CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

2.2 Transformaciones Geométricas en 2D

2.2.2 Shaders

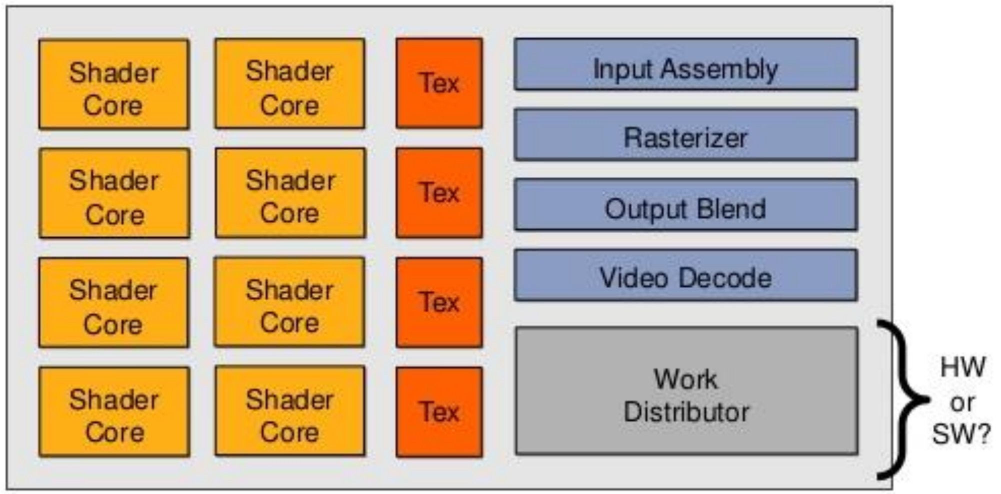
Shaders

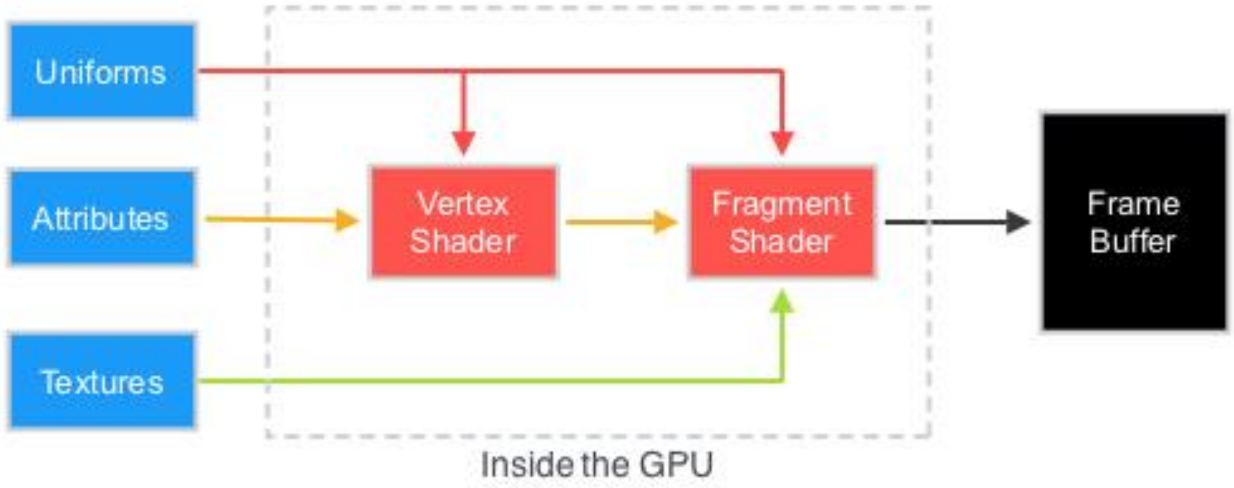
- Shaders are little programs that rest on the GPU. These programs are run for each specific section of the graphics pipeline.

- In a basic sense, shaders are nothing more than programs transforming inputs to outputs.

- Shaders are also very isolated programs in that they're not allowed to communicate with each other; the only communication they have is via their inputs and outputs.

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)





Shaders – OpenGL Shading Language (GLSL)

- GLSL es un lenguaje procedural de alto nivel.
- Desde OpenGL 2.0 es parte del estándar OpenGL.
- Se utiliza el mismo lenguaje, con unas pequeñas diferencias tanto para vertex como para fragment shaders.
- Soporta operaciones con vectores y matrices y tiene funciones "component wise".

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

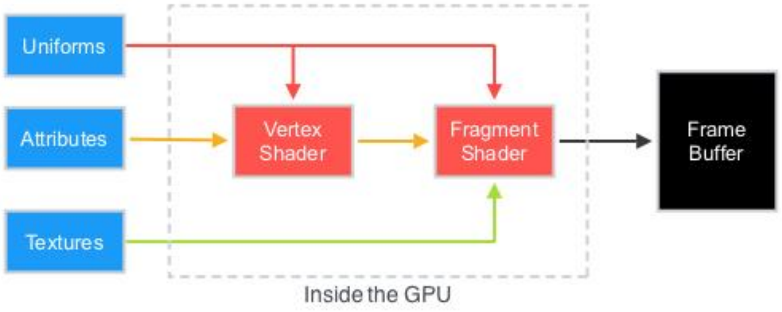
Por qué escribir un Shader?

- Porque a través del API de OpenGL no se puede cambiar la manera en que opera el Pipeline Gráfico ni el orden de las operaciones.
- Si, por ejemplo, se frustró porque OpenGL no le permitía definir la manera en que los cálculos de iluminación se realizan por vértice en vez de por fragmentos.
- O si se encontró con alguna limitación del modelo tradicional de renderizado de OpenGL.

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Vertex Processor

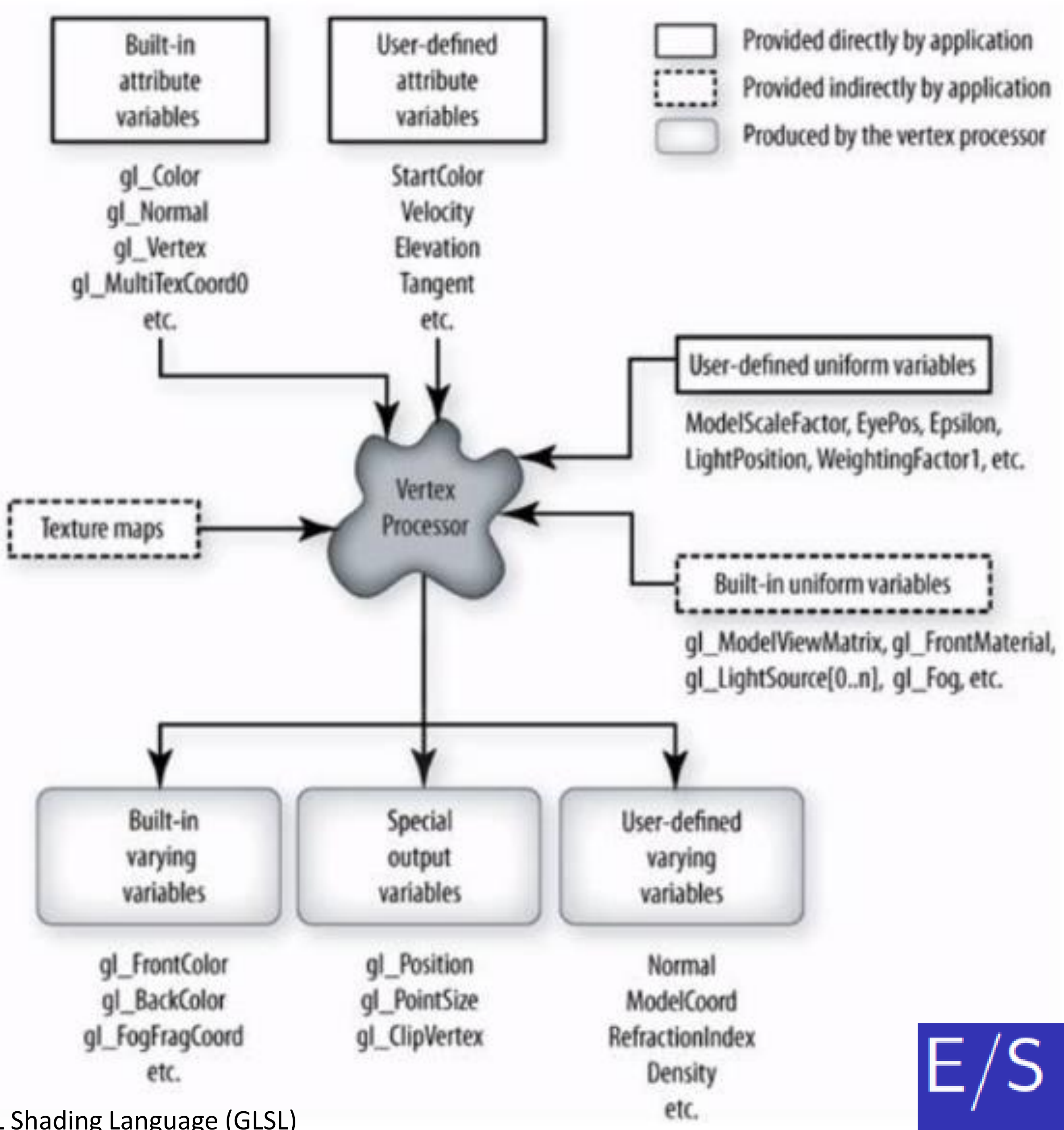


Opera sobre los valores de los vértices y sus datos asociados. Generalmente realiza:

- Transformaciones de vértices.
- Transformación de la normal y normalización.
- Generación y transformación de coordenadas de texturas.
- Iluminación.
- Aplicación del color de materiales.

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

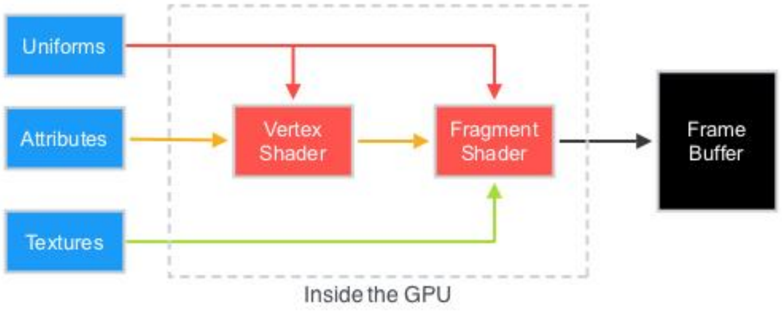


E/S del Vertex Processor

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Fragment Processor

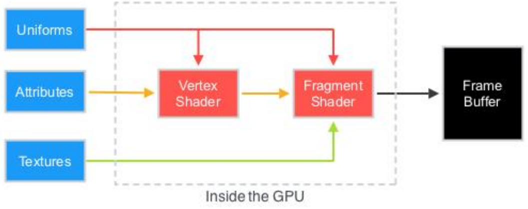


Los fragmentos son estructuras de datos por pixel que son creados por la resterizacion de primitivas gráficas. Opera con valores de fragmentos y sus datos asociados. Generalmente realiza:

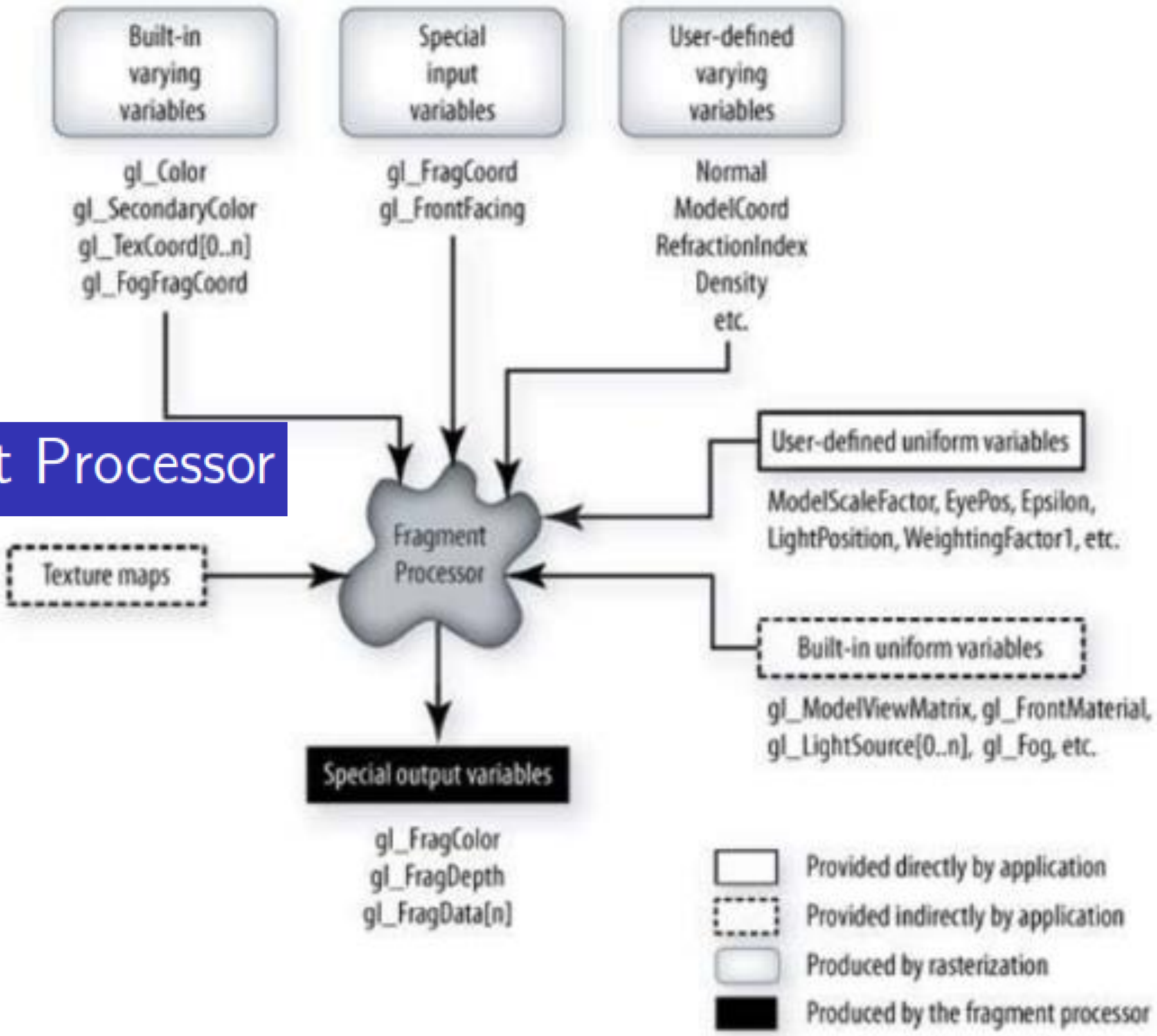
- Operaciones en valores interpolados.
- Acceso a texturas.
- Aplicación de texturas.
- Niebla.
- Suma de colores.

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)



E/S del Fragment Processor



Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Descripción básica del lenguaje

- Basado en la sintaxis de ANSI C.
- El punto de entrada de un shader es la funcion `void main();` cuyo cuerpo se delimita con llaves. Las constantes, operadores, identificadores, expresiones y declaraciones, el control de flujo para lazos, if-then-else y las llamadas a funciones son básicamente identicos a C.
- Se añaden tipos de datos vectoriales y matriciales, tales como: **vec2** (dos float), **vec3**, **vec4**, **mat2**, **mat3**, **mat4**.

Shaders – OpenGL Shading Language (GLSL)

Descripción básica del lenguaje

- *Samplers* para acceder a texturas. **Sampler1D** y **Sampler2D** para texturas en 1D y 2D respectivamente.
- Calificativos para especificar que tipo de entrada o salida realiza una variable: **attribute**, **uniform** y **varying**.
  - **attribute**: comunica un valor que cambia frecuentemente, desde la aplicación al vertex shader.
  - **uniform**: comunica un valor que no cambia frecuentemente, desde la aplicación a cualquier shader.
  - **varying**: comunica un valor interpolado (resultado de rasterización de primitivas) desde el vertex shader al fragment shader.
- Las variables predefinidas comienzan con **"gl_"**. Por ejemplo: **gl_ModelViewMatrix**, **gl_LightSource[i]**, **gl_Fog.Color**.

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Funciones incorporadas

- Trigonométricas: seno, coseno, tangente, etc.
- Exponenciales: potencia, logaritmo, raiz cuadrada, etc.
- Floor, ceiling, parte fraccionaria, módulo, etc.
- Geométricas: distancia, producto punto, producto cruz, normalización, etc.
- Racionales y vectorcomponent-wise: mayor que, menor que, igual a, etc.
- Funciones especializadas del fragment shader para calcular las derivadas y estimar el ancho de filtros para antialiasing.
- Funciones para acceder a los valores de las texturas en memoria.
- Y más...

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Hardware

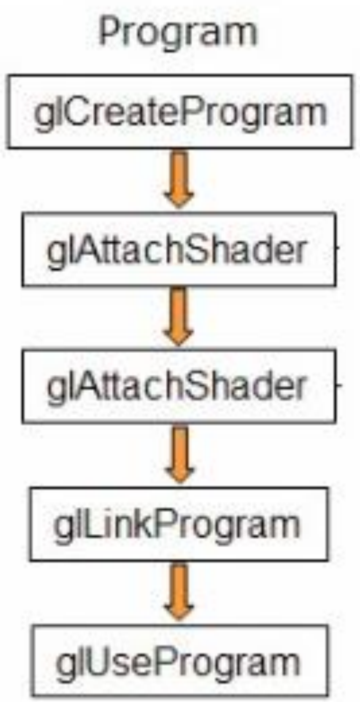
Para correr un programa que utilice shaders se necesita una GPU que acepte el lenguaje de shading. Casi todas las tarjetas gráficas desde la GeForce3 soportan shaders. Se necesita al menos una Nvidia GeForce 5200 o una ATI 9500 para trabajar correctamente con OpenGL 2.x. Asegurese que los drivers de su GPU estén actualizados, el soporte OpenGL y el compilador de Shaders estan integrados en los drivers.

Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

Shaders – OpenGL Shading Language (GLSL)

Código OpenGL y C

GLSL necesita un programa OpenGL. Tambien hay que activar el lenguaje de Shading. A continuación se presenta un esquema con los pasos necesarios para activar los shaders.



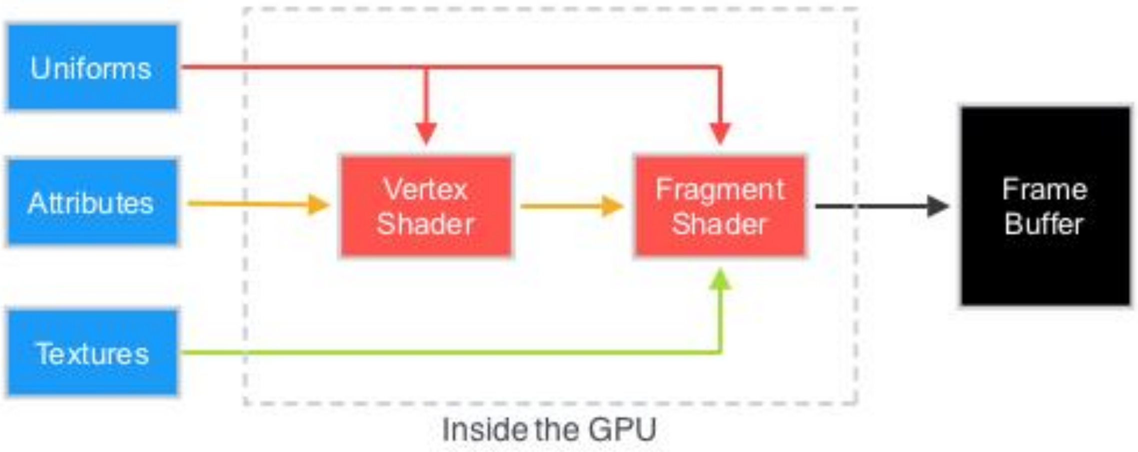
Fuente: Guido Sanchez – Introduction a OpenGL Shading Language (GLSL)

GLSL Syntax

- GLSL is like C without
  - pointers
  - recursion
  - dynamic memory allocation
- GLSL is like C with
  - Built-in vector, matrix, and sampler types
  - Constructors
  - A great math library

Language features allow us to write concise, efficient shaders.

Shaders – OpenGL Shading Language (GLSL)



- Shaders are written in the C-like language GLSL. GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

- Shaders always begin with a version declaration, followed by a list of input and output variables, uniforms and its main function.

- Each shader's entry point is at its main function where we process any input variables and output the results in its output variables.

A shader typically has the following structure:

```
#version version_number
in type in_variable_name1;
in type in_variable_name2;

out type out_variable_name1;

uniform type uniform_name1;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

GLSL – Vertex Shader

In **vertex shader** each input variable is also known as a vertex attribute.

There is a maximum number of vertex attributes we're allowed to declare limited by the hardware.

OpenGL guarantees there are always at least 16 4-component vertex attributes available, but some hardware might allow for more which you can retrieve by querying GL_MAX_VERTEX_ATTRIBS:

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes << std::endl;
```

This often returns the minimum of 16 which should be more than enough for most purposes.

Shaders – GLSL- Types

- GLSL has most of the default basic types we know from languages like C: **int, float, double, uint and bool**.
- GLSL also features two container types that we'll be using a lot throughout the tutorials, namely **vectors and matrices**.

- Scalar types: `float`, `int`, `uint`, and `bool`
- Vectors are also built-in types:
  - `vec2`, `vec3`, and `vec4`
  - Also `ivec*`, `uvec*`, and `bvec*`
- Access components three ways:
  - `.x, .y, .z, .w` ← Position or direction
  - `.r, .g, .b, .a` ← Color
  - `.s, .t, .p, .q` ← Texture coordinate

Shaders – GLSL- Vectors

A vector in GLSL is a 1,2,3 or 4 component container for any of the basic types just mentioned. They can take the following form (n represents the number of components):

- **vecn**: the default vector of n floats.
- **bvecn**: a vector of n booleans.
- **ivecn**: a vector of n integers.
- **uvecn**: a vector of n unsigned integers.
- **dvecn**: a vector of n double components.

- Scalar types: `float`, `int`, `uint`, and `bool`
- Vectors are also built-in types:
  - `vec2`, `vec3`, and `vec4`
  - Also `ivec*`, `uvec*`, and `bvec*`
- Access components three ways:
  - `.x, .y, .z, .w` ← Position or direction
  - `.r, .g, .b, .a` ← Color
  - `.s, .t, .p, .q` ← Texture coordinate

Most of the time we will be using the basic **vecn** since floats are sufficient for most of our purposes.

