Working with shaders

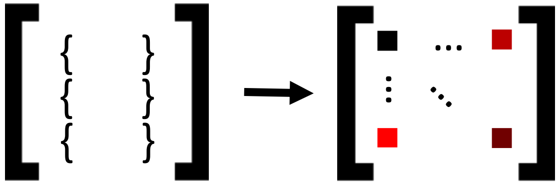
Patrick SARDINHA

What's a shader?

Small programs that run on the GPU

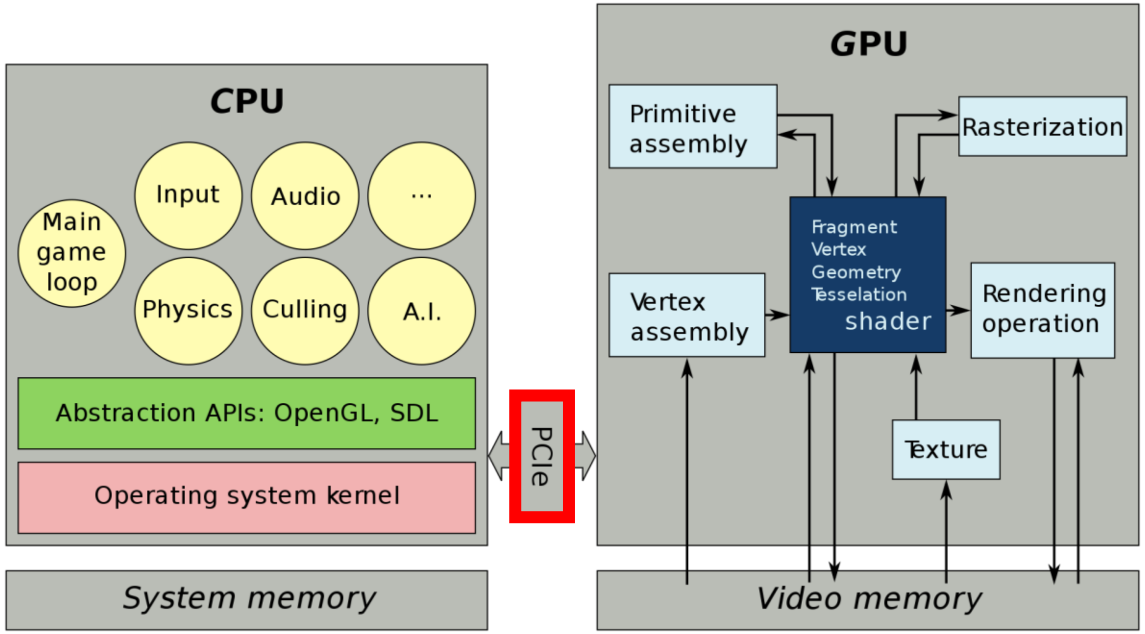
Executed for each specific section of the graphics pipeline

Isolated and not allowed to communicate with each other



It works with geometric primitives, lights, textures, ...

Shaders in the Graphics Processing Unit



Shaders are executed by the GPU & are good to be executed in parallel

Sending data to the GPU goes through the PCI, it is relatively slow
& CPU/GPU must be synchronized

Different languages

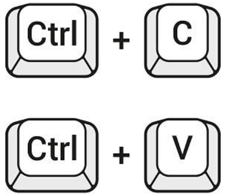
DirectX High-Level Shader Language

Cg Shader Language

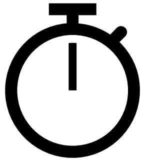
OpenGL Shading Language (GLSL)

Problem

In GLSL, there are no real data structures to easily get the attributes of a primitive (matrices, vectors, ...)



