Computer Graphics and OpenGL

Jake Michelotti

3/24/2021

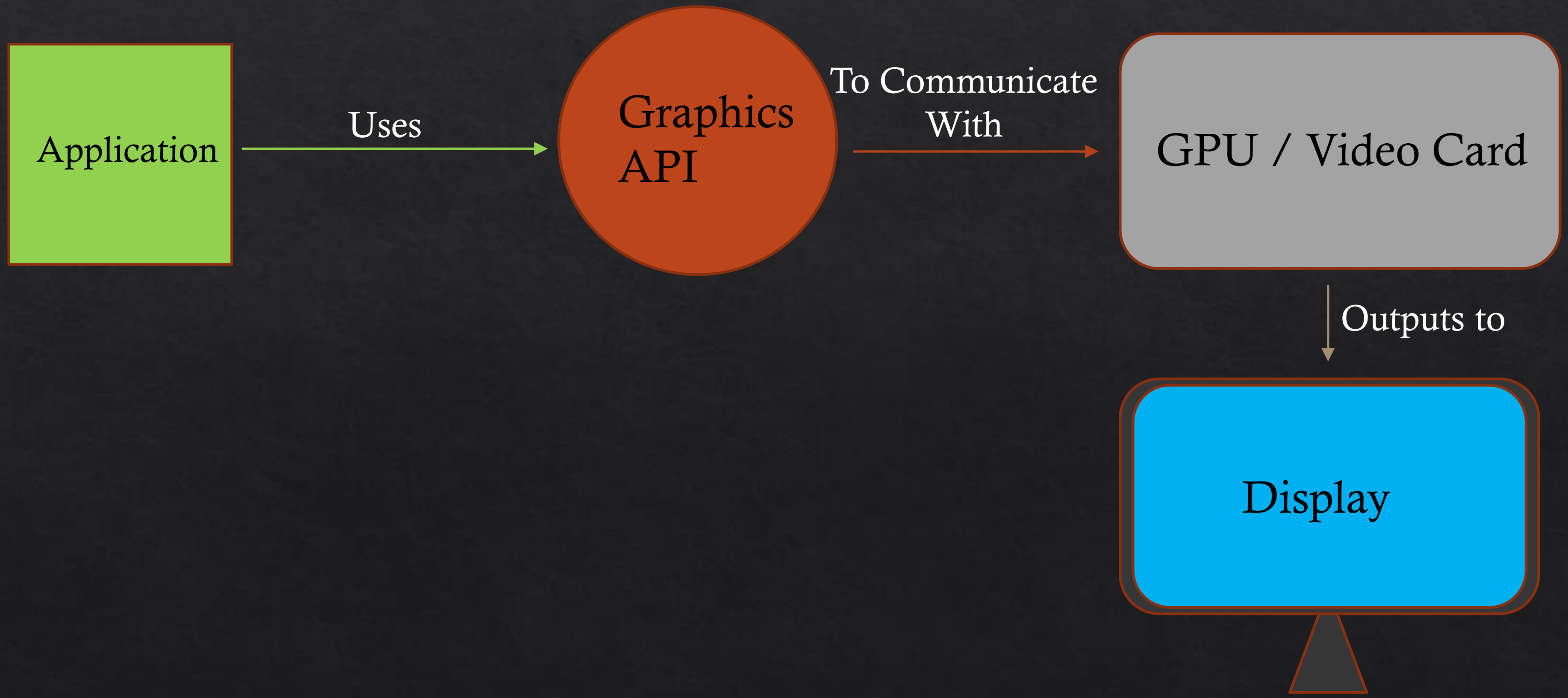
Montana Technological University

Overview

- General Overview of Computer Graphics
- OpenGL and some alternatives
- How OpenGL works
- An Example
- References

