

Working with shaders

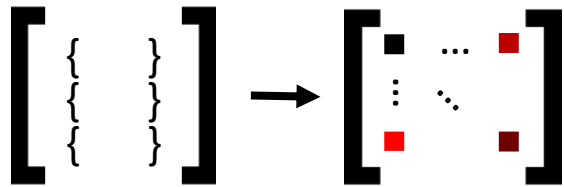
Patrick SARDINHA

What's a shader?

Small programs that run on the GPU

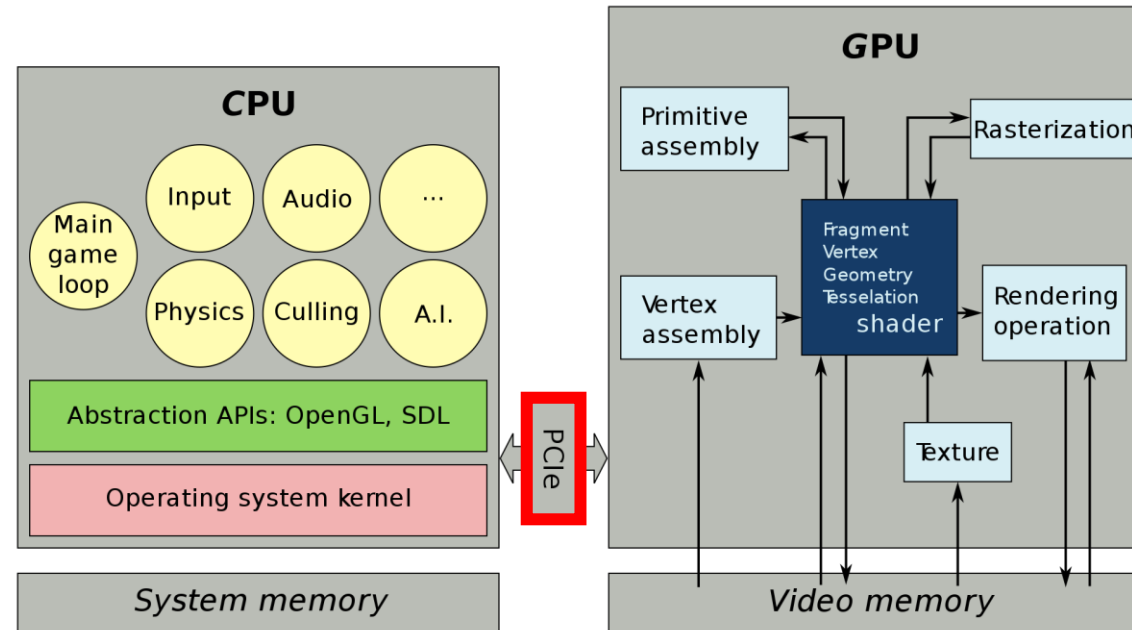
Executed for each specific section of the graphics pipeline

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

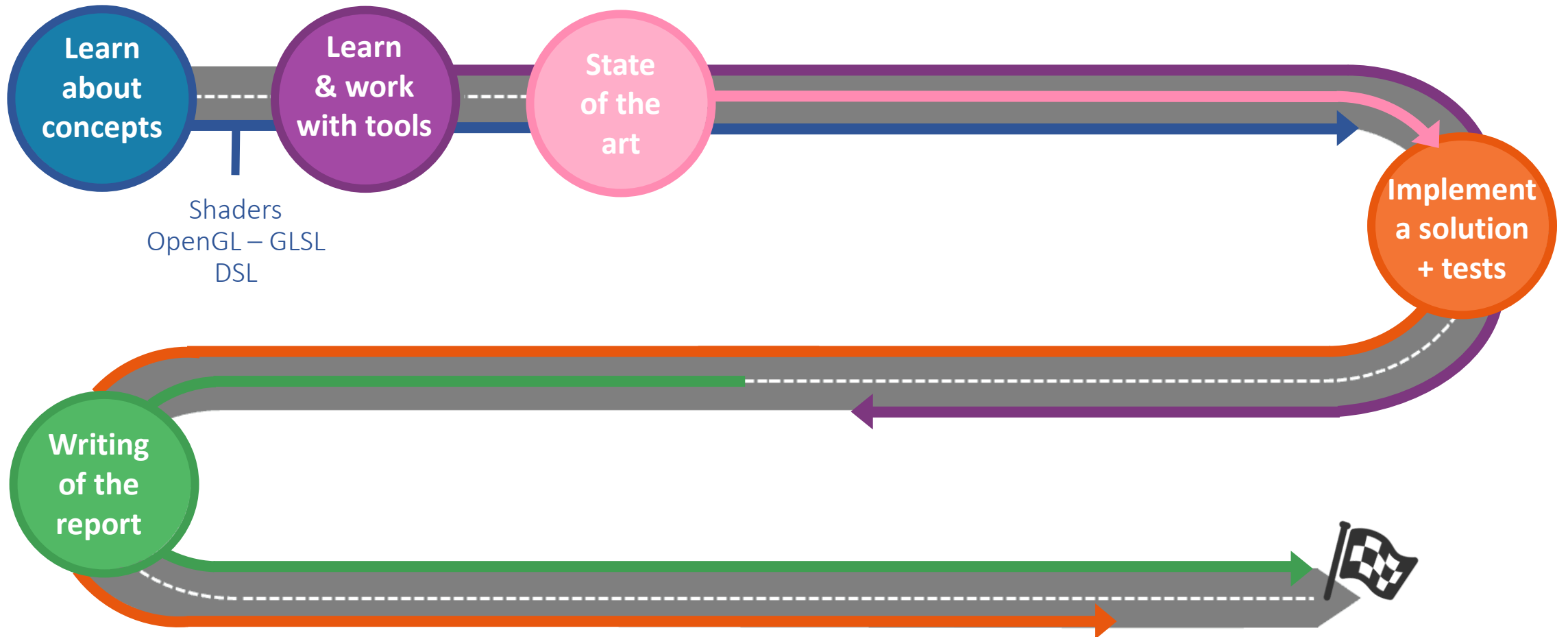


Work with the representation of the data
& abstract the types



Construct a DSL for shaders

Road map



3D space to 2D screen space

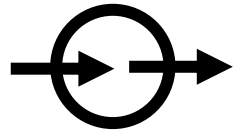


The process of transforming 3D coordinates to 2D pixel is done by the [graphics pipeline](#)

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

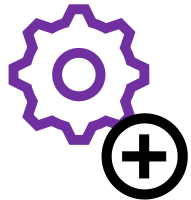


3 different shaders processing units

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 (**R****G****B**, ...)

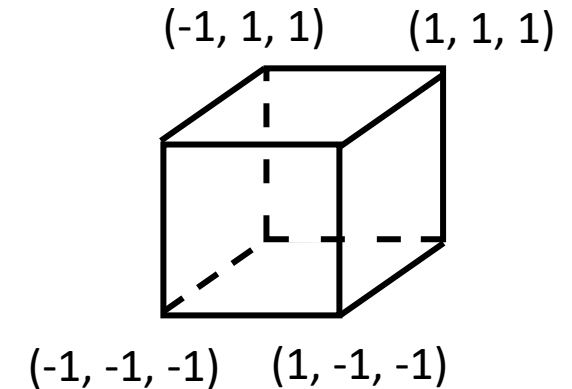


Texture coordinates



Example

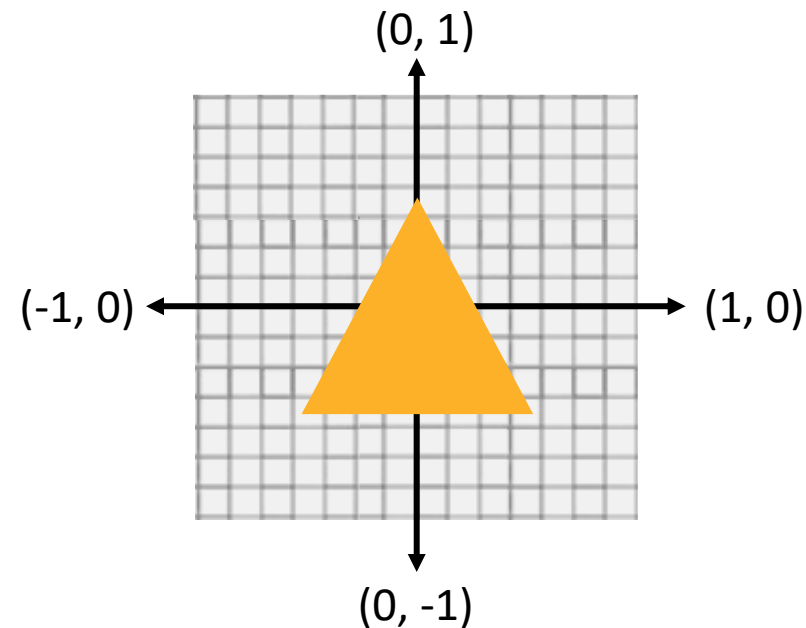
In OpenGL, only the Normalize Device Coordinates (NDC) are visible on the screen