The construction of shaders is very repetitive which implies a lot of copy and paste



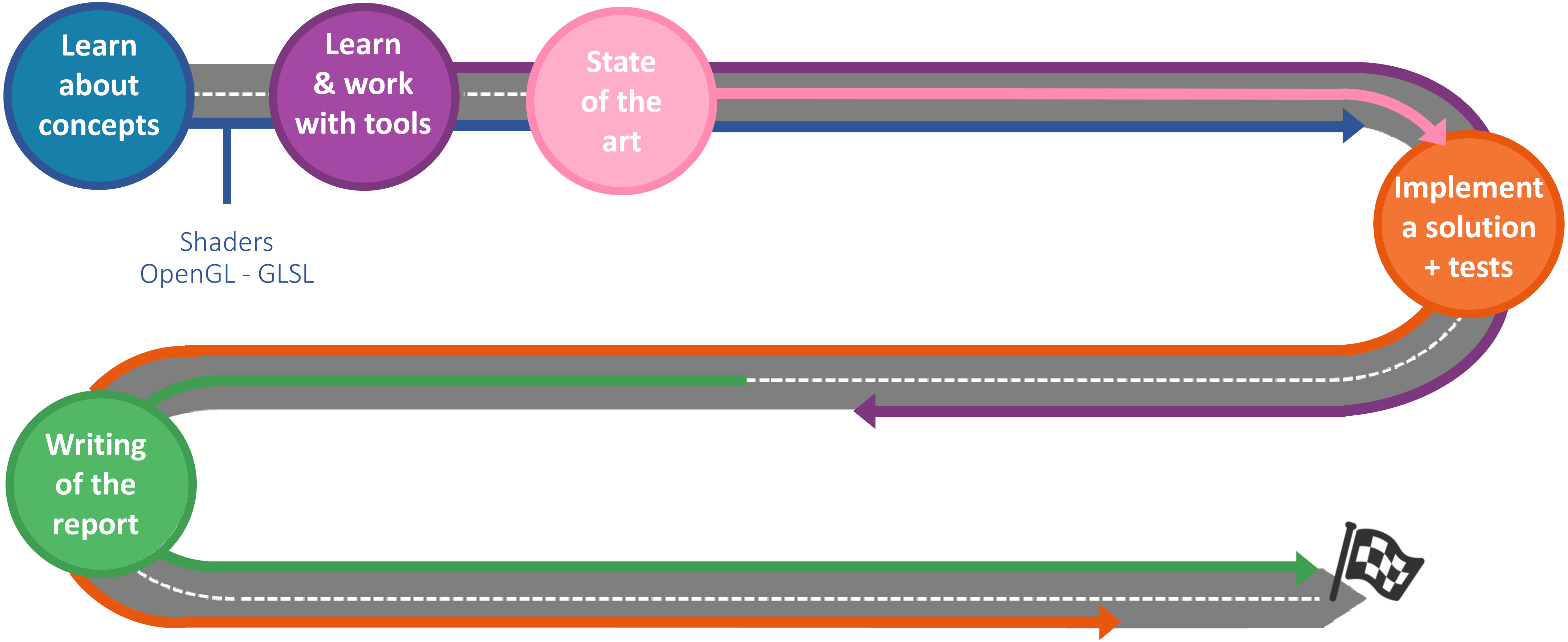
Must reduce the data sent in the PCI to avoid multiple synchronizations between CPU & GPU

Goal of the project

Work with the representation of the data
& abstract the types

Construct a DSL for shaders

Road map



Learn about concepts
Learn & work with tools
State of the art
Implement a solution + tests
Writing of the report
Shaders
OpenGL - GLSL

3D space to 2D screen space

The process of transforming 3D coordinates to 2D pixel is done by the graphics pipeline

First big part: transforms 3D coordinates into 2D coordinates

Second big part: transforms the 2D coordinates into actual colored pixels

Graphics pipeline

Input & Output Data

3 different shaders processing units

Vertex Shader

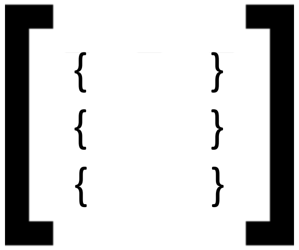
Geometry Shader

Fragment/Pixel Shader

Some others processes

Tessellation, Rasterization, Color blending

Input Data



Take as input a Vertex (or Vertices) [] which is a data structure that describes geometric primitives with certain attributes like:

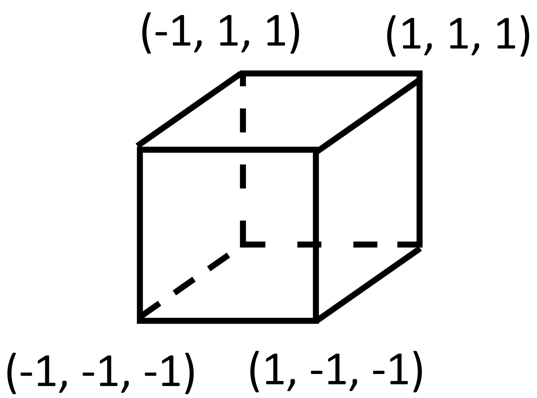
Position (2D, 3D coordinates)

Color (RGB, ...)

Texture coordinates

Example

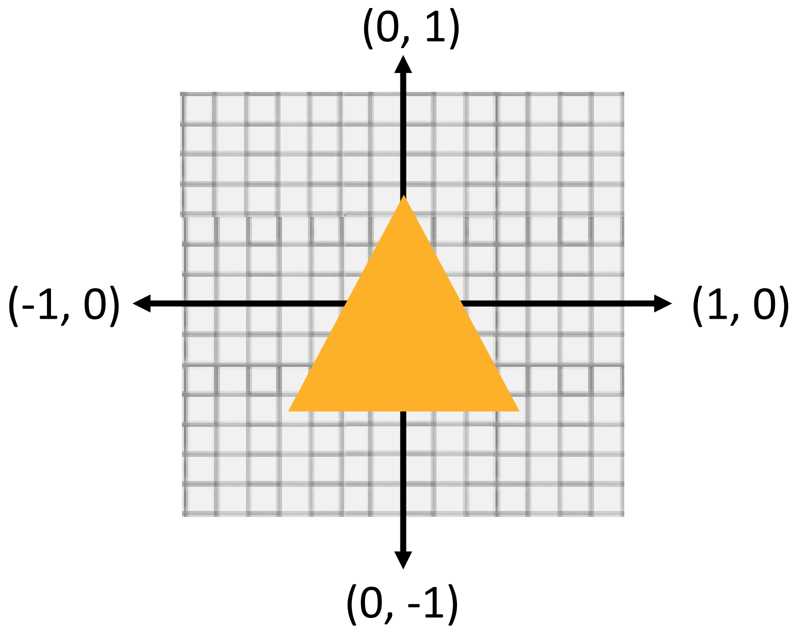
In OpenGL, only the Normalize Device Coordinates (NDC) are visible on the screen



To render a single 2D triangle:

3D position (NDC) of each vertex

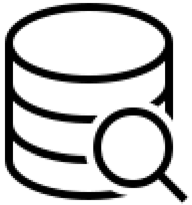
```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```



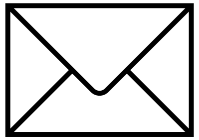
Linking vertex attributes

The input data will compose a Vertex Buffer Object (VBO)
which can store a large number of vertices in the GPU memory

Then, we specify how the vertex data should be interpreted



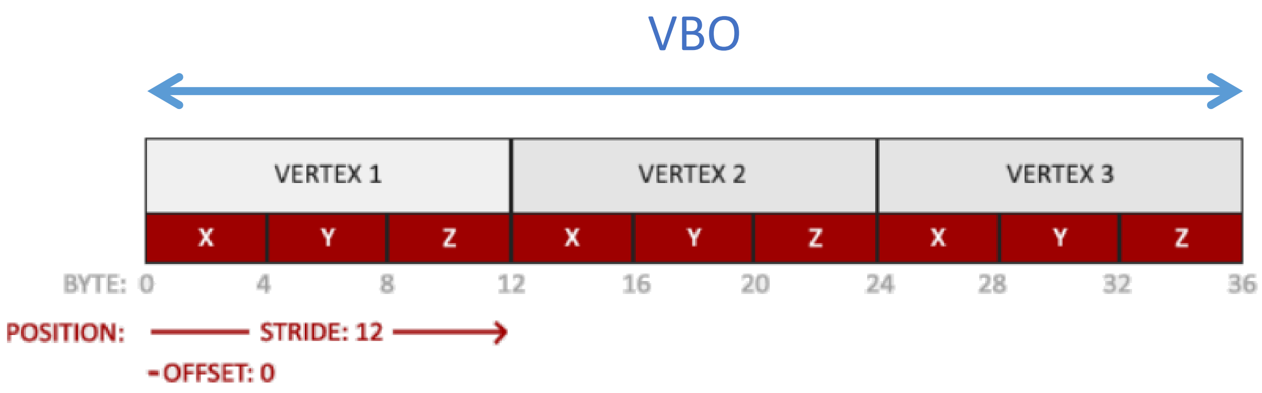
Finally, it will be sent to the Vertex Shader



Example

Triangle with position attributes:

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



The size of the vertex attribute

Specifies which vertex attribute

Normalized data or not

Offset of where the position data begins in the buffer

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

Enable the vertex attribute

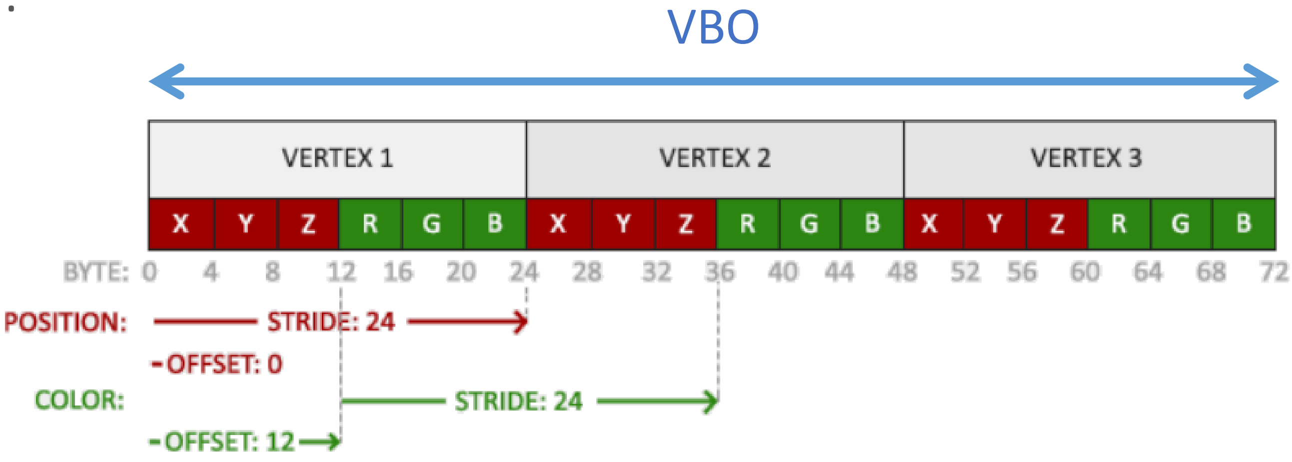
Type of the data

Stride: Space between consecutive vertex attributes

Example

Triangle with position & color attributes:

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



Position offset

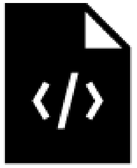
```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float))
glEnableVertexAttribArray(1);
```

Stride

Color offset

Render & draw an object

The idea now is to render and draw an object.
To do that we will have to:

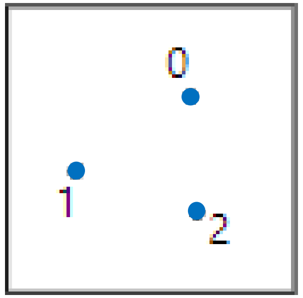


Set up a Vertex & a Fragment Shader

Compile these shaders

Link them to a shader program

Vertex Shader



Compute the projection of the vertices of primitives from 3D space into a different 3D space (NDC)

Input data: some properties of the vertices (position, color or texture coordinates)

Output data: the corresponding properties in the new space

Sample code

Shader's version (here 3.3)

All the input vertex attributes

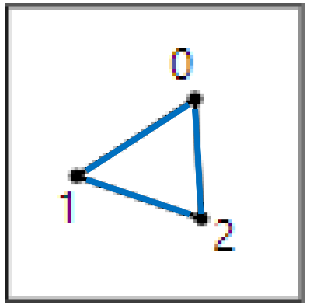
```
#version 330 core
layout (location = 0) in vec3 aPos;

