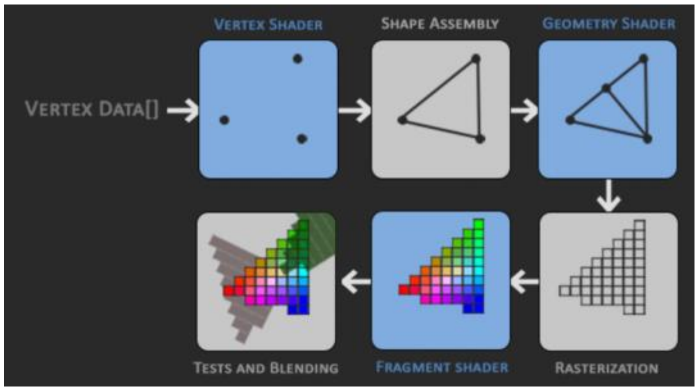
HELLO TRIANGLE



The process of transforming **3D coordinates** to **2D colored pixels** is managed by the graphics pipeline of OpenGL.

- The graphics pipeline is made up of two large parts:
  - The first part transforms 3D coordinates into 2D coordinates
  - The second part transforms those 2D coordinates into colored pixels.
- The steps of the graphics pipeline can be easily executed in **parallel**.

The processing cores run small programs, known as shaders, on the GPU for each step of the graphics pipeline.

- Shaders are written in the OpenGL Shading Language (**GLSL**).
- Some of the graphics pipeline's shaders are configurable, allowing developers to write their own shaders to replace the default ones.

The image to the right is an abstract representation of the OpenGL graphics pipeline. Sections highlighted in blue are parts of the graphics pipeline where developers can inject their own shaders.

Here's a step-by-step of how the graphics pipeline works, in terms of creating a triangle:

1. The input to the graphics pipeline is an array of **vertex data** that contains three 3D coordinates that should form a triangle.
   - A vertex is more than a point in space; it contains vertex attributes, which are all the different types of data stored for a single 3D coordinate.
   - Vertex attributes can contain any data. *For now, they will just contain a 3D position and a color value.*
   - OpenGL needs to be **hinted** at how to interpret the **vertex data**, such as whether it is a point, line, or triangle. These hints are called primitives.
   - OpenGL requires a primitive whenever you call a drawing command.
2. The **vertex shader** takes a single vertex as input, and serves to transform 3D coordinates into different 3D coordinates while allowing basic processing on the vertex's attributes.
3. The **primitive assembly** (or shape assembly) stage takes, from the vertex shader, all vertices that form a primitive as input (or a single vertex, if the point primitive type is chosen) and assembles the points based on the primitive shape given. *In this case, the primitive shape is a triangle.*
4. The **geometry shader** takes, from the primitive assembly stage, a collection of vertices that form a primitive as input, and has the ability to generate other shapes by emitting new vertices to form new (or other) primitives. *In this case, it generates a second triangle out of the given shape.*
5. The **rasterization stage** takes, from the geometry shader, all the primitives as input, and maps them to the corresponding pixels on the screen, resulting in fragments for the fragment shader to use.
   - A fragment in OpenGL is all the data required for OpenGL to render a single pixel.
   - **Clipping** is performed here, which discards all fragments that are outside of view to increase performance.
6. The **fragment shader** takes, from the rasterization stage, a fragment and calculates the final color of a pixel.
   - This is usually the stage where all the advanced OpenGL effects occur.
   - Usually, the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color, such as lights, shadows, etc.
7. The **alpha test and blending** stage takes, from the fragment shader, the corresponding depth (and stencil) value of the fragment and uses those to check if the resulting fragment is in front of other objects, or behind and needing to be discarded. This stage also checks the alpha values of objects and blends the object accordingly.
   - Even though pixel output color is calculated in the fragment shader, it may still change during blending when rendering multiple triangles.

NOTE: The tesselation stage and transform feedback loop are also part of the graphics pipeline, but they're more advanced and not required for basic rendering.

In OpenGL, there is no default vertex shader nor fragment shader (no defaults for these on the GPU), so you must define your own.

Vertex Input

OpenGL is a 3D graphics library, so all the coordinates we specify in OpenGL are in 3D (x, y, z).
OpenGL coordinates work with respect to a **normalize device coordinates range** of -1 to 1.

Create an array of vertices to represent the vertices of a right triangle.

- Array of Triangle Vertices

Next, we need send this vertex data to the the vertex shader. This is accomplished by creating 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 many vertices as we tell it to. This memory is managed by **Vertex Buffer Objects** (VBO).