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

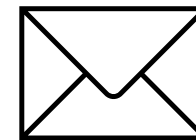


The input data will compose a **Vertex Buffer Object (VBO)** which can store a large number of vertices in the GPU memory

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  
};
```

Copy our vertices array in a buffer

Specifies the target buffer object

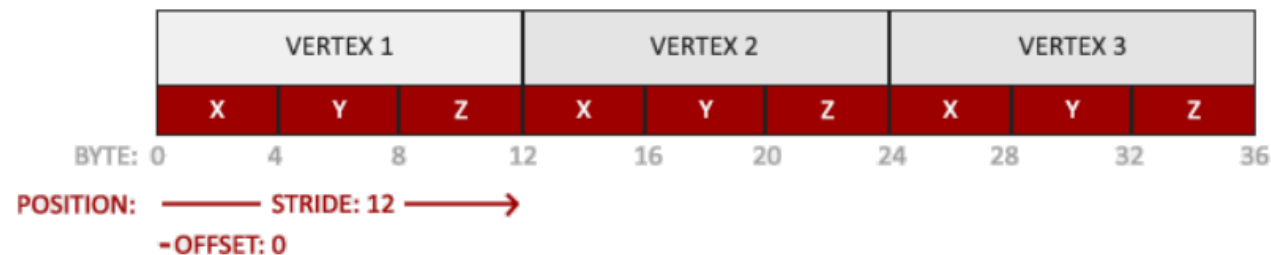
ID of the buffer which must be bind

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

Size of the buffer object

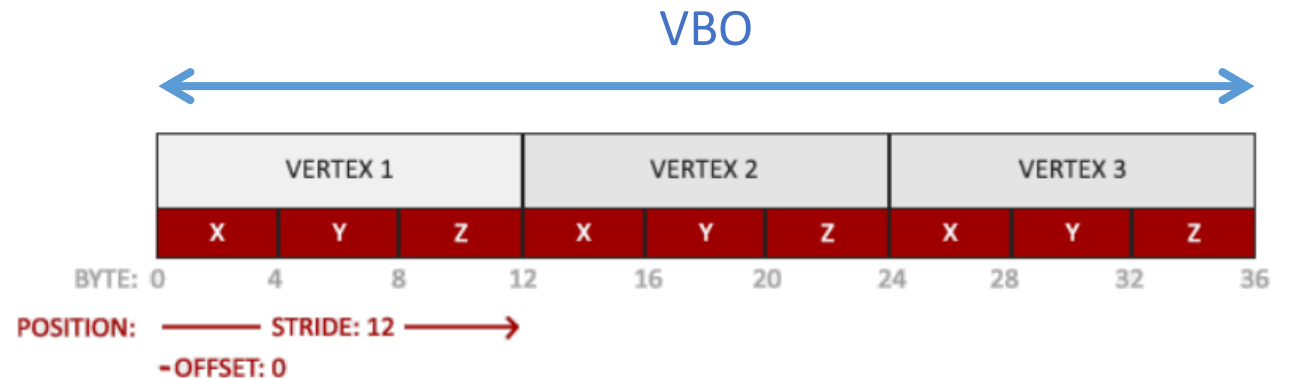
Pointer to data

VBO



Example (Cont.)

Define how the vertex data should be interpreted



Specifies which vertex attribute

The size of the 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

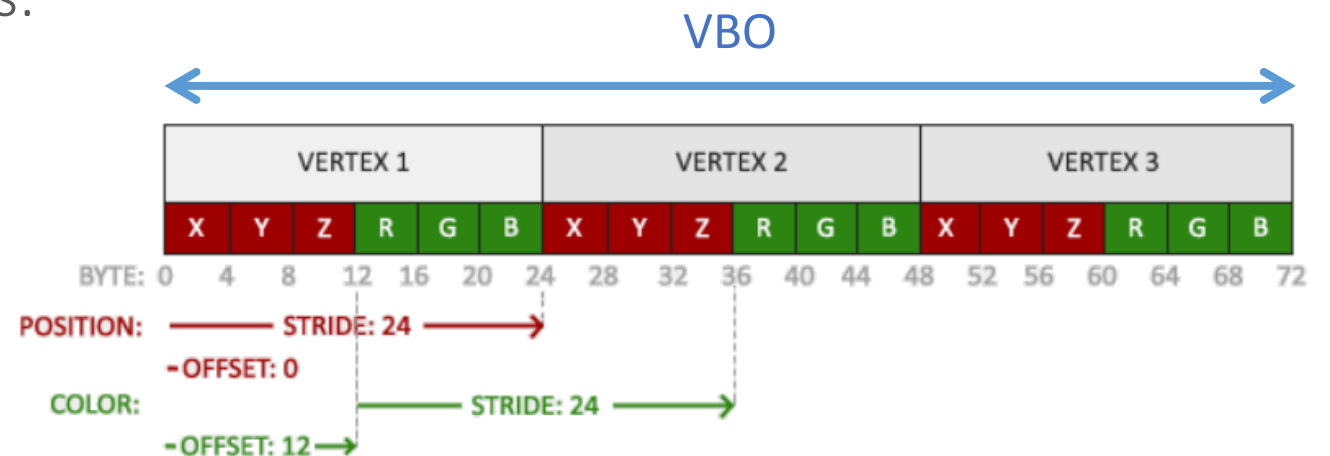
Type of the data

Stride: Space between consecutive vertex attributes

Example (Cont.)

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 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);
```

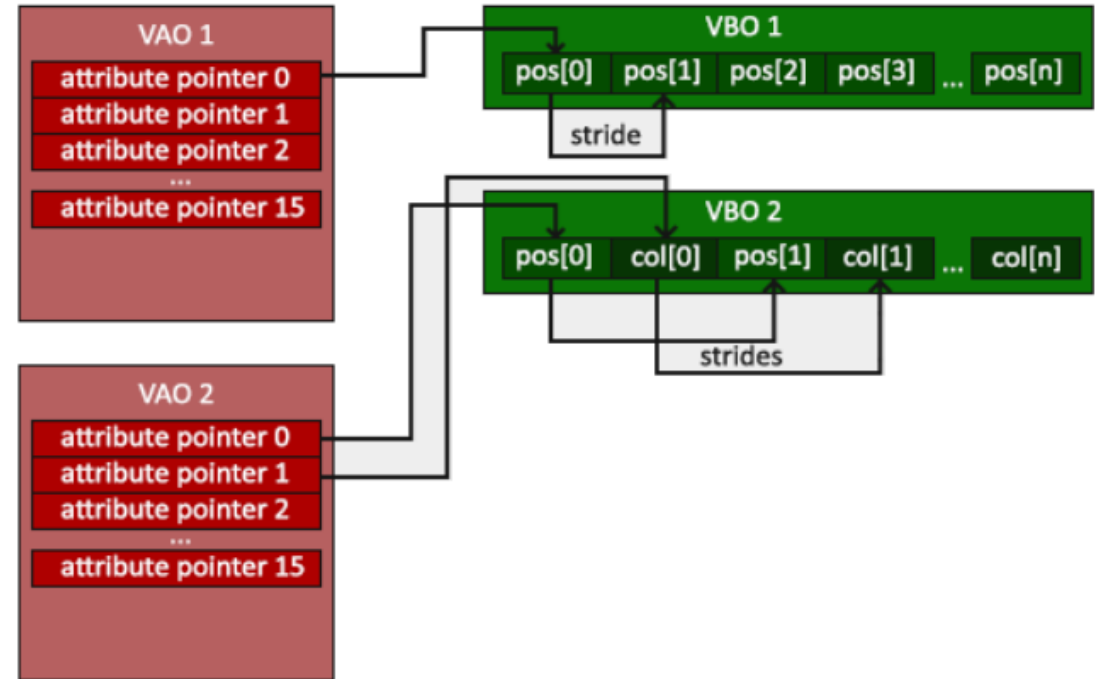
Stride

Color offset

Vertex Array Object (VAO)