Shaders – GLSL- Vectors

- Components of a vector can be accessed via **vec.x** where x is the first component of the vector.

- You can use **.x, .y, .z** and **.w** to access their first, second, third and fourth component respectively.

- GLSL also allows you to use **rgba** for colors or **stpq** for texture coordinates, accessing the same components.

- Scalar types: `float`, `int`, `uint`, and `bool`
- Vectors are also built-in types:
  - `vec2`, `vec3`, and `vec4`
  - Also `ivec*`, `uvec*`, and `bvec*`
- Access components three ways:
  - `.x, .y, .z, .w` ← Position or direction
  - `.r, .g, .b, .a` ← Color
  - `.s, .t, .p, .q` ← Texture coordinate

Shaders – GLSL- Vectors - Swizzling

- *Swizzle*: select or rearrange components

```
vec4 c = vec4(0.5, 1.0, 0.8, 1.0);

vec3 rgb = c.rgb;  // [0.5, 1.0, 0.8]

vec3 bgr = c.bgr;  // [0.8, 1.0, 0.5]

vec3 rrr = c.rrr;  // [0.5, 0.5, 0.5]

c.a = 0.5;         // [0.5, 1.0, 0.8, 0.5]
c.rb = 0.0;        // [0.0, 1.0, 0.0, 0.5]

float g = rgb[1];  // 1.0, indexing, not swizzling
```

Shaders – GLSL- Vectors - Swizzling

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

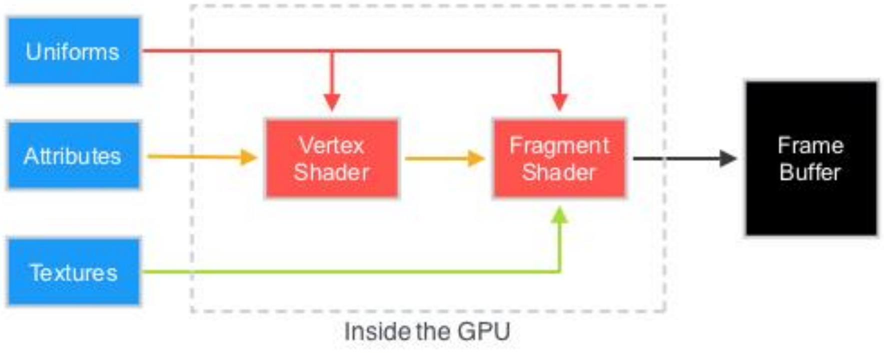
We can also pass vectors as arguments to different vector constructor calls, reducing the number of arguments required:

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

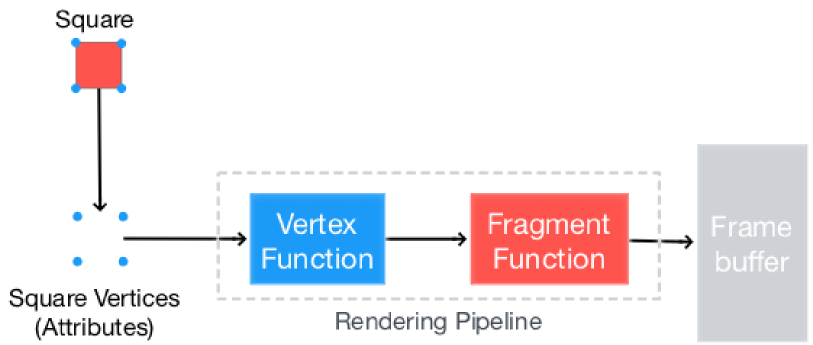
Vectors are thus a flexible datatype that we can use for all kinds of input and output.

Shaders – GLSL- Ins and Outs

- Each shader can specify inputs and outputs using keywords and wherever an output variable matches with an input variable of the next shader stage they're passed along.

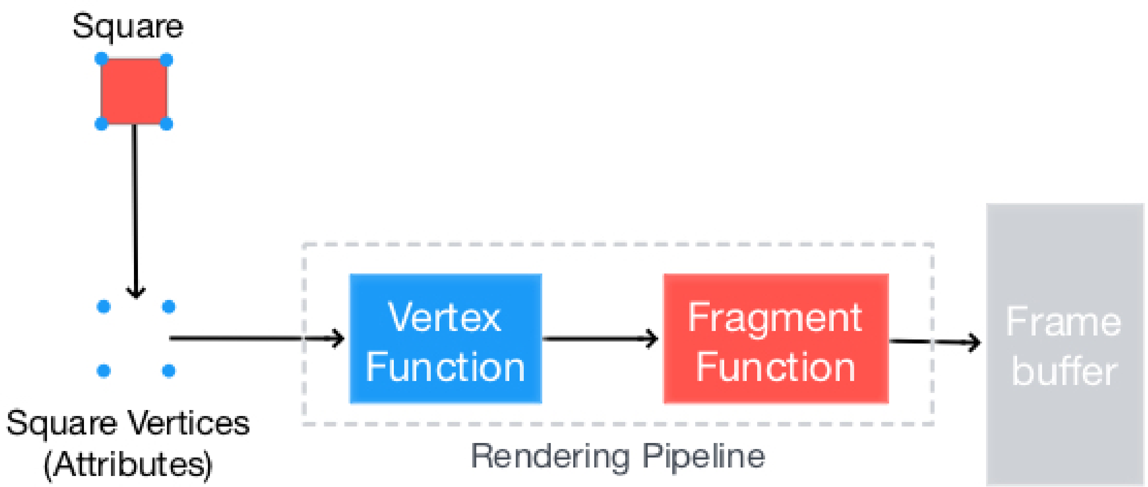


- The vertex shader should receive some form of input otherwise it would be pretty ineffective.
- The vertex shader differs in its input, in that it receives its input straight from the vertex data.
- To define how the vertex data is organized we specify the input variables with location metadata so we can configure the vertex attributes on the CPU.



Shaders – GLSL- Ins and Outs

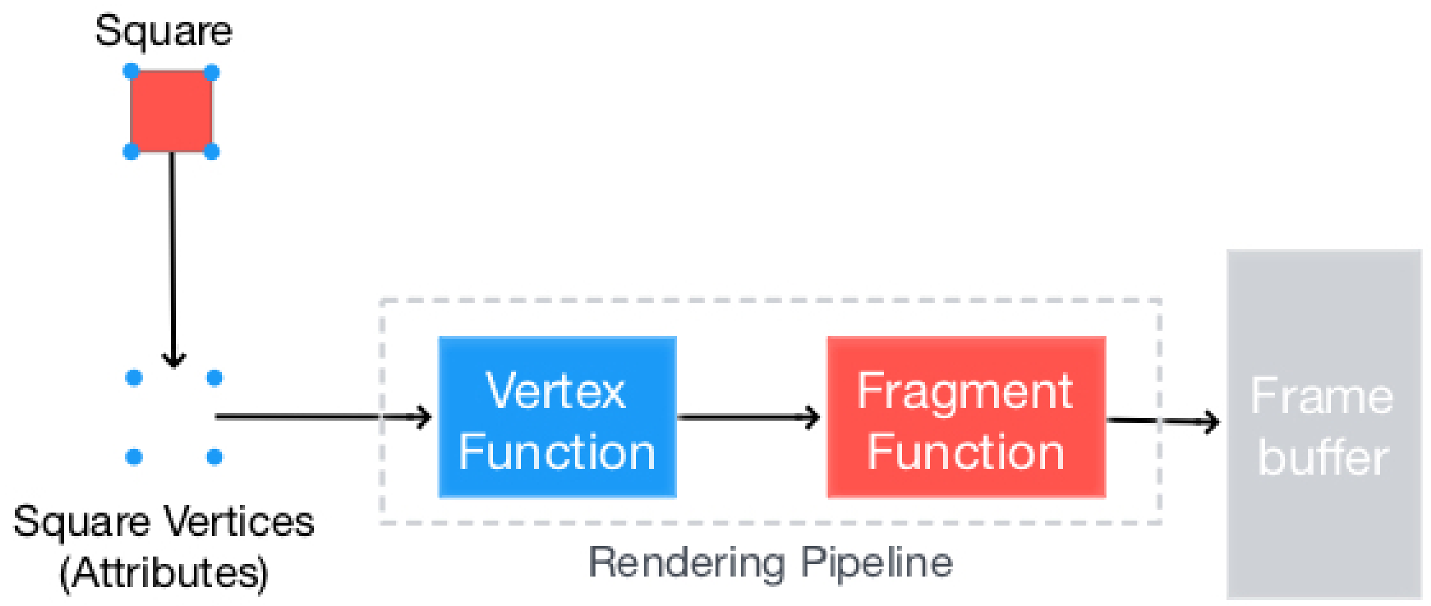
The vertex shader thus requires an extra layout specification for its inputs so we can link it with the vertex data. layout (location = 0).



The fragment shader requires a vec4 color output variable, since the fragment shaders needs to generate a final output color.

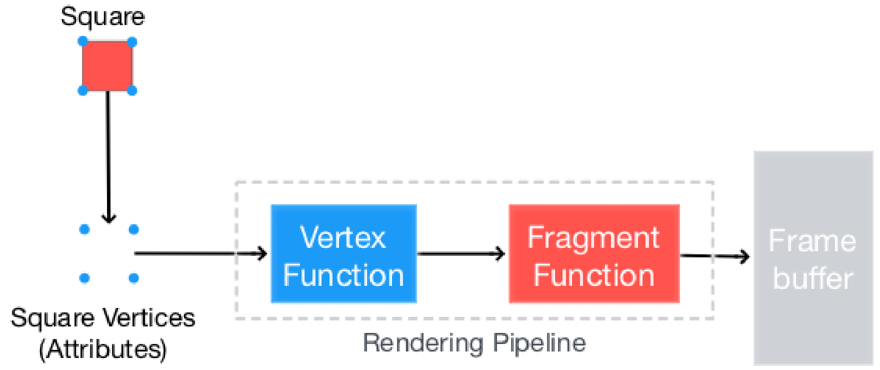
If you fail to specify an output color in your fragment shader, the color buffer output for those fragments will be undefined (which usually means OpenGL will render them either black or white).

Shaders – GLSL- Ins and Outs



- So if we want to send data from one shader to the other we'd have to declare an output in the sending shader and a similar input in the receiving shader.
- When the types and the names are equal on both sides OpenGL will link those variables together and then it is possible to send data between shaders (this is done when linking a program object).