- VBOs can store a large number of vertices in the GPU's memory.
- Since sending data from the CPU to the GPU is relatively slow, VBOs save computation time by transmitting large batches of data all at once to the graphics card, rather than sending the data of each vertex individually.
- Once the vertex data is in the graphics card's memory, the vertex shader has quick and easy access to it.

OpenGL allows binding multiple buffers at once, as long as they are different buffer types.

Configure a vertex buffer object for the vertex data.

- This consists of generating a buffer object, binding it to the correct buffer type for a VBO, and copying the vertex data into it.
- **NOTE:** Remember that OpenGL objects have a unique id.
- Configuring a Vertex Buffer Object

Now the vertex data is stored within memory on the graphics card and is being managed by the vertex buffer object you created.

Vertex Shader

The vertex shader is programmable and must be provided by developers as there is no default vertex shader.

Create a vertex shader that outputs the vertex data.

- Basic Vertex Shader

Compiling a Shader

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

Create a shader object for the vertex shader and compile the shader.

- Make sure to check for shader compilation errors!
- Compiling a Vertex Shader

Fragment Shader

The fragment shader is programmable and must be provided by developers as there is no default fragment shader.
Colors in computer graphics are represented as RGBA. In OpenGL and GLSL, the strength of each component ranges from 0 to 1, b ut typically, the strength ranges from 0 to 255 (8-bit).

Create a fragment shader that outputs an opaque color for every fragment.

- Basic Fragment Shader

Create a shader object for the fragment shader and compile the shader.
- Remember to check for shader compilation errors here, too!
- [Compiling a Fragment Shader]()

With both shaders compiled, the only thing left to do is link both shader objects into a shader program that we can use for r endering. After that, a triangle will finally be rendered to the screen.

Shader Program
A shader program object is the final linked version of multiple shaders combined.
To use compiled shaders, have to link them to a shader program object and then activate the shader program when rendering objects. After that, the shader program's shaders will be used when render calls are issued.
When shaders are linked into a shader program, the outputs of one shader are linked to the inputs of the other. Linking errors will occur here if the outputs and inputs d on't match.

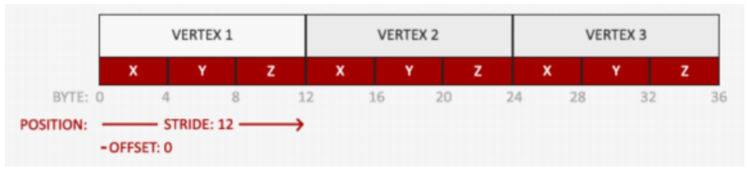
Create a shader program object, attach the vertex and fragment shaders, and link them in the shader program object.
- The shader objects can be deleted after they've been linked into the shader program.
- Make sure to check from program linking errors!
- [Creating a Shader Program]()

At this point, we've sent the input vertex data to the GPU and instructed the GPU on how to process that data with a vertex shader and fragment shader.
However, OpenGL still does not yet know how it should interpret the vertex data in memory and how it should connect the vertex data to the vertex shader's attributes.

Linking Vertex Attributes
We must manually specify what part of our input data goes to which vertex attribute in the vertex shader.
This means we must specify how OpenGL should interpret the vertex data before rendering.

On the right is a visual representation of how our vertex buffer data is currently formatted.

Tell OpenGL how to interpret the vertex data.
- [Interpretting Vertex Data]()



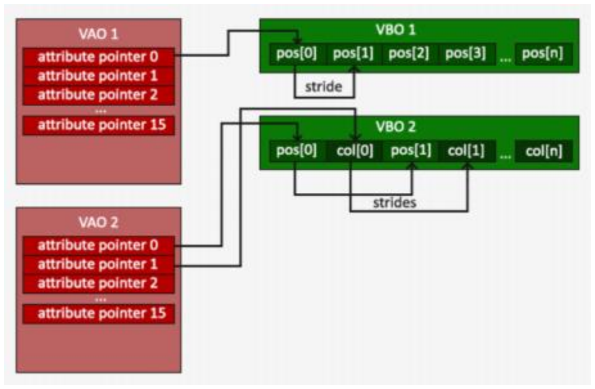
Vertex Array Object
A Vertex Array Object (VAO) can be bound just like a vertex buffer object and any subsequent vertex attribute calls from that point on will be stored inside the VAO.
- This is useful because it means that when configuring vertex attribute pointers, you only have to make those calls once and whenever we want to draw the object, we can just bind the corresponding VAO.
- This makes switching between different vertex data and attribute configurations as easy as binding a different VAO.