Allows to configure vertex attribute pointers more easily

To draw an object, just bind the corresponding VAO



We generate a VAO like a VBO

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


Summary

```
// 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);

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

Render & draw an object



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

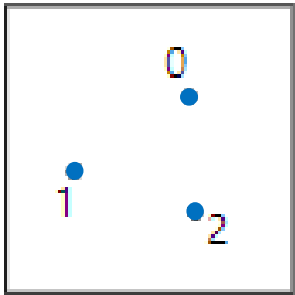


Set up a Vertex & a Fragment Shader

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

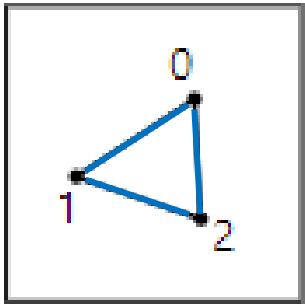
The diagram shows a GLSL vertex shader code snippet with several annotations:

- Shader's version (here 3.3)**: A red arrow points to the `#version 330 core` line, which is enclosed in a red box.
- Specifically set the location of the input variable**: A blue arrow points to the `location = 0` part of the `layout` statement, which is underlined in blue.
- All the input vertex attributes**: A pink arrow points to the `in vec3 aPos;` line, which is enclosed in a pink box.
- Output of the vertex shader**: A brown arrow points to the `gl_Position` variable in the assignment statement, which is underlined in brown.
- (x, y, z) + w component**: A green arrow points to the `aPos.x, aPos.y, aPos.z, 1.0` list of arguments in the `vec4` constructor, which is underlined in green.

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

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

Primitives Assembly



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

Sample code:

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

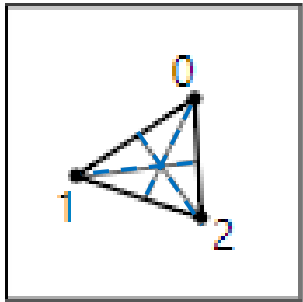
OpenGL function
that draws a shape

Kind of primitive to render

Starting index in the array

Number of vertices to render

Tessellation



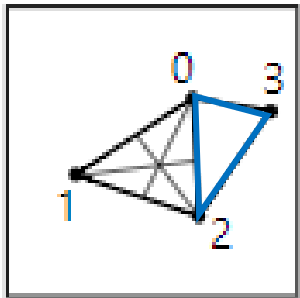
In 3D, the surfaces are built with triangular tiles

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

Geometry Shader



An unnecessary step



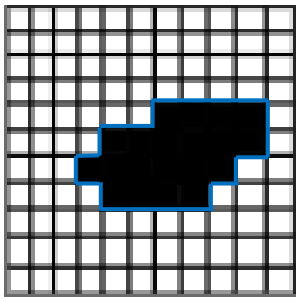
Allows to modify the geometry of each polygon and allows to create new polygons by emitting new vertices

Input data: data of a geometric primitive

Output data: data of one or more geometric primitive

Rasterization

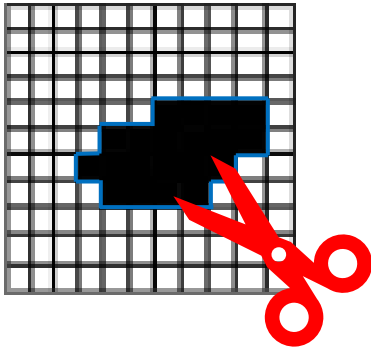
Method of converting a vector image into a raster image to be displayed on a screen



Vector image
composed of geometric objects

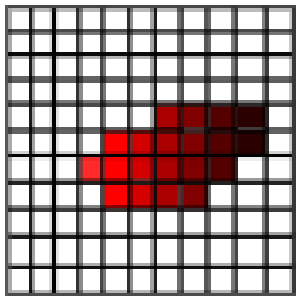
Raster image or Bitmap
composed of pixels

Clipping



This step discards 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)

Output variable which is
the final color output

```
#version 330 core
out vec4 FragColor;

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

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"
    "}\n0";
```

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);
```

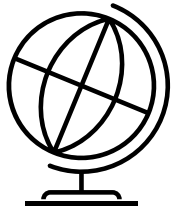
We can now activate this program to render and draw an object

```
glUseProgram(shaderProgram);
```

Final step is to delete our shader objects

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

Uniforms variables



Useful to pass data from the application on the CPU to the shaders on the GPU

These are global variables

Sample code:

Usage of uniform keyword

```
#version 330 core
out vec4 FragColor;

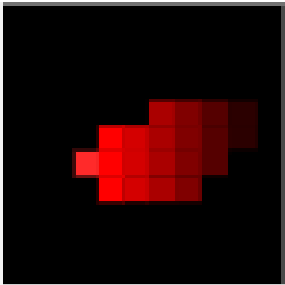
uniform vec4 ourColor;

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

Alpha test

Checks the corresponding depth value of a fragment to see if the resulting fragment is **in front** or **behind** another one

NOT DISCARDED ←  → DISCARDED 

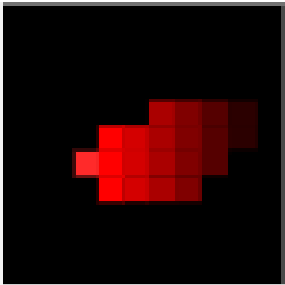


Done with the depth testing using a **Z-buffer** (in which the depth value of the fragments is stored)

```
glEnable(GL_DEPTH_TEST);
```

Then, checks for alpha values (opacity of an object)
& blends the objects

Color Blending



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

Example of a blending function

First, we have to enable the OpenGL functionality