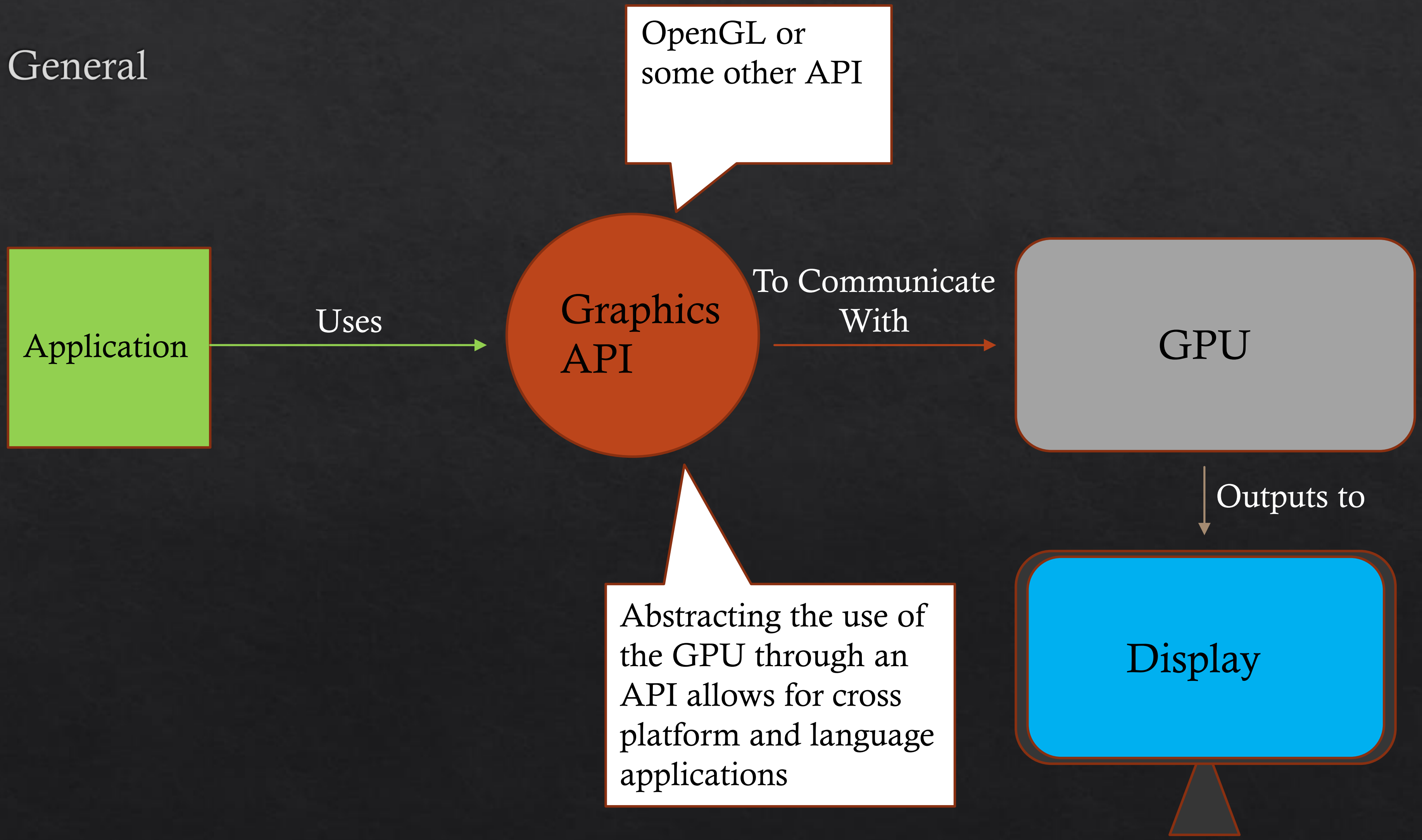
How Computers Output Graphics

- In General



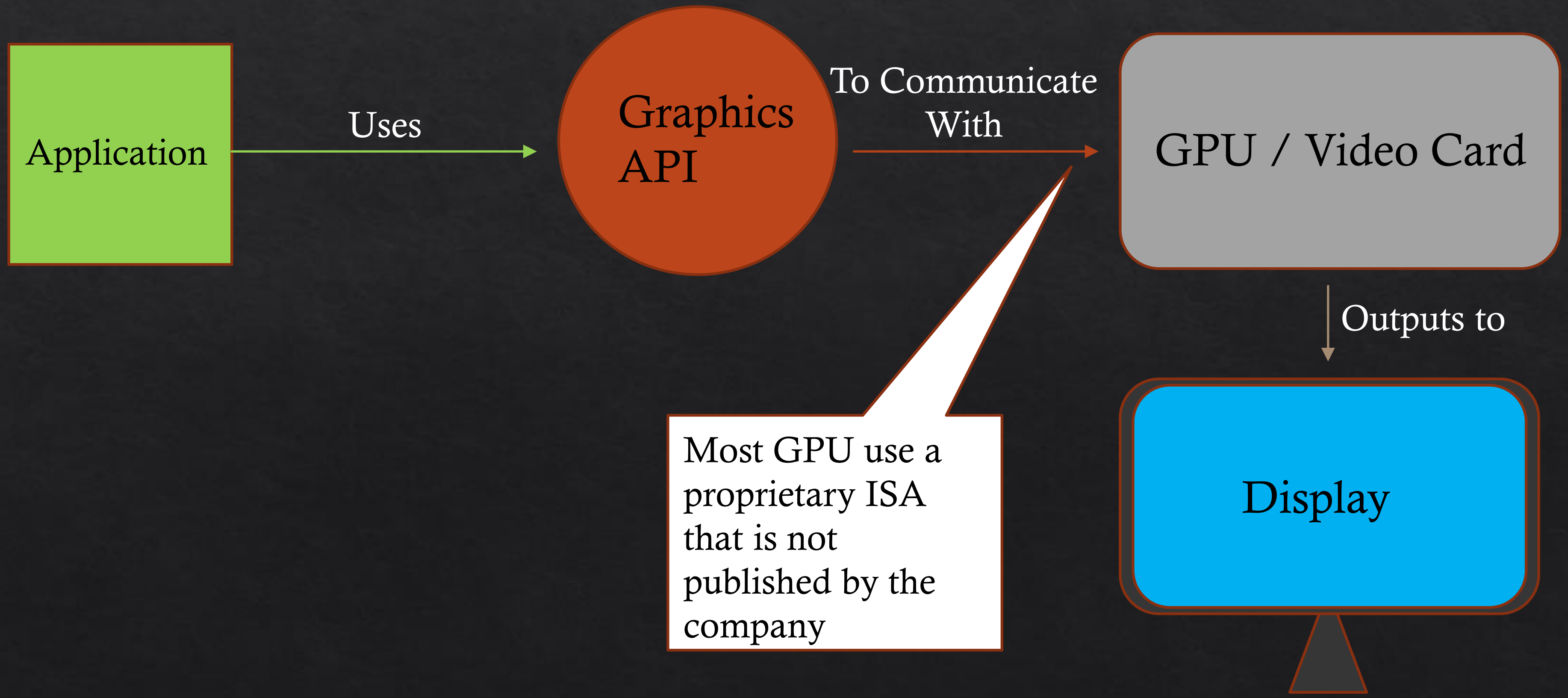
How Computers Output Graphics

- In General



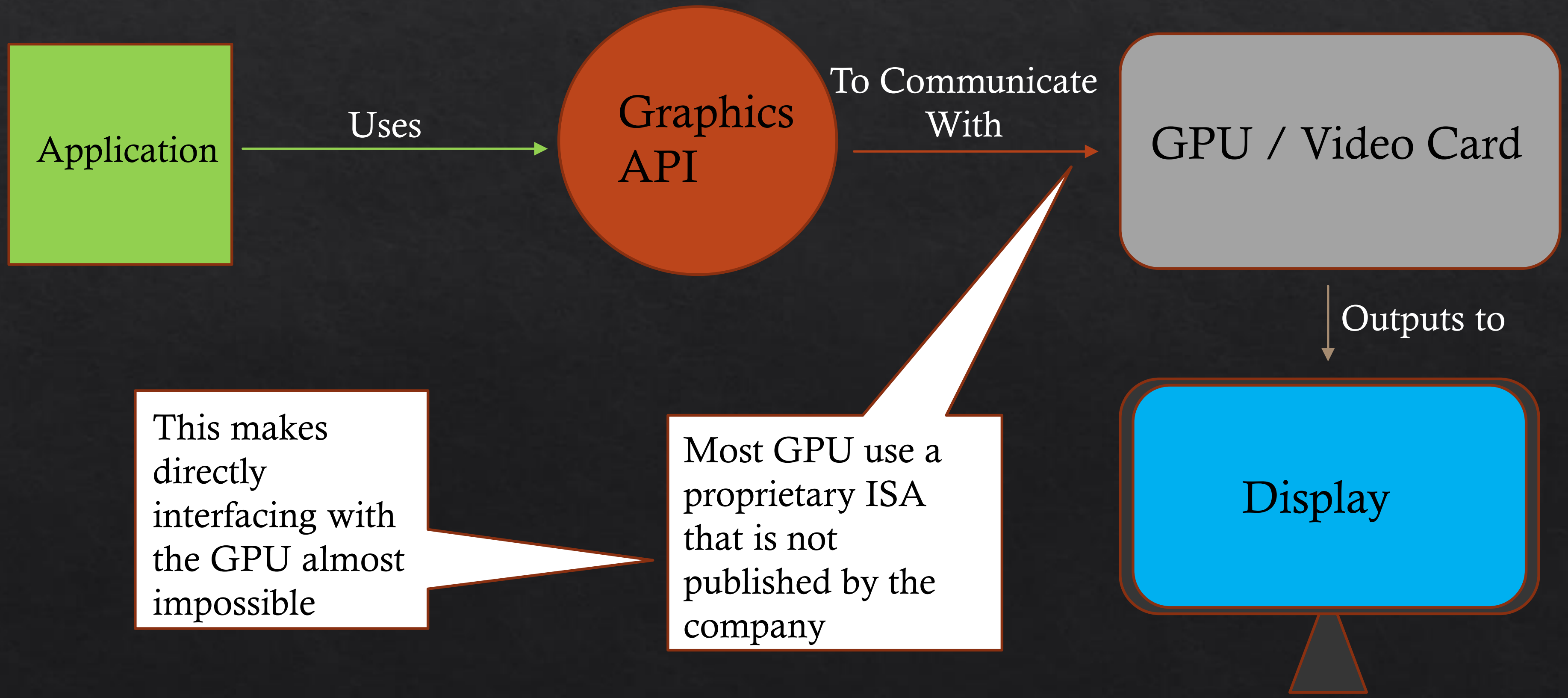
How Computers Output Graphics

- In General



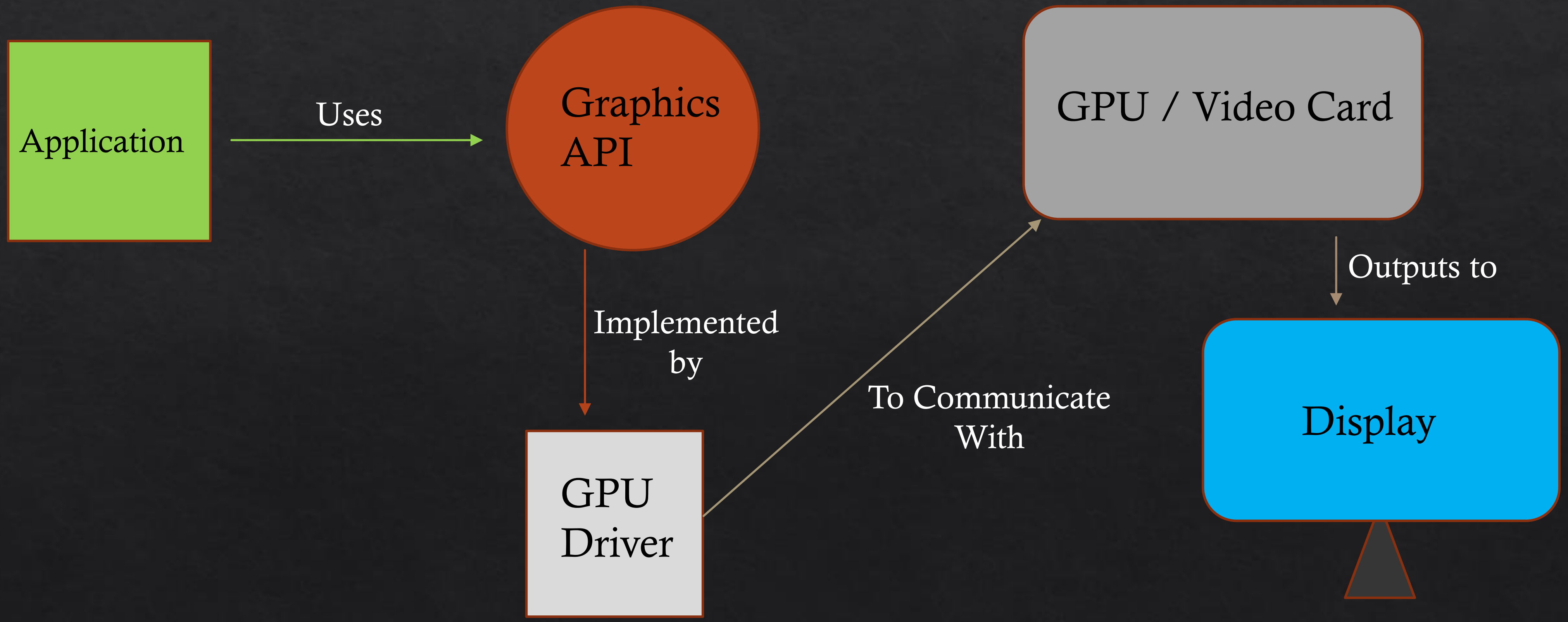
How Computers Output Graphics

- In General



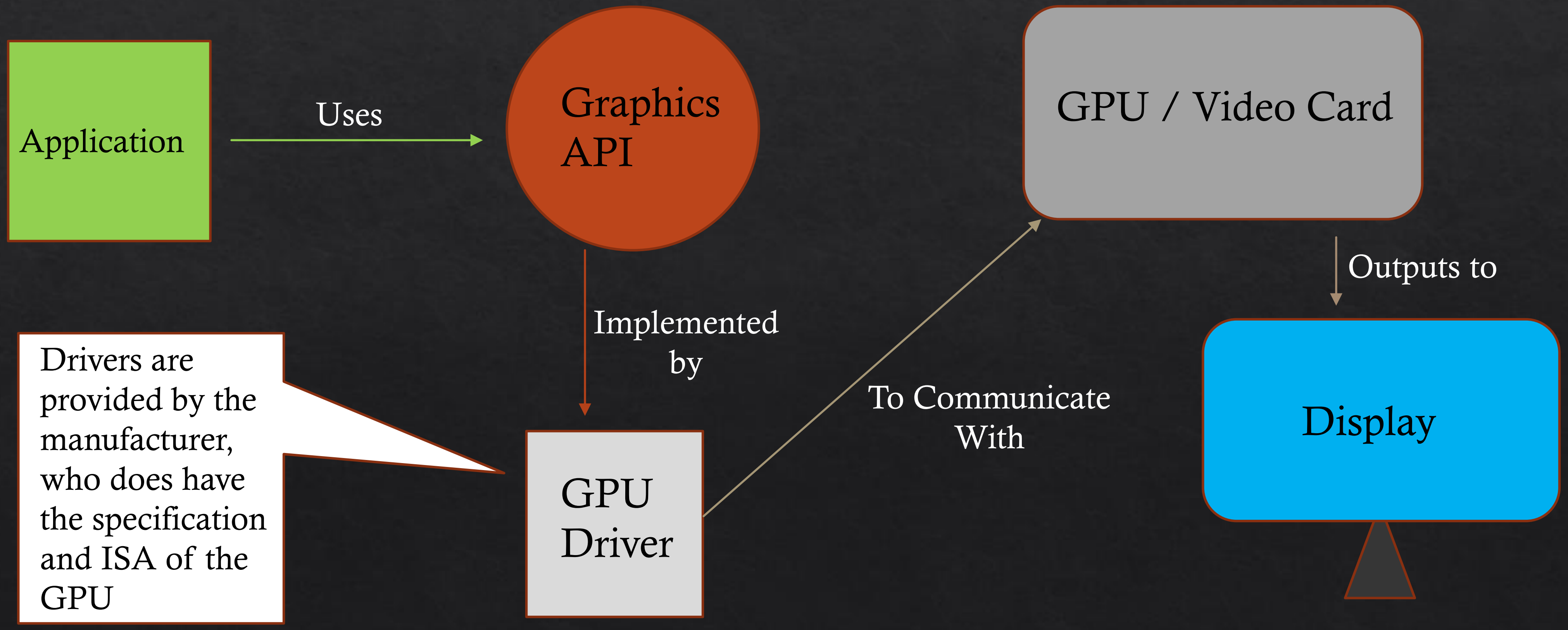
How Computers Output Graphics

- In General



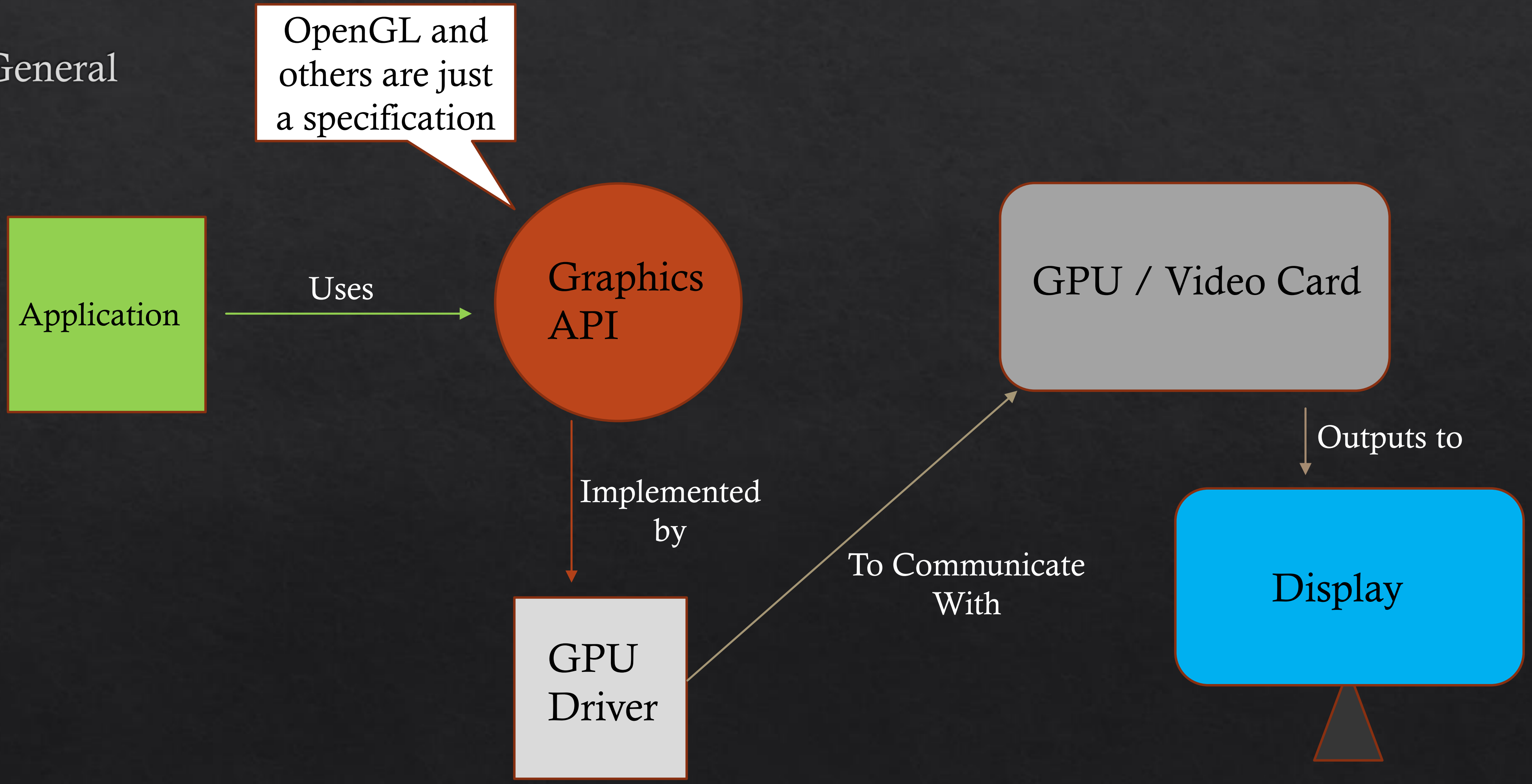
How Computers Output Graphics

- In General



