of knowledge is required before being able to render your first triangle. Once you do get to finally render your triangle at the end of this chapter you will end up knowing a lot more about graphics programming

OpenGL – First Triangle

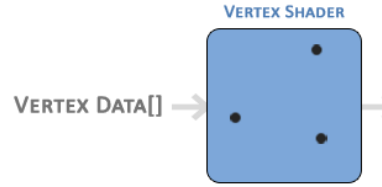
Vertex Data



OpenGL is a 3D graphics library so all coordinates that we specify in OpenGL are in 3D (x, y and z coordinate).

OpenGL only processes 3D coordinates when they're in a specific range between -1.0 and 1.0 on all 3 axes (x, y and z).

OpenGL – First Triangle



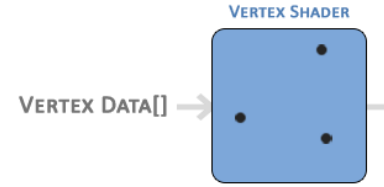
Vertex Data

We define them in normalized device coordinates (the visible region of OpenGL) in a float array:

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f, 0.5f, 0.0f
};
```

Because OpenGL works in 3D space we render a 2D triangle with each vertex having a z coordinate of 0.0. This way the depth of the triangle remains the same making it look like it's 2D.

OpenGL – First Triangle

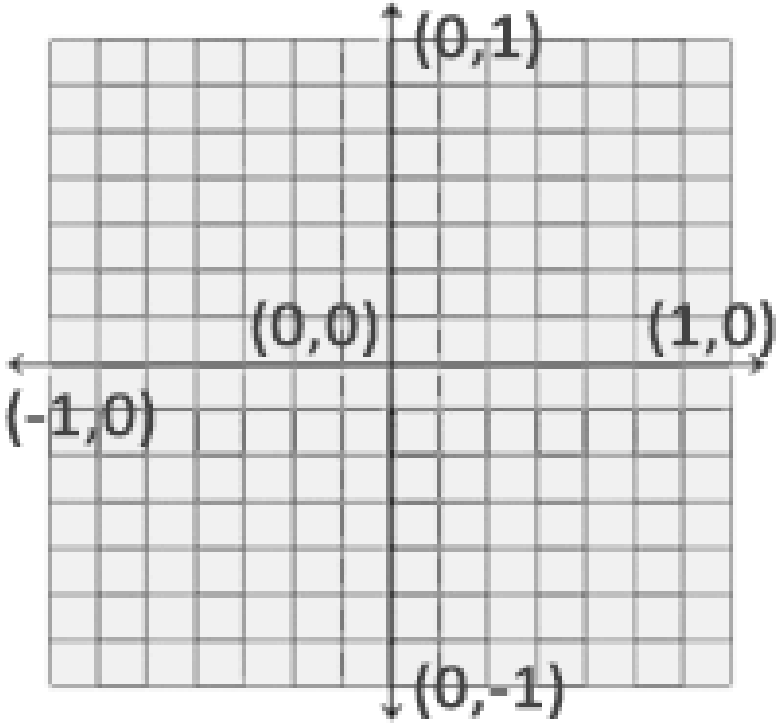


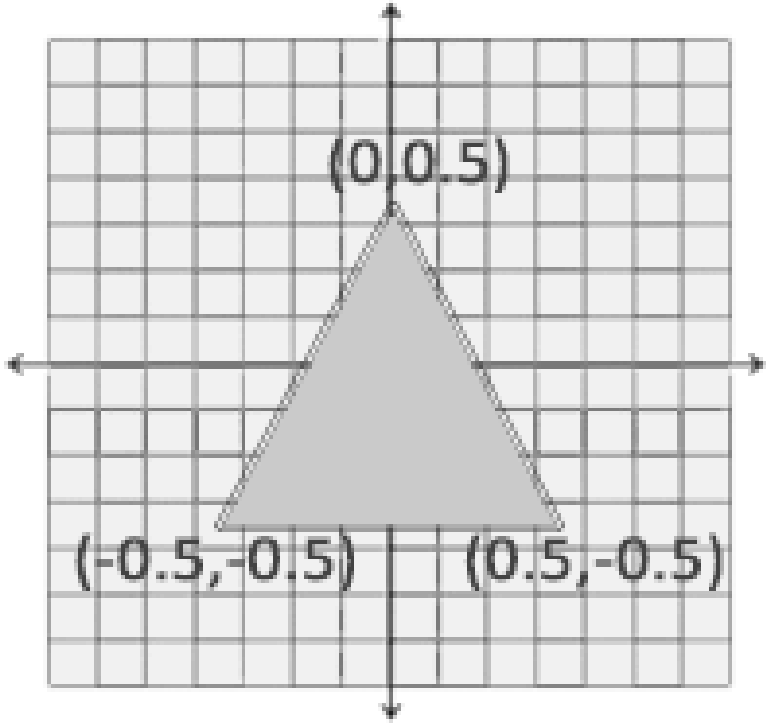
Vertex Shader

Once your vertex coordinates have been processed in the vertex shader, they should be in Normalized Device Coordinates (NDC) which is a small space where the x, y and z values vary from -1.0 to 1.0.

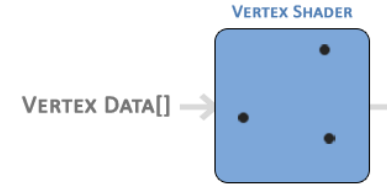
Any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen.

the triangle we specified within normalized device coordinates (ignoring the z axis): →

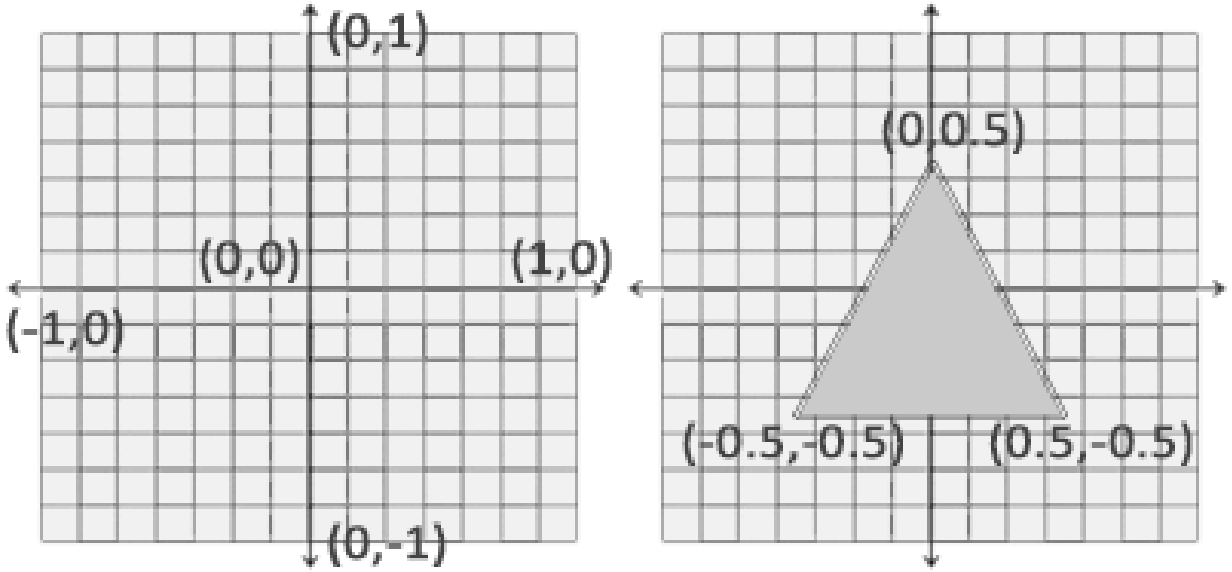




OpenGL – First Triangle



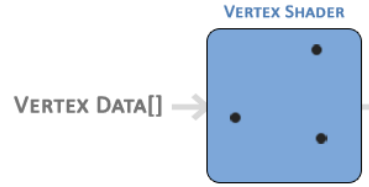
Vertex Shader



Your NDC coordinates will then be transformed to screen-space coordinates via the viewport transform using the data you provided with glViewport. The resulting screen-space coordinates are then transformed to fragments as inputs to your fragment shader.

OpenGL – First Triangle

Vertex Shader



OpenGL Context

- Active VBO (GL_ARRAY_BUFFER)
- Active IBO (GL_ARRAY_ELEMENT_BUFFER)
- Active Shader Program

CPU

modelMat	...	...	...	...	...	...	...	...	...
vPos	0.0	0.0	0.0	...	...	...	1.0	1.0	0.0
vCol	1.0	0.0	0.0	1.0	...	1.0	0.0	0.0	1.0
vIndices	0	1	2	1	3	2			
viewMat	...	...	...	...	...	...	...	...	...
projMat	...	...	...	...	...	...	...	...	...

GPU

NULL Object : 0

Shader Program Objects

Name
Vertex Shader
Fragment Shader

Buffer Objects

Name	Data								
1	0.0	0.0	0.0	...	...	...	1.0	1.0	0.0

Shader Objects

Name 1
Type GL_VERTEX_SHADER

Attributes

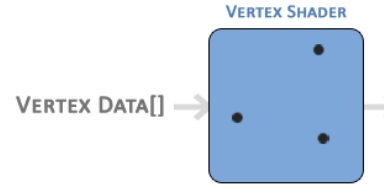
Name	Location	Type
vPos	0	vec3
vCol	1	vec4

Uniforms

Name	Location	Type
modelMat	0	mat4
viewMat	1	mat4
projMat	2	mat4

Name 2
Type GL_FRAGMENT_SHADER

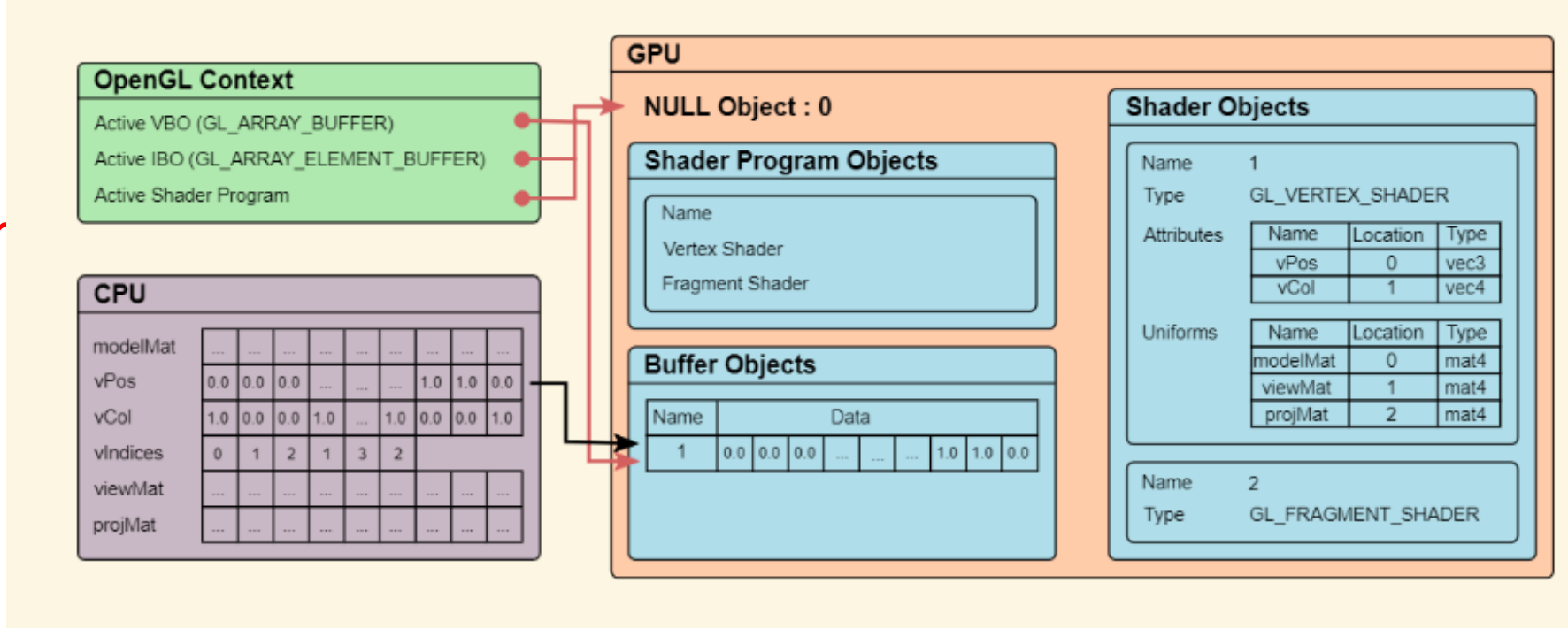
OpenGL – First Triangle



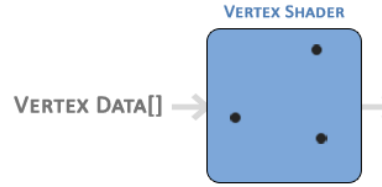
Vertex Shader

Vertex Shader creates memory on the GPU where we store the vertex data, configure how OpenGL should interpret the memory and specify how to send the data to the graphics card. The vertex shader then processes as much vertices as we tell it to from its memory

We manage this memory via so called vertex buffer objects (VBO) that can store a large number of vertices in the GPU's memory.



OpenGL – First Triangle



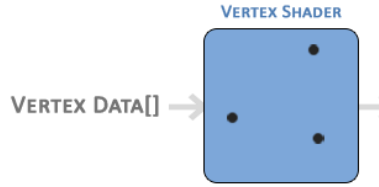
Vertex Shader

The advantage of using those buffer objects is that we can send large batches of data all at once to the graphics card, and keep it there if there's enough memory left, without having to send data one vertex at a time. Sending data to the graphics card from the CPU is relatively slow

Just like any object in OpenGL, this buffer has a unique ID corresponding to that buffer, so we can generate one with a buffer ID using the glGenBuffers function

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

OpenGL – First Triangle



Vertex Shader

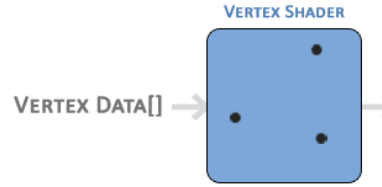
OpenGL has many types of buffer objects and the buffer type of a vertex buffer object is GL_ARRAY_BUFFER. We can bind the newly created buffer to the GL_ARRAY_BUFFER target with the glBindBuffer function:

```
glBindBuffer(GL_ARRAY_BUFFER, &VBO);
```

Then we can make a call to the glBufferData function that copies the previously defined vertex data into the buffer's memory

```
glBufferData(GL_ARRAY_BUFFER,
sizeof(vertices), vertices, GL_STATIC_DRAW);
```

OpenGL – First Triangle



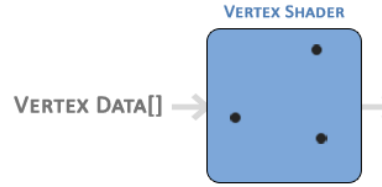
Vertex Shader

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

glBufferData is a function specifically targeted to copy user-defined data into the currently bound buffer.

- type of the buffer → GL_ARRAY_BUFFER.
- size of the data (in bytes)
- actual data we want to send → vertices
- The graphics card to manage the given data. This can take 3 forms:
  - GL_STREAM_DRAW: the data is set only once and used by the GPU at most a few times.
  - GL_STATIC_DRAW: the data is set only once and used many times.
  - GL_DYNAMIC_DRAW: the data is changed a lot and used many times.

OpenGL – First Triangle



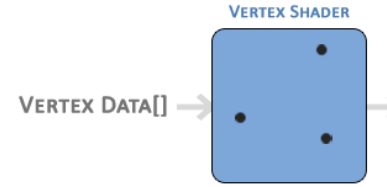
Vertex Shader

The first thing we need to do is write the vertex shader in the shader language GLSL (OpenGL Shading Language) and then compile this shader so we can use it in our application.

A very basic vertex shader in GLSL:

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

OpenGL – First Triangle



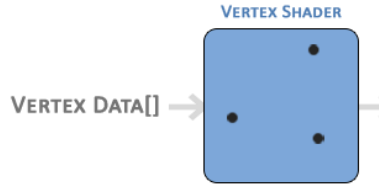
Vertex Shader

GLSL has a vector datatype that contains 1 to 4 floats based on its postfix digit. Since each vertex has a 3D coordinate we create a vec3 input variable with the name aPos.

We also specifically set the location of the input variable via layout (location = 0) and you'll later see that why we're going to need that location.

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

OpenGL – First Triangle



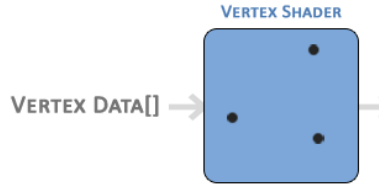
Vertex Shader

```
gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
```

Vector

- In graphics programming we use the mathematical concept of a vector quite often, since it neatly represents positions/directions in any space and has useful mathematical properties.
- A vector in GLSL has a maximum size of 4 and each of its values can be retrieved via vec.x, vec.y, vec.z and vec.w respectively where each of them represents a coordinate in space.
- Note that the vec.w component is not used as a position in space (we're dealing with 3D, not 4D) but is used for something called perspective division (later).

OpenGL – First Triangle

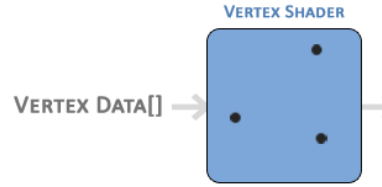


Vertex Shader – Compiling a shader

We take the source code for the vertex shader and store it in a const C string at the top of the code file for now:

```
const char *vertexShaderSource = "#version
330 core\n"
   "layout (location = 0) in vec3 aPos;\n"
   "void main()\n"
   "{\n"
   "   gl_Position = vec4(aPos.x, aPos.y, aPos.z,
1.0);\n"
   "}\0";
```

OpenGL – First Triangle



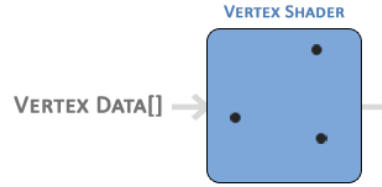
Vertex Shader – Compiling a shader

In order for OpenGL to use the shader it has to dynamically compile it at run-time from its source code.

The first thing we need to do is create a **shader object**, again referenced by an ID. So we store the vertex shader as an unsigned int and create the shader with glCreateShader:

```
unsigned int vertexShader;
vertexShader = glCreateShader (GL_VERTEX_SHADER);
```

OpenGL – First Triangle

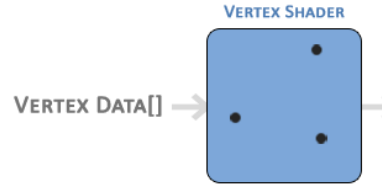


Vertex Shader – Compiling a shader

Next we attach the shader source code to the shader object and compile the shader:

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

OpenGL – First Triangle

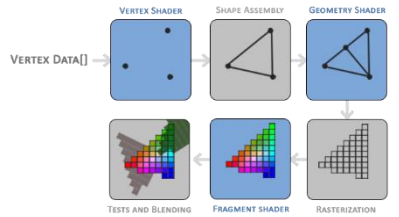


Vertex Shader – Compiling a shader

You probably want to check if compilation was successful after the call to glCompileShader and if not, what errors were found so you can fix those. Checking for compile-time errors is accomplished as follows:

```
int  success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);

if(!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

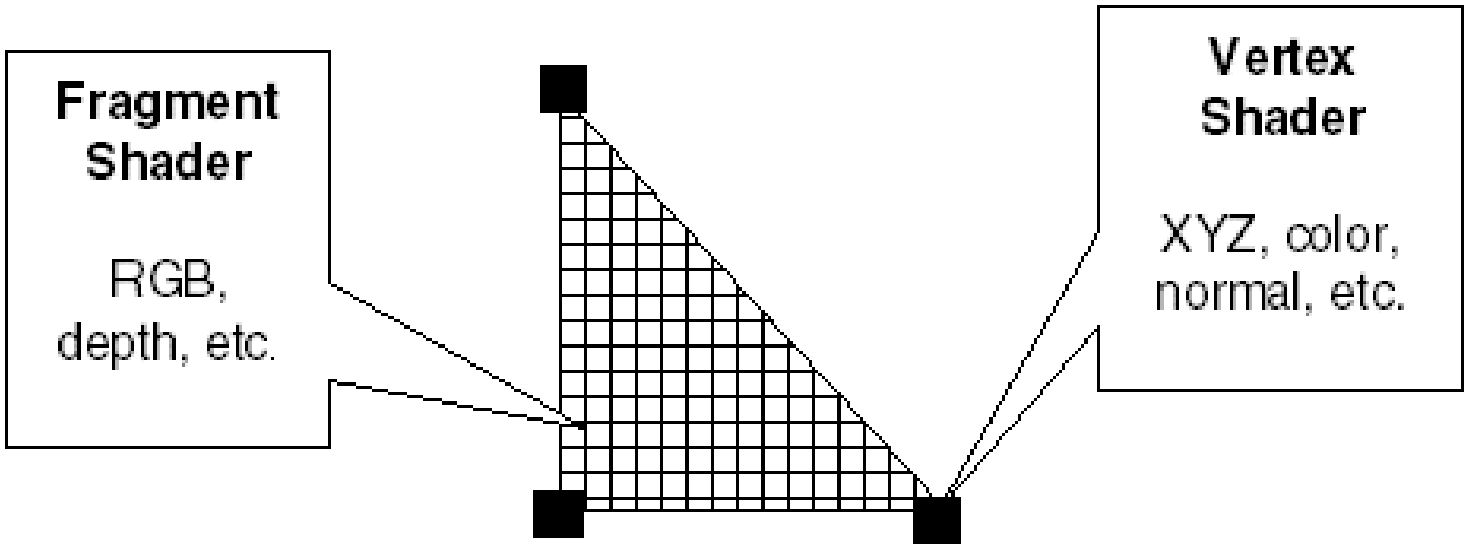
OpenGL – First Triangle



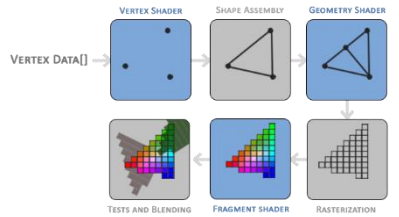
Fragment Shader

The fragment shader is the second and final shader we're going to create for rendering a triangle. The fragment shader is all about calculating the color output of your pixels.