```
glEnable(GL_BLEND);
```

Then, blending can follow this equation:

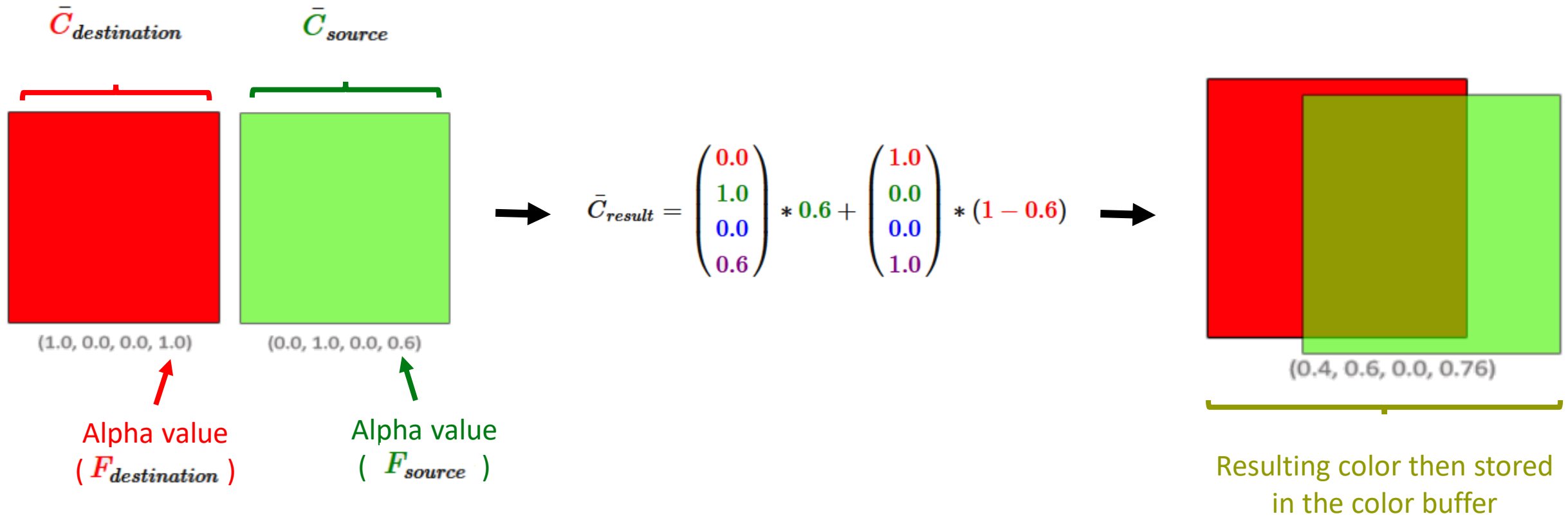
$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

The diagram illustrates the blending equation with color-coded terms and arrows explaining their meaning:

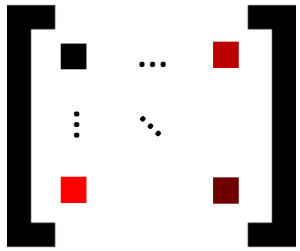
- \bar{C}_{result} : Final color of the fragment (indicated by a blue arrow)
- \bar{C}_{source} : Color output of the fragment shader (indicated by a green arrow)
- F_{source} : Impact of the alpha value (indicated by a green arrow)
- $\bar{C}_{destination}$: Color currently stored in the color buffer (indicated by a red arrow)
- $F_{destination}$: Impact of the alpha value (indicated by a red arrow)

Example (Cont.)

$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$



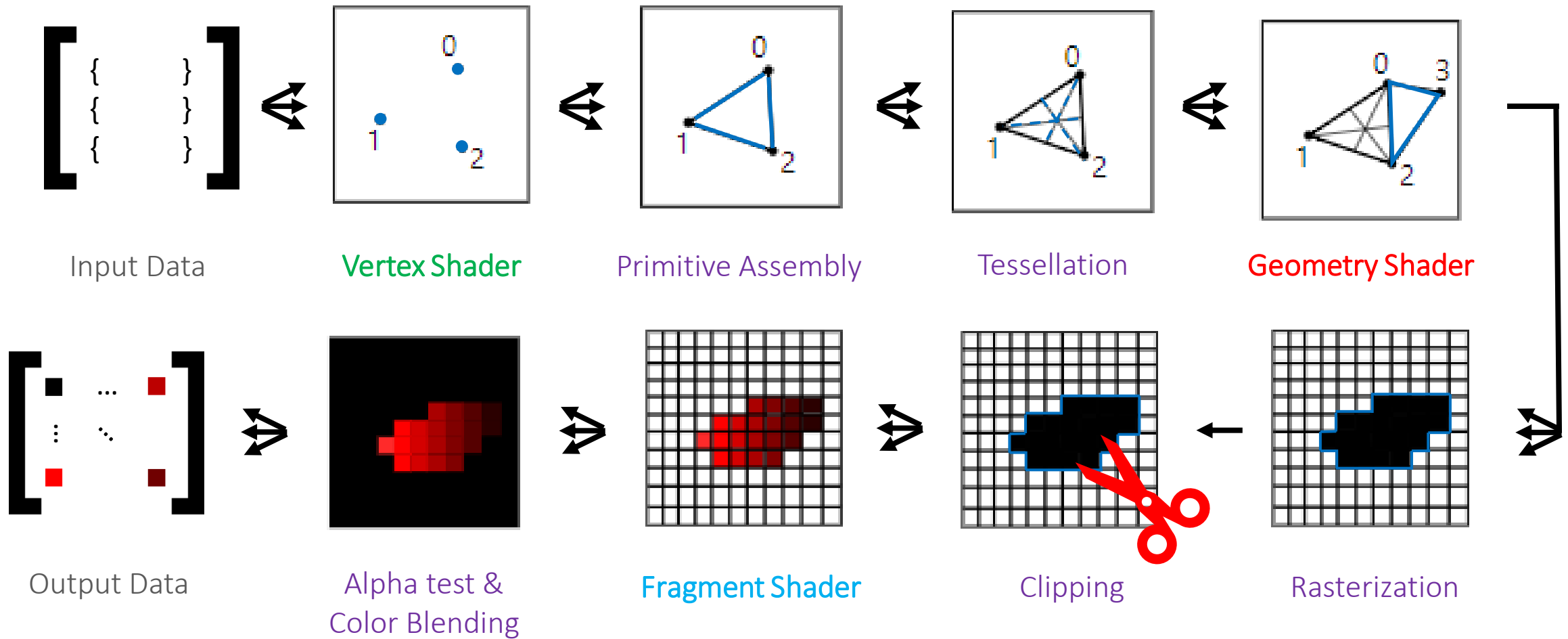
Output Data



Return a **Framebuffer**

The information in this buffer are the values of the color components (**R****G****B**) for each pixel

Overall view



Textures

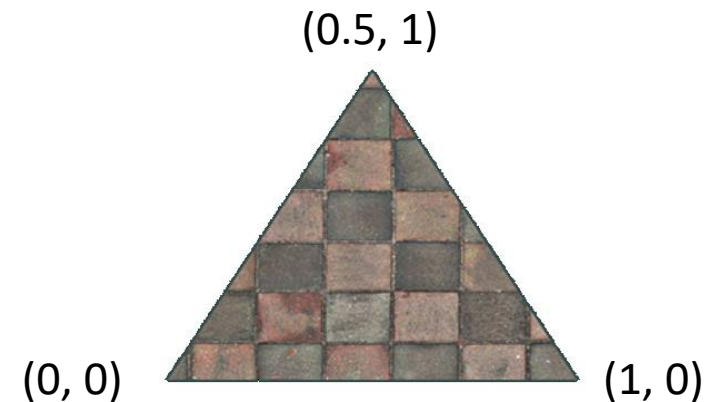
Allows to give the illusion the object is detailed
without having to specify vertices

Associate each vertex to a texture coordinate

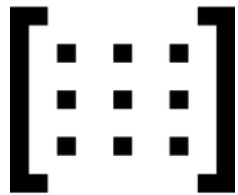
A fragment interpolation is then done for the other fragments

Sample code:

```
float texCoords[] = {  
    0.0f, 0.0f, // lower-left corner  
    1.0f, 0.0f, // lower-right corner  
    0.5f, 1.0f // top-center corner  
};
```



Transformations



Make an object dynamic using matrix objects
& by combining the matrices



Some library can be used like the GLM (OpenGL
Mathematics) library

Useful matrices

Scaling Matrix

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Rotation Matrix

Around X-axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Around Y-axis

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x + \sin \theta \cdot z \\ y \\ -\sin \theta \cdot x + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

Around Z-axis

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta \cdot x - \sin \theta \cdot y \\ \sin \theta \cdot x + \cos \theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Sample code

Translating a vector of (1,0,0) by (1,1,0)

The diagram illustrates the process of translating a vector using GLM. It features a code block with two lines of C++ code. Four annotations with arrows point to specific parts of the code: a green arrow points to the first 'trans' argument in the `glm::translate` function call, labeled 'The matrix to transform (identity Matrix4)'; a blue arrow points to the `glm::vec3(1.0f, 1.0f, 0.0f)` argument, labeled 'The translation vector to apply to the matrix'; a purple arrow points to the `vec = trans * vec;` line, labeled 'Multiply vec by the translation matrix'; and a light blue arrow points to the `glm::translate` function name, labeled 'A translate function'.