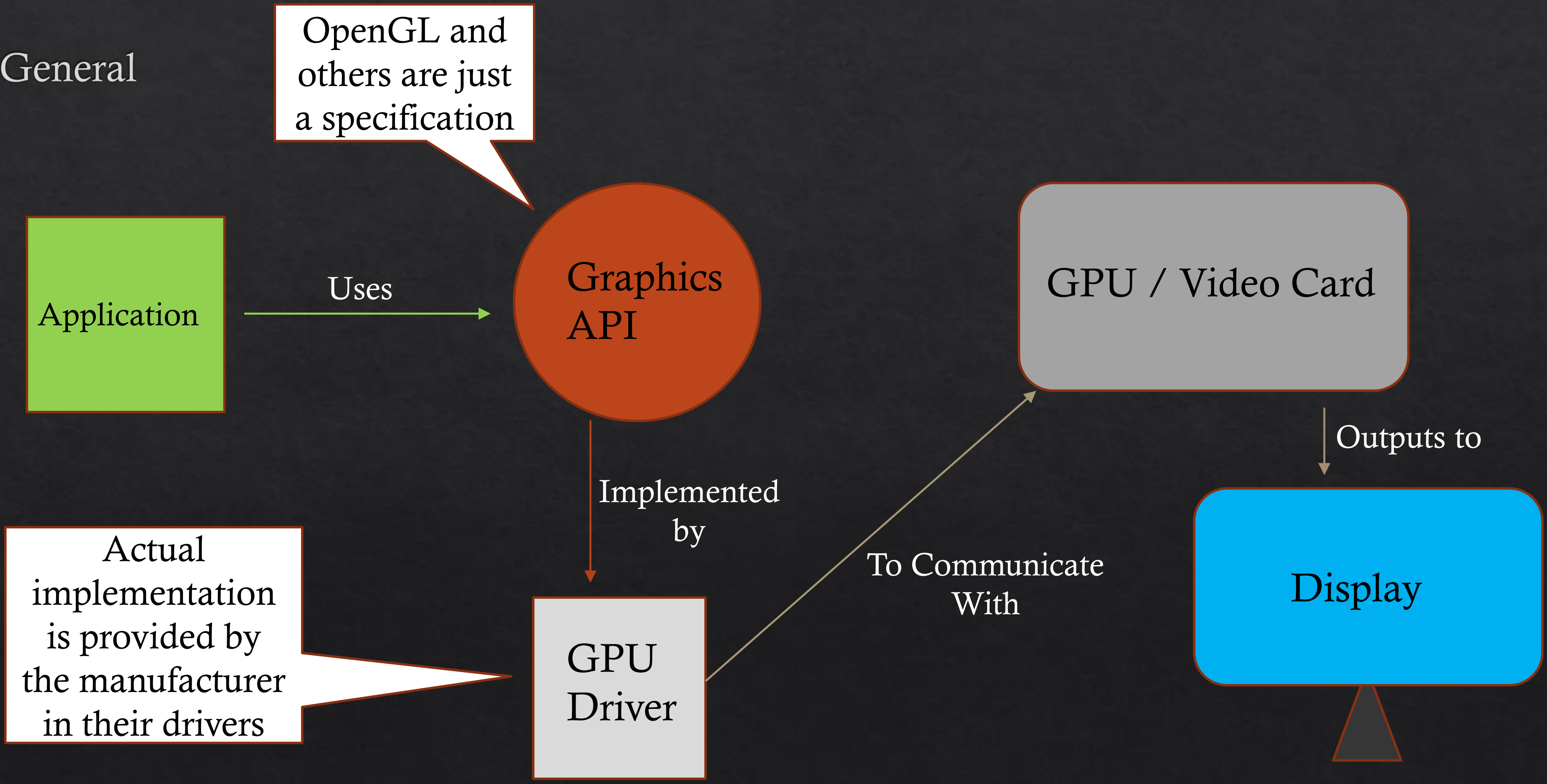
How Computers Output Graphics

- In General



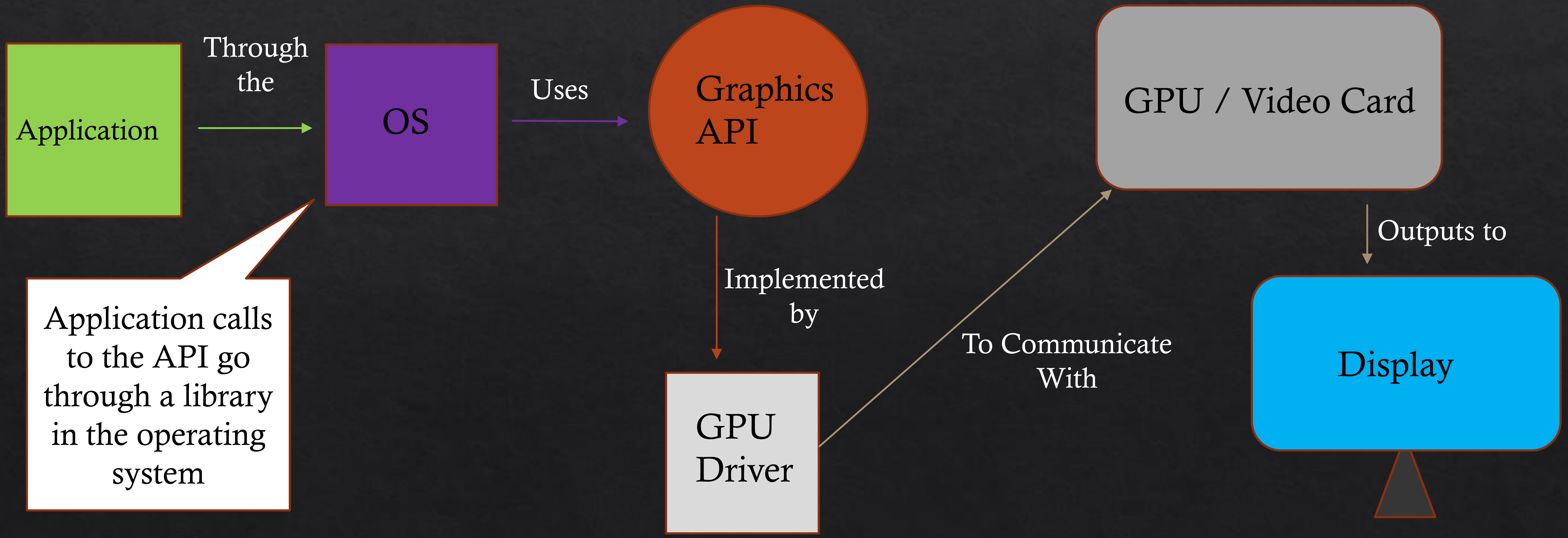
How Computers Output Graphics

- In General



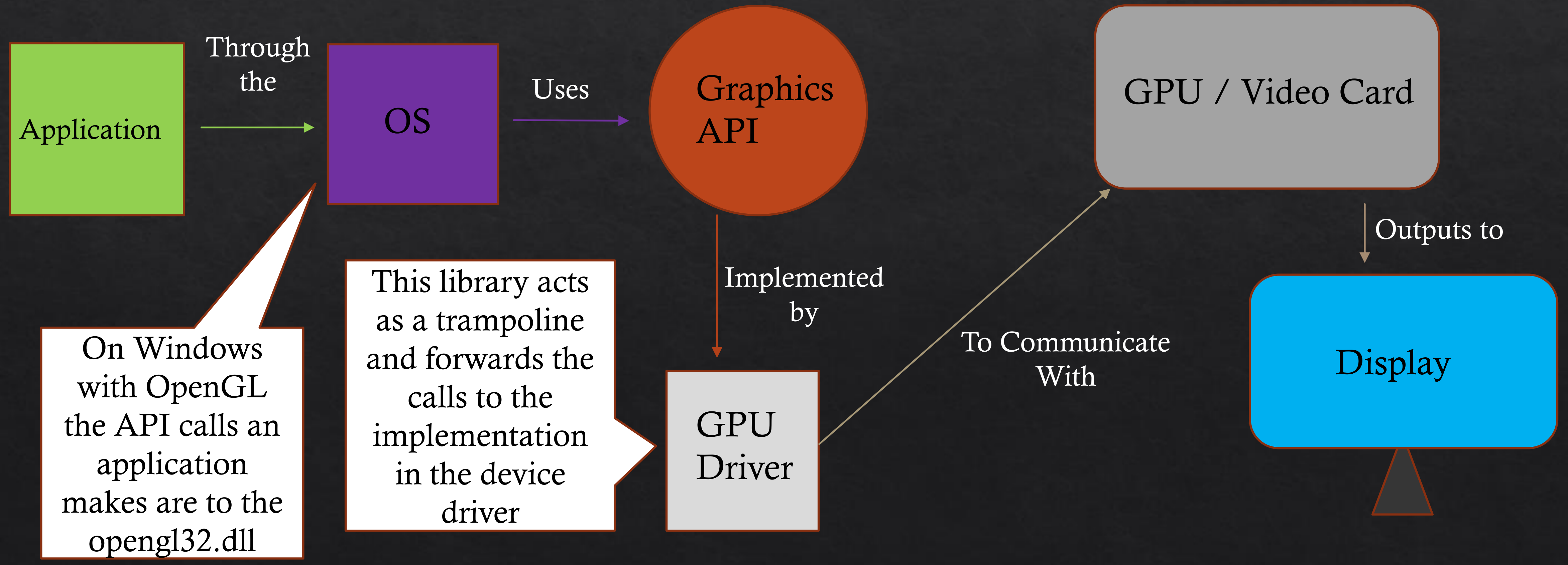
How Computers Output Graphics

- In General



How Computers Output Graphics

- In General



What is OpenGL

- Application Programmer Interface
- Specification not an Implementation
- Cross-Platform (OS)
- Cross-Language
- Specification is maintained by the **Khronos Group** consortium
- Widely used across all platforms and applications