To keep things simple the fragment shader will always output an orange-ish color



OpenGL – First Triangle

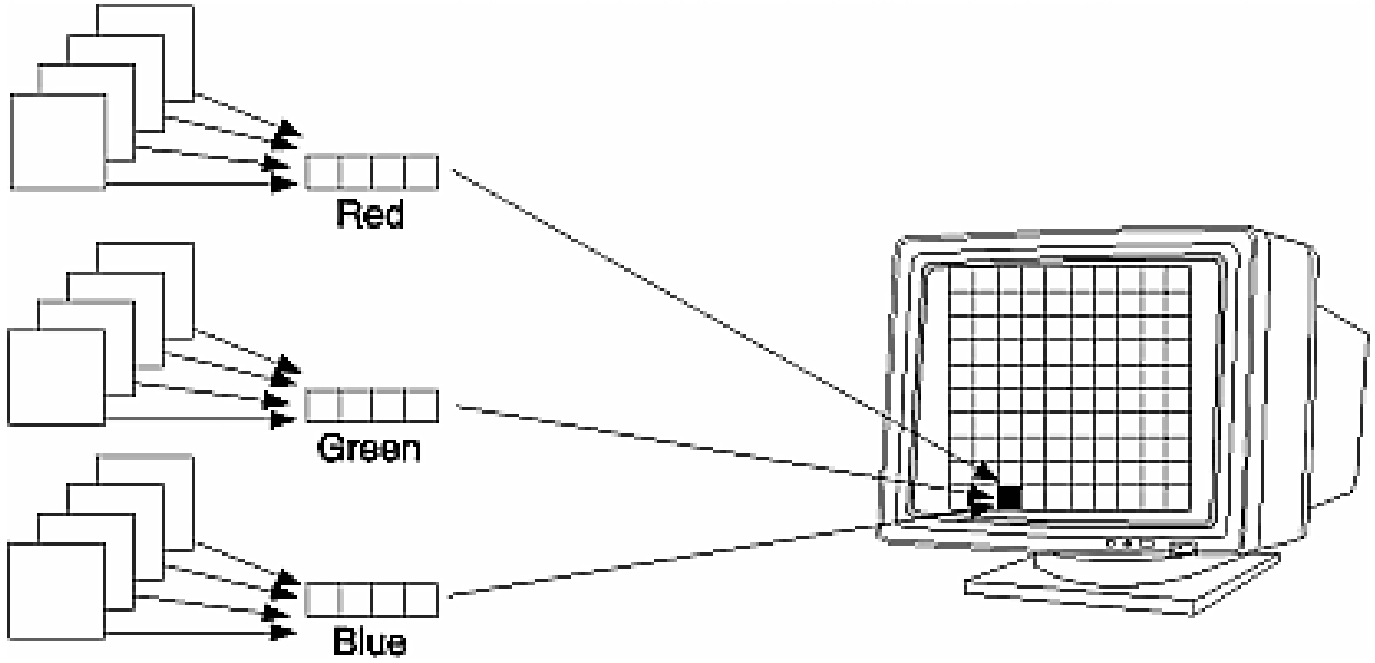


Fragment Shader - Colors

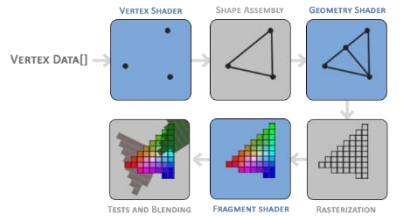
Colors in computer graphics are represented as an array of 4 values: the red, green, blue and alpha (opacity) component, commonly abbreviated to RGBA.

When defining a color in OpenGL or GLSL we set the strength of each component to a value between 0.0 and 1.0.

Given those 3 color components we can generate over 16 million differen colors!



OpenGL – First Triangle



Fragment Shader - Colors

How Many Colors?

Color number =2color_depth

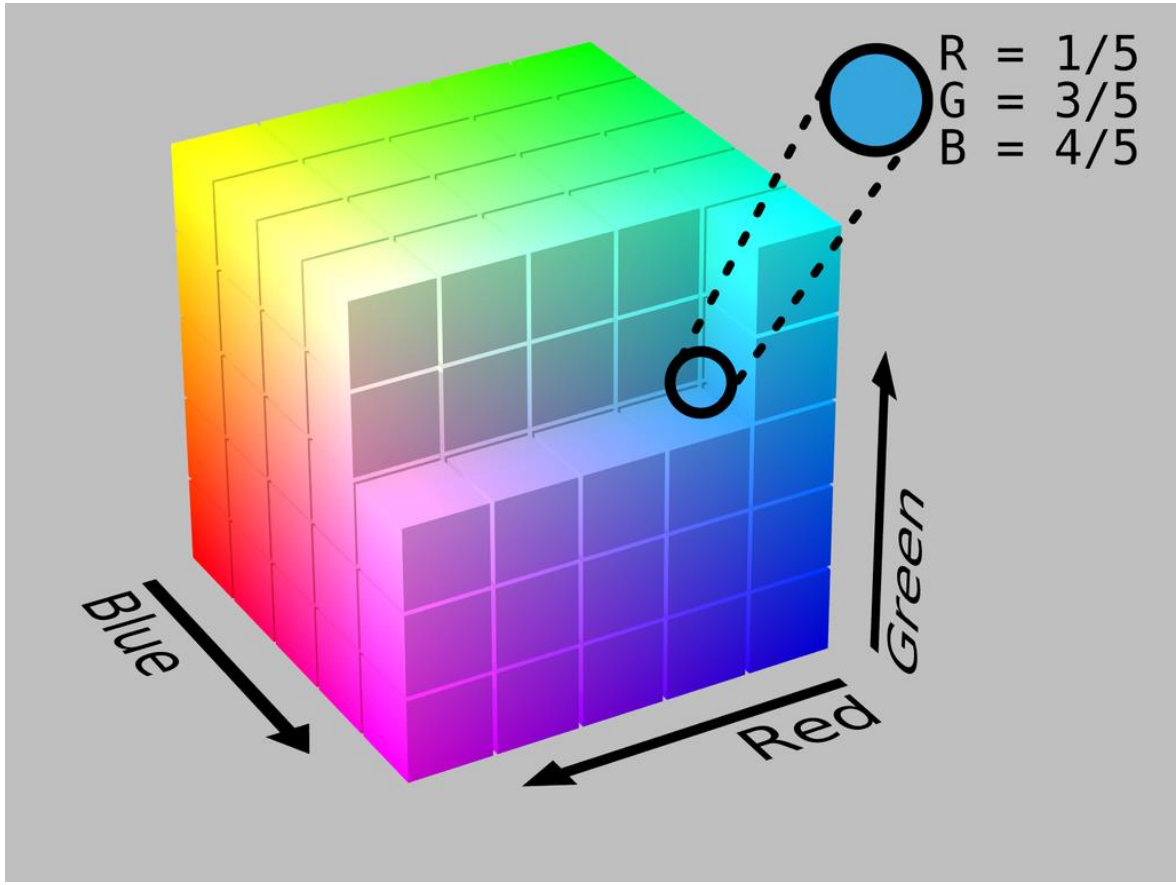
For example:

- 4-bit color

  2^4 = 16 colors
- 8-bit color

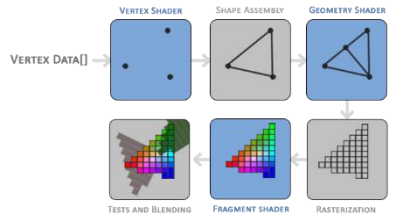
  2^8 = 256 colors
- 24-bit color

  2^24= 16.77 million colors



https://rgbcolorcode.com/color/aero

OpenGL – First Triangle



Fragment Shader - Colors

How Much Memory?

Buffer size = width * height *color depth

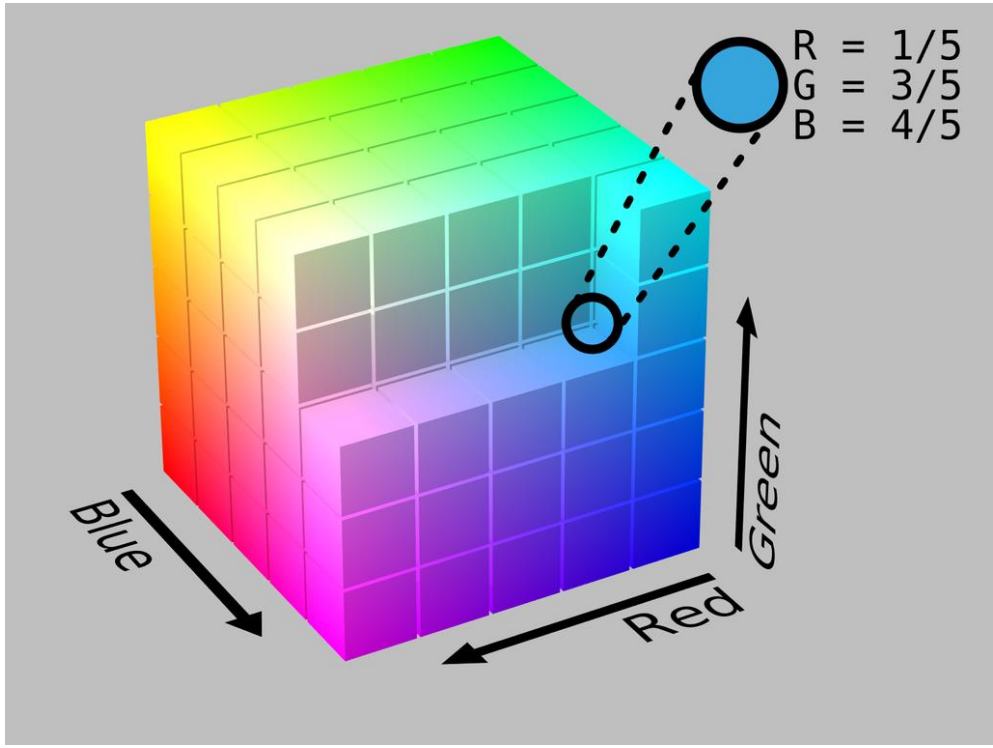
For example:

If width = 640, height = 480, color depth = 24 bits

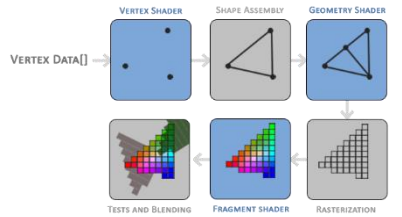
Buffer size = 640 * 480 * 24 = 921,600 bytes

If width = 640, height = 480, color depth = 32 bits

Buffer size = 640 * 480 * 32 = 1,228,800 bytes



https://rgbcolorcode.com/color/aero



Fragment Shader – Colors- Alpha Component

Alpha value

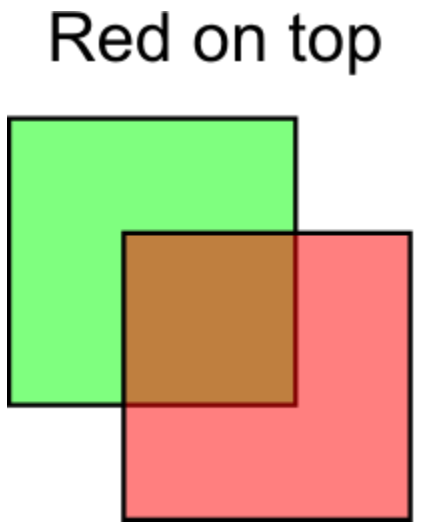
A value indicating the pixels opacity

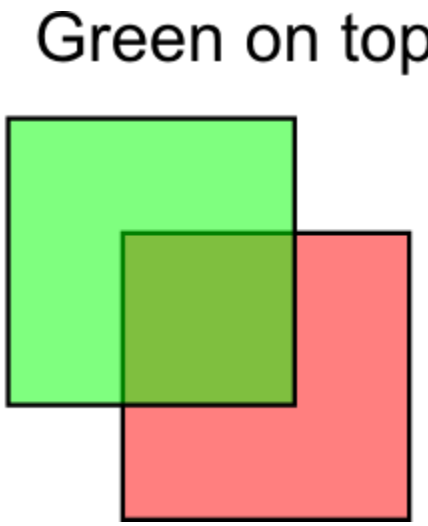
Zero usually represents totally transparent and the maximum value represents completely opaque

Alpha buffer

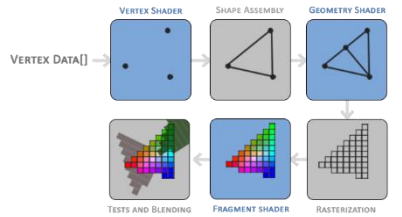
Hold the alpha value for every pixel

Alpha values are commonly represented in 8 bits, in which case transparent to opaque ranges from 0 to 255





OpenGL – First Triangle



Fragment Shader – Colors

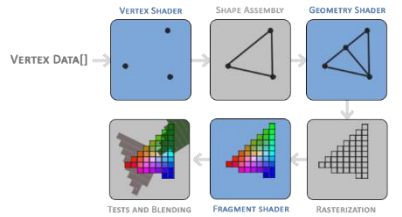
The fragment shader only requires one output variable and that is a vector of size 4 that defines the final color output that we should calculate ourselves.