```
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));  
vec = trans * vec;
```

The matrix to transform
(identity Matrix4)

The translation vector to
apply to the matrix

Multiply vec by the
translation matrix

A translate function

Coordinates system

Transforming coordinates to NDC is done by a process regrouping several intermediate coordinate systems



Local Space



World Space



View Space

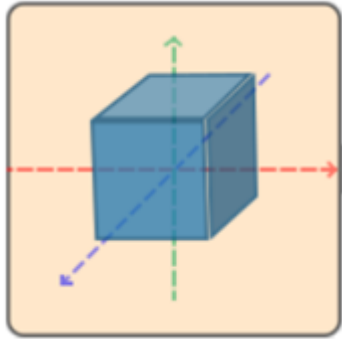
Clip Space



Screen Space



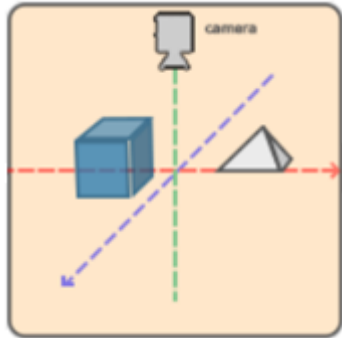
Local Space



Coordinates of the object relative to its local origin

In general, all new objects have $(0, 0, 0)$ as initial position

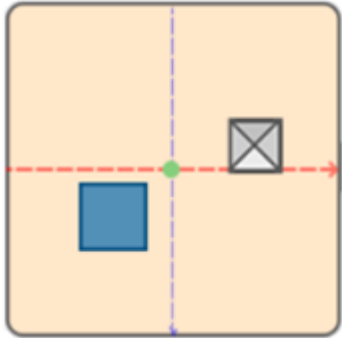
World Space



Coordinates of all the objects are relative to some global origin of the world

We use a model matrix which translates, scales and/or rotates the object to place it in the world

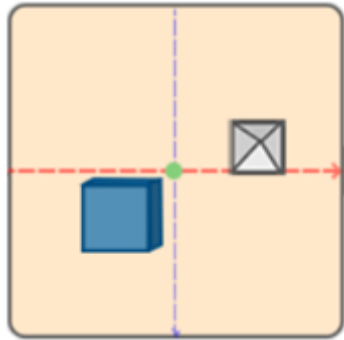
View Space



Each coordinates is seen from the camera's point of view

This is done by a combination of translations & rotations of the scene which is stored in a view matrix

Clip Space



Each coordinates is seen from the camera's point of view

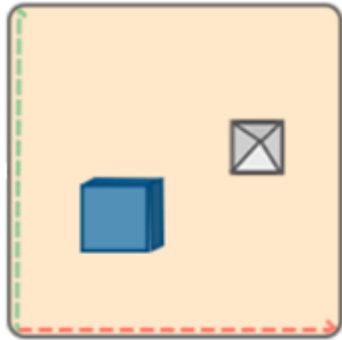
For this step, we use a projection matrix which transform the coordinates into NDC

Example:

Specified range $[-1000, 1000]$
for each dimension

{	$(1250, 500, 750)$	→	Not visible
	$(900, 500, 750)$	→	Visible