Shaders – GLSL- Ins and Outs



Example:

Alter the shaders from the previous example to let the vertex shader decide the color for the fragment shader

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
   gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
   vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

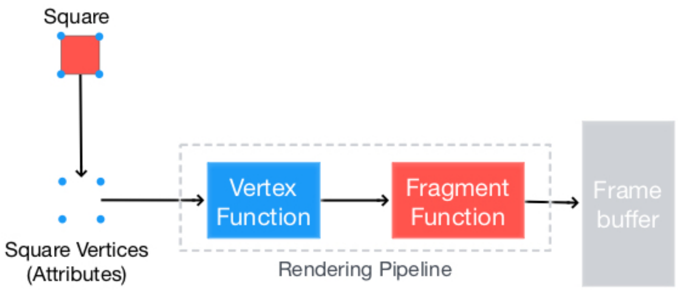
Fragment Shader

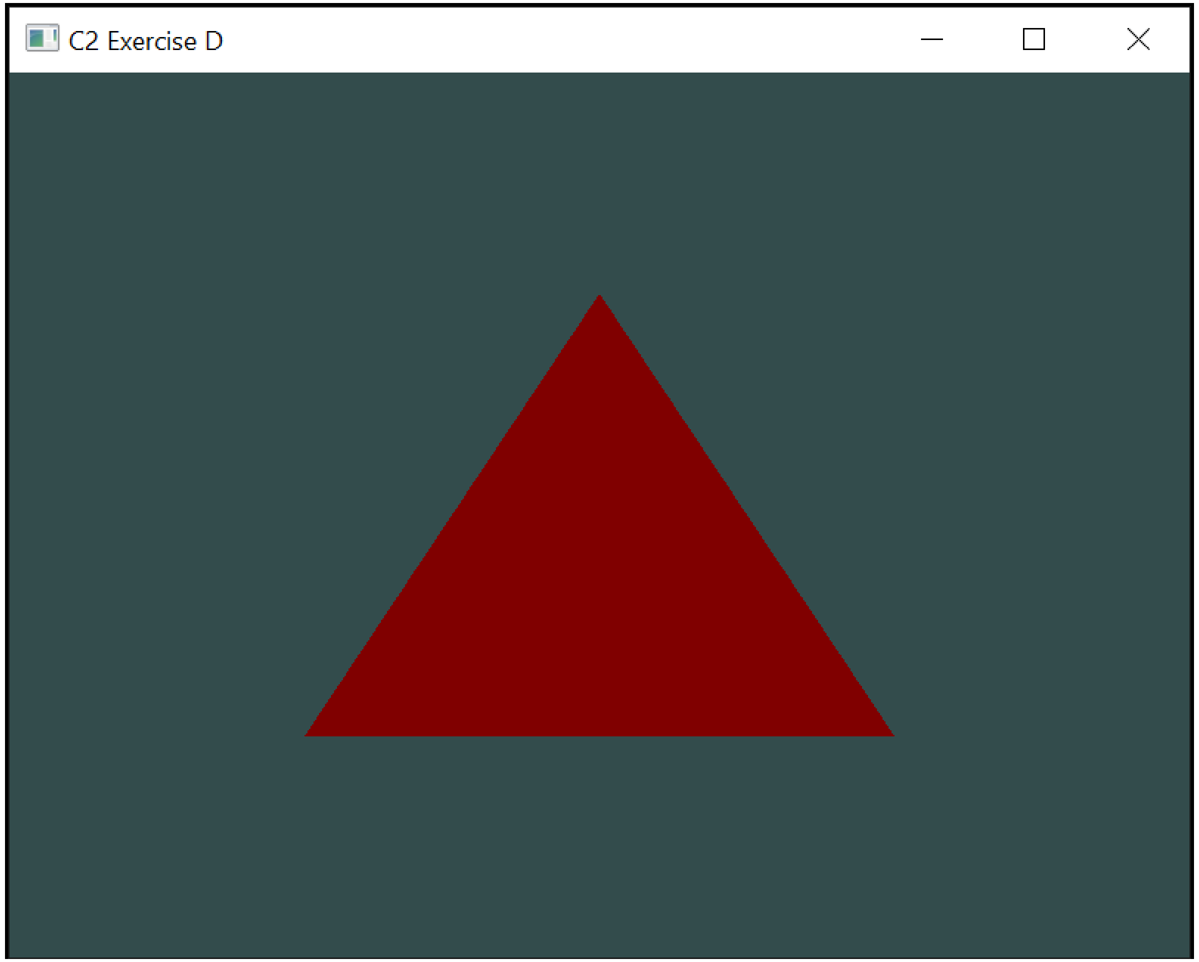
```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)

void main()
{
   FragColor = vertexColor;
}
```

Shaders – GLSL- Ins and Outs

Example:

Exercise 4: Modify the code of the shaders of the First Triangle.





Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
   gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
   vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

Fragment Shader

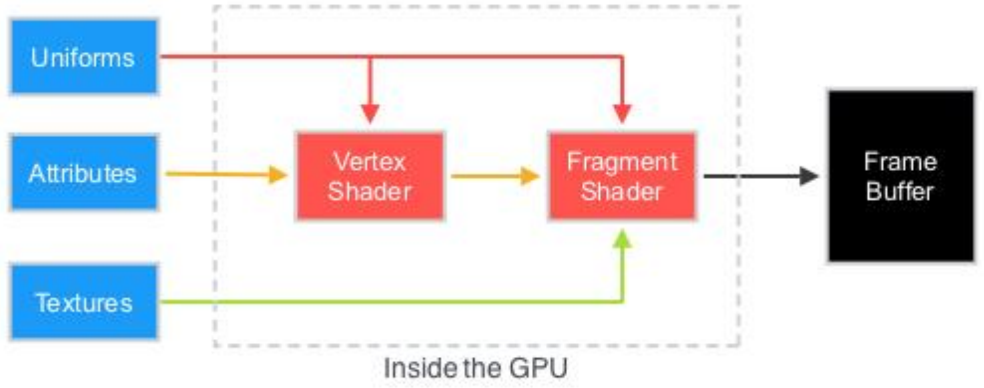
```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)

void main()
{
   FragColor = vertexColor;
}
```

Shaders – GLSL- Uniforms

Next Challenge:

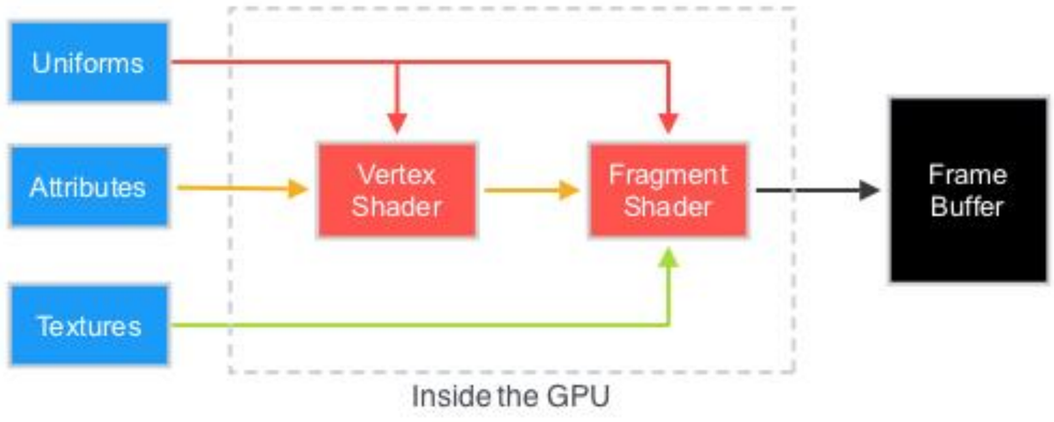
Send Information (a color) from our application to the fragment shader!

- **UNIFORMS** are another way to pass data from our application on the CPU to the shaders on the GPU.
- Uniforms are however slightly different compared to vertex attributes. First of all, **uniforms are global**.
- Global, meaning that a uniform variable is **unique per shader program object**, and can be accessed from any shader at any stage in the shader program.
- Second, whatever you set the uniform value to, uniforms will **keep their values until they're either reset or updated**.



Shaders – GLSL- Uniforms

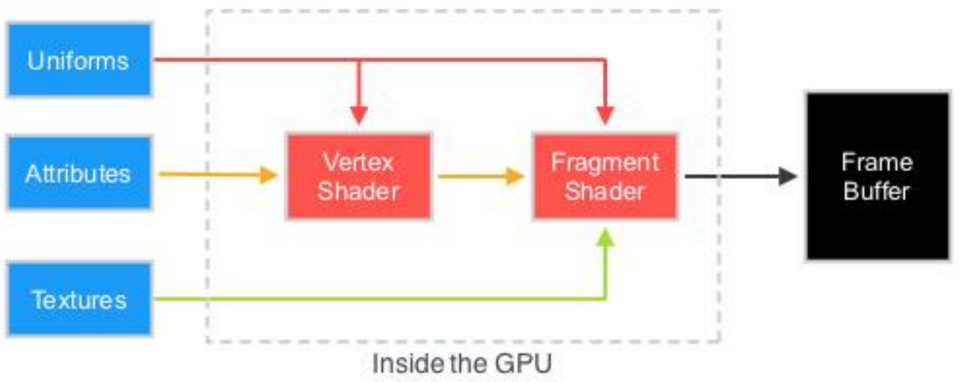
- To declare a uniform in GLSL we simply add the uniform keyword to a shader with a type and a name.
- From that point on we can use the newly declared uniform in the shader.




```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
   FragColor = ourColor;
}
```

We declared a uniform vec4 ourColor in the fragment shader and set the fragment's output color to the content of this uniform value.

Shaders – GLSL- Uniforms



- Since uniforms are global variables, we can define them in any shader stage we'd like so no need to go through the vertex shader again to get something to the fragment shader.

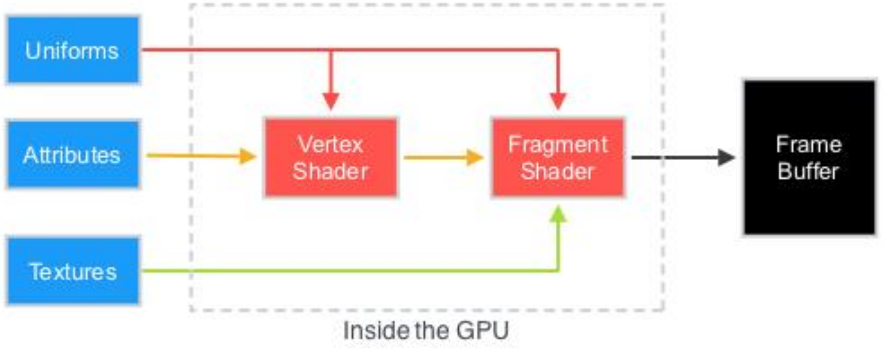
- We're not using this uniform in the vertex shader so there's no need to define it there.

