Working with shaders

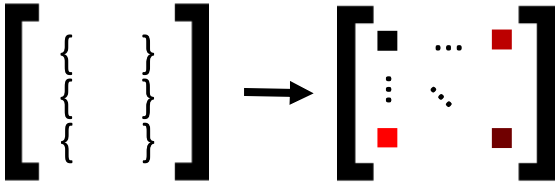
Patrick SARDINHA

What’s a shader?

Small programs that run on the GPU

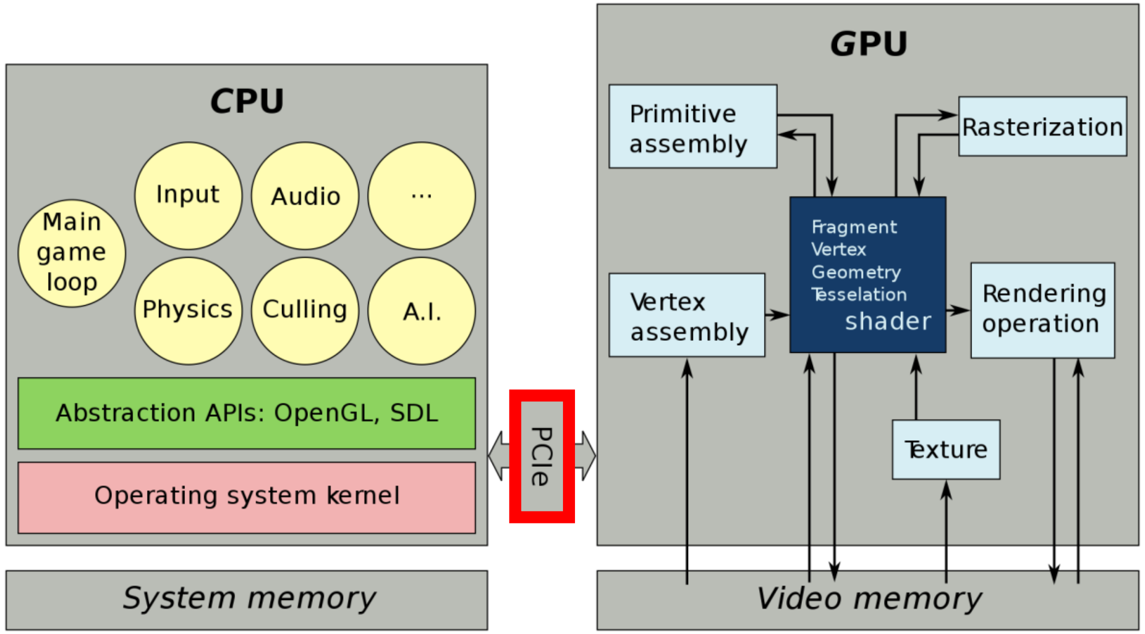
Executed for each specific section of the graphics pipeline

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

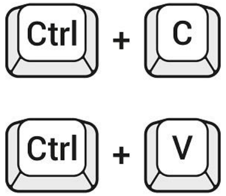
DirectX High-Level Shader Language

Cg Shader Language

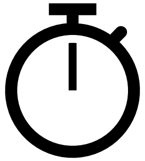
OpenGL Shading Language (GLSL)

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



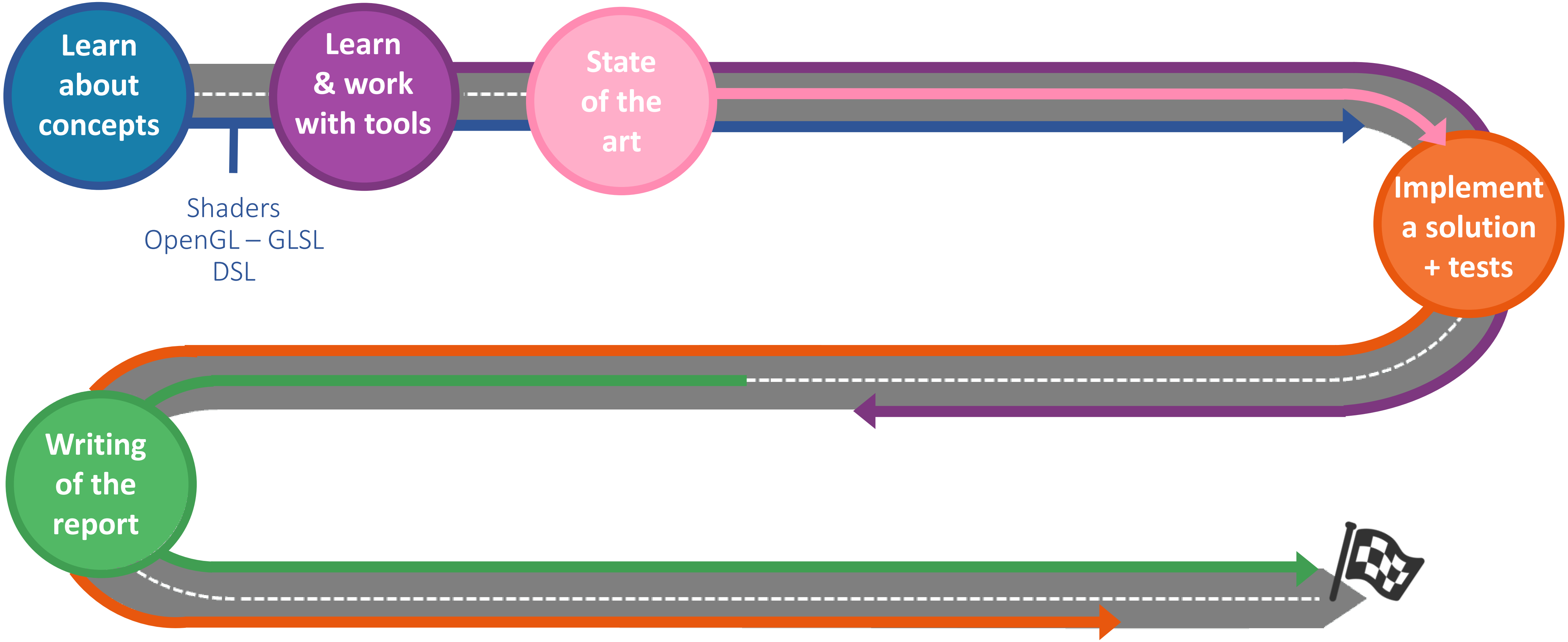
Must reduce the data sent in the PCI to avoid multiple synchronizations between CPU & GPU

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
DSL

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

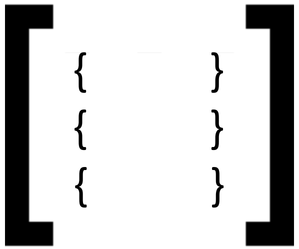
3 different shaders processing units

- 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:

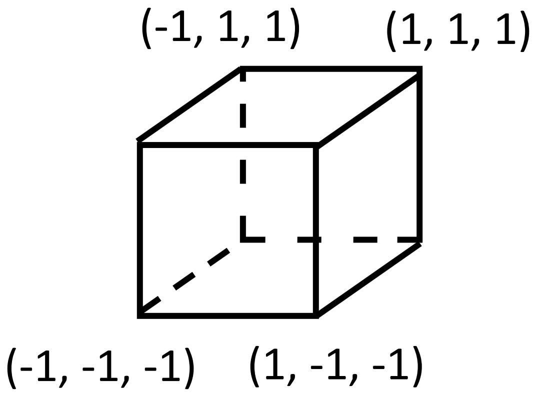
Position (2D, 3D coordinates)

Color (RGB, ...)