Screen Space

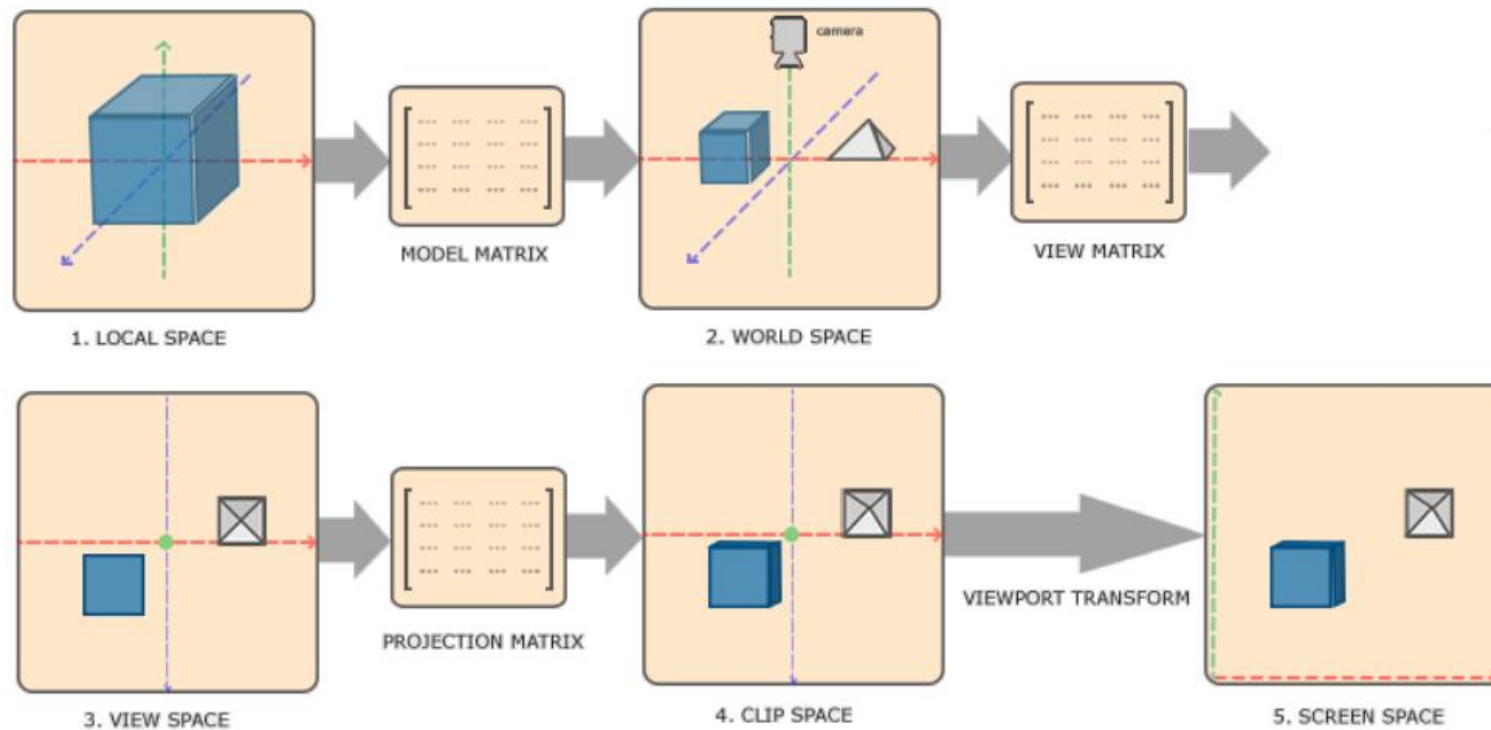


Transforms the NDC coordinates to the window coordinates with the *glViewport()* function

Resulting coordinates are then sent to the rasterizer

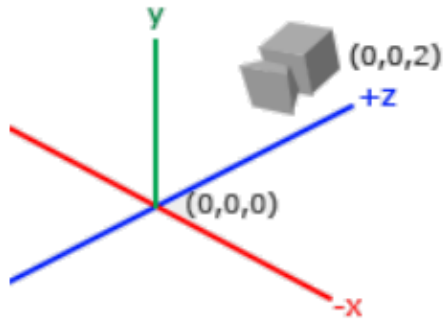
Overall view

A vertex coordinate is transformed to clip coordinates as follow: $V_{clip} = \underline{M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}}$

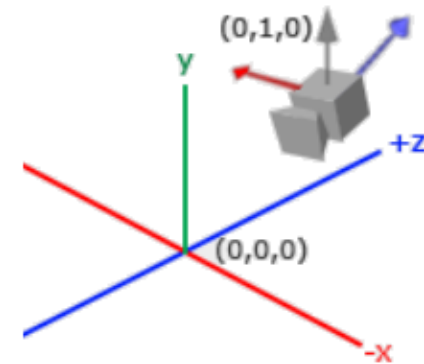


Camera

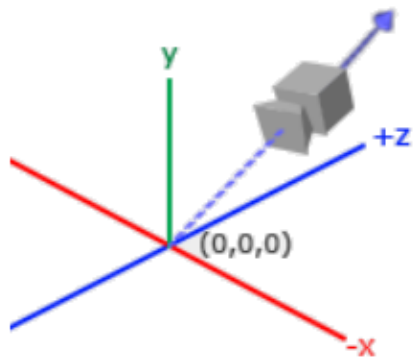
To define a camera we need 4 pieces of information



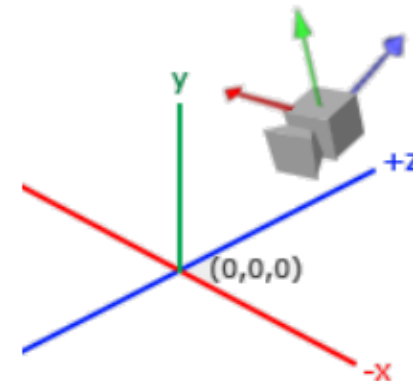
1. Its position in the world space



3. A vector pointing to the right

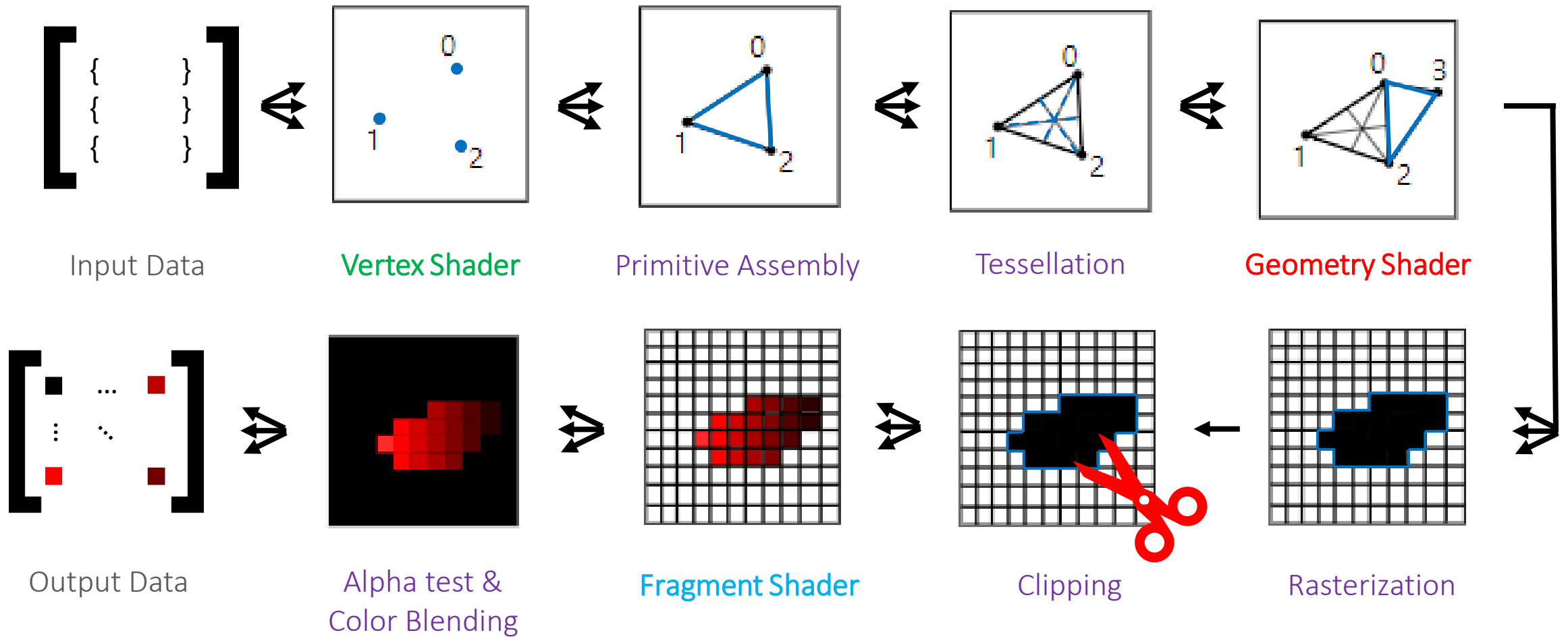


2. Its direction



4. A vector pointing upwards

Recall : Graphics pipeline



Pipeline abstraction

We can see the pipeline as a function composition which can give us:

$$output_data = (cb \circ at \circ fs \circ c \circ r \circ t \circ pa \circ vs) (input_data)$$

Fragment Shader function

Vertex Shader function

Context

We can define the notion of context that gives us the valid constants for a run (see after [Uniforms](#))

The formula below is applied for every run

$$output_data = (cb \circ at \circ fs \circ c \circ r \circ t \circ pa \circ vs) (input_data)$$

Recall: Input data



A VBO is built containing the attributes of all vertices which give us a huge vector of data

To work with that, we have to use offsets, strides, etc.