If you declare a uniform that isn't used anywhere in your GLSL code the compiler will silently remove the variable from the compiled version which is the cause for several frustrating errors; **keep this in mind!**

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
   FragColor = ourColor;
}
```

Shaders – GLSL- Uniforms

- Send Data → we first need to find the index/location of the uniform attribute in our shader.
- Once we have the index/location of the uniform, we can update its values
- **Example**: Instead of passing a single color to the fragment shader, let's spice things up by gradually changing color over time:




```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
   FragColor = ourColor;
}
```

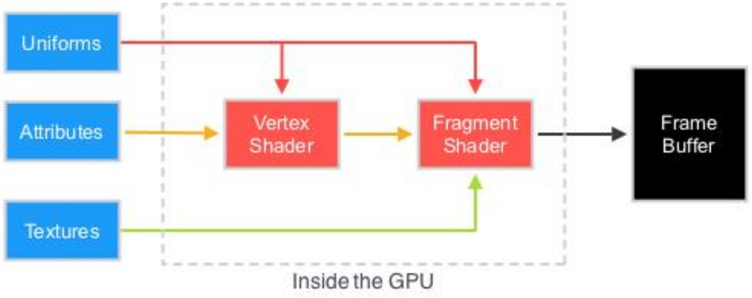
the running time is calculated in seconds via **glfwGetTime**(). Then we vary the color in the range of 0.0 - 1.0 by using the sin function and store the result in greenValue.

Shaders – GLSL- Uniforms

```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
   FragColor = ourColor;
}
```

- Then we query for the location of the **ourColor** uniform using glGetUniformLocation.
- We supply the shader program and the name of the uniform (that we want to retrieve the location from) to the query function. If glGetUniformLocation returns -1, it could not find the location.
- Lastly we can set the uniform value using the glUniform4f function.



- Note that finding the uniform location does not require you to use the shader program first, but **updating a uniform does require you to first use the program (by calling glUseProgram**), because it sets the uniform on the currently active shader program.

Shaders – GLSL- Uniforms

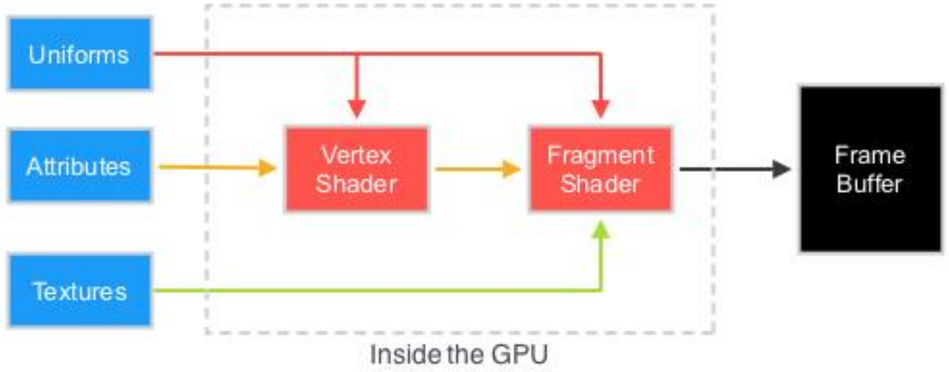
```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

Because OpenGL is in its core a C library it does not have native support for function overloading, so wherever a function can be called with different types OpenGL defines new functions for each type required; `glUniform` is a perfect example of this. The function requires a specific postfix for the type of the uniform you want to set. A few of the possible postfixes are:

- `f`: the function expects a `float` as its value.
- `i`: the function expects an `int` as its value.
- `ui`: the function expects an `unsigned int` as its value.
- `3f`: the function expects 3 `float`s as its value.
- `fv`: the function expects a `float` vector/array as its value.

Whenever you want to configure an option of OpenGL simply pick the overloaded function that corresponds with your type. In our case we want to set 4 floats of the uniform individually so we pass our data via `glUniform4f` (note that we also could've used the `fv` version).

Shaders – GLSL- Uniforms



- Now that we know how to set the values of uniform variables, we can use them for rendering.

- If we want the color to gradually change, we want to update this uniform every frame, otherwise the triangle would maintain a single solid color if we only set it once.

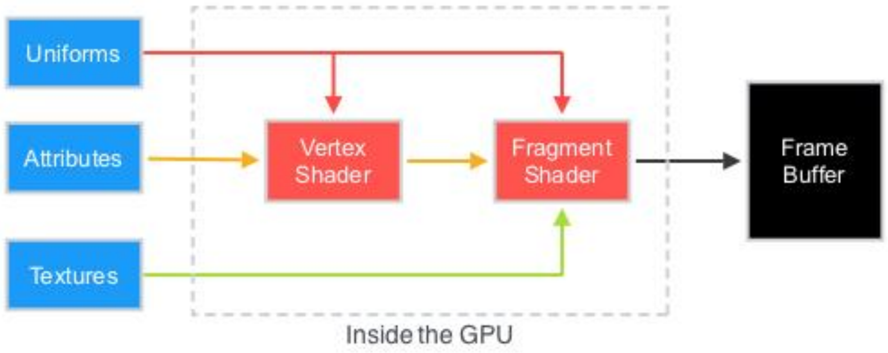
- So we calculate the greenValue and update the uniform each render iteration:

```
while(!glfwWindowShouldClose(window))
{
   // input
   processInput(window);
   // render
   // clear the colorbuffer
   glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
   glClear(GL_COLOR_BUFFER_BIT);
   // be sure to activate the shader
   glUseProgram(shaderProgram);
    // update the uniform color
   float timeValue = glfwGetTime();
   float greenValue = sin(timeValue) / 2.0f + 0.5f;
   int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
   glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
   // now render the triangle
   glBindVertexArray(VAO);
   glDrawArrays(GL_TRIANGLES, 0, 3);
    // swap buffers and poll IO events
   glfwSwapBuffers(window);
   glfwPollEvents();
}
```

Shaders – GLSL- Uniforms

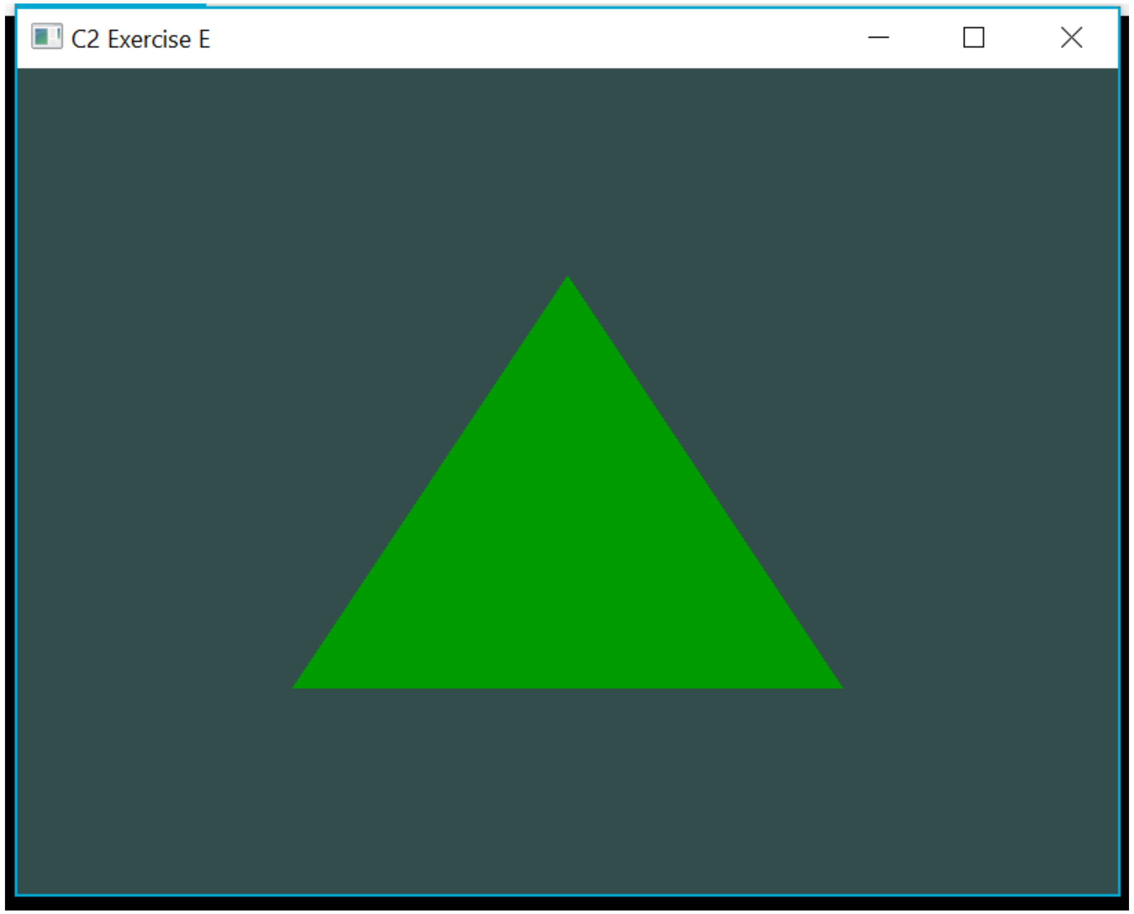
Exercise 5:

Modify the shaders and render loop to draw a triangle with time based green color change.

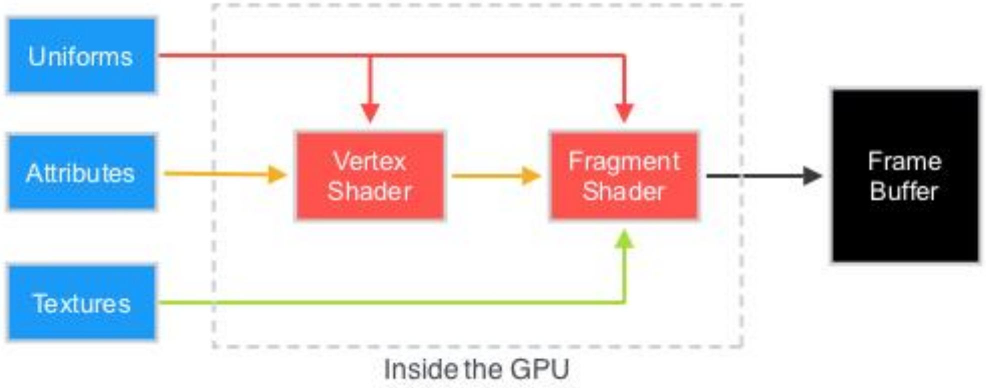



```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
   FragColor = ourColor;
}
```

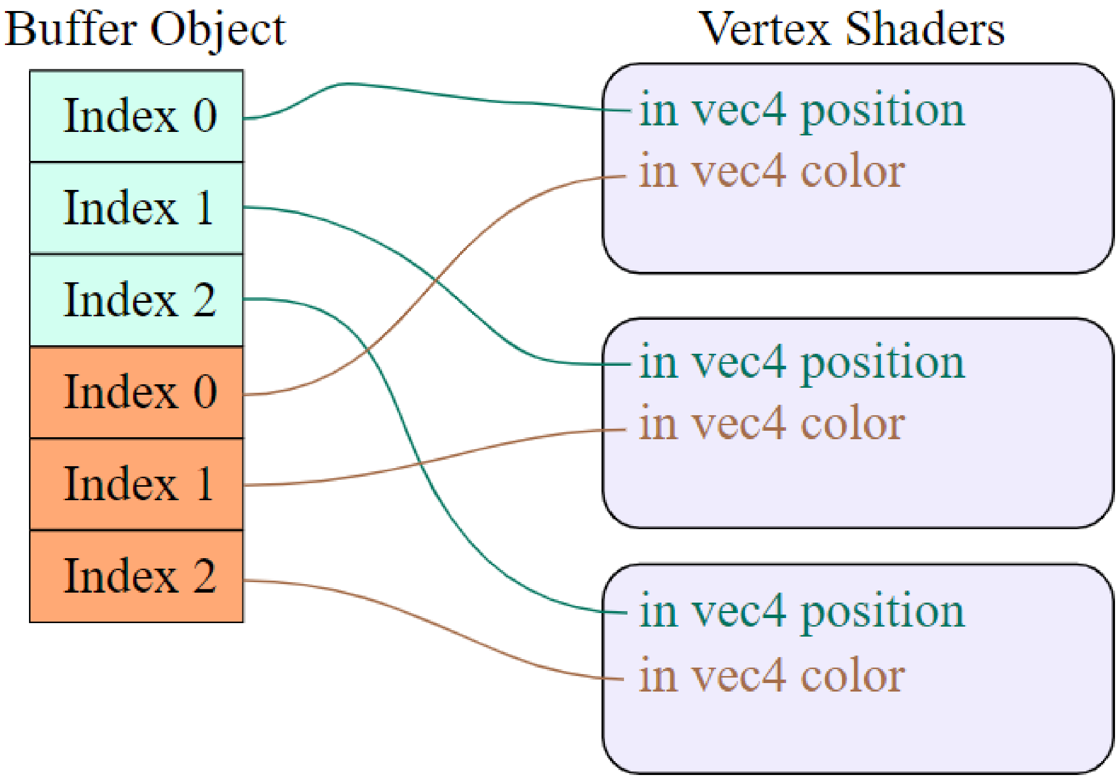
```
while(!glfwWindowShouldClose(window))
{
   // input
   processInput(window);
   // render
   // clear the colorbuffer
   glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
   glClear(GL_COLOR_BUFFER_BIT);
   // be sure to activate the shader
   glUseProgram(shaderProgram);
    // update the uniform color
   float timeValue = glfwGetTime();
   float greenValue = sin(timeValue) / 2.0f + 0.5f;
   int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
   glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
   // now render the triangle
   glBindVertexArray(VAO);
   glDrawArrays(GL_TRIANGLES, 0, 3);
    // swap buffers and poll IO events
   glfwSwapBuffers(window);
   glfwPollEvents();
}
```