Texture coordinates

Example

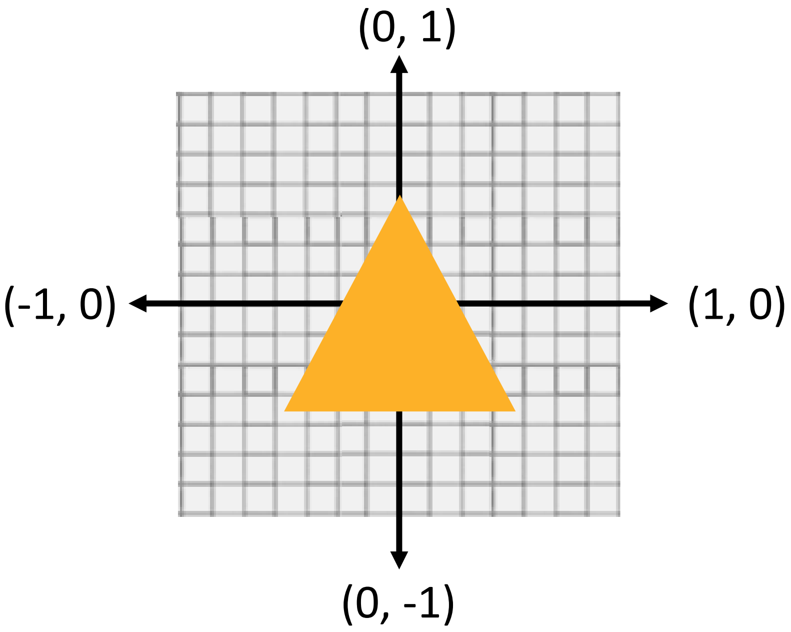
In OpenGL, only the Normalize Device Coordinates (NDC) are visible on the screen



To render a single 2D triangle:

3D position (NDC) of each vertex

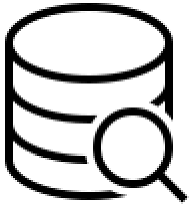
```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```



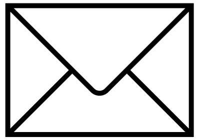
Linking vertex attributes

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
};
```

Copy our vertices array in a buffer

ID of the buffer which must be bind

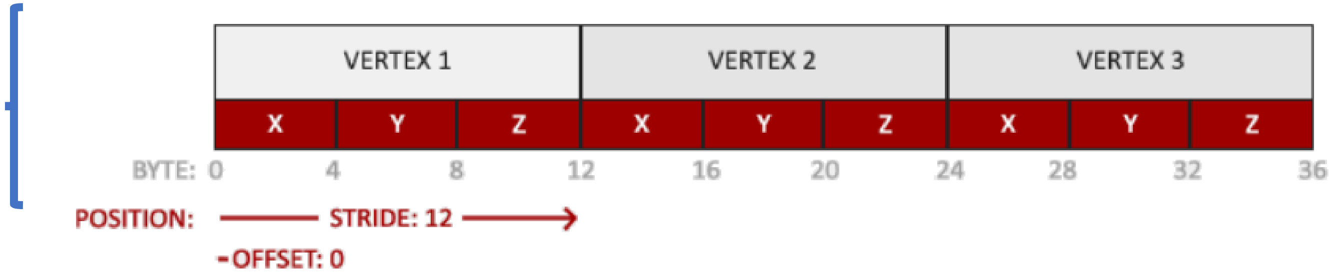
Specifies the target buffer object

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

Size of the buffer object

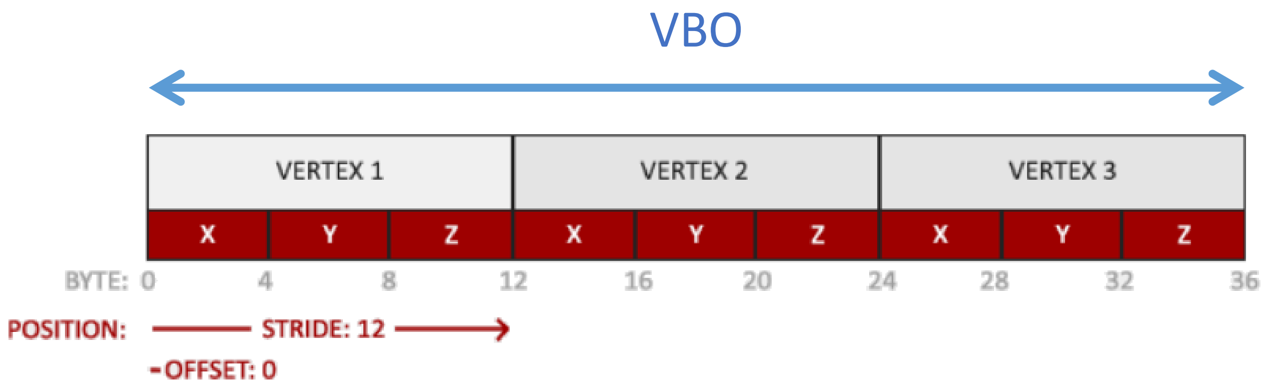
Pointer to data

VBO



Example (Cont.)

Define how the vertex data should be interpreted



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

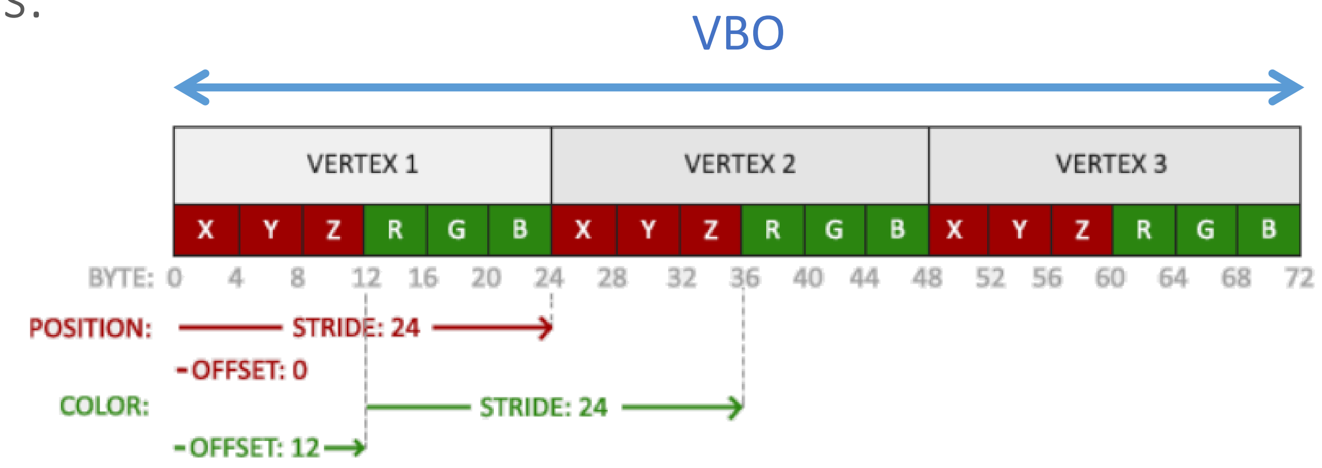
Type of the data

Stride: Space between consecutive vertex attributes

Example (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 offset

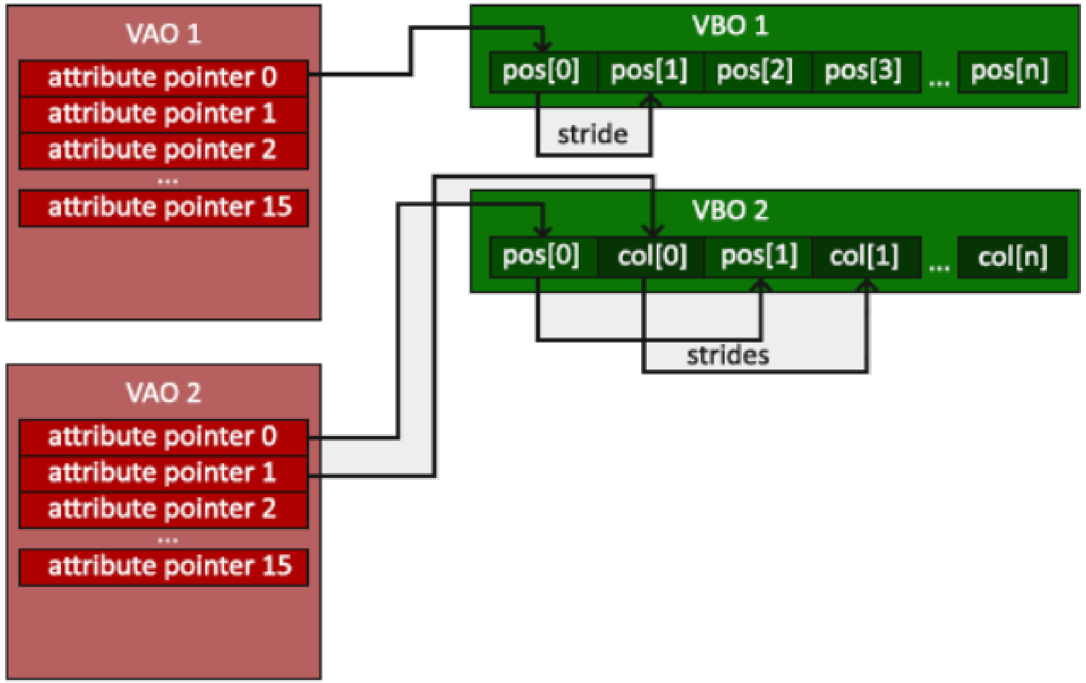
```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float))
glEnableVertexAttribArray(1);
```