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

Specifically set the location of the input variable

Output of the vertex shader

(x, y, z) + w component

Primitives Assembly



This process takes all the vertex given by the step before and assemble them in order to create a geometric shape

Starting index in the array

Sample code:

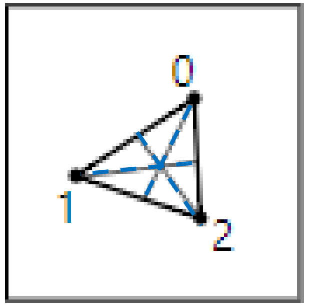
```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Number of vertices to render

OpenGL function that draws a shape

Kind of primitive to render

Tessellation

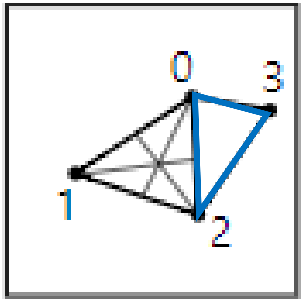


In 3D, the surfaces are built with triangular tiles

Tessellation allows to double triangles on a given surface and therefore increase the level of details

Geometry Shader

An unnecessary step

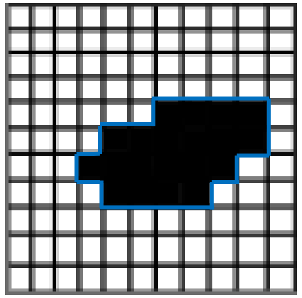


Allows to modify the geometry of each polygon and allows to create new polygons by emitting new vertices

Input data: data of a geometric primitive

Output data: data of one or more geometric primitive

Rasterization

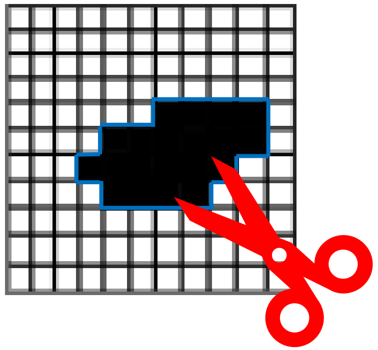


Method of converting a vector image into a raster image to be displayed on a screen

Vector image
composed of geometric objects

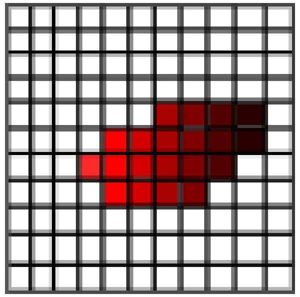
Raster image or Bitmap
composed of pixels

Clipping



This step discard all fragments (which is the required data to render a single pixel) that are outside the view, increasing the performance

Fragment/Pixel Shader



Calculates the final color of a pixel

Input data: pixel data
(position, texture coordinates, color)

Output data: the pixel color

Sample code

Shader's version (here 3.3)

```
#version 330 core
out vec4 FragColor;

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

Output variable which is the final color output

RGB + alpha component

Compile a Shader

First, we store the code in a string constant

```
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";
```

Then, we store and create the shader

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

Type of shader we want to create

Finally, we link the source code to the object and compile it

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

Shader program

First, we create a program object

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

We attach the previously compiled shaders to the program object and link them

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

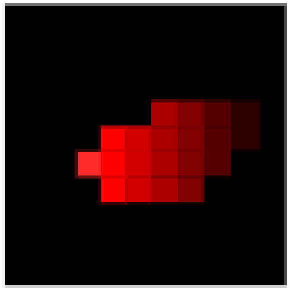
We can now activate this program to render and draw an object

```
glUseProgram(shaderProgram);
```