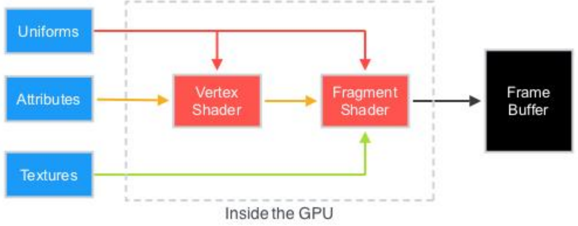
Shaders – GLSL- Uniforms



- As you can see, uniforms are a useful tool for setting attributes that might change every frame, or for interchanging data between your application and your shaders, but what **if we want to set a color for each vertex?**
- In that case we'd have to declare as many uniforms as we have vertices.
- A better solution would be to include more data in the vertex attributes which is what we're going to do now.

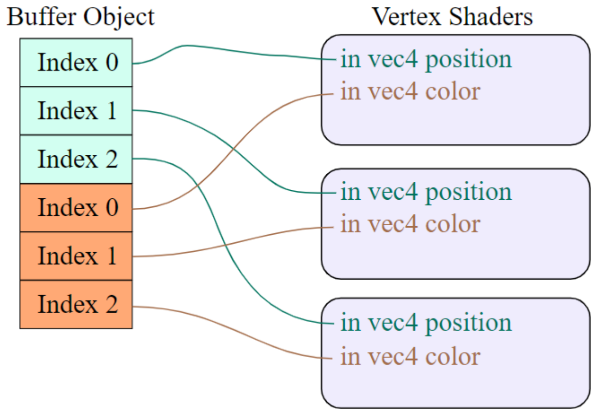


Shaders – GLSL- Attributes



The best procedure is fill a **VBO**, configure vertex attribute pointers and store it all in a **VAO**.

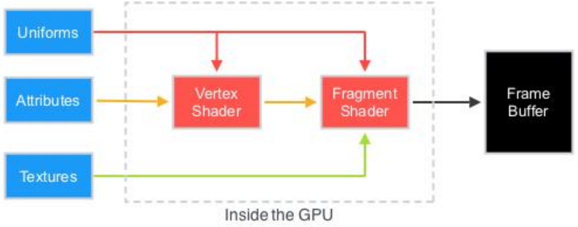
This time, we also want to add color data to the vertex data. We're going to add color data as 3 floats to the vertices array.

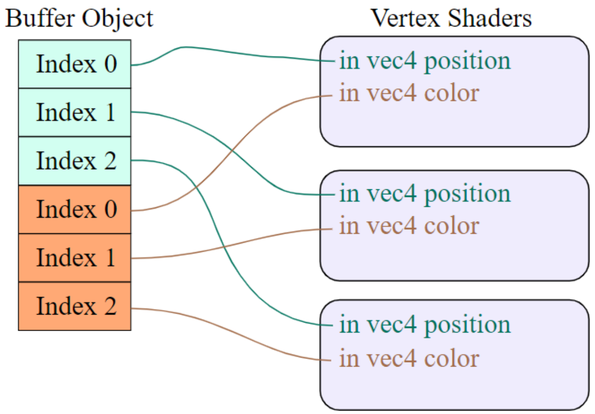


- We assign a red, green and blue color to each of the corners of our triangle respectively:

```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,   // bottom left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // top
};
```

Shaders – GLSL- Attributes



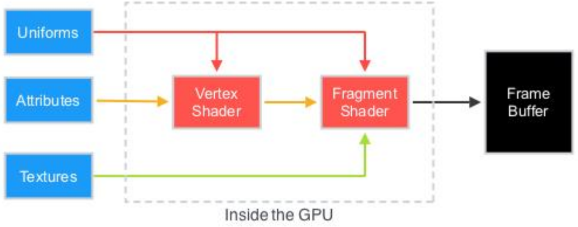


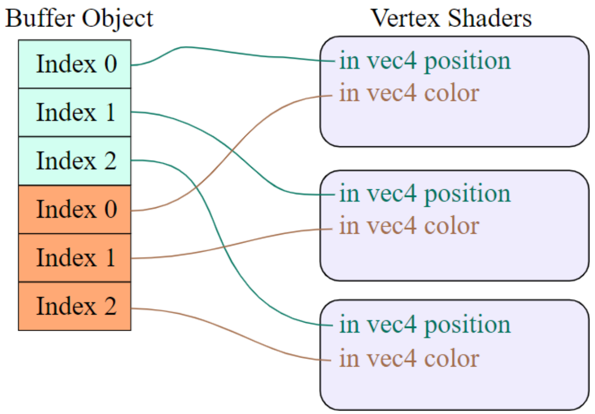
Since we now have more data to send to the vertex shader, it is necessary to **adjust the vertex shader** to also **receive our color value as a vertex attribute input**.

Note that we set the location of the aColor attribute to 1 with the layout specifier:

```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{
   gl_Position = vec4(aPos, 1.0);
   ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

Shaders – GLSL- Attributes



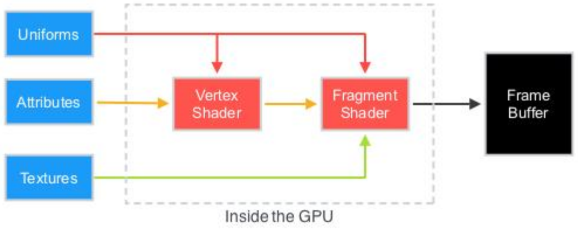


Since we no longer use a uniform for the fragment's color, but now use the ourColor output variable we'll have to change the fragment shader as well:

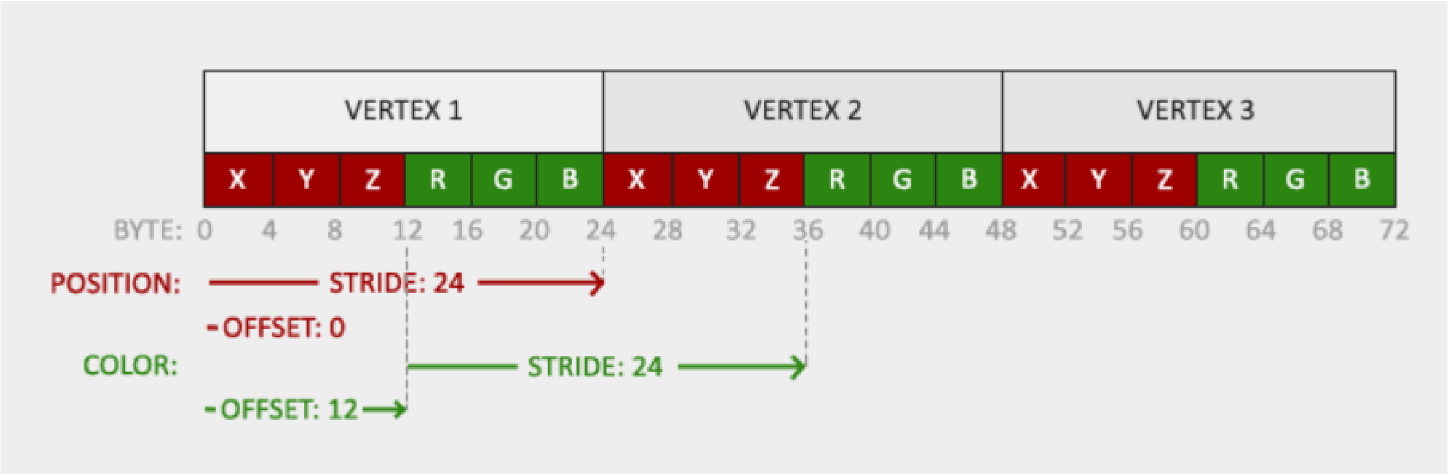
```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
   FragColor = vec4(ourColor, 1.0);
}
```

Because we added another vertex attribute and updated the VBO's memory we have to **re-configure the vertex attribute pointers**.

Shaders – GLSL- Attributes



The updated data in the VBO's memory now looks a bit like this:

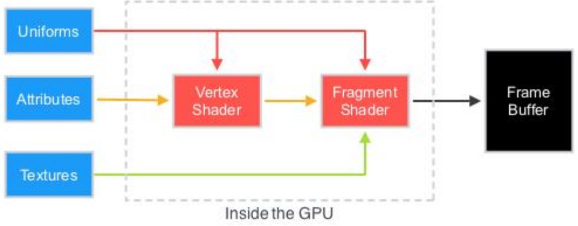


The vertex format with glVertexAttribPointer:

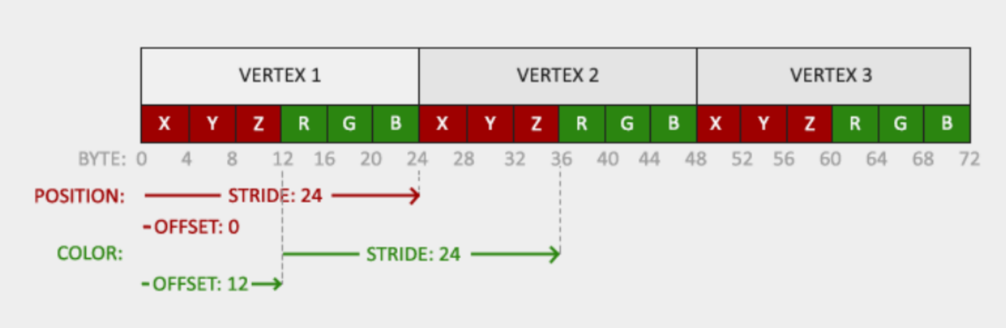
```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```

The first few arguments of glVertexAttribPointer are relatively straightforward. This time we are configuring the vertex attribute on attribute location 1. The color values have a size of 3 floats and we do not normalize the values.