Stride

Color offset

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);
```

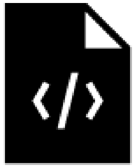
Summary

```
// 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();
```

Source: https://learnopengl.com/Getting-started/Hello-Triangle

Render & draw an object

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

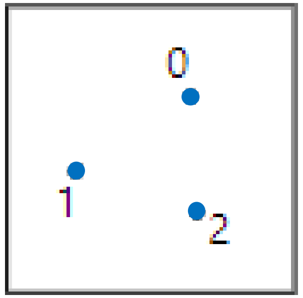


Set up a Vertex & a Fragment Shader

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

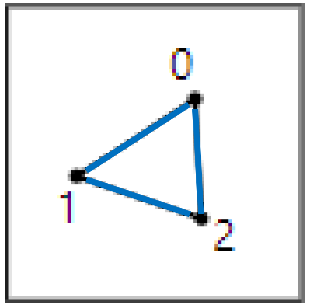
```
#version 330 core
layout (location = 0) in vec3 aPos;

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

Specifically set the location of the input variable

(x, y, z) + w component

Output of the vertex shader

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:

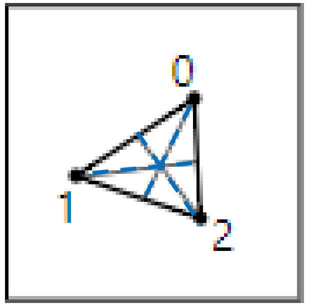
```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

OpenGL function that draws a shape

Kind of primitive to render

Number of vertices to render

Tessellation

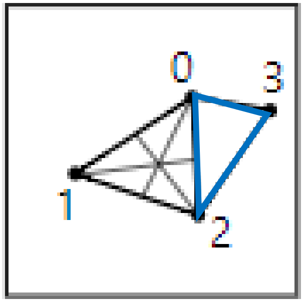


In 3D, the surfaces are built with triangular tiles

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

Geometry Shader

An unnecessary step

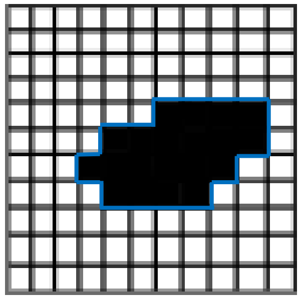


Allows to modify the geometry of each polygon and allows to create new polygons by emitting new vertices

Input data: data of a geometric primitive

Output data: data of one or more geometric primitive

Rasterization

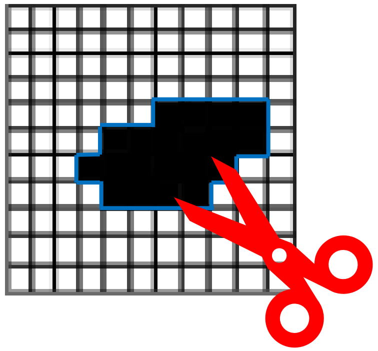


Method of converting a vector image into a raster image to be displayed on a screen

Vector image
composed of geometric objects

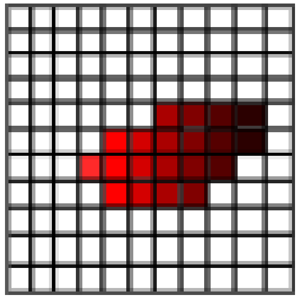
Raster image or Bitmap
composed of pixels

Clipping



This step 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);
```

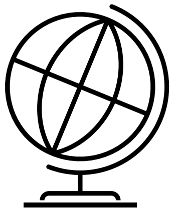
We can now activate this program to render and draw an object

```
glUseProgram(shaderProgram);
```

Final step is to delete our shader objects

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

Uniforms variables



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

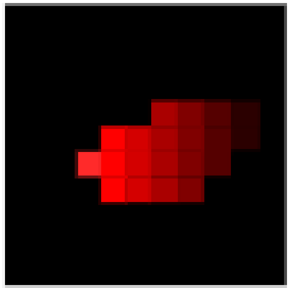
These are global variables

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

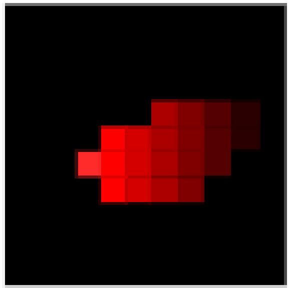
NOT DISCARDED ← in front | behind → DISCARDED

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:

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

Impact of the alpha value

Final color of the fragment

Impact of the alpha value

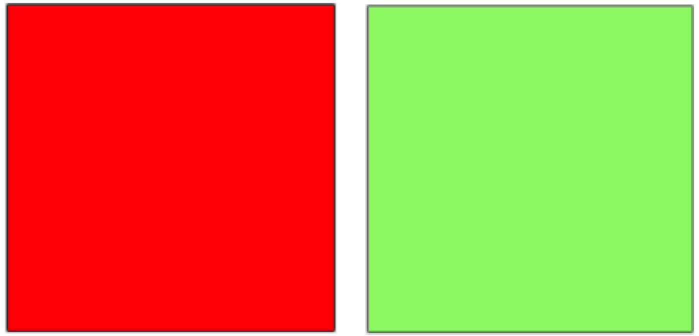
Color output of the fragment shader

Color currently stored in the color buffer

Example (Cont.)

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

$\bar{C}_{destination}$ $\bar{C}_{source}$

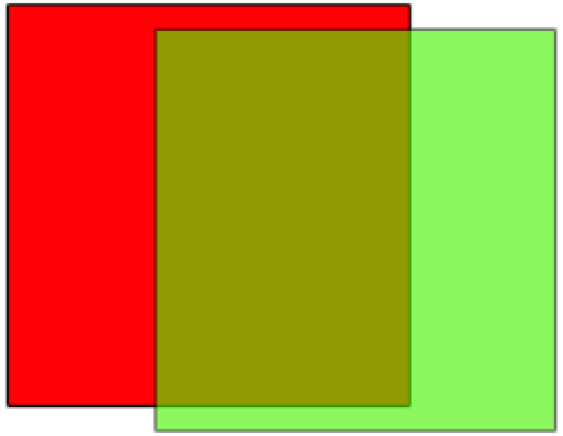


(1.0, 0.0, 0.0, 1.0) (0.0, 1.0, 0.0, 0.6)

Alpha value ($F_{destination}$)

Alpha value (F_{source})

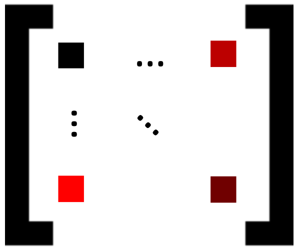
$$\bar{C}_{result} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} * 0.6 + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} * (1 - 0.6)$$



(0.4, 0.6, 0.0, 0.76)