We can declare output values with the out keyword, that we here promptly named FragColor.

Next we simply assign a vec4 to the color output as an orange color with an alpha value of 1.0 (1.0 being completely opaque).

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

OpenGL – First Triangle



Fragment Shader

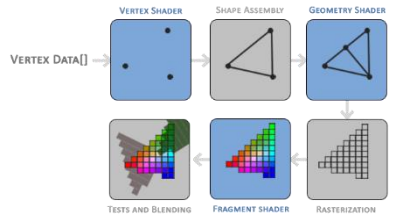
```
const char* fragmentShaderSource =
"#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f,
1.0f);\n"
"}\n\0";
```

The process for compiling a fragment shader is similar to the vertex shader, although this time we use the GL_FRAGMENT_SHADER constant as the shader type:

```
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

Make sure to check for compile errors here as well!

OpenGL – First Triangle

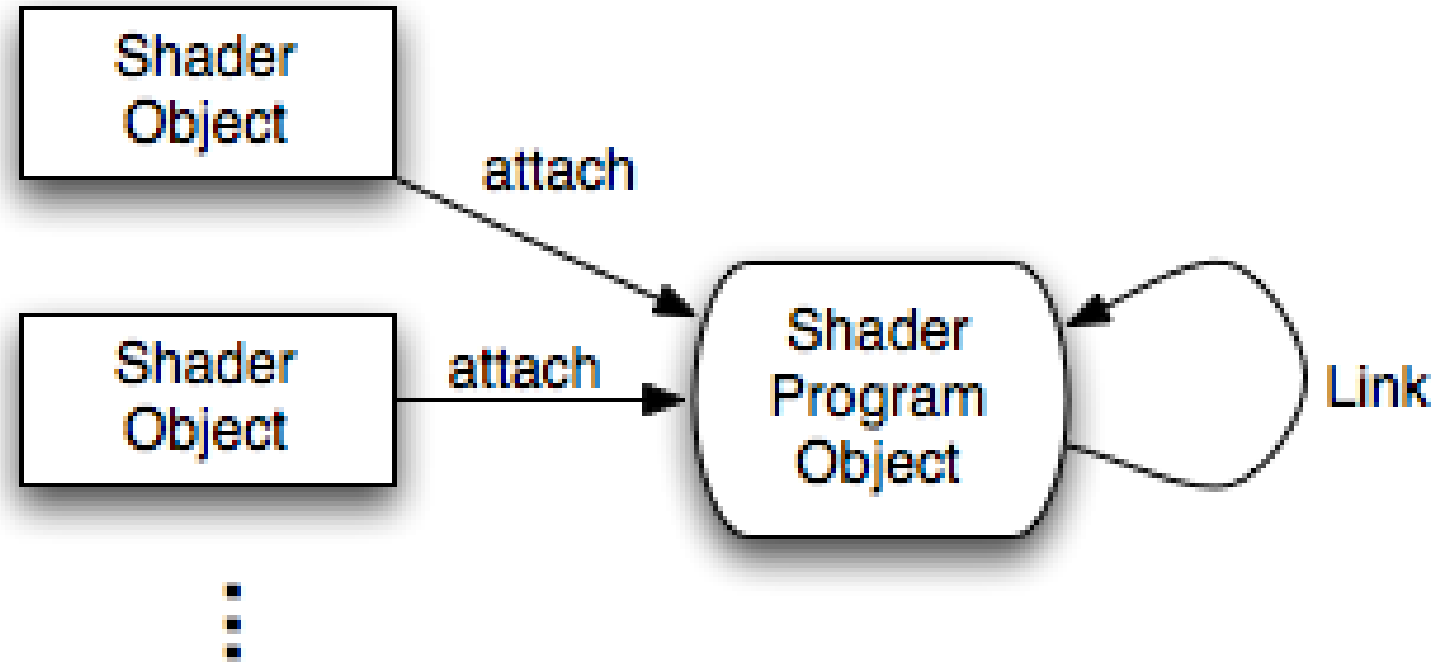


Shader Program

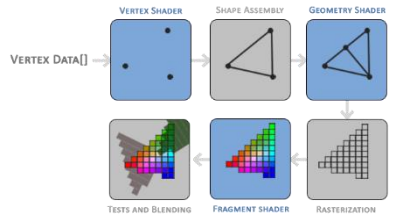
A shader program object is the final linked version of multiple shaders combined.

To use the recently compiled shaders we have to link them to a shader program object and then activate this shader program when rendering objects.

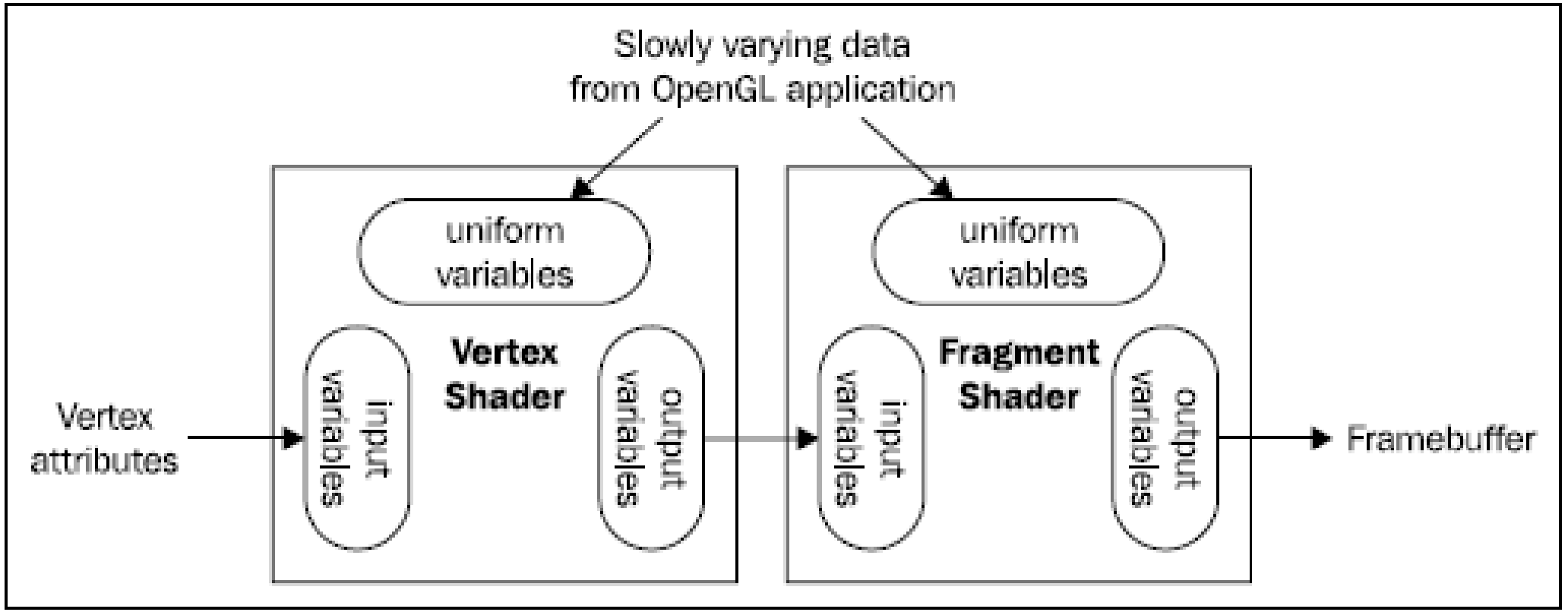
The activated shader program's shade will be used when we issue render call



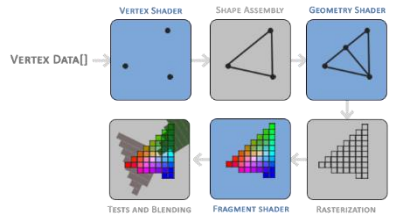
OpenGL – First Triangle



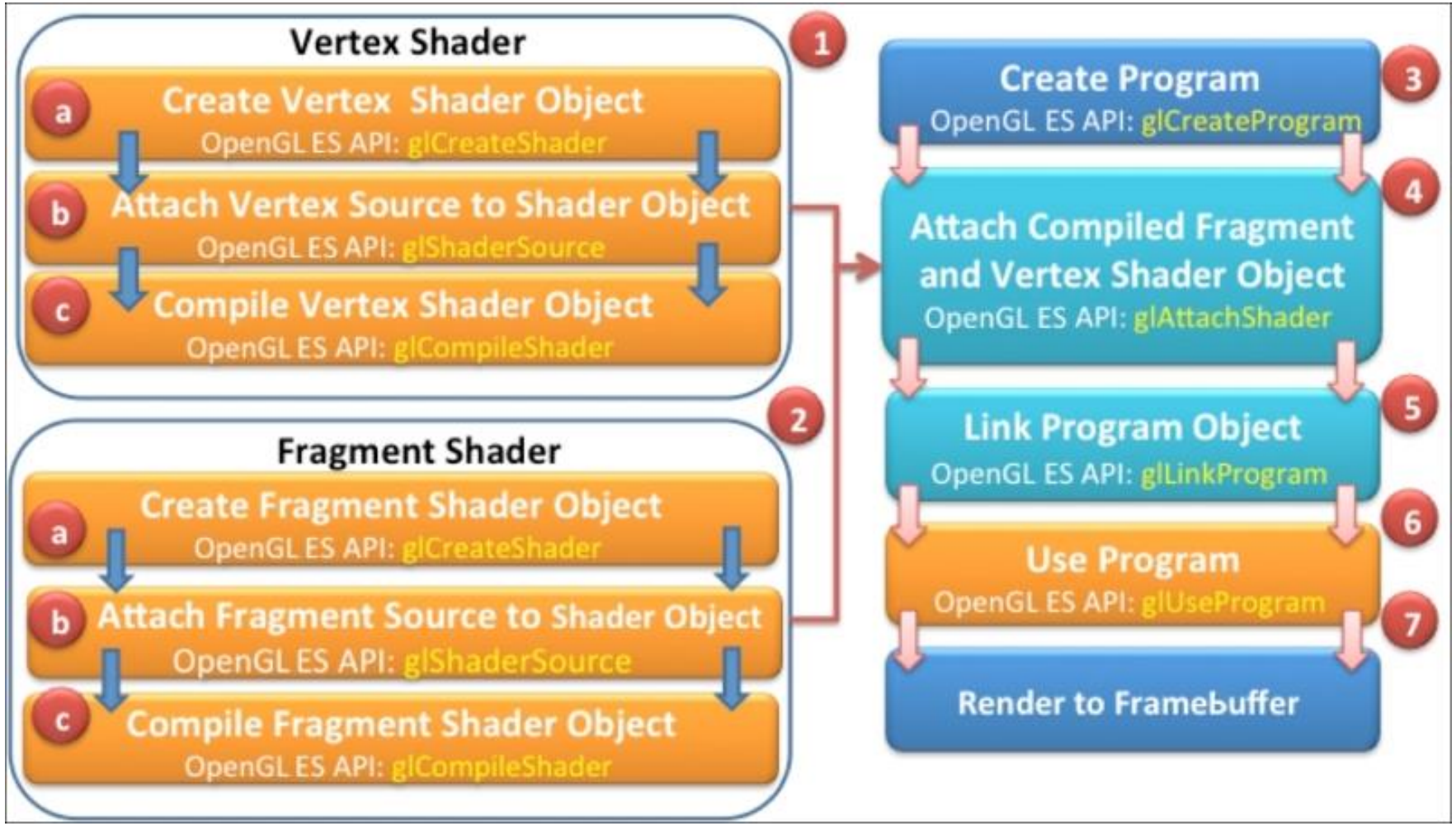
Shader Program



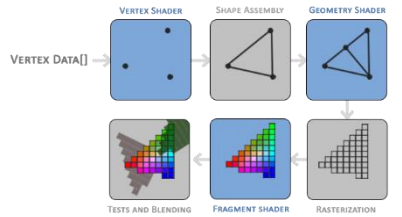
OpenGL – First Triangle



Shader Program



OpenGL – First Triangle



Shader Program Linking

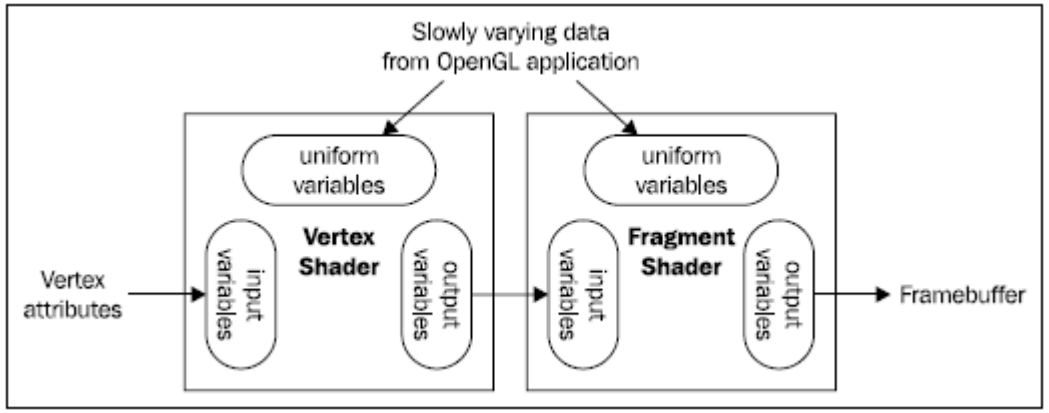
Creating a program object is easy:

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

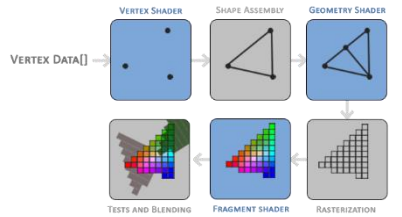
The glCreateProgram function creates a program and returns the ID reference to the newly created program object.

Now we need to attach the previously compiled shaders to the **program object** and then link them with glLinkProgram:

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```



OpenGL – First Triangle

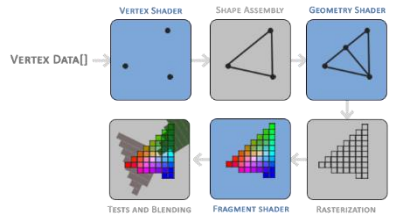


Shader Program Linking

Just like shader compilation we can also check if linking a shader program failed and retrieve the corresponding log.