Shaders – GLSL- Attributes



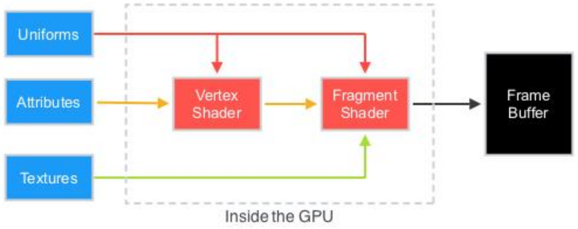
The updated data in the VBO's memory now looks a bit like this:



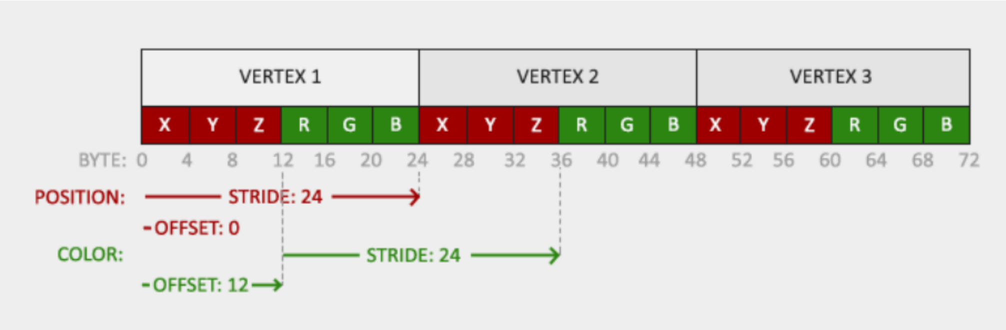

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```

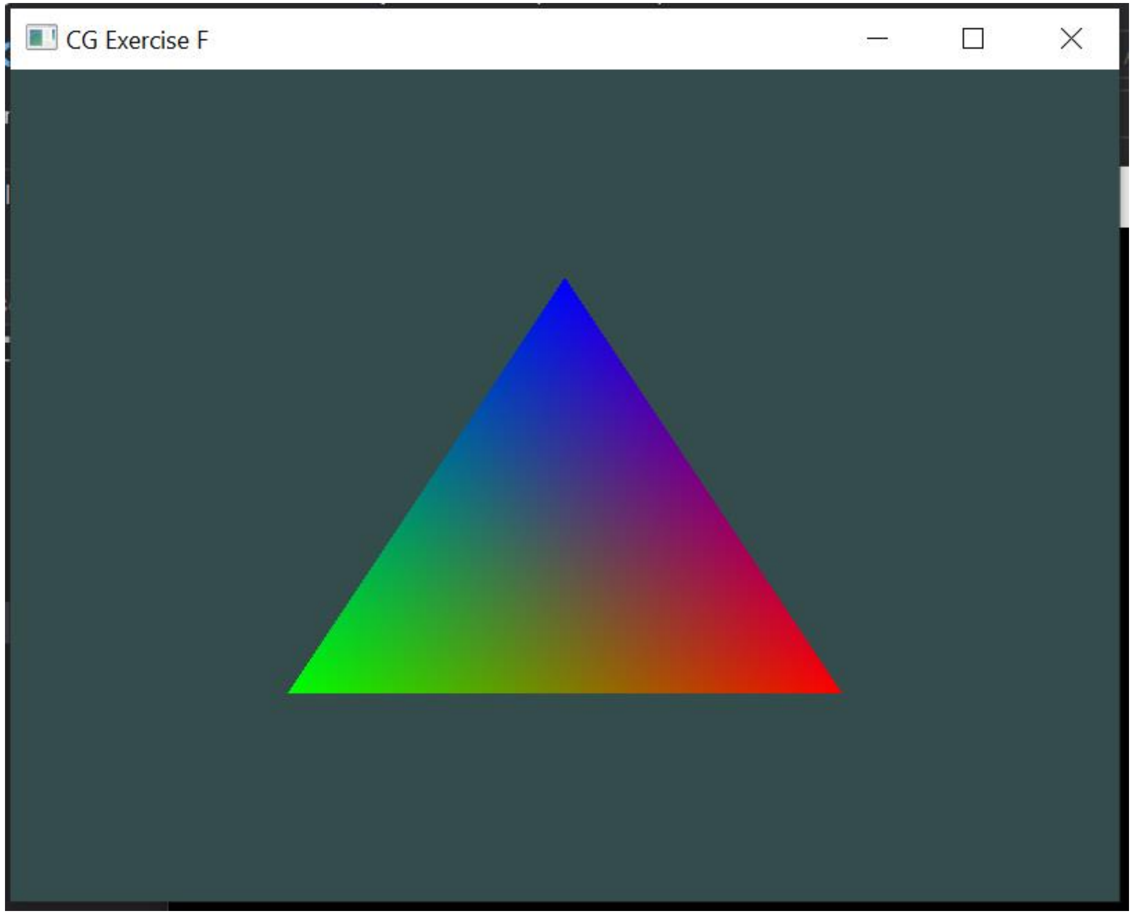
- Since we now have two vertex attributes we have to re-calculate the stride value.
- To get the next attribute value (e.g. the next x component of the position vector) in the data array we have to move 6 floats to the right, three for the position values and three for the color values. This gives us a stride value of 6 times the size of a float in bytes (= 24 bytes).
- Also, this time we have to specify an offset. For each vertex, the position vertex attribute is first so we declare an offset of 0.
- The color attribute starts after the position data so the offset is 3 * sizeof(float) in bytes (= 12 bytes).

Shaders – GLSL- Attributes



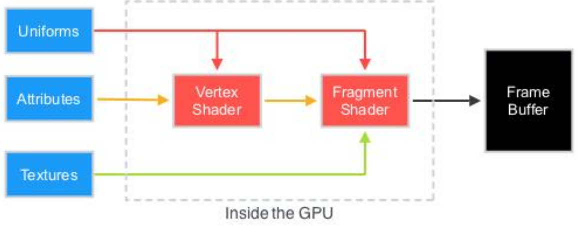
Exercise 6: Modify the VBO and Vertex Attributes in triangle example



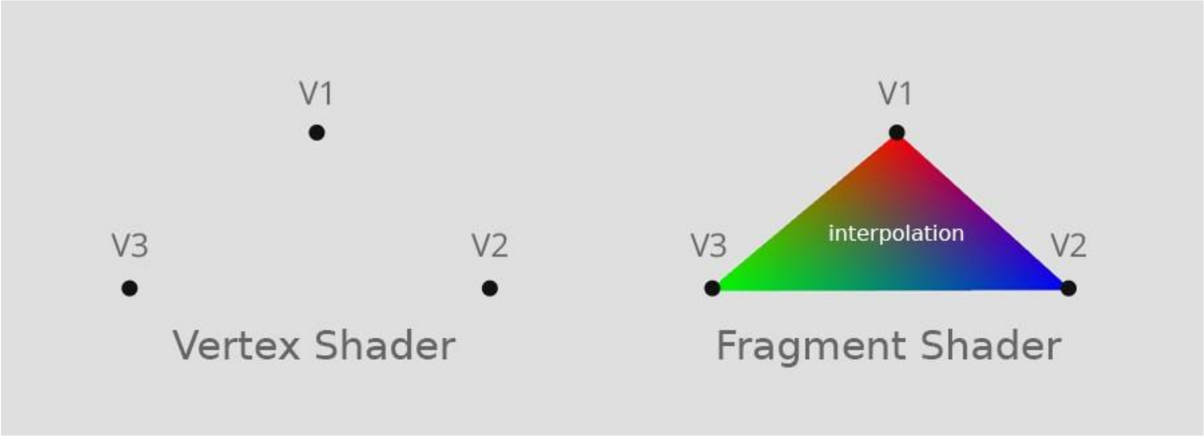


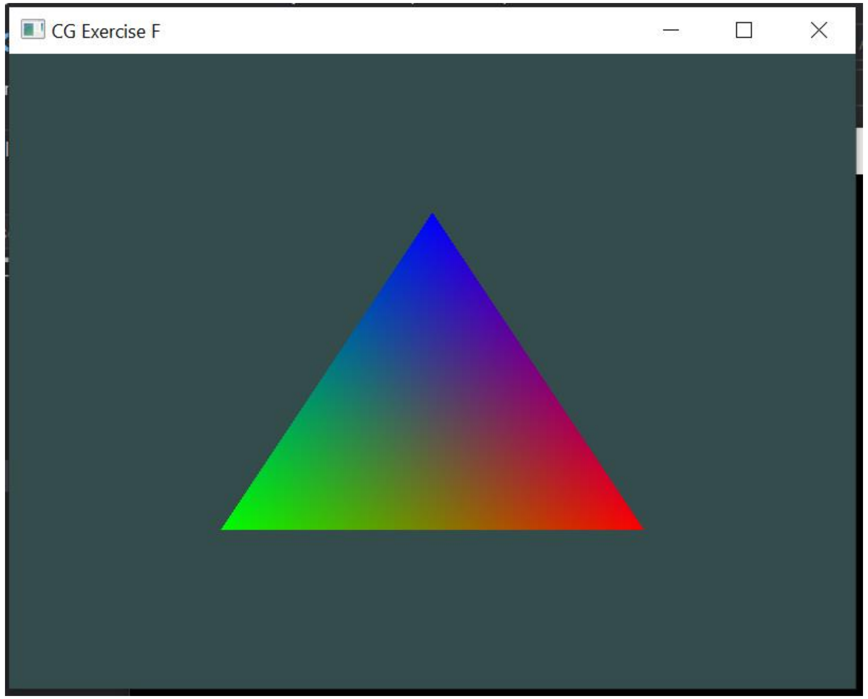
The image might not be exactly what you would expect, since we only supplied 3 colors, not the huge color palette we're seeing right now.

Shaders – GLSL- Fragment Interpolation



Exercise 6: Modify the VBO and Vertex Attributes in triangle example



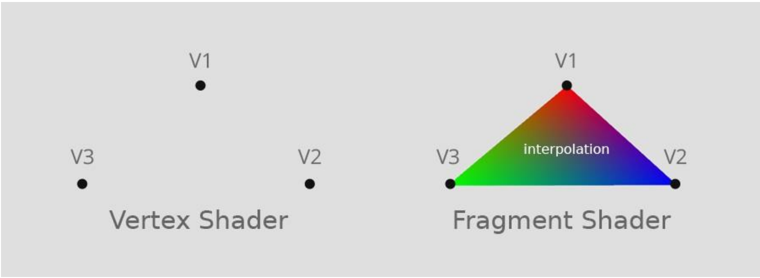


This is all the result of something called fragment interpolation in the fragment shader.

When rendering a triangle the rasterization stage usually results in a lot more fragments than vertices originally specified.