A VAO stores the following:
- Calls to `glEnableVertexAttribArray` or `glDisableVertexAttribArray`.
- Vertex attribute configurations via `glVertexAttribPointer`.
- Vertex buffer objects associated with vertex attributes by calls to `glVertexAttribPointer`.

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.

Core OpenGL requires that we use a VAO so it knows what to do with our vertex inputs.

Configure a vertex array object that encompasses the current vertex attribute configurations of the vertex buffer object.
- [Configuring a Vertex Array Object]()



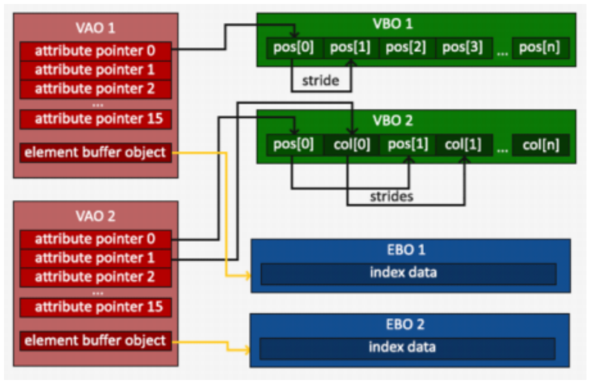
The Triangle We've All Been Waiting For
Draw that god-forsaken triangle to the screen.
- [Drawing a Triangle]()

Element Buffer Objects
An Element Buffer Object (EBO) is a buffer that stores indices that OpenGL uses to decide what vertices to draw.
- These indices are used to prevent overlap in the vertex data (when two triangles use one of more of the same vertex).

Create a new set of vertices that will form a rectangle. Then, create the indices for the EBO.
- [Rectangle Vertices and Indices]()

Configure an element buffer object for the indices and draw the rectangle.
- [Configuring an Element Buffer Object]()

A vertex array object can also keep track of element buffer object bindings. Just bind the EBO before unbinding the VAO.



If, for testing purposes, you want to draw your triangles in wireframe mode, you can configure how OpenGL draws its primitives via `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`, which applies the setting to the front and back of all triangles and draws them as lines.
Setting it back to normal is a matter of calling `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)`.

EXERCISES

1. Try to draw 2 triangles next to each other using glDrawArrays by adding more vertices to your data.
2. Now, create the same 2 triangles using two different VAOs and VBOs for their data.
3. Create two shader programs where the second program uses a different fragment shader that outputs the color yellow. Draw both triangles again when one outputs the color yellow.

Array of Triangle Vertices

Wednesday, March 16, 2022 5:21 PM

Because OpenGL works in 3D space, one of the coordinate planes of the triangle must be all zeros (we chose the z-coordinate plane here) so the depth of the triangle is uniform.

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

Configuring a Vertex Buffer Object

Wednesday, March 16, 2022 5:41 PM

OpenGL has many buffer types.
The buffer type of a VBO is GL_ARRAY_BUFFER.
After binding the new buffer, any buffer calls we make on the GL_ARRAY_BUFFER target will be used to configure the currently bound buffer, which is vbo.
Calling glBufferData on GL_ARRAY_BUFFER allows us to copy the vertex data into the buffer's memory.

```
// Generates a buffer and stores its id
unsigned int vbo;
glGenBuffers(1, &vbo);
// Binds the newly generated buffer as a Vertex Buffer Object
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// Inserts the vertex data into the Vertex Buffer Object
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

The GL_STATIC_DRAW argument is a specification of how we want the graphics card to manage the given data.

- GL_STREAM_DRAW - The data is set only once and used by the GPU only 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.

Since the position data of the triangle does not change, is used a lot, and stays the same for every render call, its best usage type is GL_STATIC_DRAW.

Basic Vertex Shader

Wednesday, March 16, 2022 8:42 PM

Each shader begins with a declaration of its version and the OpenGL profile used. As of OpenGL 3.3, the GLSL version numbers match the OpenGL version.
Layouts are used to declare all the input vertex attributes in the vertex shader at a specific location.

- Right now, we only need one layout because the only data we're storing in vertices[] is their normalized device coordinates.

gl_Position is actually the output of the vertex shader.