Resulting color then stored in the color buffer

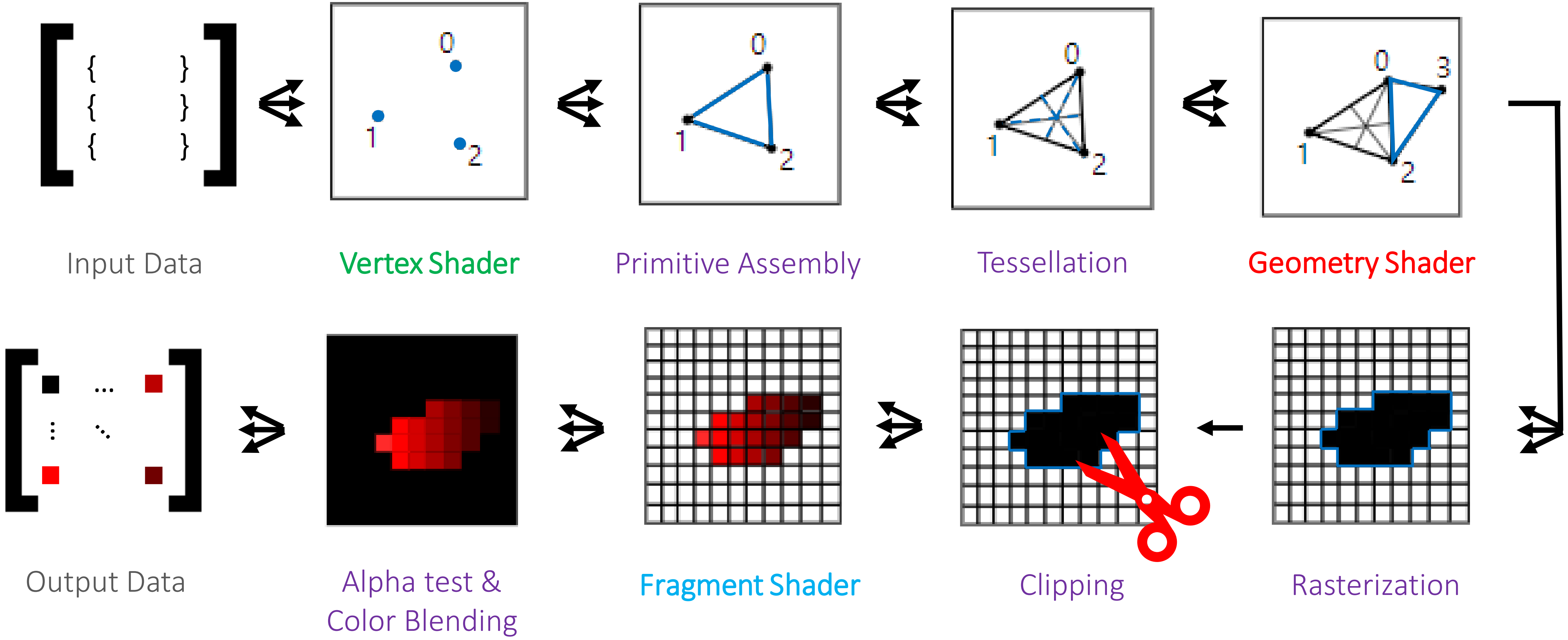
Output Data



Return a Framebuffer

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

Overall view



Textures

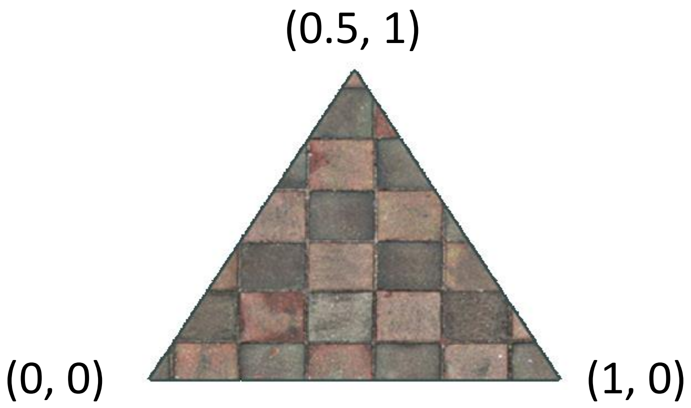
Allows to give the illusion the object is detailed without having to specify vertices

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 matrix to transform
(identity Matrix4)

The translation vector to
apply to the matrix

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

Multiply vec by the
translation matrix

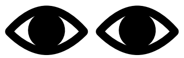
A translate function

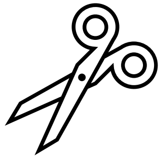
Coordinates system

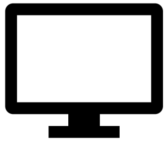
Transforming coordinates to NDC is done by a process regrouping several intermediate coordinate systems


Local Space

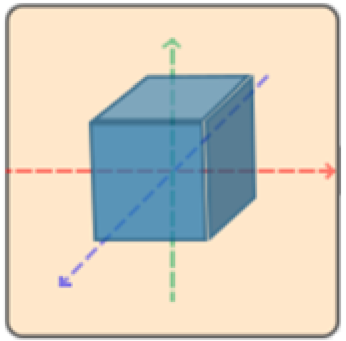

World Space


View Space

Clip Space


Screen Space


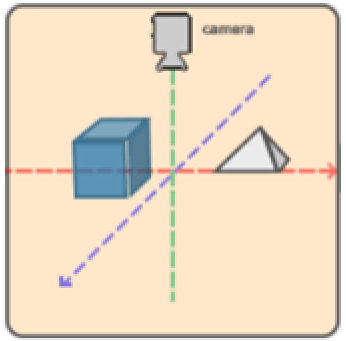
Local Space



Coordinates of the object relative to its local origin

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

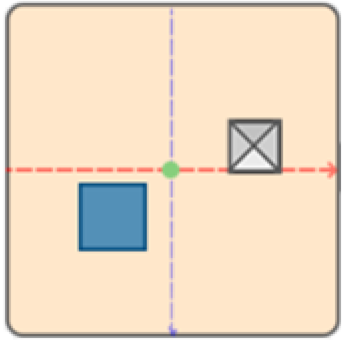
World Space



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

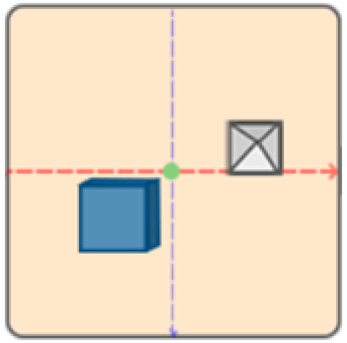
View Space



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

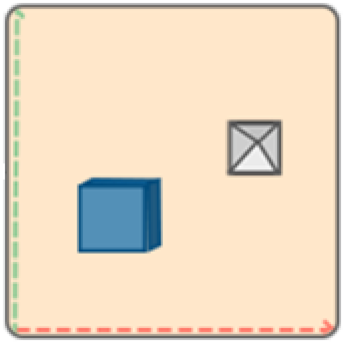
Example:

Specified range [-1000, 1000] for each dimension

(1250, 500, 750) ➞ Not visible

(900, 500, 750) ➞ Visible

Screen Space

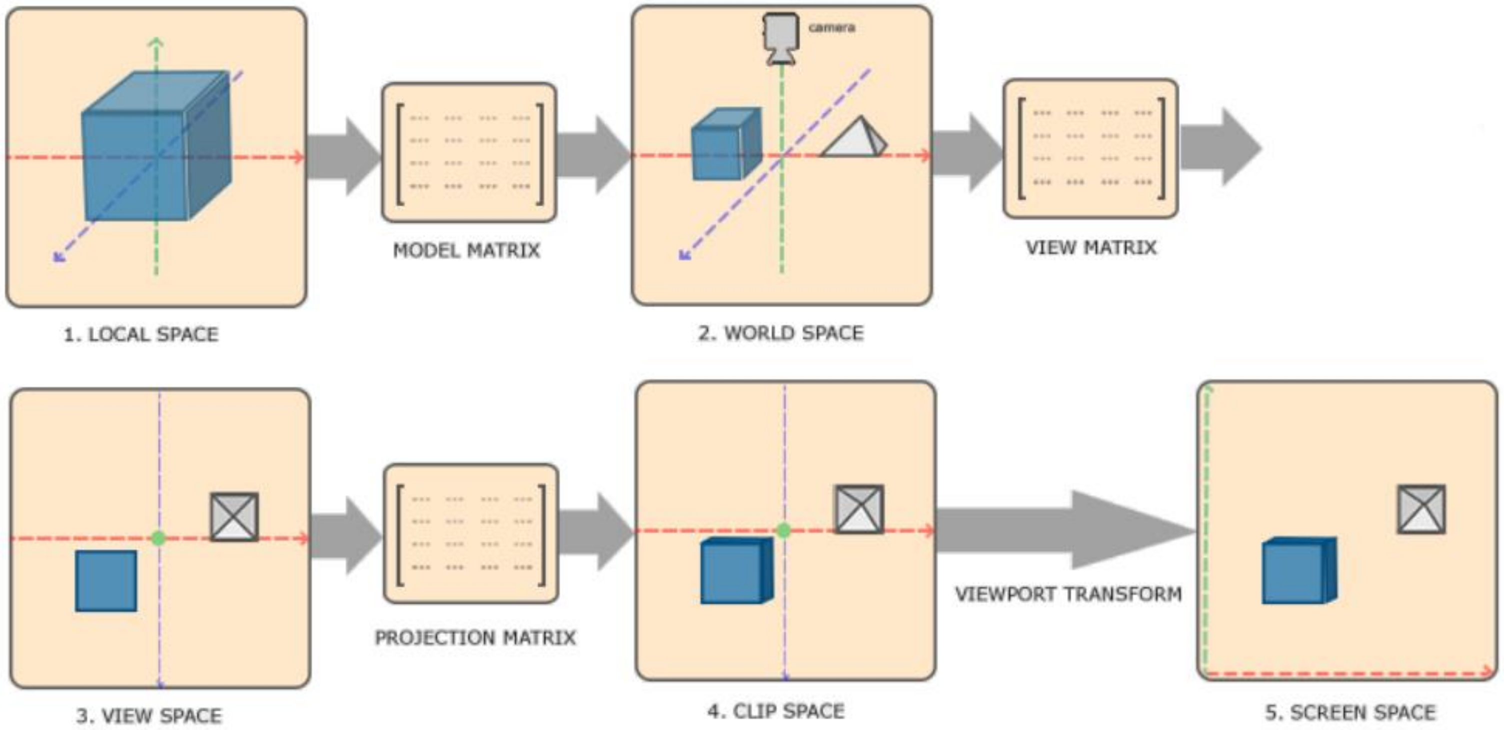


Transforms the NDC coordinates to the window coordinates with the *glViewport()* function