Other Graphics APIs – Direct3d

- Microsoft's 3D graphics API
- 3D graphics API within the DirectX libraries
- Is widely used in for video games
- Only available on Windows

Other Graphics APIs - Vulcan

- Also maintained by the Khronos Group
- Developed to be the successor of OpenGL
- Provide a lower-level API giving more direct access to the GPU
- Improved performance over OpenGL
- Specification was released in 2016
- Starting to see wider adoption across the industry

Computer Graphics Without an API?

- It is entirely possible to calculate an image and display within a window
- On windows it would require an operating system call like **BitBLt**
- Windows also provides the graphic device interface (GDI)
  - Allows the users simple access to graphical programing without using the GPU or video card
  - Cannot do 3D graphics
  - Struggles with 2D animation
- Generally everything displayed on the monitor is at least piped through the GPU / Video Card

General Computing APIs

- GPU's are extremely useful for computing outside of graphics
- Until fairly recently, utilizing GPUs was done by bending OpenGL to accomplish the task
- OpenCL is focused on heterogeneous computing
  - Is also maintained by the Khronos Group
- Cuda is focused on general purpose use of GPUs
  - Maintained by and only functions on GPUs manufactured by Nvidia

How OpenGL Works

- OpenGL is a large state machine
- A large collection of variables define how OpenGL should currently operate
- For example, to change OpenGL from drawing triangles to lines we change the context variable that defines how it should draw
- The current state of OpenGL is called the context

How OpenGL Works

- The Birds Eye View of steps to operating OpenGL
  - Create a Window
  - Create an OpenGL context
  - Give OpenGL data to render
  - Set OpenGL to the desired context
  - Have OpenGL render

How OpenGL Works

- The Birds Eye View of steps to operating OpenGL
  - Create a Window
  - Create an OpenGL context
  - Give OpenGL data to render
  - Set OpenGL to the desired context
  - Have OpenGL render

OpenGL cannot operate without a window

OpenGL Objects

- The context and data OpenGL is working with are controlled by OpenGL objects
- Objects are an abstraction that allows for easier translation to higher languages
- OpenGL libraries are usually implemented in C, so objects can be thought of as more as structs

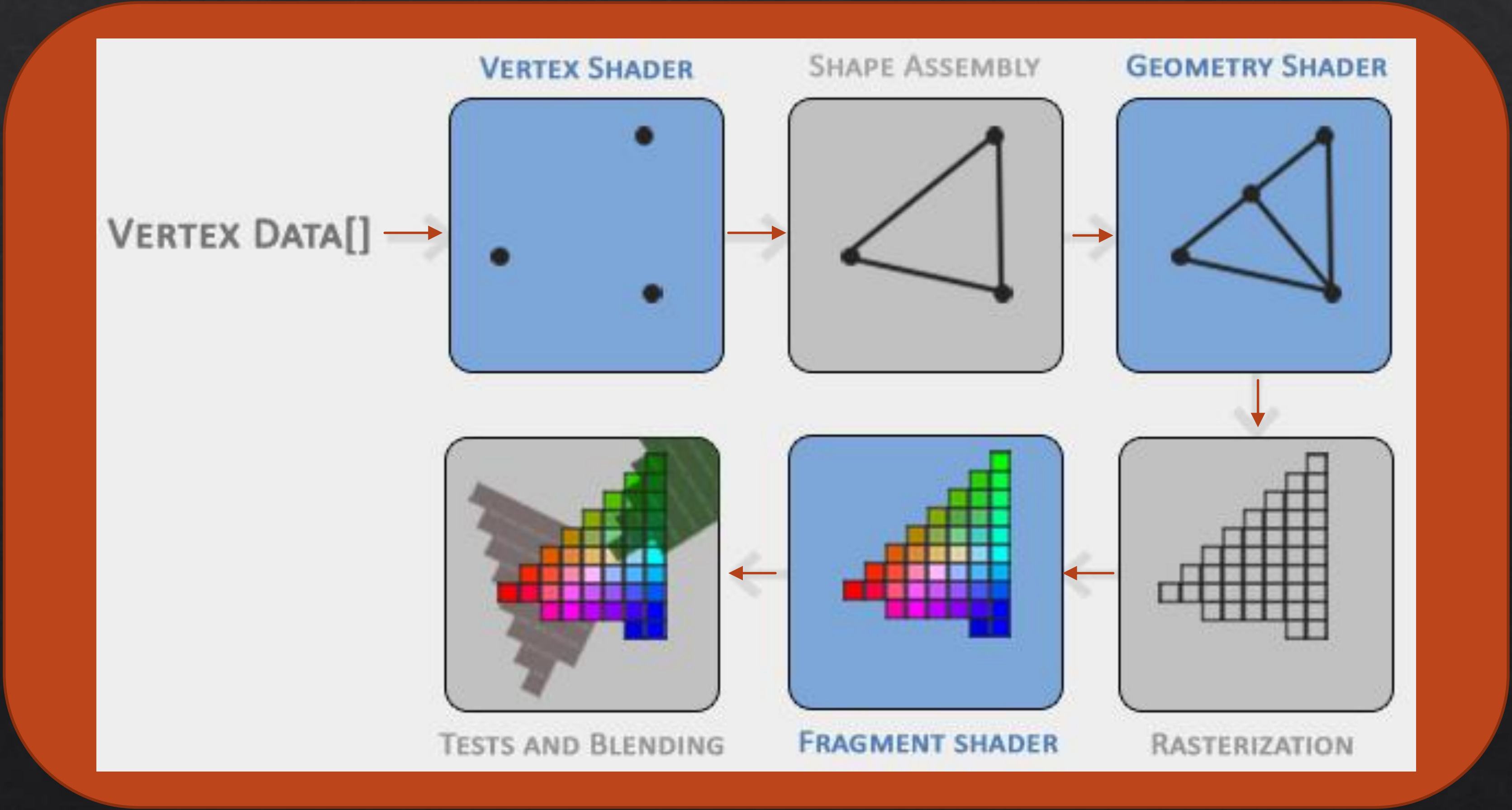
OpenGL Objects

- Creating and using an object involves the following workflow

```
// create object
unsigned int objectId = 0;
glGenObject(1, &objectId);
// bind/assign object to context
glBindObject(GL_WINDOW_TARGET, objectId);
// set options of object currently bound to GL_WINDOW_TARGET
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH,  800);
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_HEIGHT, 600);
// set context target back to default
glBindObject(GL_WINDOW_TARGET, 0);
```