```
#version 330 core

layout (location = 0) in vec3 aPos;

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

Compiling a Vertex Shader

Wednesday, March 16, 2022 9:05 PM

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

```
// Creates the vertex shader object
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
// Stores the source code for the shader
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
// Compiles the shader
glCompileShader(vertexShader);

// Error checking for vertex shader compilation
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success) {
    glGetShaderLogInfo(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

Basic Fragment Shader

Wednesday, March 16, 2022 9:30 PM

The fragment shader only requires one output variable: the color of the fragment.
Output values are declared with the **out** keyword.
Here, the outputted fragment color is always an opaque orange.

```
#version 330 core

out vec4 FragColor

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

Compiling a Fragment Shader

Wednesday, March 16, 2022 9:34 PM

The process of compiling a fragment shader is similar to compiling a vertex shader, except you used GL_FRAGMENT_SHADER instead.

```
// Fragment shader source code, stored in a string.
const char* fragmentShaderSource =
"#version 330 core\n"
"out vec4 FragColor;\n"
"void main() {\n"
"   FragColor = (1.0f, 0.5, 0.2f, 1.0f);\n"
"}\0";
```

```
// Creates a fragment shader object and compiles the shader
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

You should perform error checking for the fragment shader compilation like you did with the vertex shader compilation.

Creating a Shader Program

Wednesday, March 16, 2022 10:10 PM

```
// Creates a shader program object
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
// Attaches the vertex and fragment shaders to the shader program
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
// Links the shaders in the shader program
glLinkProgram(shaderProgram);
```

Error checking for a program is slightly different, in that we want to check if the program successfully linked or not.

```
// Error checking for program linking
int success;
char infoLog[512];
glGetProgramiv(id, GL_LINK_STATUS, &success);
if (!success) {
   glGetProgramInfoLog(id, 512, NULL, infoLog);
   std::cout << "ERROR::PROGRAM::" << type << "::LINKING_FAILED\n" << infoLog << std::endl;
}
```

Finally, we can use the shaders through the shader program.

```
glUseProgram(shaderProgram);
```

Now, every shader and rendering call will use our shader program object (and thus, the shaders).

Make sure to delete the shader objects after they've been linked into the shader program.

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

Interprетting Vertex Data

Wednesday, March 16, 2022 10:45 PM

```
// Tells OpenGL how to interpret the vertex data
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

The parameters of glVertexAttribPointer are pretty self-explanatory, except for the last one, *pointer*.

- *pointer* 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.

We use glEnableVertexAttribArray to enable the vertex attribute at location 0.

- This is because vertex attributes are disabled by default.

Configuring a Vertex Array Object

Wednesday, March 16, 2022 11:51 PM

```
// Generate a vertex array object and stores its id
unsigned int vao;
glGenVertexArray(1, &vao);
```

To use a VAO, all you have to do is bind it. From that point on, we should bind/configure the corresponding VBO(s) and attrib ute pointer(s) and then unbind the VAO for later use, almost like setting up a prefab.
As soon as we want to draw an object, we just need to bind the VAO that has the the settings we want before drawing the object. That's it.

Below are the typical steps taken to set up a VAO:

```
// ..:: Initialization code (done once, unless your object frequently changes) ::..
// 1. Bind the vertex array object
glBindVertexArray(vao);
// 2. Copy the vertices array into a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. Set the vertex attributes pointers
glSetVertexAttribPointer(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();
```

Drawing a Triangle

Thursday, March 17, 2022 12:15 AM

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

Rectangle Vertices and Indices

Thursday, March 17, 2022 1:00 AM

```
// Vertices of a rectangle
float vertices[] = {
// Coords
    0.5f,  0.5f, 0.0f,       // top right
    0.5f, -0.5f, 0.0f,       // bottom right
   -0.5f, -0.5f, 0.0f,       // bottom left
   -0.5f,  0.5f, 0.0f,       // top left
};

// Indices of a rectangle given the above
vertices
unsigned int indices[] = {
   0, 1, 3,
   1, 2, 3
};
```

Configuring an Element Buffer Object

Thursday, March 17, 2022 1:12 AM

```
// Generates and configures the element buffer object for the indices
unsigned int ebo;
glGenBuffers(1, &ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), &indices, GL_STATIC_DRAW);
```

```
// DRAW
// Replace glDrawArrays with this:
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

The parameters of glDrawElements are pretty self-explanatory, except for the last one, *indices*.

- *indices* allows us to specify an offset in the EBO (or pass in an index array, but that is when you're not using element buffer objects).

After everything is said and done, your code should, in some shape or form, resemble the following:

```
// ..:: Initialization code :: ..
// 1. bind Vertex Array Object
glBindVertexArray(vao);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in a element buffer for OpenGL to use
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ..:: Drawing code (in render loop) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```