Resulting coordinates are then sent to the rasterizer

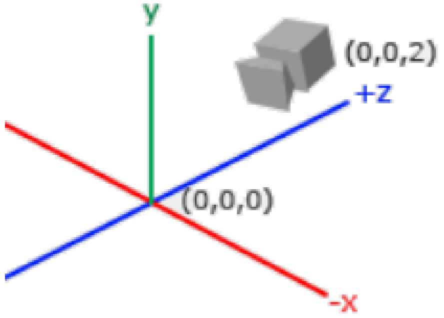
Overall view

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

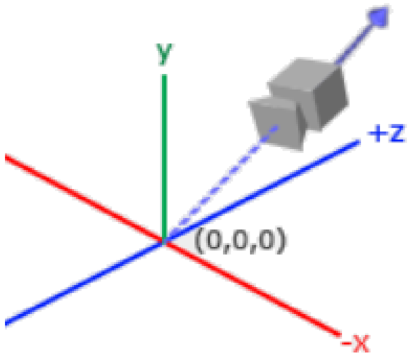


Camera

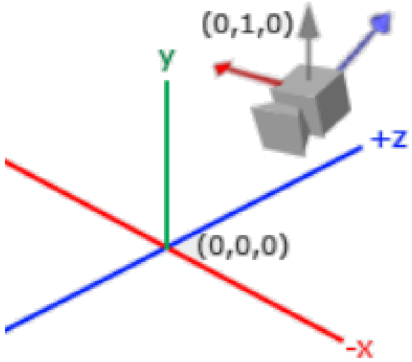
To define a camera we need 4 pieces of information



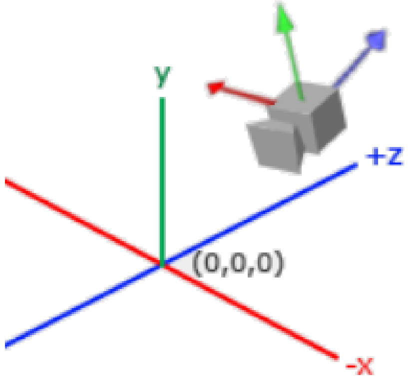
1. Its position in the world space



2. Its direction

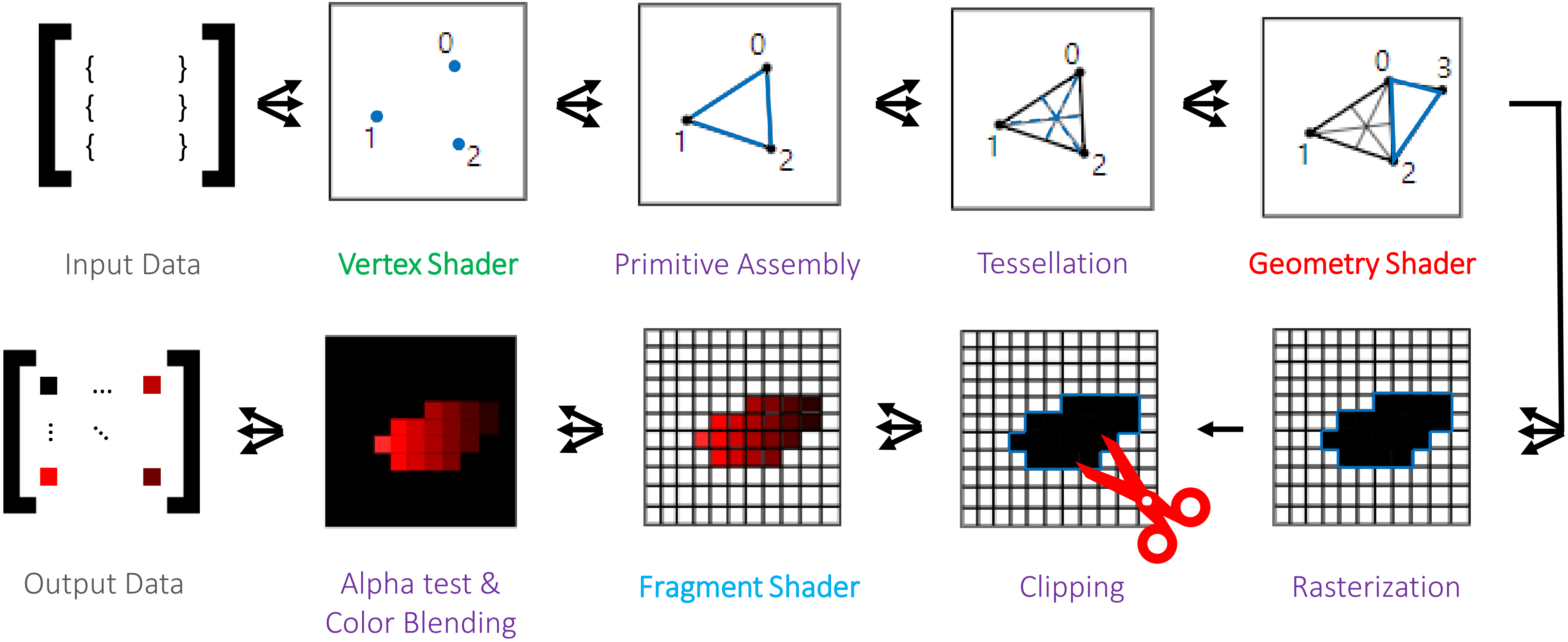


3. A vector pointing to the right



4. A vector pointing upwards

Recall : Graphics pipeline



Pipeline abstraction

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

$$output\_data = (\mathrm{cb} \circ \mathrm{at} \circ \mathrm{fs} \circ \mathrm{c} \circ \mathrm{r} \circ \mathrm{t} \circ \mathrm{pa} \circ \mathrm{vs})\ (input\_data)$$

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

$$\boxed{output\_data = (\mathrm{cb} \circ \mathrm{at} \circ \mathrm{fs} \circ \mathrm{c} \circ \mathrm{r} \circ \mathrm{t} \circ \mathrm{pa} \circ \mathrm{vs})\ (input\_data)}$$