```
// check for linking errors
   glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
   if (!success) {
      glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
      std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
   }
```

OpenGL – First Triangle



Shader Program

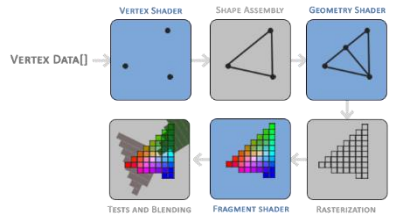
The result is a program object that we can activate by calling glUseProgram with the newly created program object as its argument:

```
glUseProgram(shaderProgram);
```

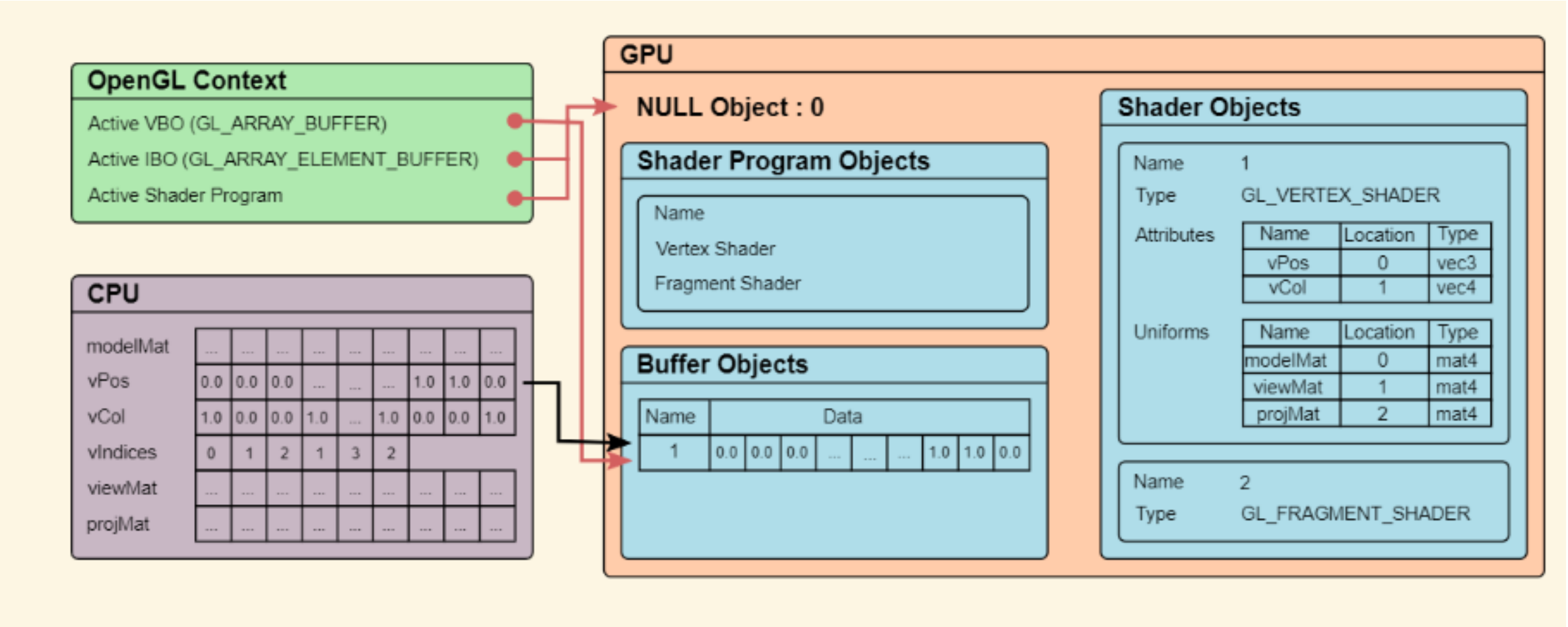
Delete the shader objects once we've linked them into the program object; we no longer need them anymore:

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

OpenGL – First Triangle

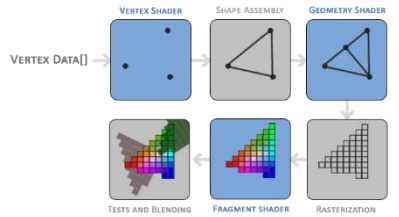


Linking Vertex Atributes



Right now we sent the input vertex data to the GPU and instructed the GPU how it should process the vertex data within a vertex and fragment shader. We're almost there, but not quite yet.

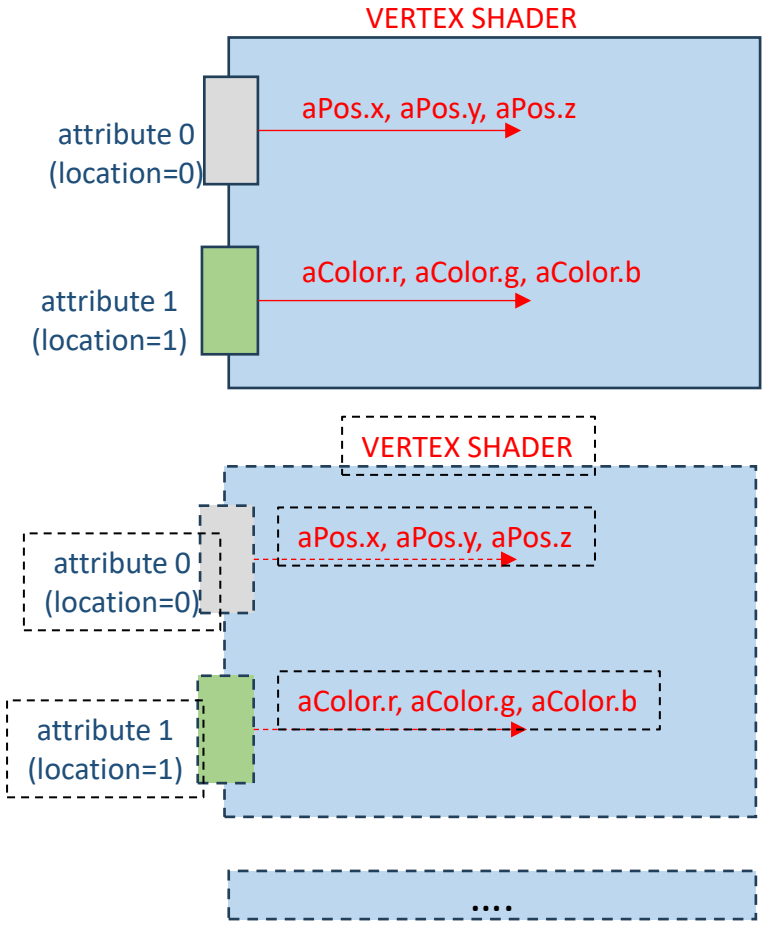
OpenGL – First Triangle



Linking Vertex Atributes

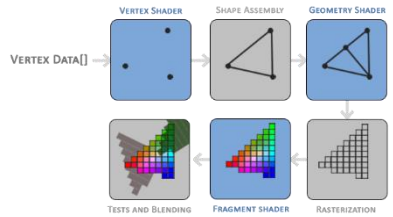
VERTEX BUFFER OBJECT (VBO)

(x1,y1,z1,r1,g1,b1,x2,y2,z2,r2,g2,b2…)

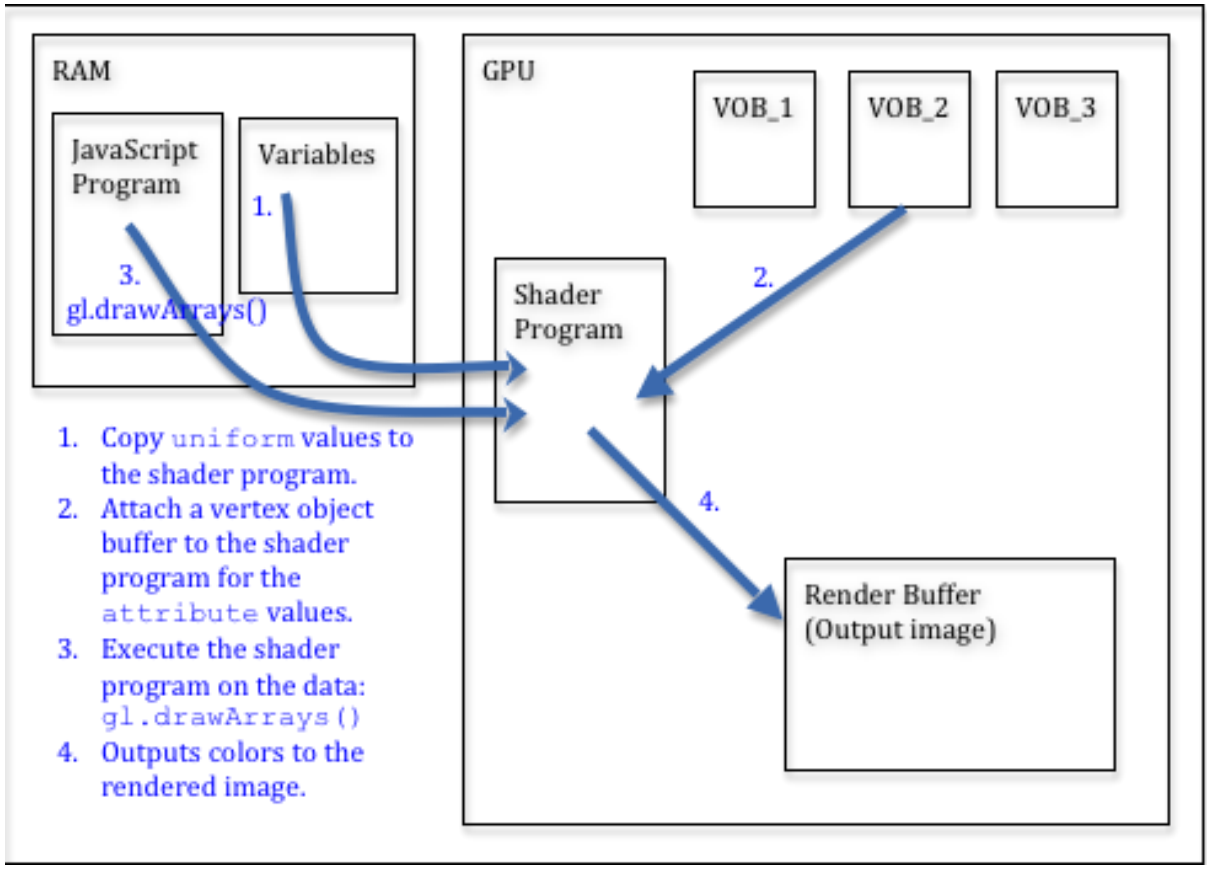


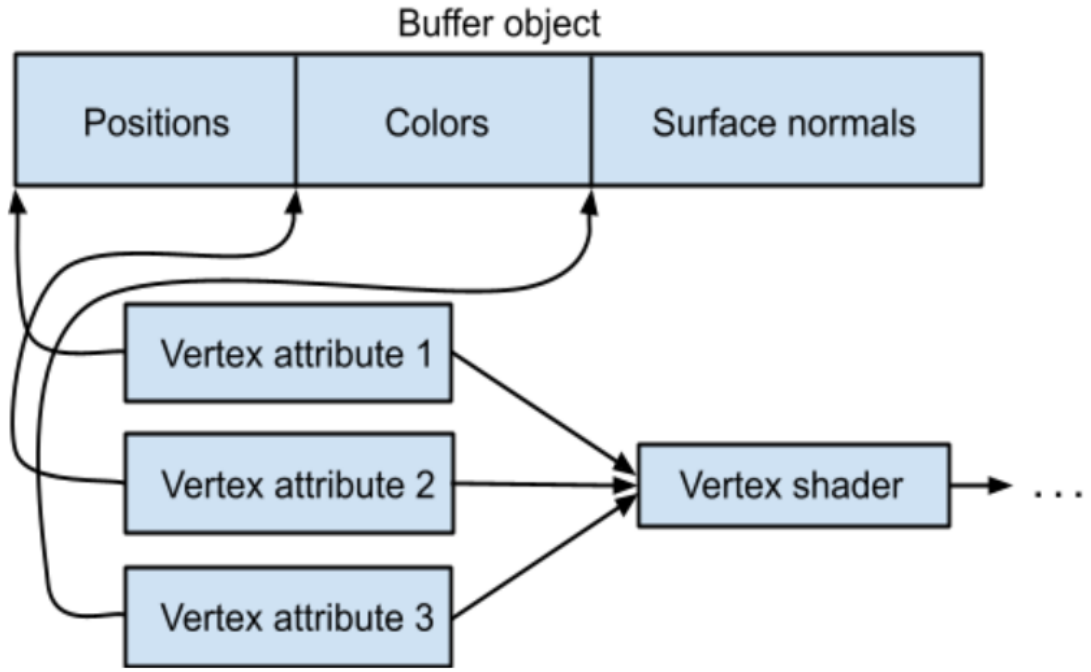
The vertex shader allows us to specify any input we want in the form of vertex attributes and while this allows for great flexibility

OpenGL – First Triangle

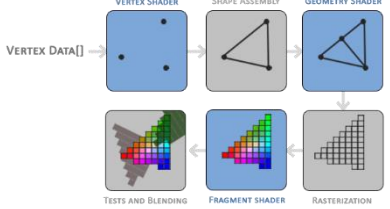


Linking Vertex Attributes





This means we have to specify how OpenGL should interpret the vertex data before rendering.



OpenGL – First Triangle

Linking Vertex Attributes

With this knowledge we can tell OpenGL how it should interpret the vertex data (per vertex attribute) using glVertexAttribPointer.

glVertexAttribPointer

Shaders

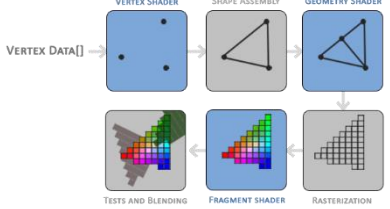
The function `glVertexAttribPointer` specifies how OpenGL should interpret the vertex buffer data whenever a drawing call is made. The interpretation specified is stored in the currently bound vertex array object saving us all quite some work.

The parameters of `glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer)` are as follows:

- `index`: Specifies the index of the vertex attribute.
- `size`: Specifies the number of components per vertex attribute. Must be 1, 2, 3, 4.
- `type`: Specifies the data type of each component in the array.
- `normalized`: Specifies wheter data should be normalized (clamped to the range -1 to 1 for signed values and 0 to 1 for unsigned values).
- `stride`: Specifies the byte offset between consecutive vertex attributes. If stride is 0, the generic vertex attributes are understood to be tightly packed in the array
- `pointer`: Specifies an offset of the first component of the first vertex attribute in the array.

Example usage