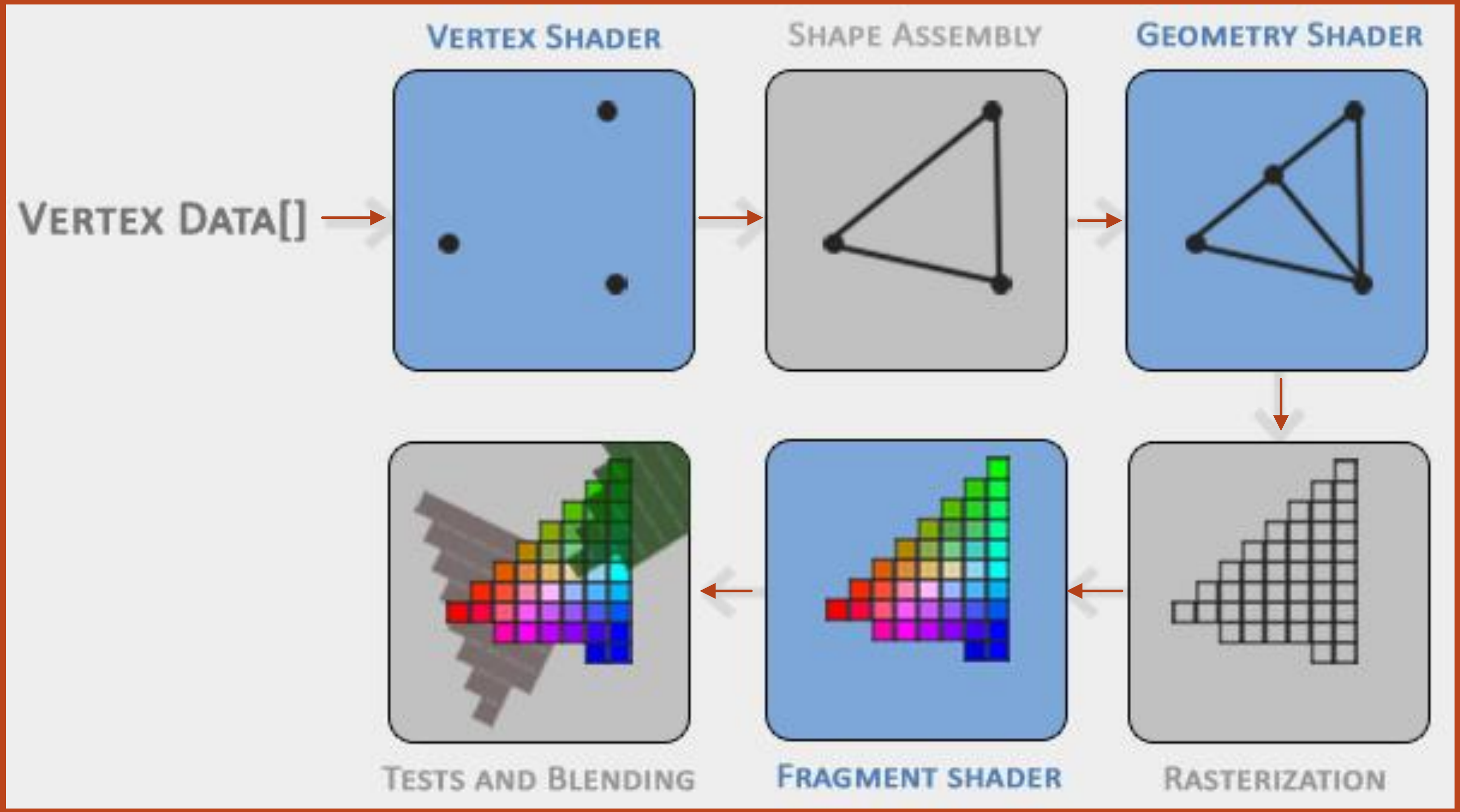
The OpenGL Graphics Pipeline

- Well a simplified version of it



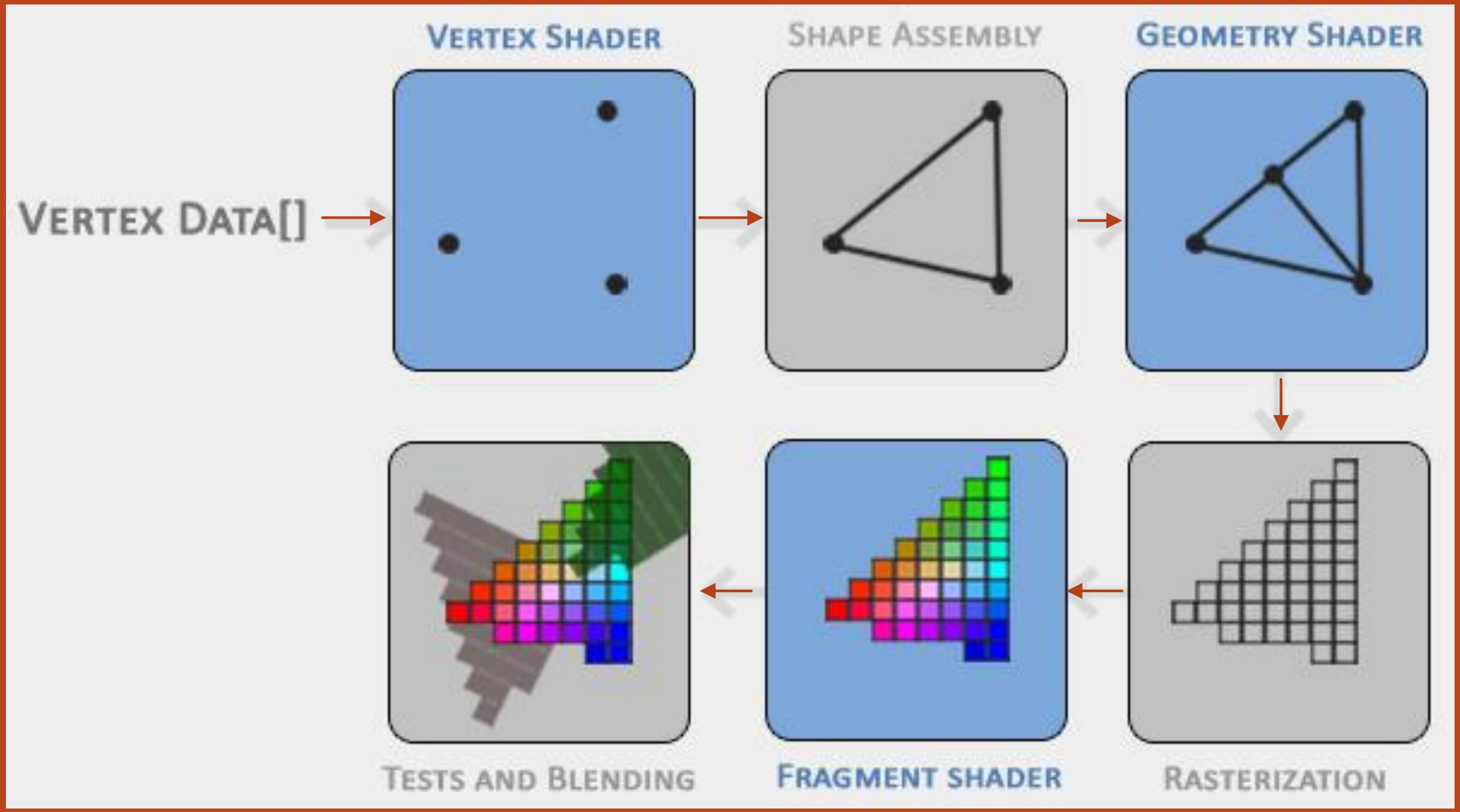
The OpenGL Graphics Pipeline

- The output of each step is input to the next step
- A small program is run on a processing core for each step of the pipeline
- These programs are called shaders



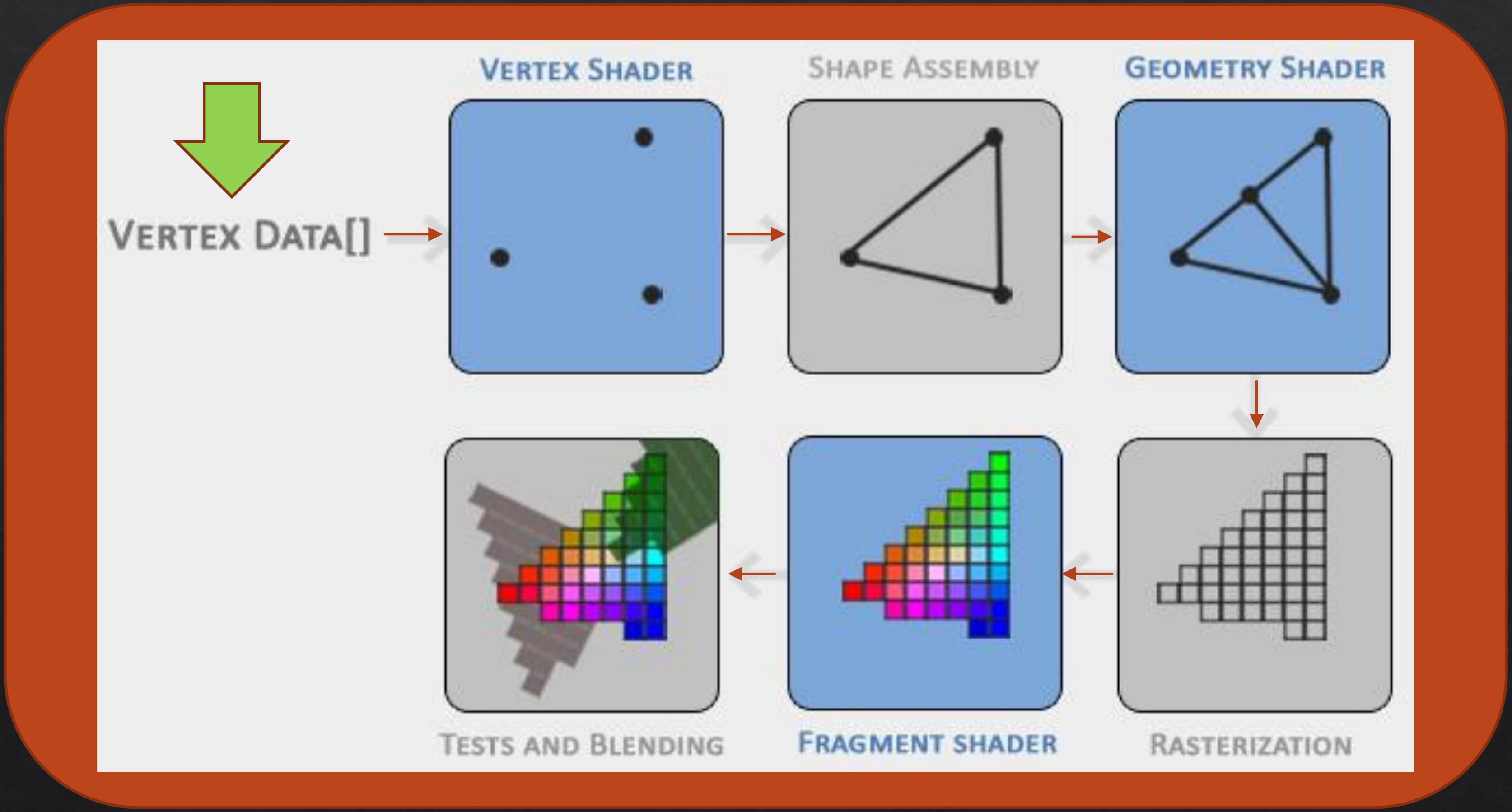
The OpenGL Graphics Pipeline

- The vertex data fed in initially consist usually of 3 or 4 vector coordinates
- This is done with the OpenGL Shading Language
- GLSL