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

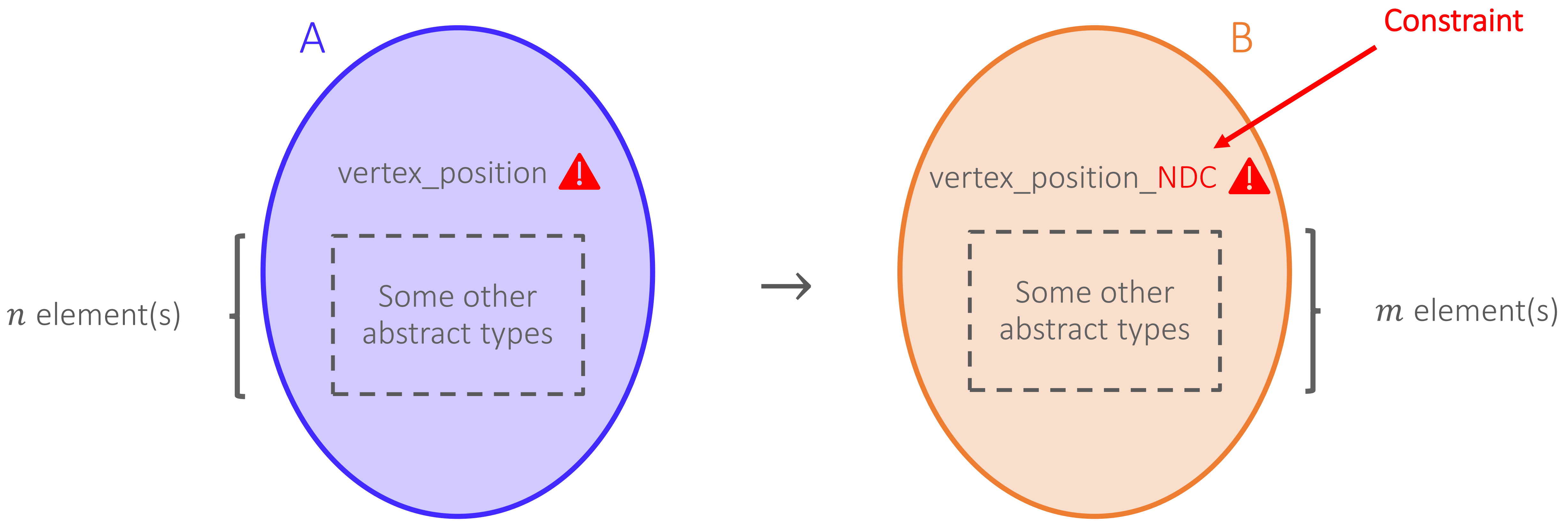
No longer working with containers of type

Ex: ~~vec3, vec4, ivec4~~, mat4, …

But with abstract type objects

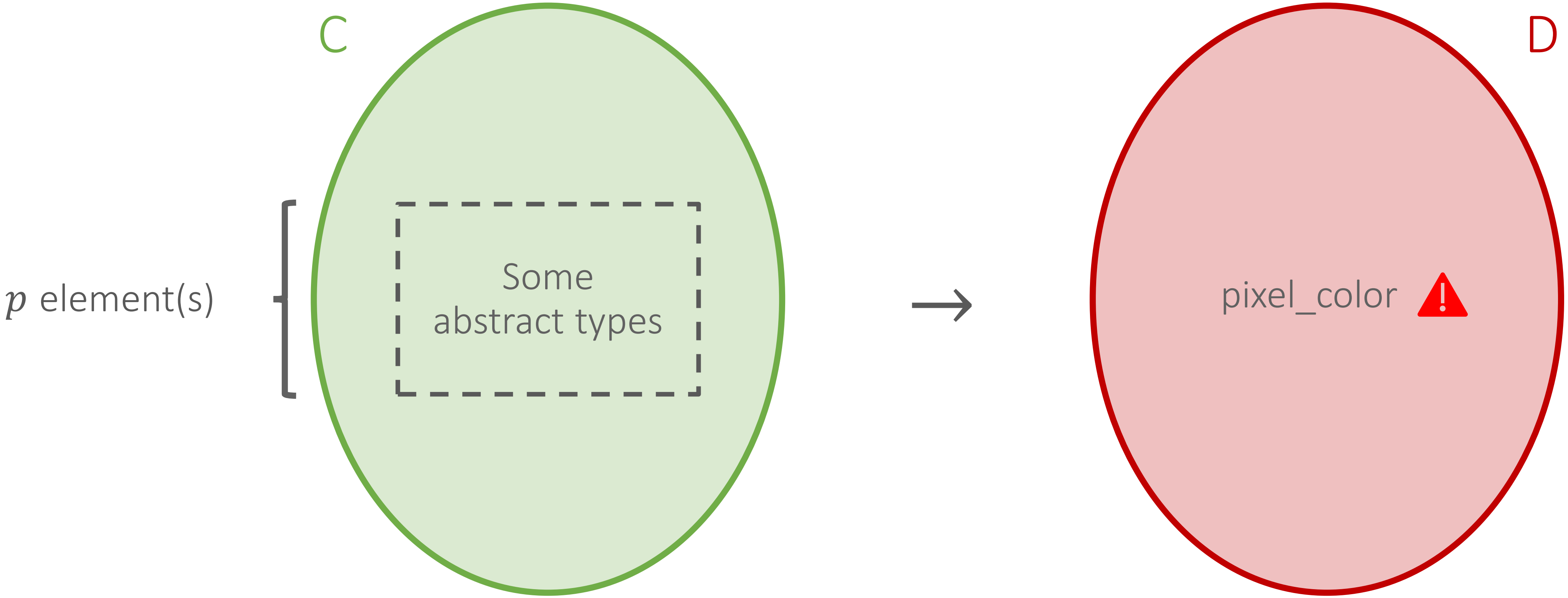
Ex: color, position, textures, …

Vertex Shader function $\quad \mathrm{vs} : A \rightarrow B$



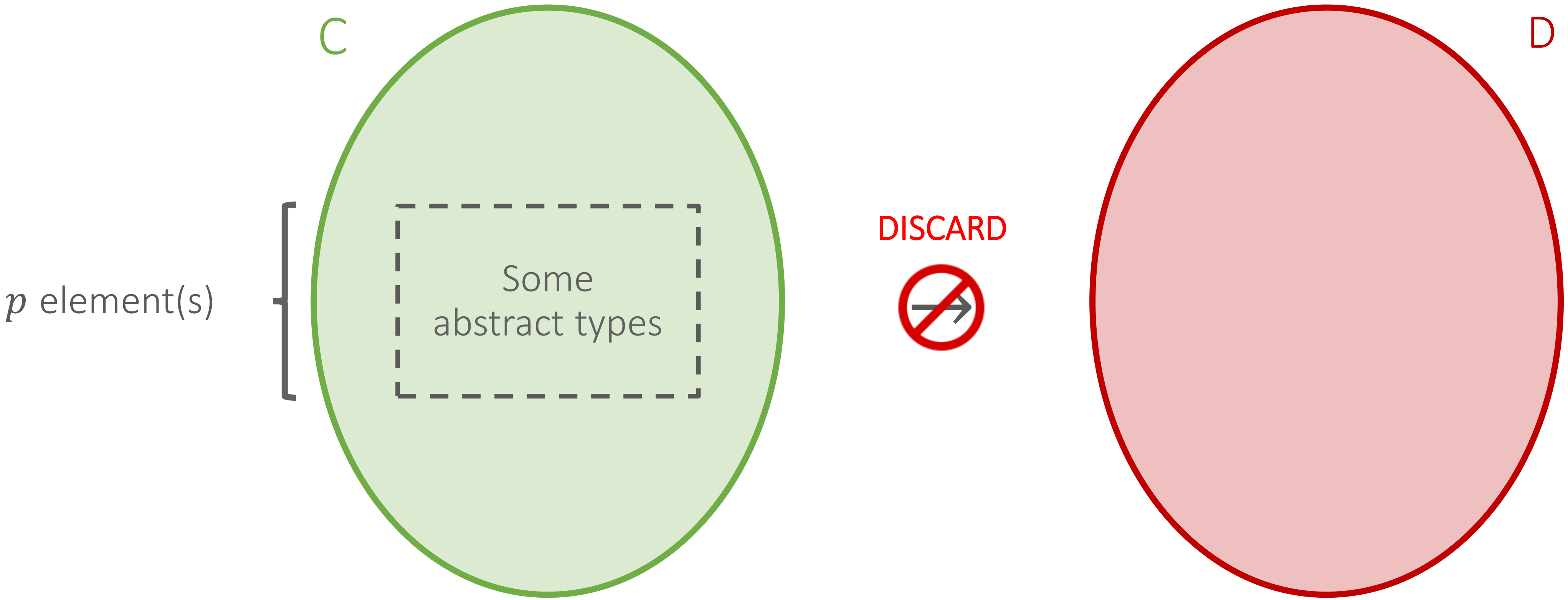
$$\mathrm{vs}(\_ : \mathit{vertex\_position}, \cup \_ : (A_i \setminus \mathit{vertex\_position}))$$

Fragment Shader function   fs : C → D



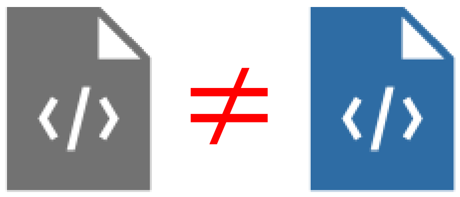
fs(_ : C_i)

Fragment Shader function Alternative   fs : C → D



fs(_ : C_i)