Final step is to delete our shader objects

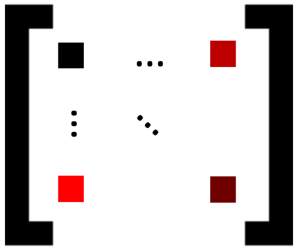
```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Color Blending



The technique of gently blending two or more colors to create a gradual transition

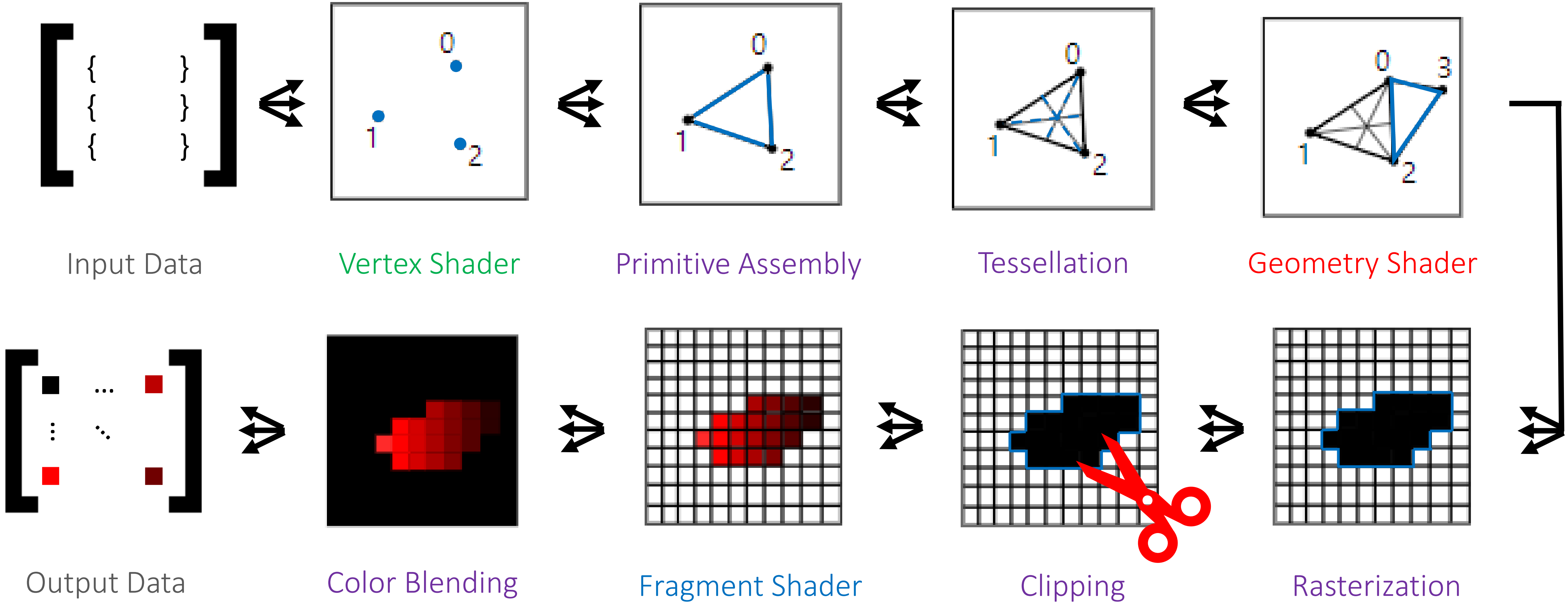
Output Data



Return a Framebuffer

The information in this buffer are the values of the color components (RGB) for each pixel

Overall view



Work incoming

Learn more about OpenGL - GLSL & shaders

- Coordinate systems
- Textures
- Lighting
- Camera
- Transformations
- Geometry shaders

Take a look about CUDA

Learn about DSL & SIMD language

Begin to work with Rendery

References

https://learnopengl.com

https://fr.wikipedia.org/wiki/Shader

https://fr.wikipedia.org/wiki/OpenGL

https://fr.wikipedia.org/wiki/DirectX

https://developer.apple.com/metal

https://github.com/RenderyEngine/Rendery

https://www.khronos.org/opengl/wiki

Working with shaders

Patrick SARDINHA