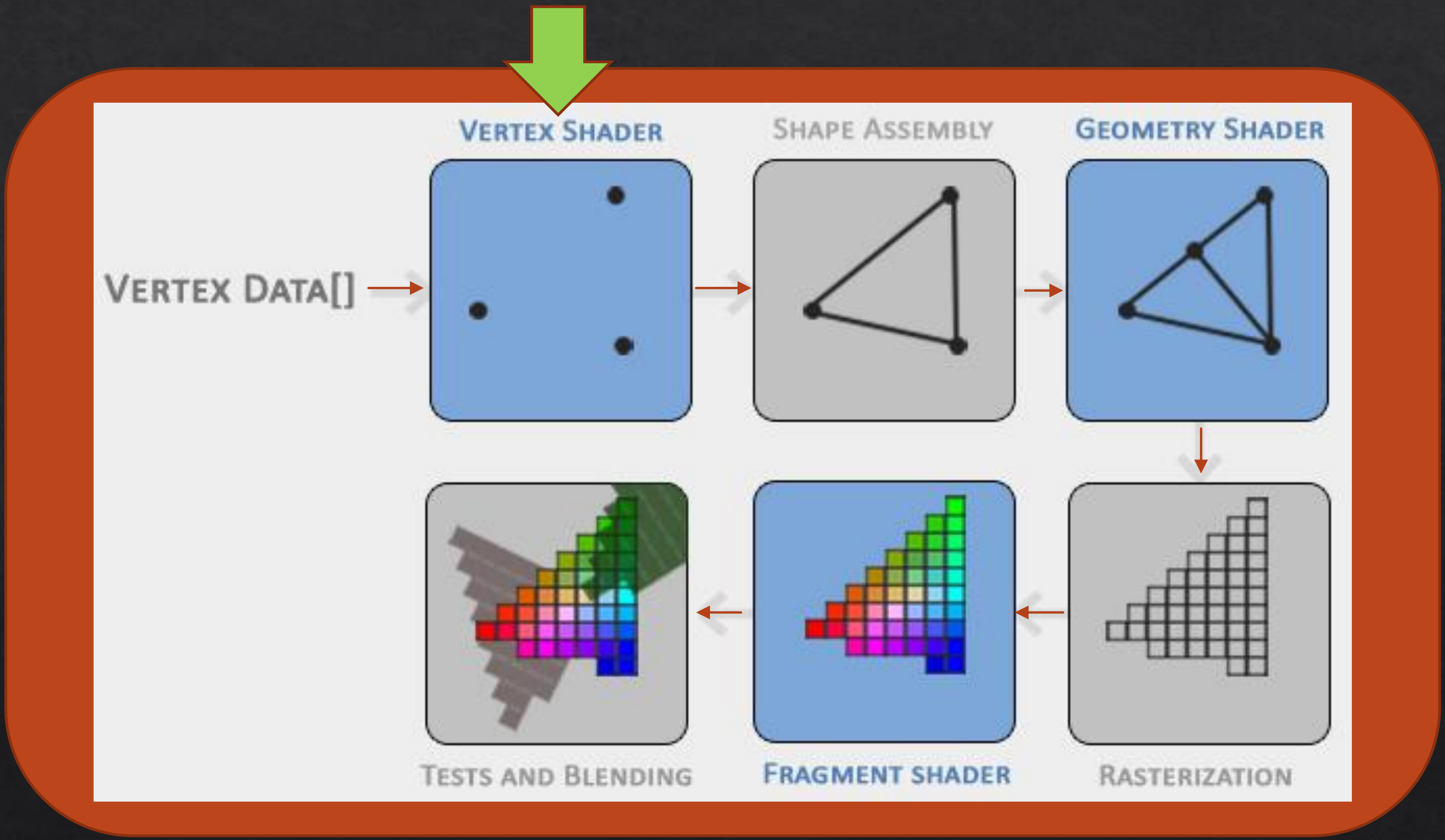
The OpenGL Graphics Pipeline

- The vertex data is an array of coordinates representing points in 3d space



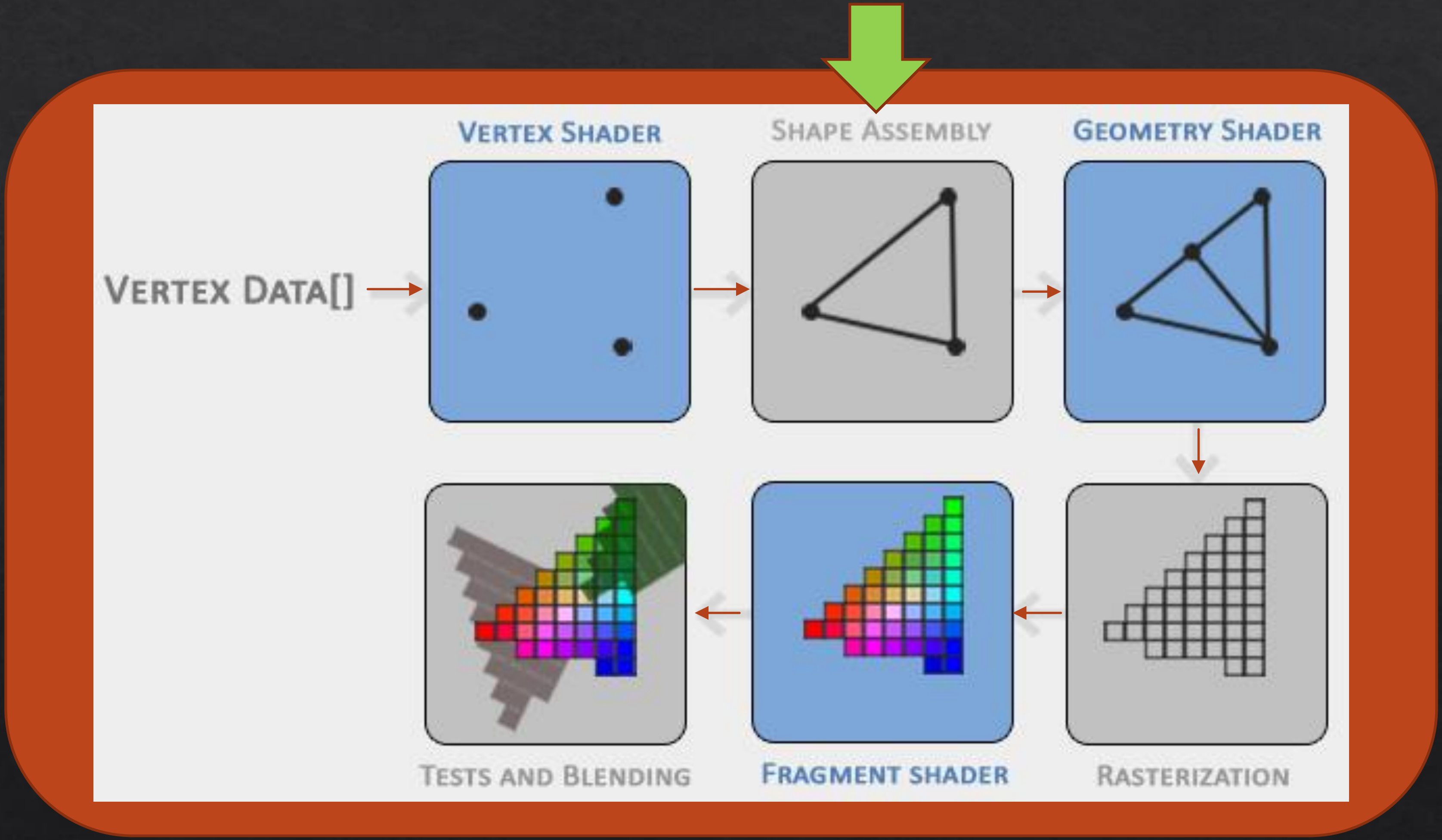
The OpenGL Graphics Pipeline

- The vertex shader is provided by the user
- Its purpose is to calculate 3d transformations that occur



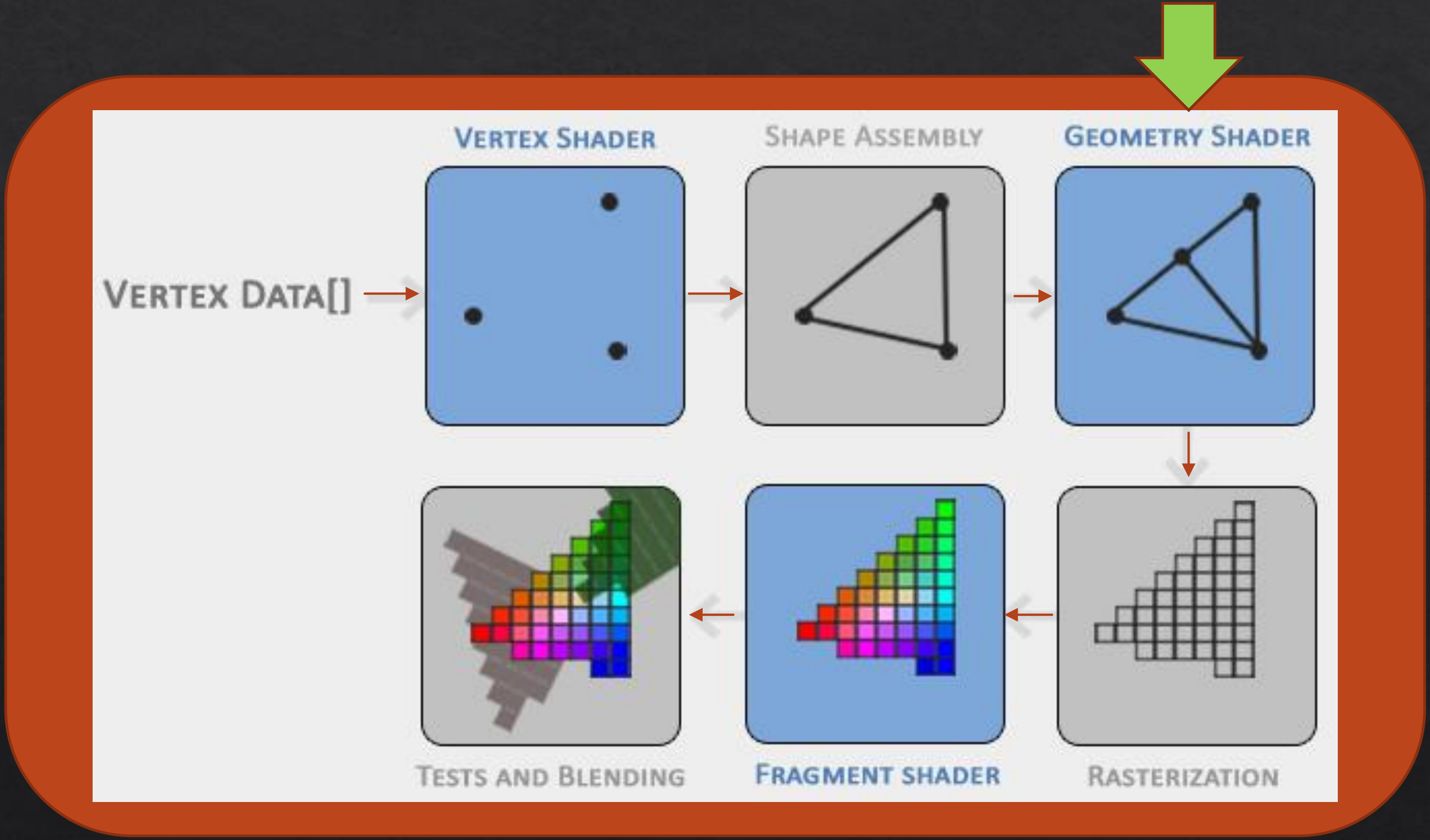
The OpenGL Graphics

- Shape assembly links the 3d points it is provided into primitive shapes
- These shapes are most often triangles in OpenGL
- They can be lines or the 3d points can be left alone