Several signatures

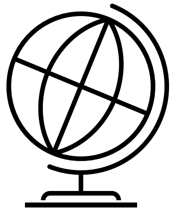


Depending on why a shader is created,
the signature will be different

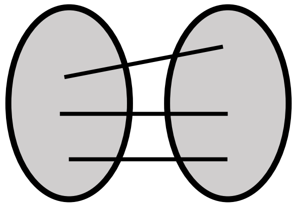
Examples:

vs(_ : position)	fs()
vs(_ : position, _ : color)	fs(_ : fragment, _ : light:)
vs(_ : position, _ : color, _ : texture:)	fs(_ : fragment, _ : light, _ : texture:)

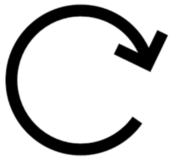
Uniforms



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

```
// 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

Output

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

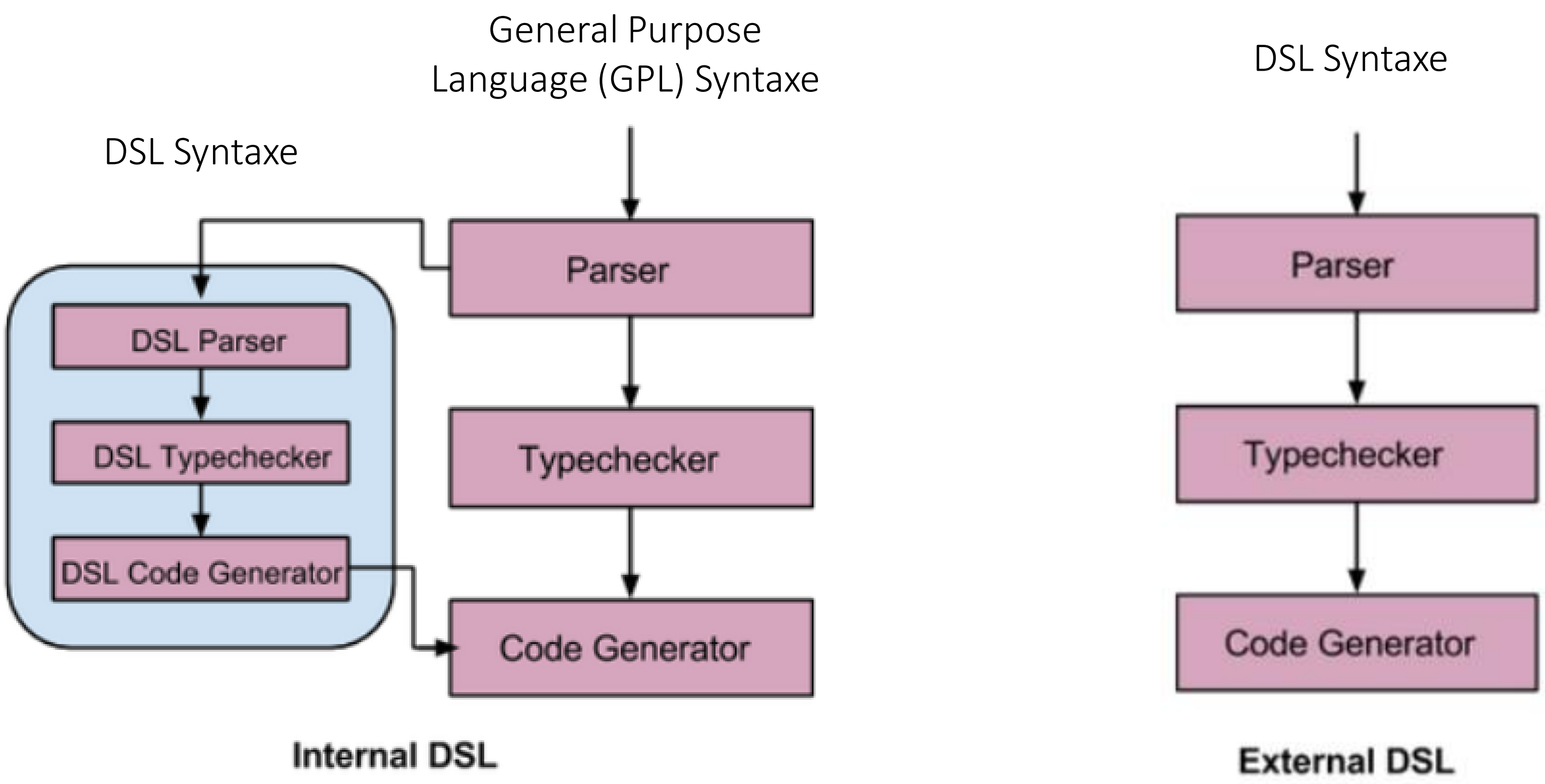
DSL can be reused for other purposes

✗

DSL maintenance is complicated

The cost of a DSL is expensive

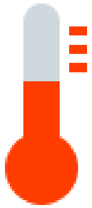
Different types of DSL



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

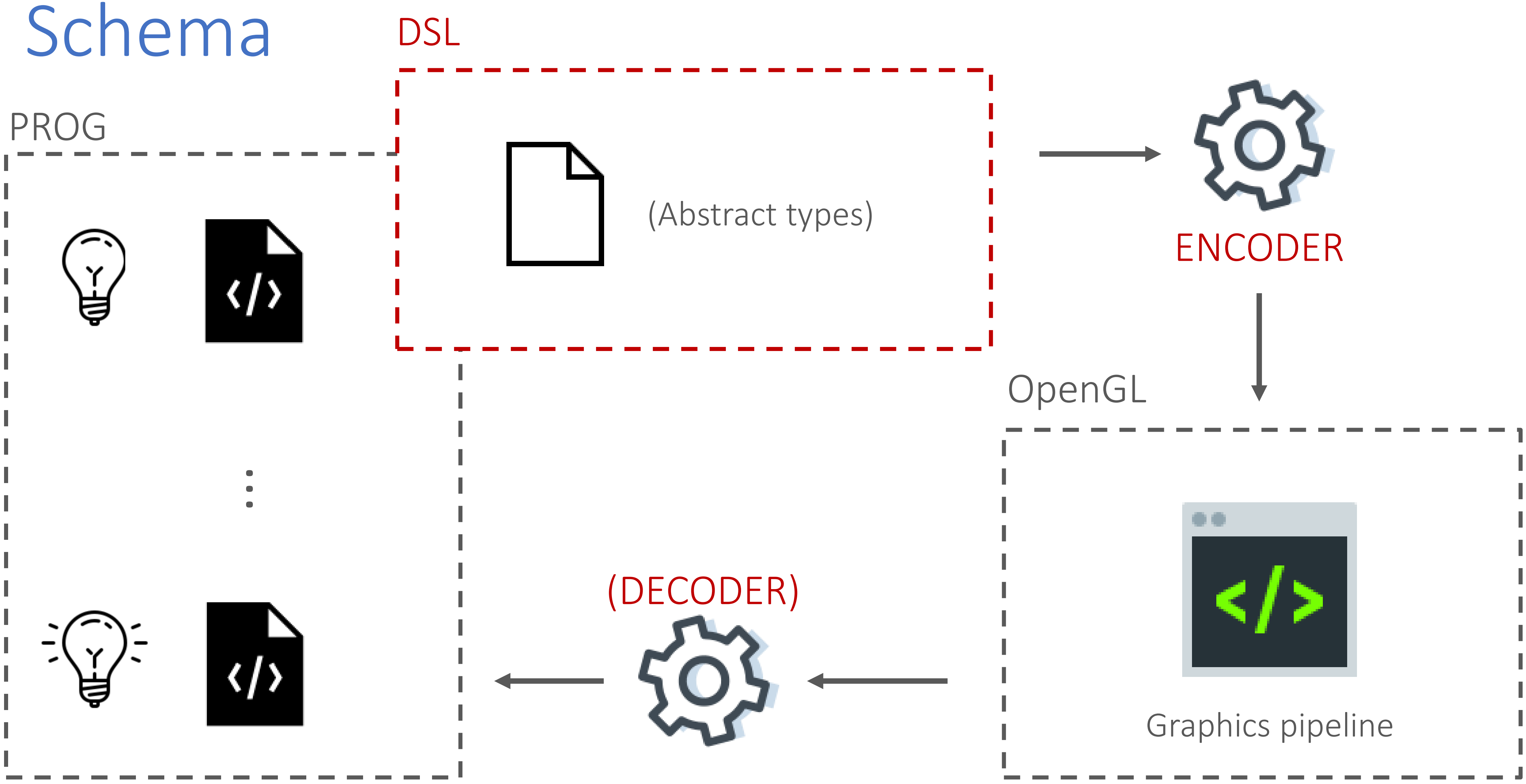
We write in our program a
vertex & a fragment shader with our DSL

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

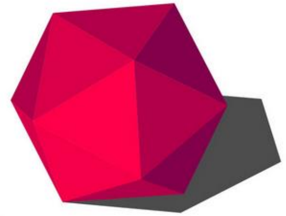
(Then a decoder allows us to get the results
with the desired abstract types)

Schema

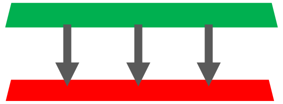

DSL
PROG
(Abstract types)
ENCODER
OpenGL
(DECODER)
Graphics pipeline

Gator

Language created by *Dietrich Geisler*, *Irene Yoon*, *Aditi Kabra*, *Horace He*, *Yinnon Sanders* & *Adrian Sampson*

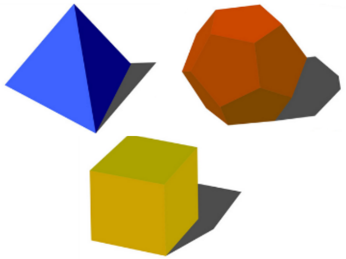


Higher level programming model that allows focus on the geometric semantics of programs

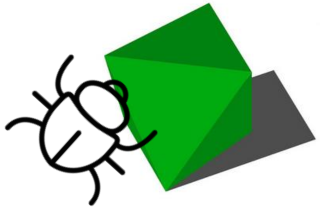


Gator is a surface language with an extended type system based on a target language with a type set (GLSL)
A type-directed translation allows to compile Gator to GLSL

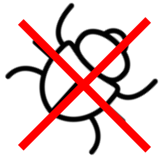
Problem & ideas



3D scenes consist of many individual objects & the rendering code must combine vectors of different coordinate systems

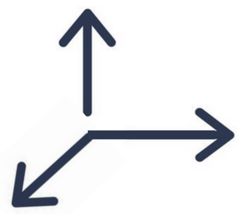


Geometry bugs are difficult to detect



Introduce a type system to eliminate this class of bugs & implement a mechanism that can exclude some bugs by construction

A geometry type



"Geometry types describe the coordinate system representing each value and the transformations that manipulate them"

A geometry type is made up of 3 components:

- Reference frame
- Geometric object
- Coordinate scheme

Define which operations are legal

Syntax

Geometry types give more information about the objects they represent than simple vector types in GLSL

Syntax for a geometry type is *scheme*<*frame*>.*object*

Example:

cart3<world>.point

represents the type of a point in world space represented in a 3D cartesian coordinate scheme

Example: Diffuse Lighting

GLSL implementation