Idea of the abstraction



No longer working with containers of type

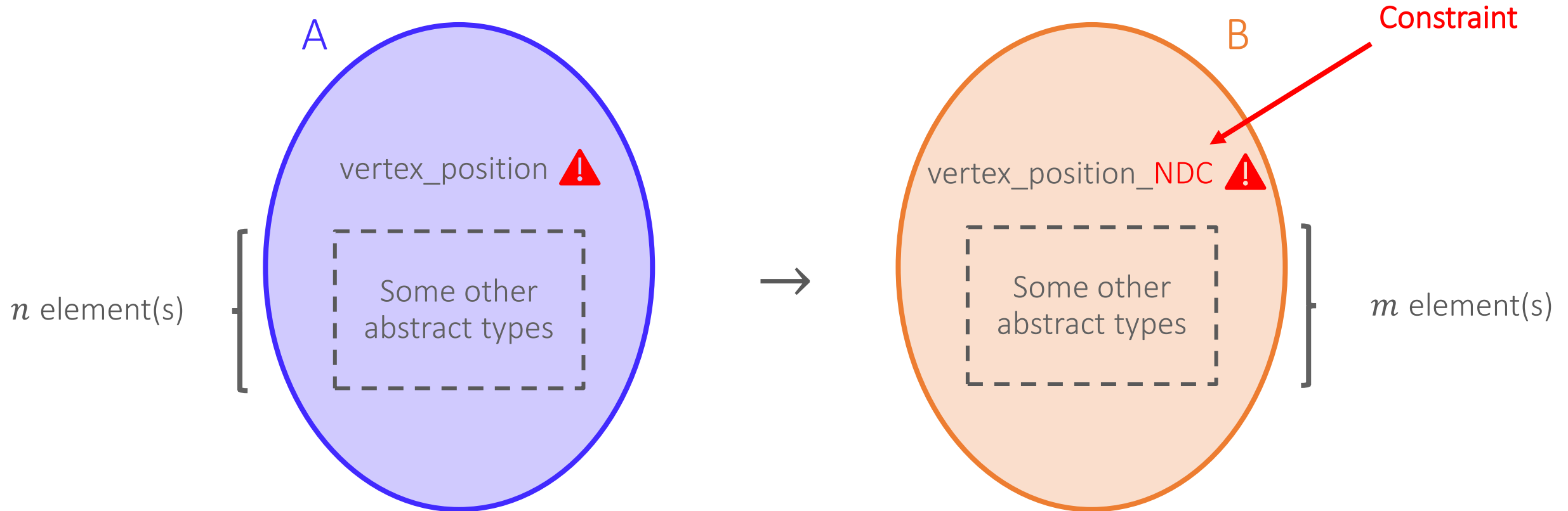
Ex: ~~vec3, vec4, ivec4, mat4, ...~~



But with abstract type objects

Ex: color, position, textures, ...

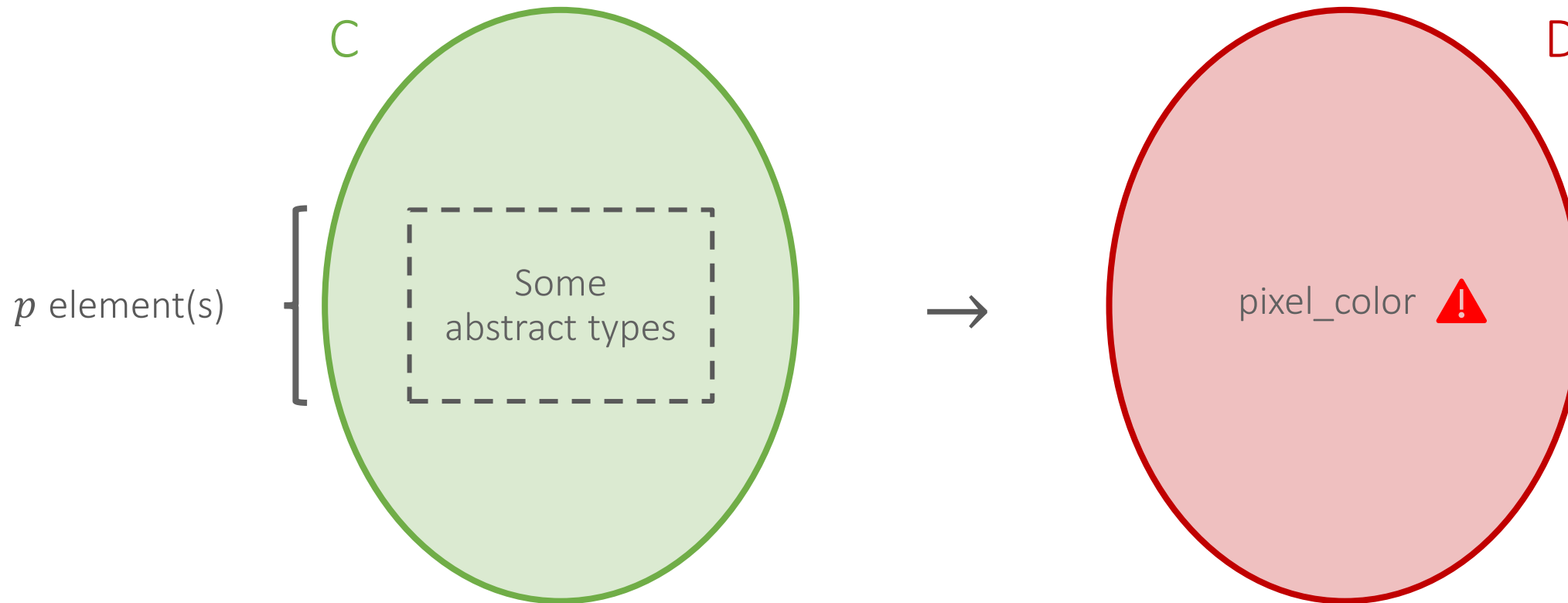
Vertex Shader function $vs : A \rightarrow B$



$vs(vertex_position: \cup P_{k \leq n}(A \setminus vertex_position):)$

Fragment Shader function

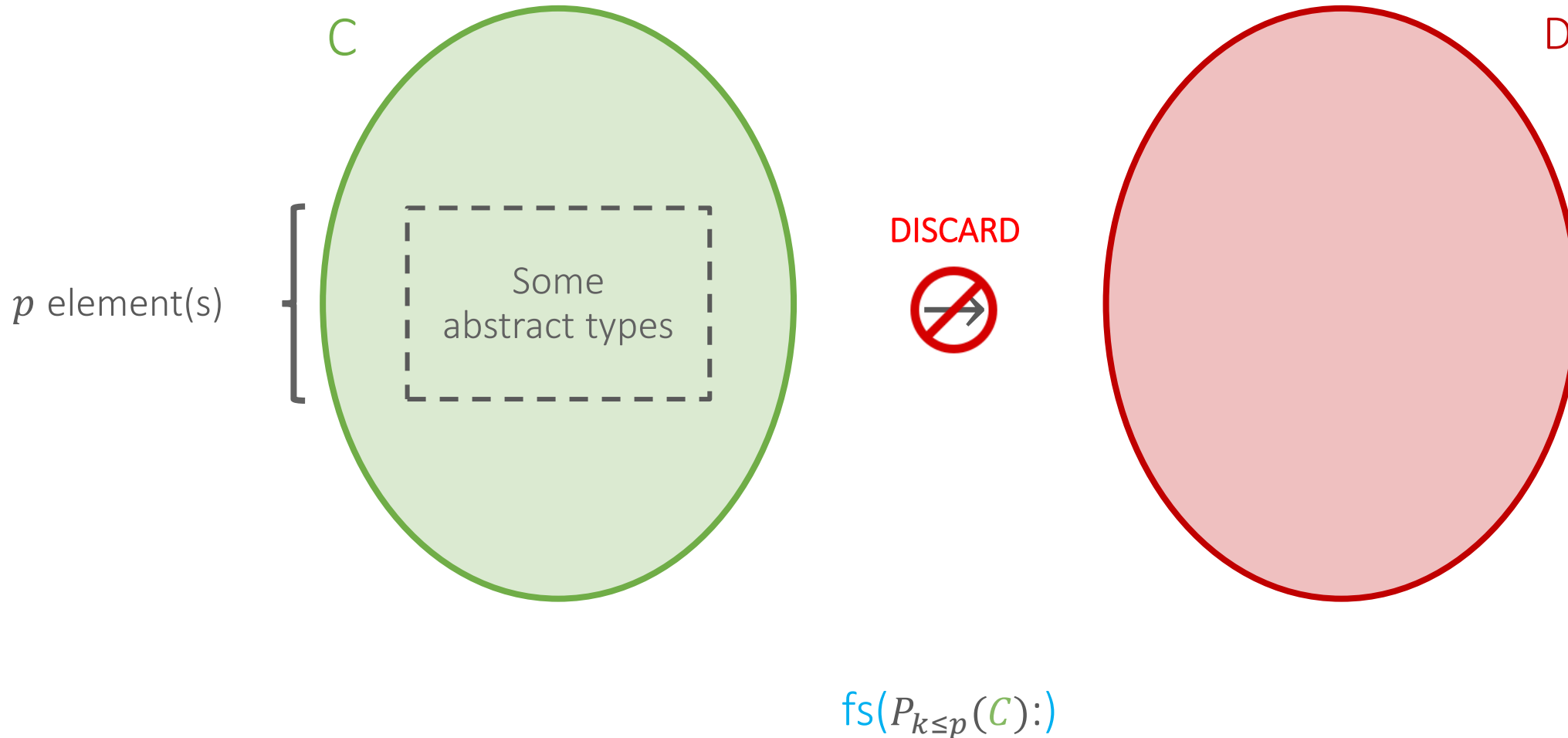
$$fs : C \rightarrow D$$



$$fs(P_{k \leq p}(C):)$$

Fragment Shader function Alternative

$$fs : C \rightarrow D$$



Several signatures



Depending on why a shader is created,
the signature will be different

Examples:

`vs(position:)`

`fs()`

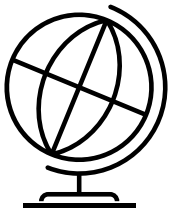
`vs(position: color:)`

`fs(fragment: light:)`

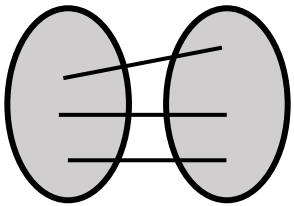
`vs(position: color: texture:)`

`fs(fragment: light: texture:)`

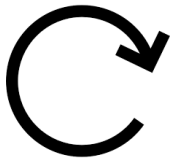
Uniforms



We saw that uniform variables are global variables



They are part of the domain and
the codomain of the `vs()` & `fs()` function



These variables are set for a run and define the context

Type checking between `vs()` & `fs()`

Check that names and types variables shared between the vertex & the fragment shader are identical

Example:

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

out vec4 vertexColor;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0);
}
```