```
GLfloat data[] = {
  // Position  // Color           // TexCoords
  1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 0.0f, 0.5f,
  0.0f, 1.0f,  0.2f, 0.8f, 0.0f, 0.0f, 1.0f
};
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));
```



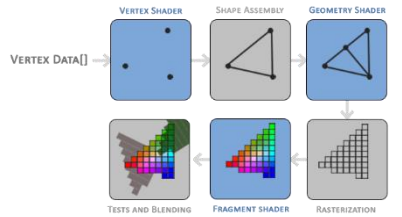
OpenGL – First Triangle

Linking Vertex Attributes

Example usage

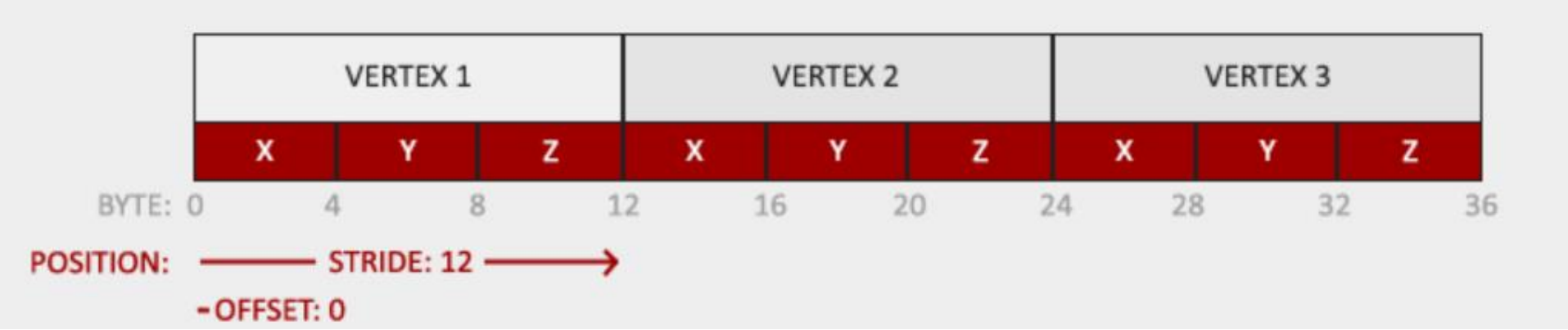
```
GLfloat data[] = {
  // Position  // Color             // TexCoords
  1.0f, 0.0f,  0.5f, 0.5f, 0.5f, 0.0f, 0.5f,
  0.0f, 1.0f,  0.2f, 0.8f, 0.0f, 0.0f, 1.0f
};
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
        7 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));
```

OpenGL – First Triangle



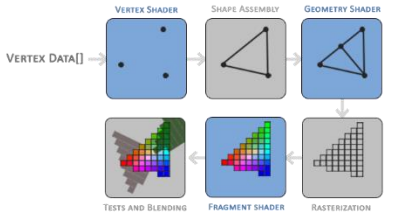
Linking Vertex Attributes

Our vertex buffer data is formatted as follows:



- The position data is stored as 32-bit (4 byte) floating point values.
- Each position is composed of 3 of those values.
- There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
- The first value in the data is at the beginning of the buffer.

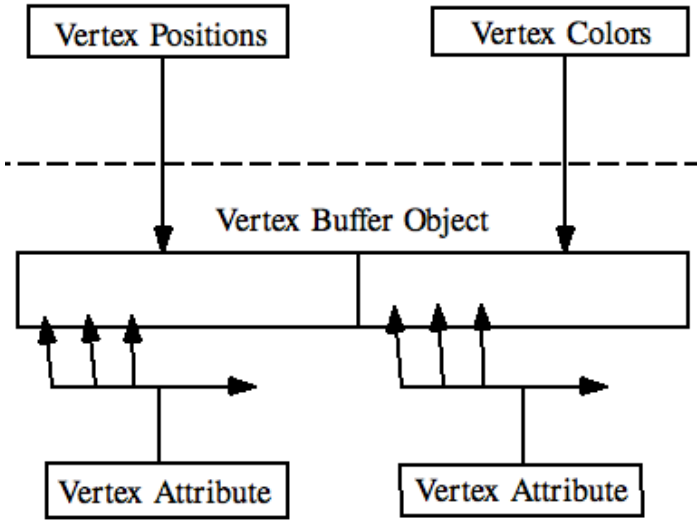
OpenGL – First Triangle



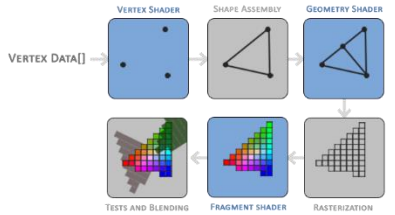
Linking Vertex Attributes

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

- The first parameter specifies which vertex attribute we want to configure. Remember that we specified the location of the position vertex attribute in the vertex shader with layout (location = 0). This sets the location of the vertex attribute to 0 and since we want to pass data to this vertex attribute, we pass in 0.
- The next argument specifies the size of the vertex attribute. The vertex attribute is a vec3 so it is composed of 3 values
- The third argument specifies the type of the data which is GL_FLOAT (a vec* in GLSL consists of floating point values).



OpenGL – First Triangle



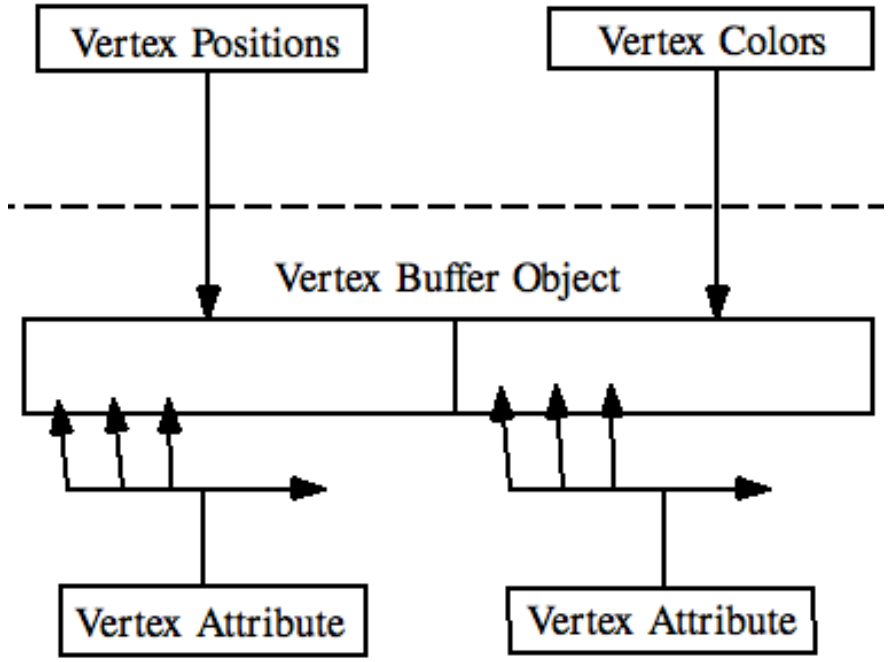
Linking Vertex Attributes

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

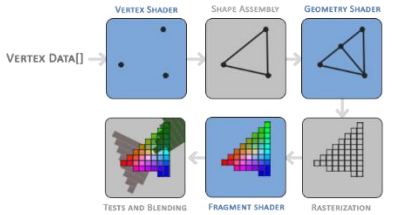
- The next argument specifies if we want the data to be normalized.

If we're inputting integer data types (int, byte) and we've set this to GL_TRUE, the integer data is normalized to 0 (or -1 for signed data) and 1 when converted to float.

This is not relevant for us so we'll leave this at GL_FALSE.



OpenGL – First Triangle



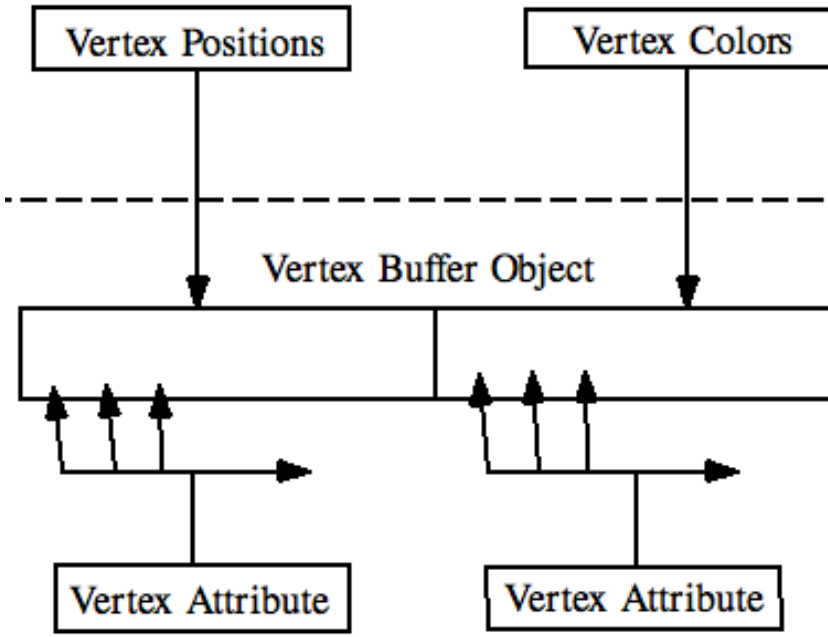
Linking Vertex Attributes

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

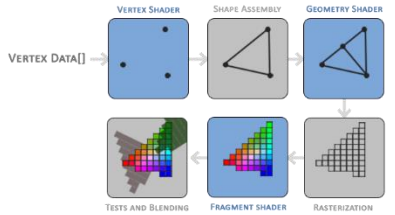
- The fifth argument is known as the stride and tells us the space between consecutive vertex attributes. Since the next set of position data is located exactly 3 times the size of a float away we specify that value as the stride.

Note that since we know that the array is tightly packed (there is no space between the next vertex attribute value) we could've also specified the stride as 0 to let OpenGL determine the stride (this only works when values are tightly packed).

Whenever we have more vertex attributes we have to carefully define the spacing between each vertex attribute.



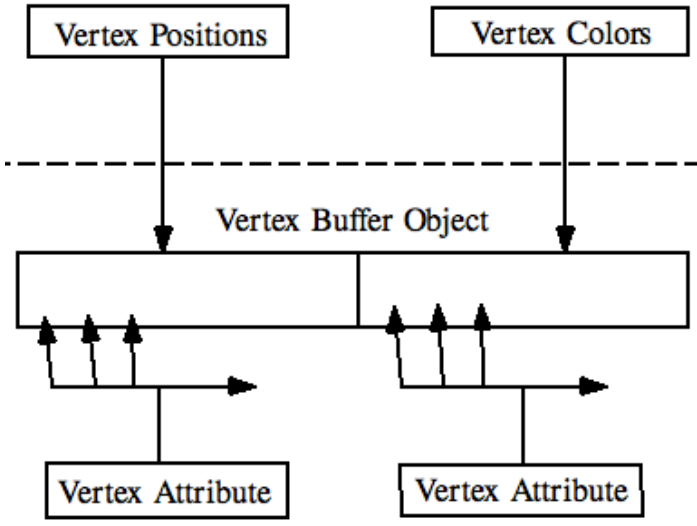
OpenGL – First Triangle



Linking Vertex Attributes

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

- The last parameter is of type void* and thus requires that weird cast. This is the offset of where the position data begins in the buffer. Since the position data is at the start of the data array this value is just 0.

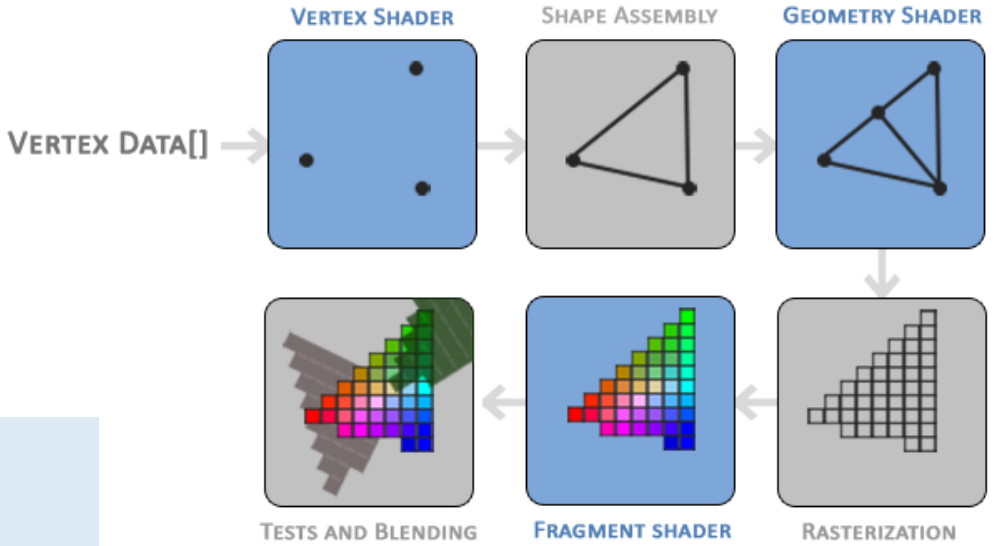


Now that we specified how OpenGL should interpret the vertex data we should also enable the vertex attribute with glEnableVertexAttribArray giving the vertex attribute location as its argument; vertex attributes are disabled by default.

OpenGL – First Triangle

Linking Vertex Attributes

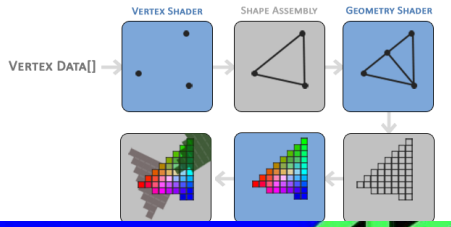
From that point on we have everything set up: we initialized the vertex data in a buffer using a vertex buffer object, set up a vertex and fragment shader and told OpenGL how to link the vertex data to the vertex shader's vertex attributes.



Drawing an object in OpenGL would now look something like this:

```
// 0. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. use our shader program when we want to render an object
glUseProgram(shaderProgram);
// 3. now draw the object
someOpenGLFunctionThatDrawsOurTriangle();
```

OpenGL – First Triangle



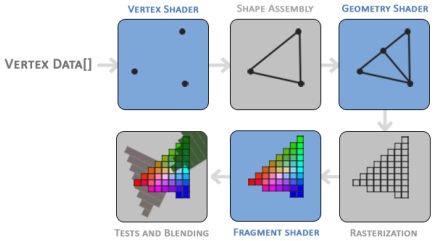
Vertex Array Object

- We have to repeat this process every time we want to draw an object.
- It may not look like that much, but imagine if we have over 5 vertex attributes and perhaps 100s of different objects (which is not uncommon).
- Binding the appropriate buffer objects and configuring all vertex attributes for each of those objects quickly becomes a cumbersome process.
  - What if there was some way we could store all these state configurations into an object and simply bind this object to restore its state?

model without VAO

```
glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();
```

OpenGL – First Triangle



Vertex Array Object

- What if there was some way we could store all these state configurations into an object and simply bind this object to restore its state?

Rendering Loop

A vertex array object (also known as VAO) can be bound just like a vertex buffer object and any subsequent vertex attribute calls from that point on will be stored inside the VAO.

This has the advantage that when configuring vertex attribute pointers you only have to make those calls once and whenever we want to draw the object, we can just bind the corresponding VAO.

model without VAO

```
glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();

glBindBuffer();
glEnableVertexAttribArray();
glVertexAttribPointer();
```

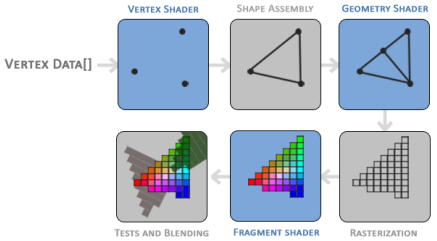
or

model with VAO