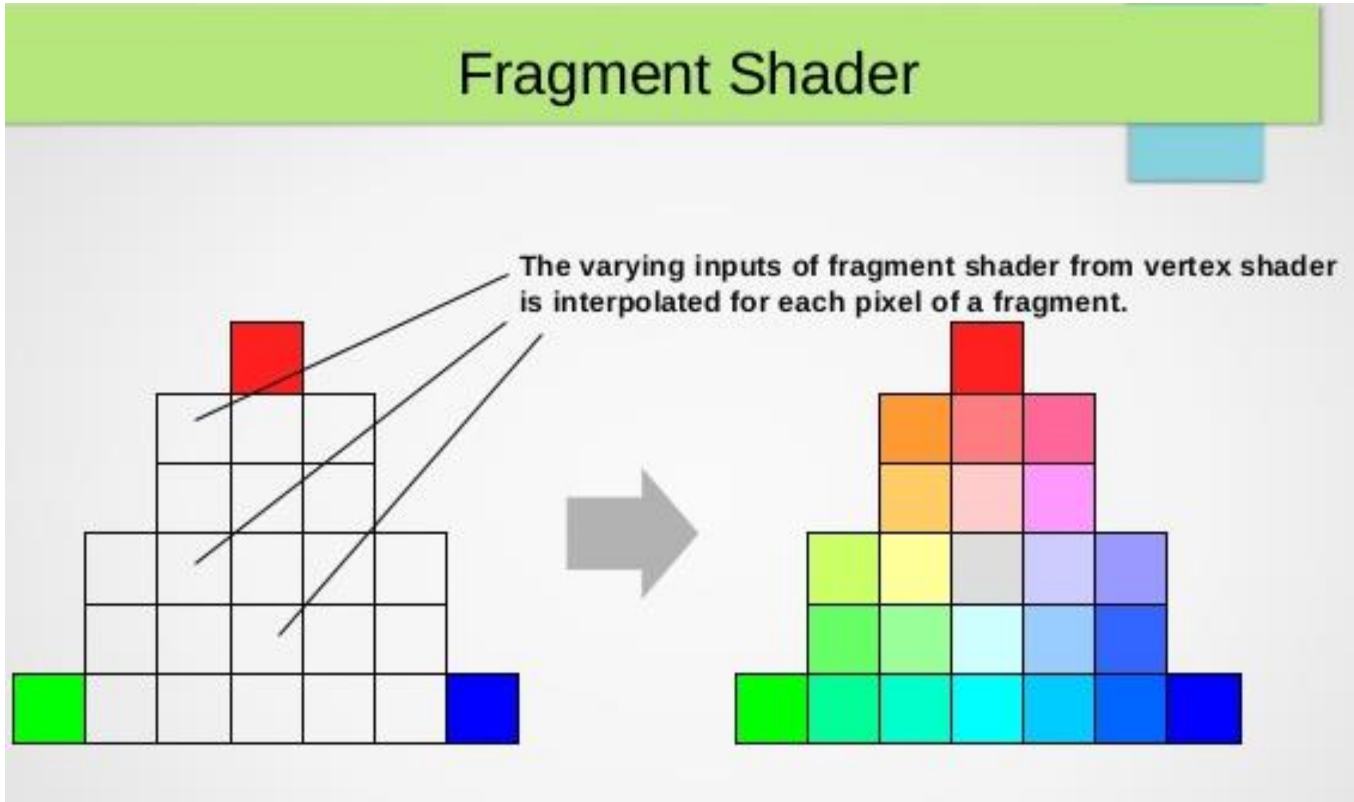
The rasterizer then determines the positions of each of those fragments based on where they reside on the triangle shape.

Shaders – GLSL- Fragment Interpolation

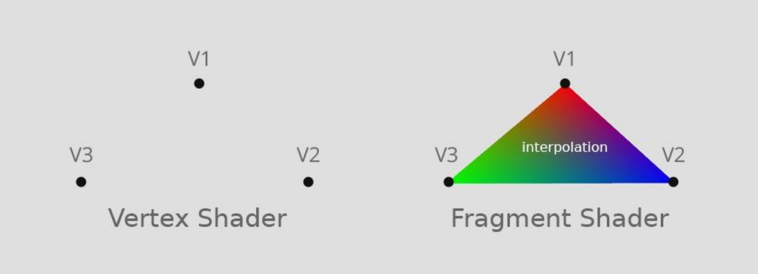


Exercise 6: Modify the VBO and Vertex Attributes in triangle example

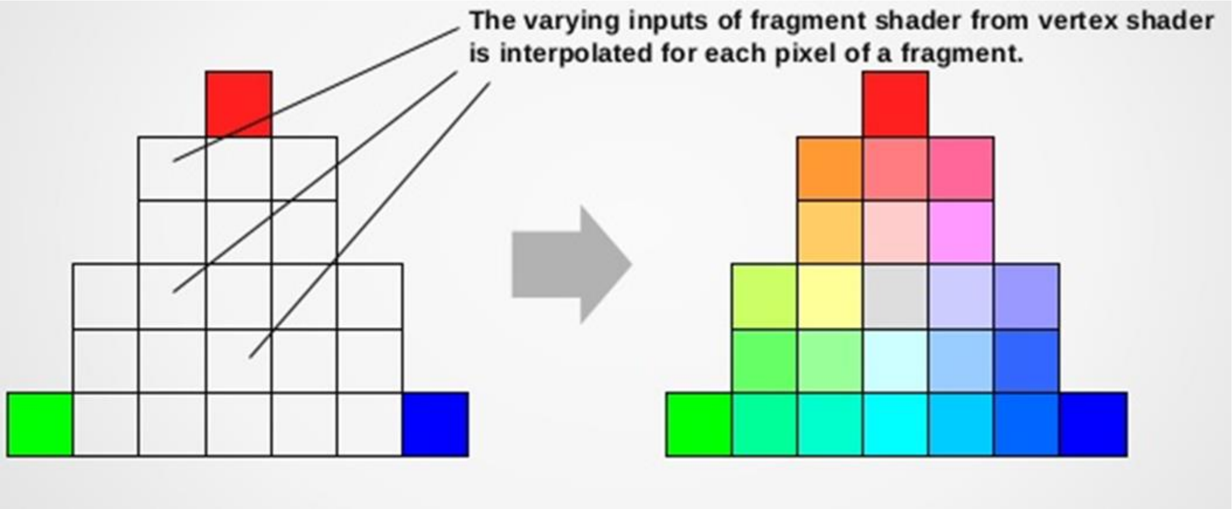
- Then rendering a triangle the rasterization stage usually results in a lot more fragments than vertices originally specified.
- The rasterizer then determines the positions of each of those fragments based on where they reside on the triangle shape.
- Based on these positions, it interpolates all the fragment shader's input variables.

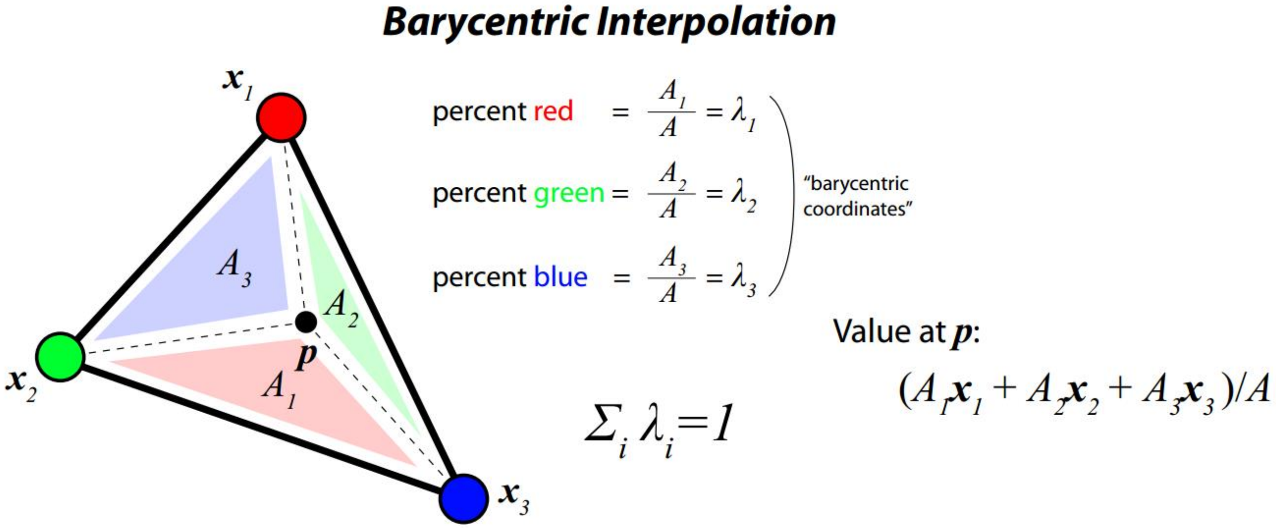


Shaders – GLSL- Fragment Interpolation



Exercise 6: Modify the VBO and Vertex Attributes in triangle example

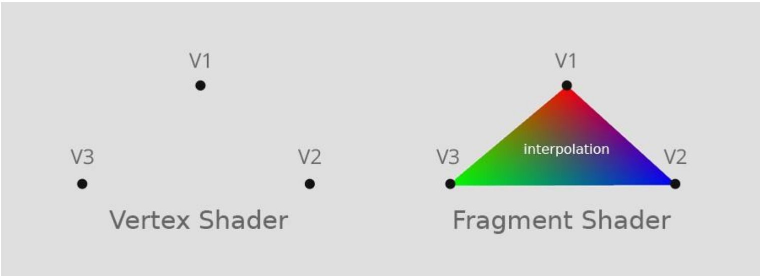




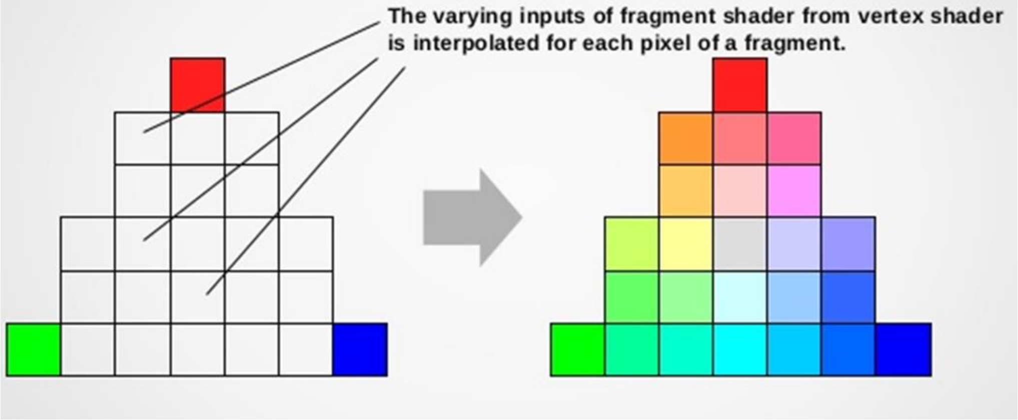
Say for example we have a line where the upper point has a green color and the lower point a blue color.

If the fragment shader is run at a fragment that resides around a position at 70% of the line, its resulting color input attribute would then be a linear combination of green and blue; to be more precise: 30% blue and 70% green.

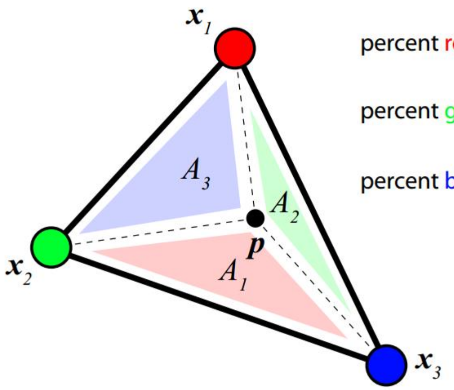
Shaders – GLSL- Fragment Interpolation



Exercise 6: Modify the VBO and Vertex Attributes in triangle example



Barycentric Interpolation



percent red $= \frac{A_1}{A} = \lambda_1$

percent green $= \frac{A_2}{A} = \lambda_2$

percent blue $= \frac{A_3}{A} = \lambda_3$

"barycentric coordinates"

$$\Sigma_i \lambda_i = 1$$

Value at $\boldsymbol{p}$:

$$(A_1\boldsymbol{x}_1 + A_2\boldsymbol{x}_2 + A_3\boldsymbol{x}_3)/A$$

- We have 3 vertices and thus 3 colors, and judging from the triangle's pixels it probably contains around 50000 fragments, where the fragment shader interpolated the colors among those pixels.
- If you take a good look at the colors you'll see it all makes sense: red to blue first gets to purple and then to blue.
- Fragment interpolation is applied to all the fragment shader's input attributes.