Vertex shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor;

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

Fragment shader

Recall : Different languages



DirectX High-Level Shader Language
(Unreal Engine)



Cg Shader Language
(Unity)



OpenGL Shading Language

Similar structures

A sample Cg vertex shader:

Types definition

Calculate output
coordinates & colors

Output

```
// input vertex
struct VertIn {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
};

// output vertex
struct VertOut {
    float4 pos    : POSITION;
    float4 color  : COLOR0;
};

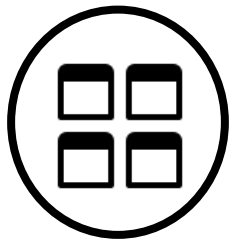
// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    {
        OUT.pos      = mul(modelViewProj, IN.pos);
        OUT.color    = IN.color;
    }
    return OUT;
}
```

Uniform keyword

Same abstraction



The different shader languages are very similar



We could therefore use the same abstraction for any language

Domain-Specific Language (DSL)

A DSL is a programming language whose specifications allow to overcome some constraints in a specific domain



The specific domain will be for us the shaders and especially vertex & fragment shaders

Advantages & disadvantages



DSL will allow us to gain in productivity

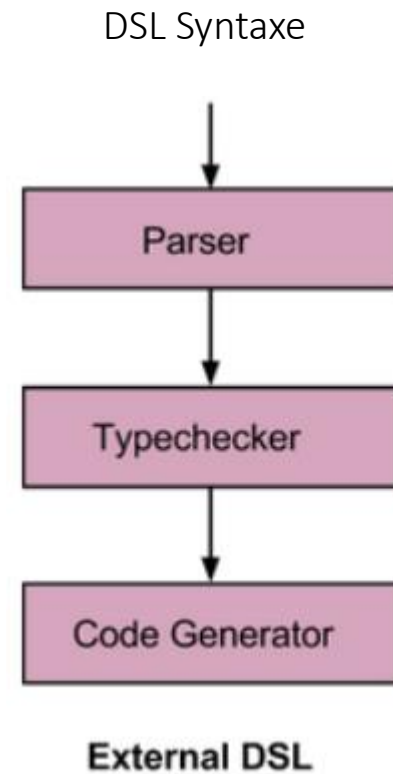
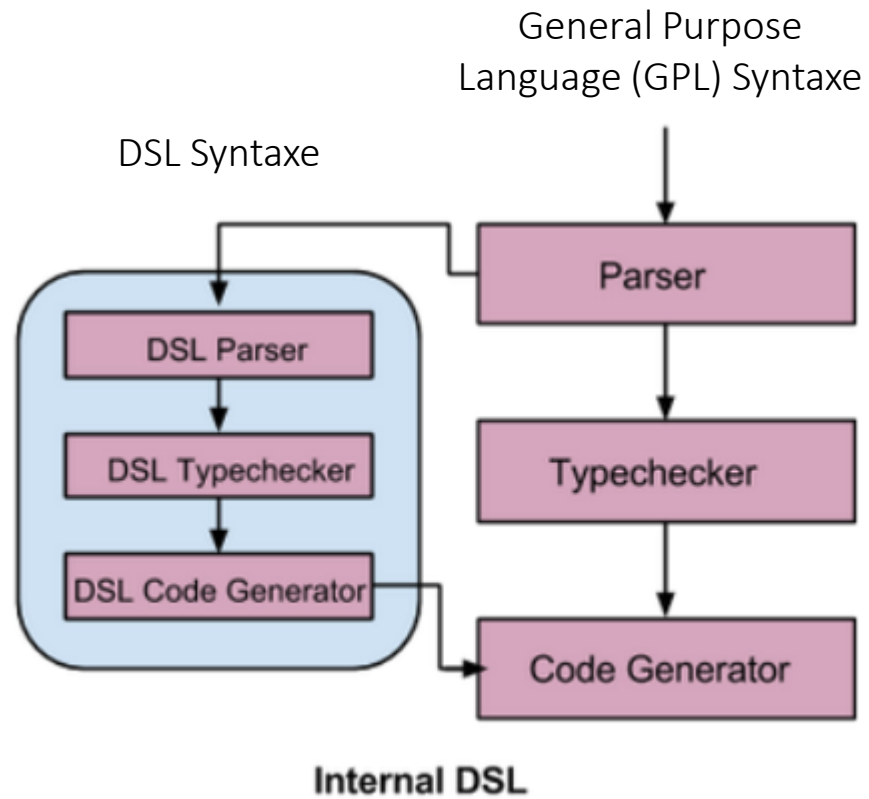
DSL can be reused for other purposes



DSL maintenance is complicated

The cost of a DSL is expensive

Different types of DSL



Our way



First, we will go on an Internal DSL based on the Swift language

Later, we can potentially encounter a lot of constraints relating to Swift



If so, we will go on an External DSL at this time

Main idea



We write in our program a vertex & a fragment shader with our DSL



We send them to an encoder which will translate the abstract types into containers of types



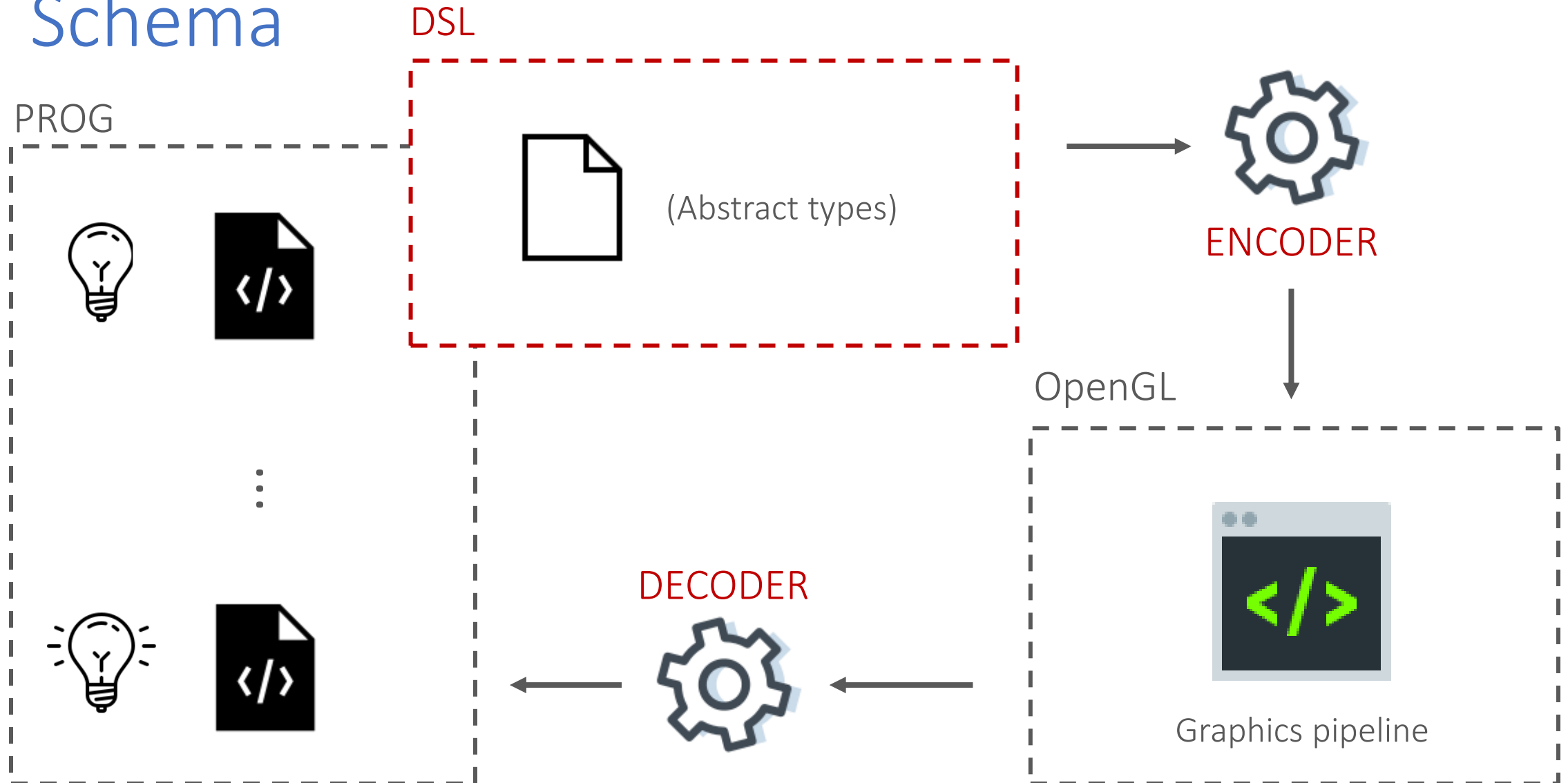
This translation can then be evaluated by the graphics pipeline of OpenGL



Then a decoder allows us to get the results with the desired abstract types



Schema



Work incoming



Learn more about DSL & the creation of abstract types

Steps to create an Internal DSL in Swift

How to build the abstraction

How to link the DSL to the OpenGL pipeline



(To document more about other shader languages)



Begin to work with Rendery

References

<https://learnopengl.com>

<https://developer.apple.com/metal>

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

<https://github.com/RenderyEngine/Rendery>

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

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

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

[https://en.wikipedia.org/wiki/Domain-specific language](https://en.wikipedia.org/wiki/Domain-specific_language)

<https://tomassetti.me/domain-specific-languages/>

Working with shaders

Patrick SARDINHA