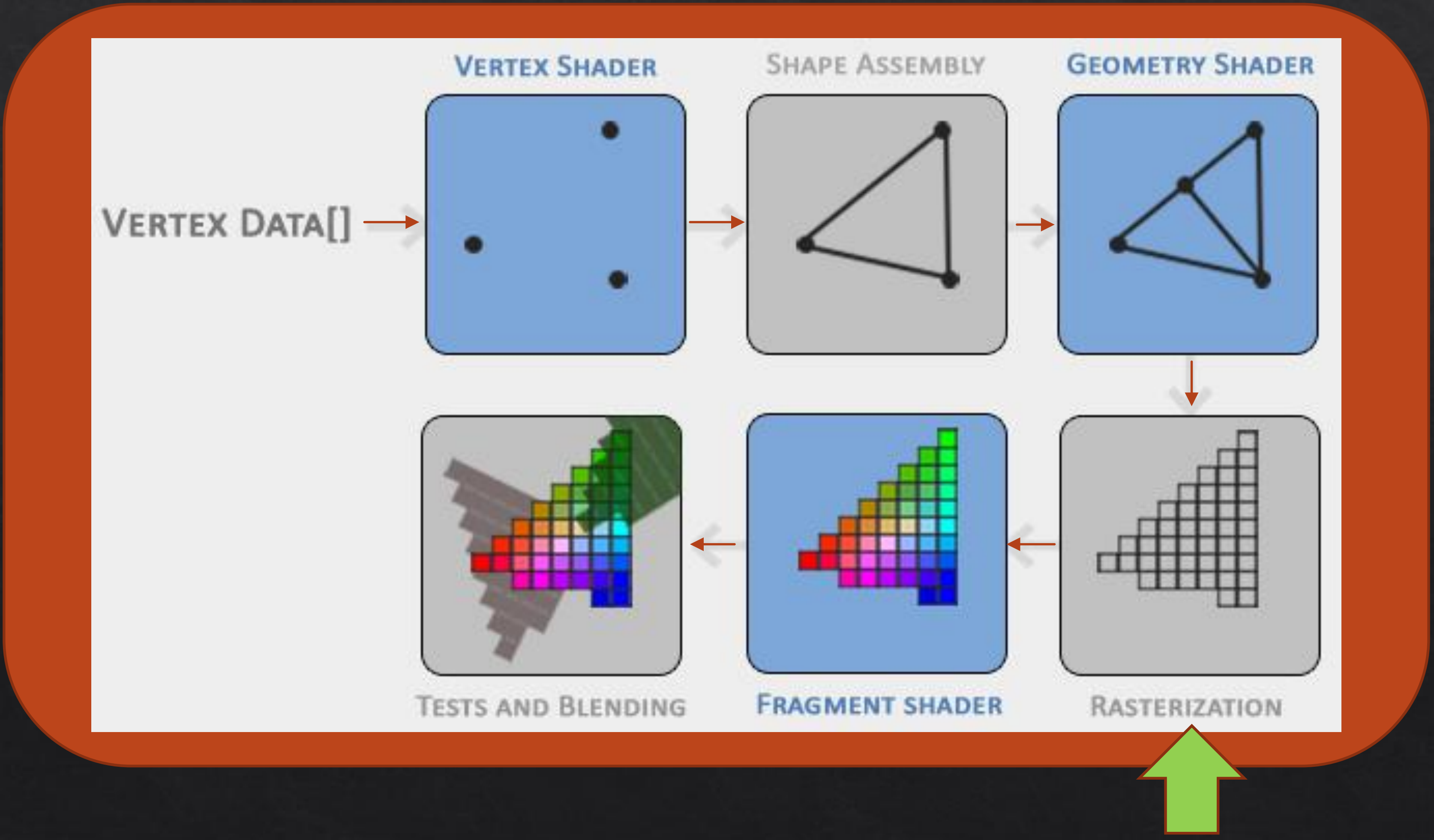
The OpenGL Graphics Pipeline

- The Geometry shader takes primitive shapes and can form more complex ones
- It can be defined by the developer but has a default shader program as well



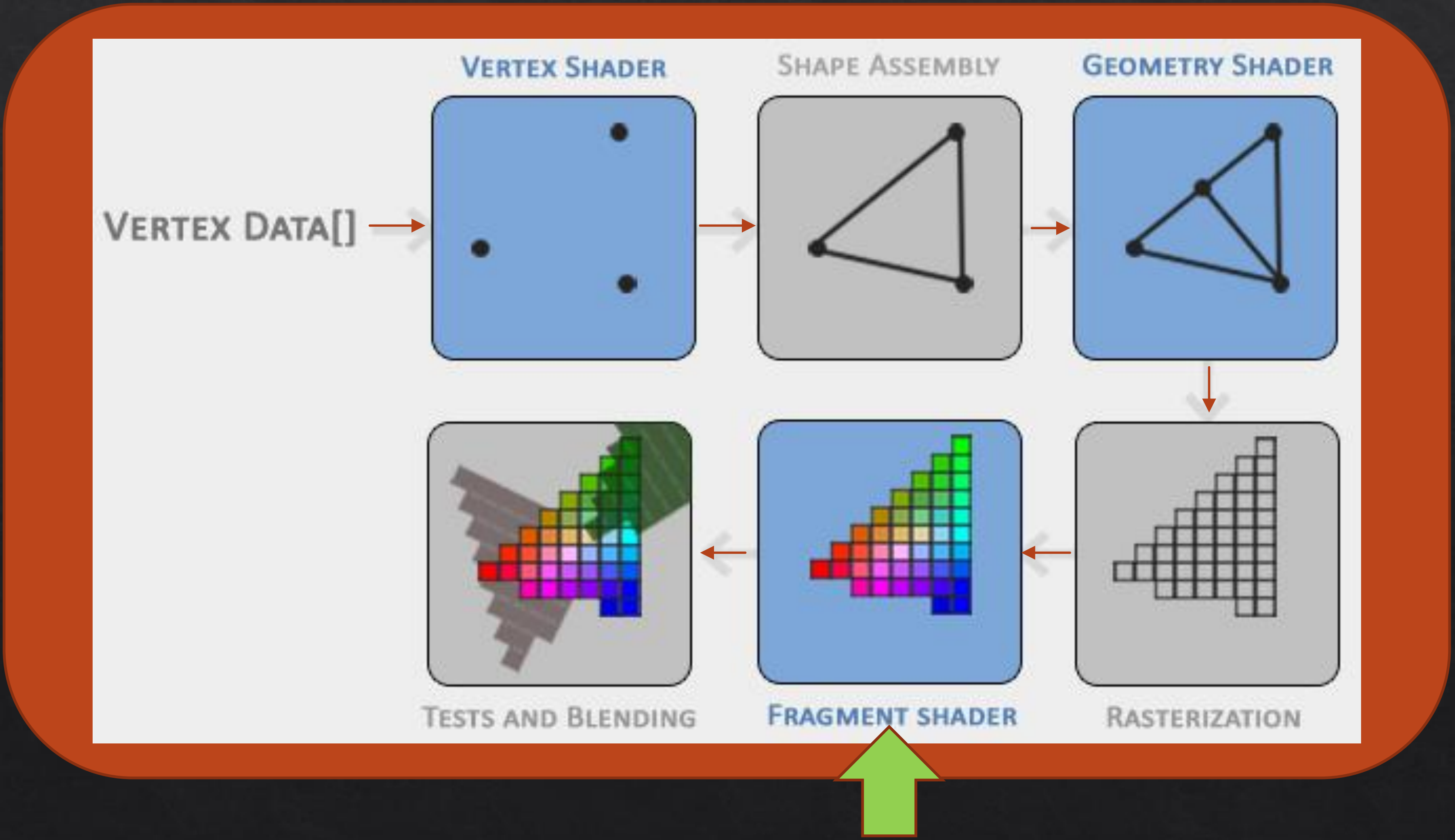
The OpenGL Graphics Pipeline - Data

- Rasterization maps the primitive shapes to their corresponding pixels on the final display
- It also performs clipping, removing ay primitive shapes outside of the display, increasing performance



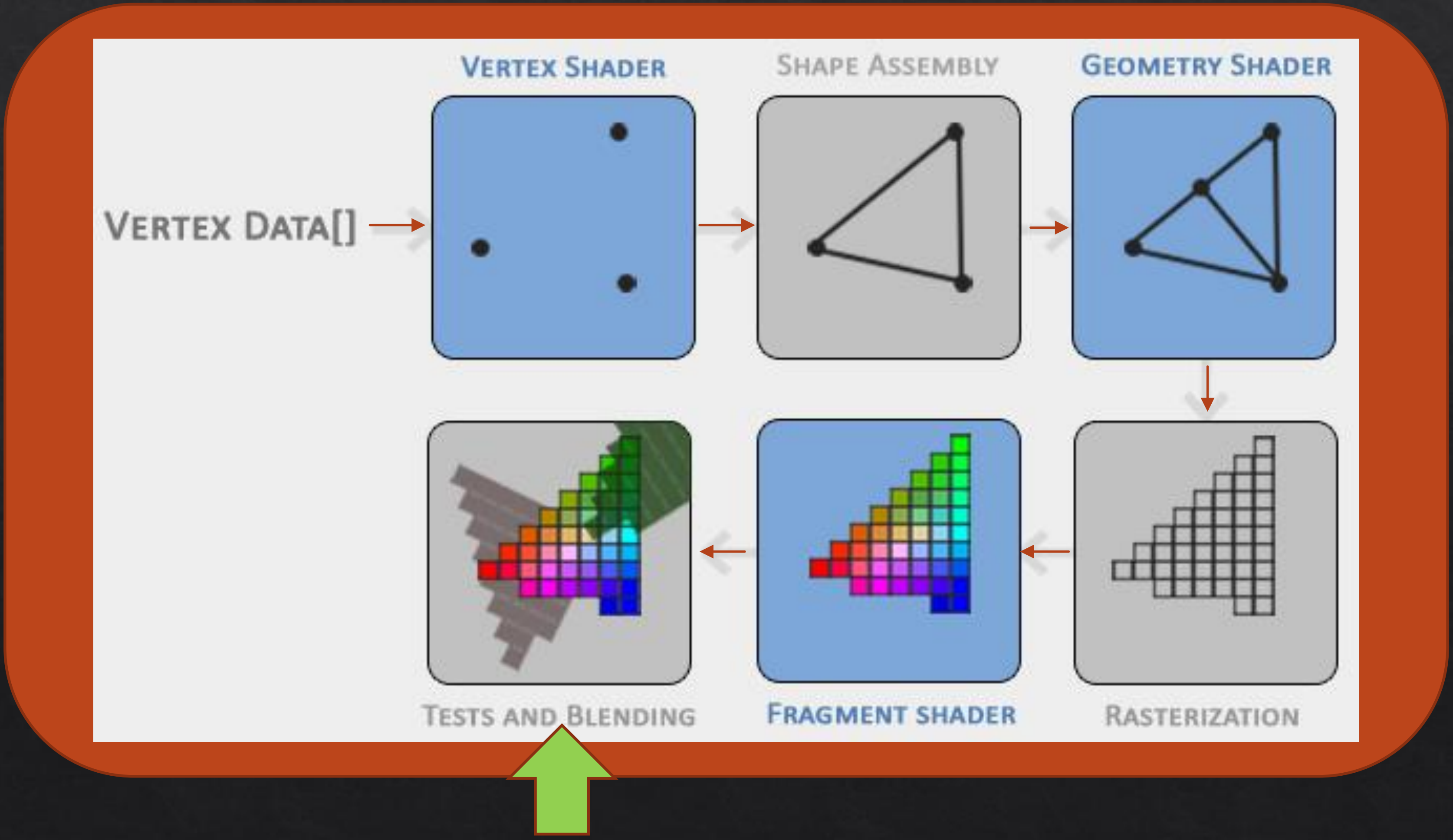
The OpenGL Graphics Pipeline - Data

- The fragment shader decides what color each pixel is going to be
- This shader needs to be defined by the developer
- Calculations like light, shadows, color of light, etc would be performed here



The OpenGL Graphics Pipeline - Data

- Tests and Blending checks what pixels are behind other pixels
- It also applies opacity to pixels



Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. Send OpenGL data
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read our data
  5. Set the Context for the rendering
  6. Render the triangle
- Code will be shown when helpful
- This example was done using C++