```
float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
  vec3 lightDir = normalize(lightPos - fragPos);
  return max(dot(lightDir, normalize(fragNorm)), 0.);
}
```

lightPos & fragPos have the same type but they are not geometrically compatible
We have different vectors in different coordinate systems

Subtraction between fragPos (model space) & lightPos (world space)

GLSL implementation (Cont.)

```
float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
  vec3 lightDir = normalize(lightPos - uModel * fragPos));
  return max(dot(lightDir, normalize(fragNorm)), 0.);
}
```

To correct the problem we transform the two vectors into a common coordinate system

We define a transformation matrix to go from model to world space

GLSL implementation (Cont.)

```
float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
  vec3 lightDir = normalize(lightPos - vec3(uModel * vec4(fragPos, 1.)));
  return max(dot(lightDir, normalize(fragNorm)), 0.);
}
```

3x3 Cartesian transformation matrices allow only linear transformations

4x4 transformation matrices in Homogeneous coordinates can express affine transformations

Cartesian to Homogeneous: $[x, y, z] \rightarrow [x, y, z, 1.]$
Homogeneous to Cartesian: $[x, y, z, w] \rightarrow [x/w, y/w, z/w]$

GLSL implementation (Cont.)

```
float naiveDiffuse(vec3 lightPos, vec3 fragPos, vec3 fragNorm) {
  vec3 lightDir = normalize(lightPos - vec3(uModel * vec4(fragPos, 1.)));
  return max(dot(lightDir, normalize(vec3(uModel * vec4(fragNorm, 0.)))));
}
```

The final calculation of the diffuse intensity

We must transform now fragNorm into world space
It's a direction so w should be 0

GLSL implementation (Cont.)

(a) Correct implementation. (b) With geometry bug.

Subtle differences can imply errors

```
frame model has dimension 3;
frame world has dimension 3;
```

Gator implementation

```
float diffuseNaive(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm) {
  cart3<world>.direction lightDir = normalize(lightPos - fragPos);
  return max(dot(lightDir, normalize(fragNorm)), 0.0);
}
```

lightPos & fragPos are both positions but their reference frames are different : <world> vs <model>

The subtraction implies an error

Gator implementation (Cont.)

```
with frame(3) r:
coordinate cart3 : geometry {
  object vector is float[3];
  ...
}
```

```
float diffuse(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm,
    hom3<model>.transformation<world> uModel) {
  cart3<world>.direction lightDir =
    normalize(lightPos - (uModel * fragPos));
  return max(dot(lightDir, normalize(uModel * fragNorm)), 0.0);
```

We need to define an affine transformation matrix to transform fragPos & fragNorm into world reference frame

Multiplying uModel & fragPos implies an error because the coordinate schemes are different

```
coordinate hom3 : geometry {
  object point is float[4];
  object direction is float[4];
  with frame(3) r:
  object transformation is float[4][4];
  ...
}
```

Gator implementation (Cont.)

```
float diffuse(
    cart3<world>.point lightPos,
    cart3<model>.point fragPos,
    cart3<model>.direction fragNorm,
    hom3<model>.transformation<world> uModel) {
  cart3<world>.direction lightDir =
     normalize(lightPos - reduce(uModel * homify(fragPos)));
  return max(dot(lightDir, normalize(reduce(uModel * homify(fragNorm)))), 0.0);
}
```

homify() allows us to go from cart3<model>.point to hom3<model>.point (w=1)
or to go from cart3<model>.direction to hom3<model>.direction (w=0)

reduce() allows to map Homogeneous to Cartesian coordinates

Subtyping in Gator

Object & type declarations extend existing types

All types must be given a supertype which can be a primitive type (**bool**, **int**, **float**, **string**, *array*) or a geometry type

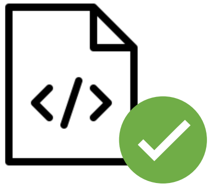
Subtype of float

Example:

```
type angle is float;
type acute is angle;
type obtuse is angle;
```

Subtype of angle

Conclusion



The Gator type system avoids statically incorrect coordinate system transformation codes

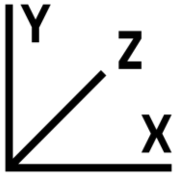
We can thus automatically generate a correct transformation code by construction

→ Programmers do not write vector-matrix multiplication calculations
→ Let the compiler find the right transformations

Gator helps to limit the number of geometry bugs

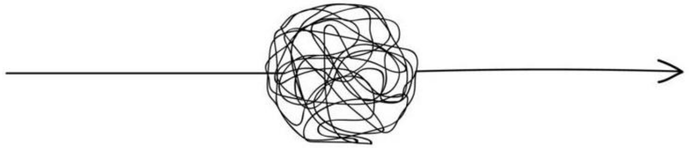
Limitations

The created abstraction remains low level

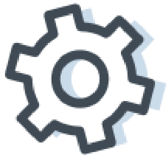


It’s only based on coordinate system transformations

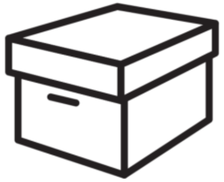
The syntax is a bit complicated



Inspiration



The notion of surface language



New types based on primitives

color, light, texture, normal, (position)

A little less complicated syntax

Work incoming

Learn more about DSL & the creation of abstract types

- Construct abstract types
- Look steps to create an Internal DSL in Swift
- How to link the DSL to the OpenGL pipeline

(To document more about other shader languages)

Begin to work with Rendery

References (Links)

https://learnopengl.com

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

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

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

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

https://developer.apple.com/metal

https://github.com/RenderyEngine/Rendery

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

https://en.wikipedia.org/wiki/Domain-specific_language

References (Research)

Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. 2020. Geometry Types for Graphics Programming. Proc. ACM Program. Lang. 4, OOPSLA, Article 173 (November 2020), 25 pages.

Working with shaders

Patrick SARDINHA