```
glBindVertexArray();
```

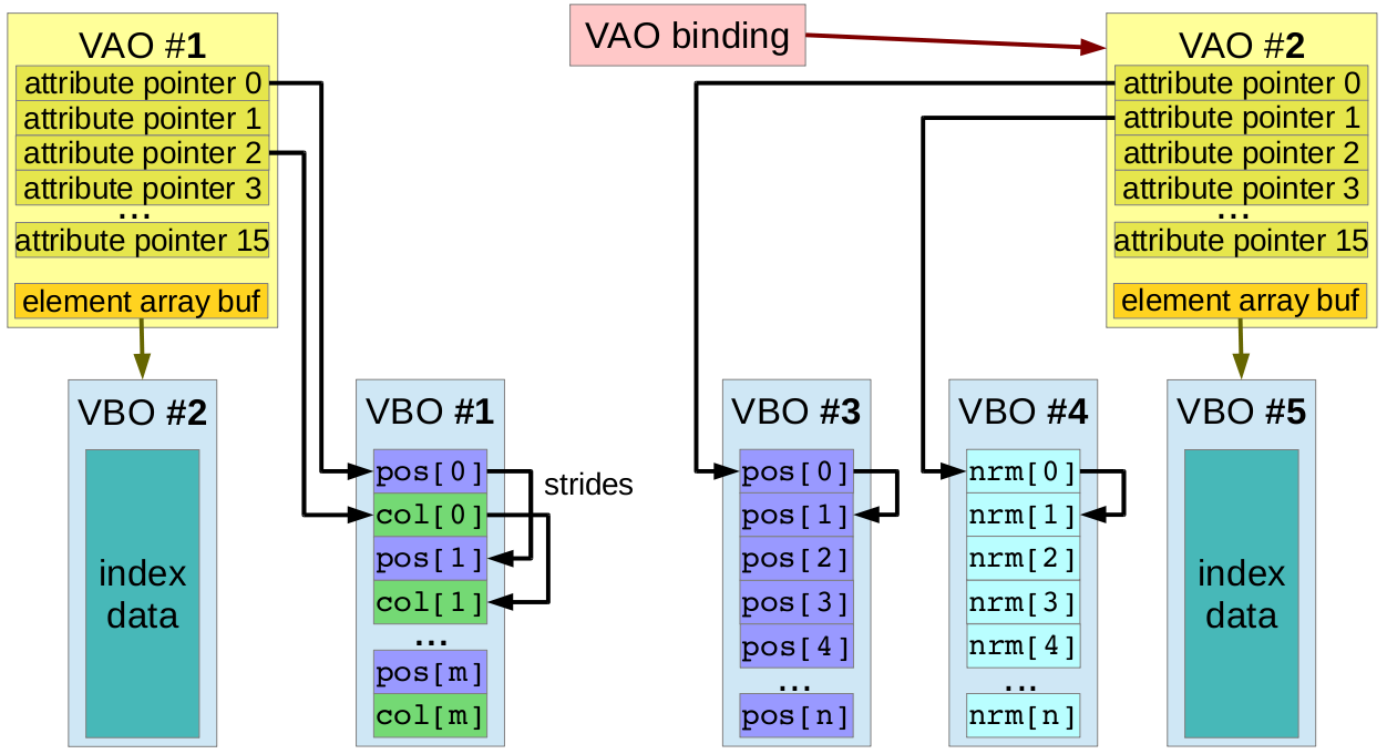
That's all ☺

OpenGL – First Triangle



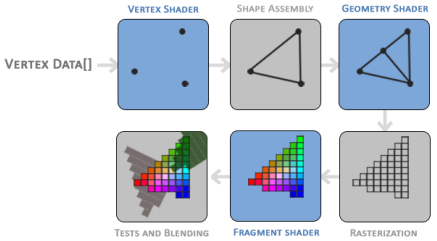
Vertex Array Object

This makes switching between different vertex data and attribute configurations as easy as binding a different VAO. All the state we just set is stored inside the VAO.



Core OpenGL requires that we use a VAO so it knows what to do with our vertex inputs. If we fail to bind a VAO, OpenGL will most likely refuse to draw anything.

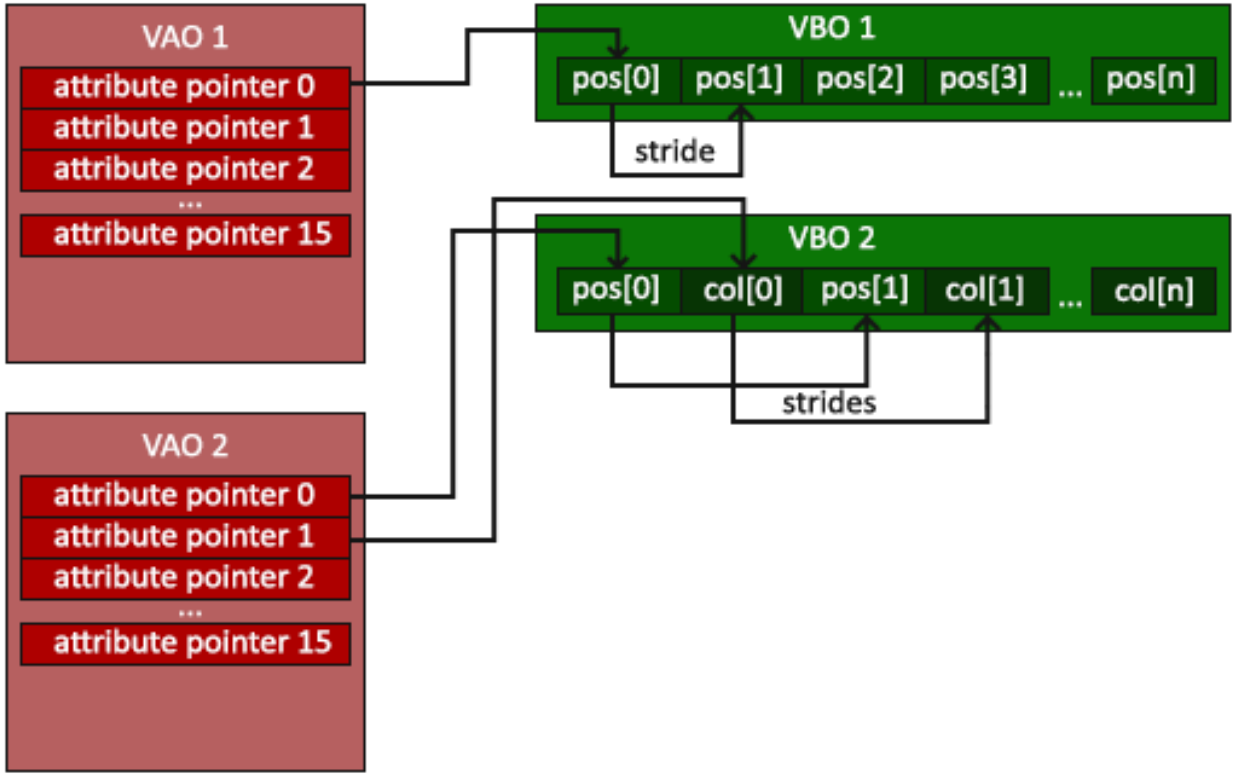
OpenGL – First Triangle



Vertex Array Object

A vertex array object stores the following:

- Calls to glEnableVertexAttribArray or glDisableVertexAttribArray.
- Vertex attribute configurations via glVertexAttribPointer.
- Vertex buffer objects associated with vertex attributes by calls to glVertexAttribPointer.

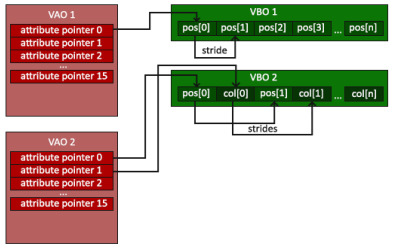


The process to generate a VAO looks simliar to that of a VBO: →

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

OpenGL – First Triangle

Vertex Array Object

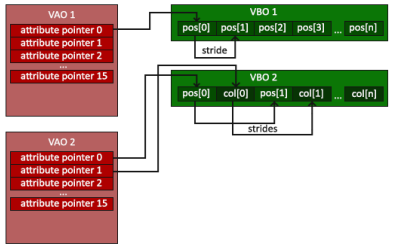


- To use a VAO all you have to do is bind the VAO using glBindVertexArray.
- From that point on we should bind/configure the corresponding VBO(s) and attribute pointer(s) and then unbind the VAO for later use.
- As soon as we want to draw an object, we simply bind the VAO with the prefered settings before drawing the object and that is it.
- In code this would look a bit like this:

```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

 [...]

// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

OpenGL – First Triangle

Vertex Array Object



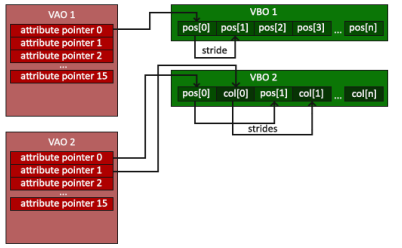
- Usually when you have multiple objects you want to draw, you first generate/configure all the VAOs (and thus the required VBO and attribute pointers) and store those for later use.

- The moment we want to draw one of our objects, we take the corresponding VAO, bind it, then draw the object and unbind the VAO again.

```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

 [...]

// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

OpenGL – First Triangle

Vertex Array Object



- A VAO that stores our vertex attribute configuration and which VBO to use.
- Usually when you have multiple objects you want to draw, you first generate/configure all the VAOs (and thus the required VBO and attribute pointers) and store those for later use.
- The moment we want to draw one of our objects, we take the corresponding VAO, bind it, then draw the object and unbind the VAO again.

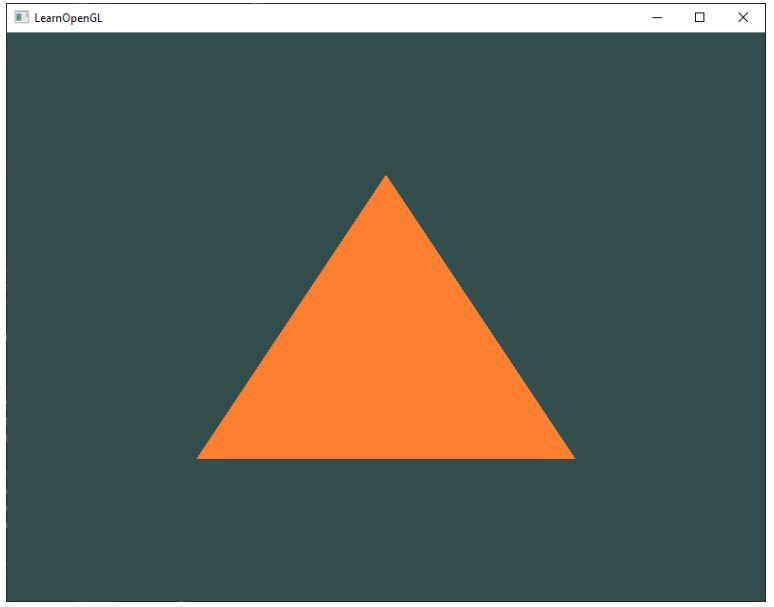
```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

 [...]

// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

OpenGL – First Triangle

Triangle Object

To draw our objects of choice, OpenGL provides us with the glDrawArrays function that draws primitives using the currently active shader, the previously defined vertex attribute configuration and with the VBO's vertex data (indirectly bound via the VAO)

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```



- The glDrawArrays function takes as its first argument the OpenGL primitive type we would like to draw. Since I said at the start we wanted to draw a triangle, and I don't like lying to you, we pass in GL_TRIANGLES.
- The second argument specifies the starting index of the vertex array we'd like to draw; we just leave this at 0.
- The last argument specifies how many vertices we want to draw, which is 3 (we only render 1 triangle from our data, which is exactly 3 vertices long).