Example: Drawing a Triangle

- Steps
  1. **Create a window and OpenGL context**
  2. Send OpenGL data
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read our data
  5. Set the Context for the rendering
  6. Render the triangle

Create A Window and OpenGL Context

- Can be done with a library called **GLFW**
- It will perform all the necessary OS calls to set up a window and context
- GLFW also accounts for user input like window resizing or escapes
- There are many alternatives to **GLFW**

Create A Window and OpenGL Context

- With hand waving, GLFW is used to:
  - **Create a window**
  - **Create an OpenGL context**
  - **Handle if the user resizes the window**
  - **Handle if the user quits**
  - **Handle the render loop**
- The code is long and not that interesting

Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. **Send OpenGL data**
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read the data
  5. Set the Context for the rendering
  6. Render the triangle

Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. **Send OpenGL data**
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read the data
  5. Set the Context for the rendering
  6. Render the triangle

These last two steps are performed repeatedly within the render loop

Send OpenGL data

- Here is the data we are going to send OpenGL

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

- It consists of three 3d points
- Even though we are only drawing a 2d triangle, OpenGL only works in 3d

Send OpenGL data

- Data is sent to the GPU and set to the current context through objects
- To do this create a buffer object

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

- Bind it to the OpenGL context variable GL_ARRAY_BUFFER

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

- Then put the data into the buffer currently bound to GL_ARRAY_BUFFER

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

Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. Send OpenGL data
  3. **Create a Vertex shader and Fragment shader**
  4. Tell OpenGL how to read the data
  5. Set the Context for the rendering
  6. Render the triangle

Create a Vertex and Fragment shader

- The Vertex and Fragment shaders need to be provided by the developer

Create a Vertex and Fragment shader

- The Vertex shader will work as a pipe

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

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

- Taking the input and sending it to the output

Create a Vertex and Fragment shader

- The Fragment shader will just output the color orange

```
#version 330 core
out vec4 FragColor;

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

Create a Vertex and Fragment shader

- To compile the shaders you create a shader object

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

- The code is then given to the shader object in the form of C string

```
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";
```

- And compiled

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

Create a Vertex and Fragment shader

- After the shaders are compiled they are linked into a program

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

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

- And are ready for use

Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. Send OpenGL data
  3. Create a Vertex shader and Fragment shader
  4. **Tell OpenGL how to read the data**
  5. Set the Context for the rendering
  6. Render the triangle

Tell OpenGL how to read the data

- The vertices data did not need to be formatted this way

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

- OpenGL does not specify how vertex attributes should be inputted to the vertex shader
- So we need to tell it how to read the data given

Tell OpenGL how to read the data

- This is done through a vertex attribute pointer

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

Tell OpenGL how to read the data

- This is done through a vertex attribute pointer

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

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

- This links the currently bound array buffer

Tell OpenGL how to read the data

- This is done through a vertex attribute pointer

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

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

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

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

- This links the currently bound array buffer to the input variables of the vertex shader

Tell OpenGL how to read the data

- This is done through a vertex attribute pointer

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

- It also specifies how the data should be interpreted

Tell OpenGL how to read the data

- We can have multiple Vertex buffer objects, each with multiple vertex attribute pointers
- This could be a nightmare to manager
- To track all of this OpenGL gives Vertex Array Objects
- VAO’s automatically keep track of the connections between VBOs and attribute pointers
- OpenGL will not render without them

Tell OpenGL how to read the data

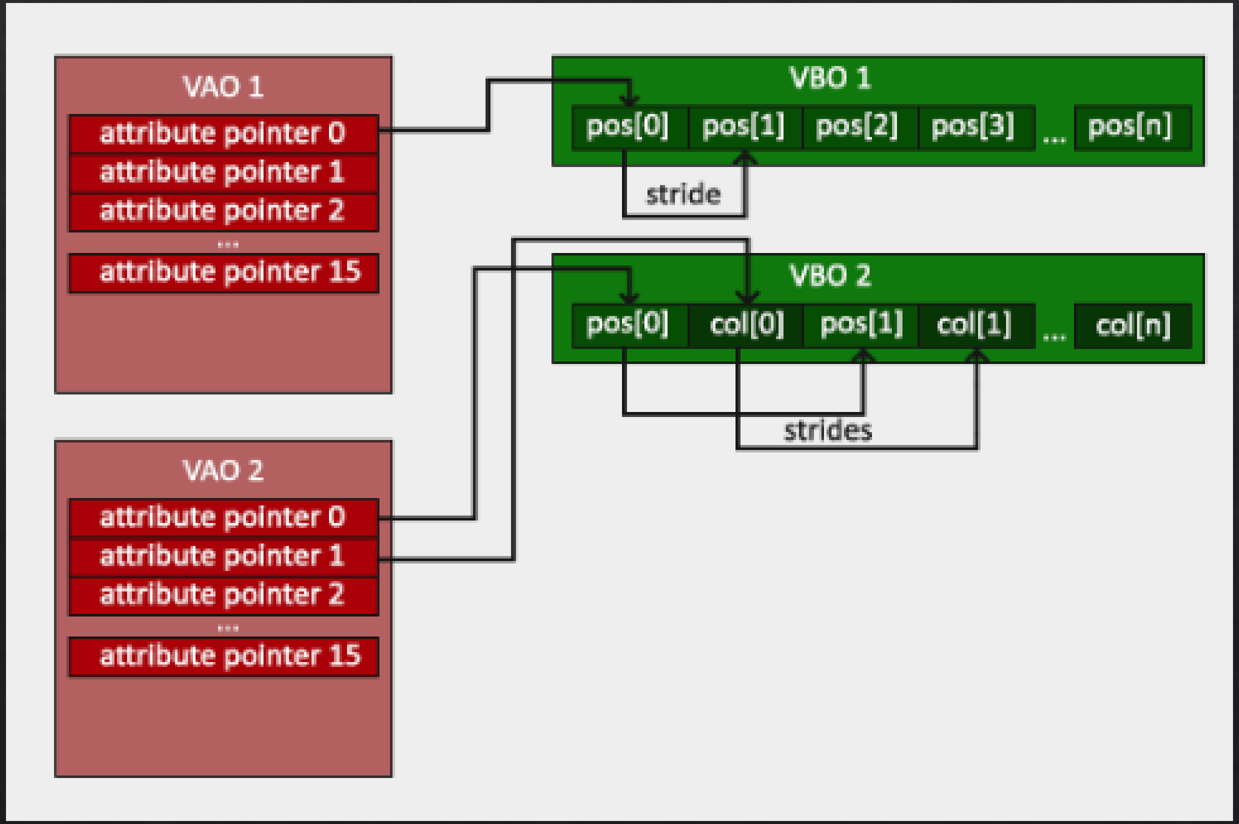
- To use them we create them, bind them to the current context, and then initialize the data like usual

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

- OpenGL automatically makes the connections

Tell OpenGL how to read the data

- Vertex array objects can be seen as containers linking the data (VBO) with how it is interpreted (attribute pointers)



Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. Send OpenGL data
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read the data
  5. **Set the Context for the rendering**
  6. Render the triangle

Set the Context for Rendering

- The hard part is done
- The shader program is complete
- The vertex array object has the data and how it should interpret it
- All that's needed is to set the OpenGL context to them

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
```

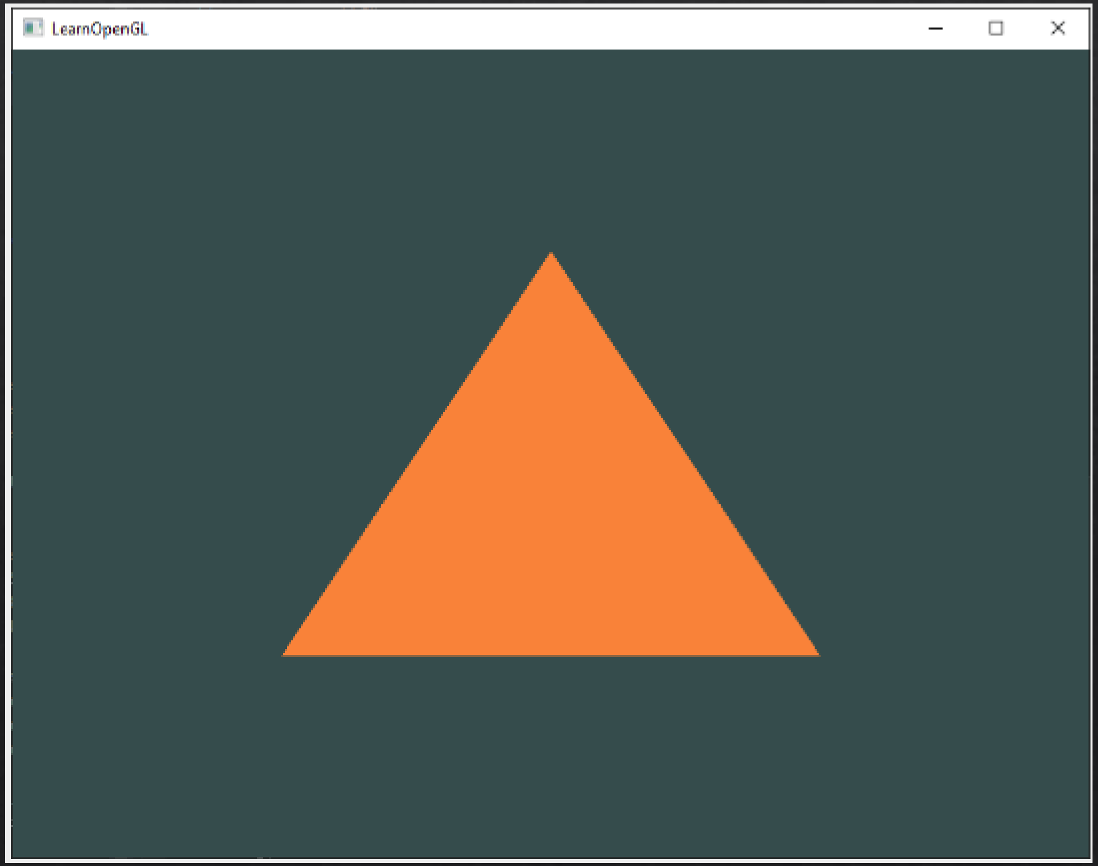
Example: Drawing a Triangle

- Steps
  1. Create a window and OpenGL context
  2. Send OpenGL data
  3. Create a Vertex shader and Fragment shader
  4. Tell OpenGL how to read the data
  5. Set the Context for the rendering
  6. **Render the triangle**

Render the Triangle

- Easy

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```



References

- De Vries, Joey. (2020). *Learn OpenGL – Computer Graphics.* Kendall & Welling Publishing.
- Free online at: https://learnopengl.com/
- Microsoft. (2018, 05 31). *Windows GDI*. Docs.microsoft.com. https://docs.microsoft.com/en-us/windows/win32/gdi/windows-gdi
- Microsoft. (2018, 05 31). *Direct 3d 11 Graphics*. Docs.microsoft.com. https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11
- Khronos Group. *Vulcan Overview*. Khronos.org. https://www.khronos.org/vulkan/