

Working with shaders

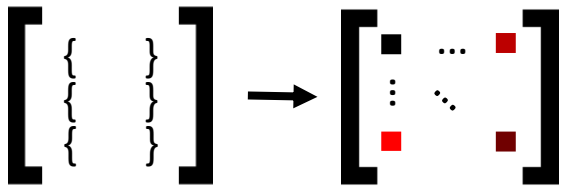
Patrick SARDINHA

What's a shader?

Small programs that run on the GPU

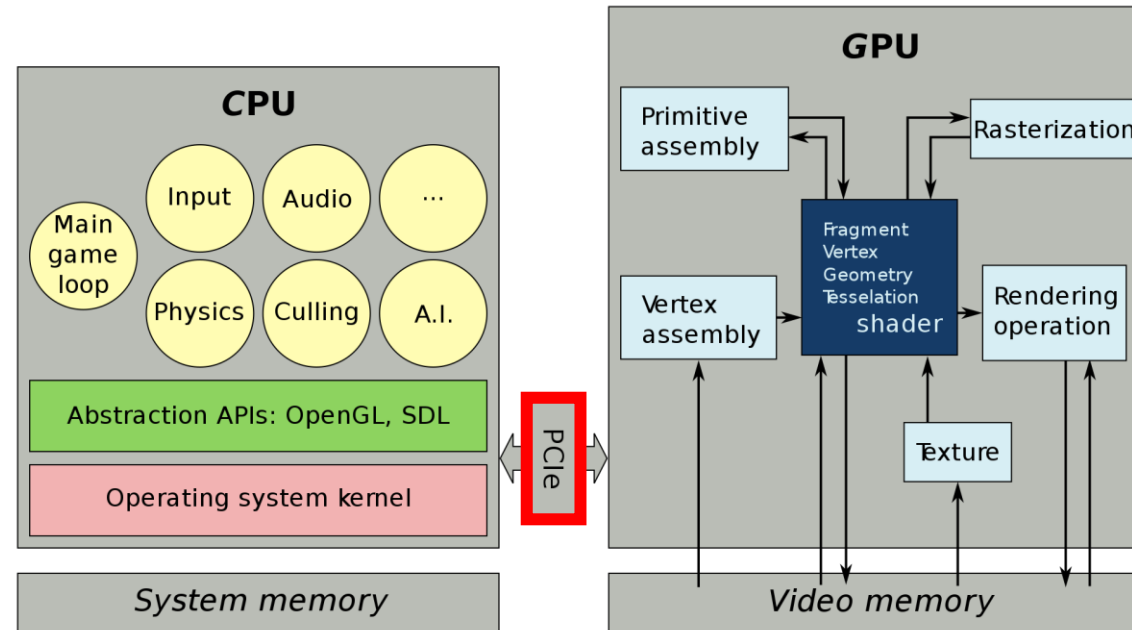
Executed for each specific section of the graphics pipeline

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

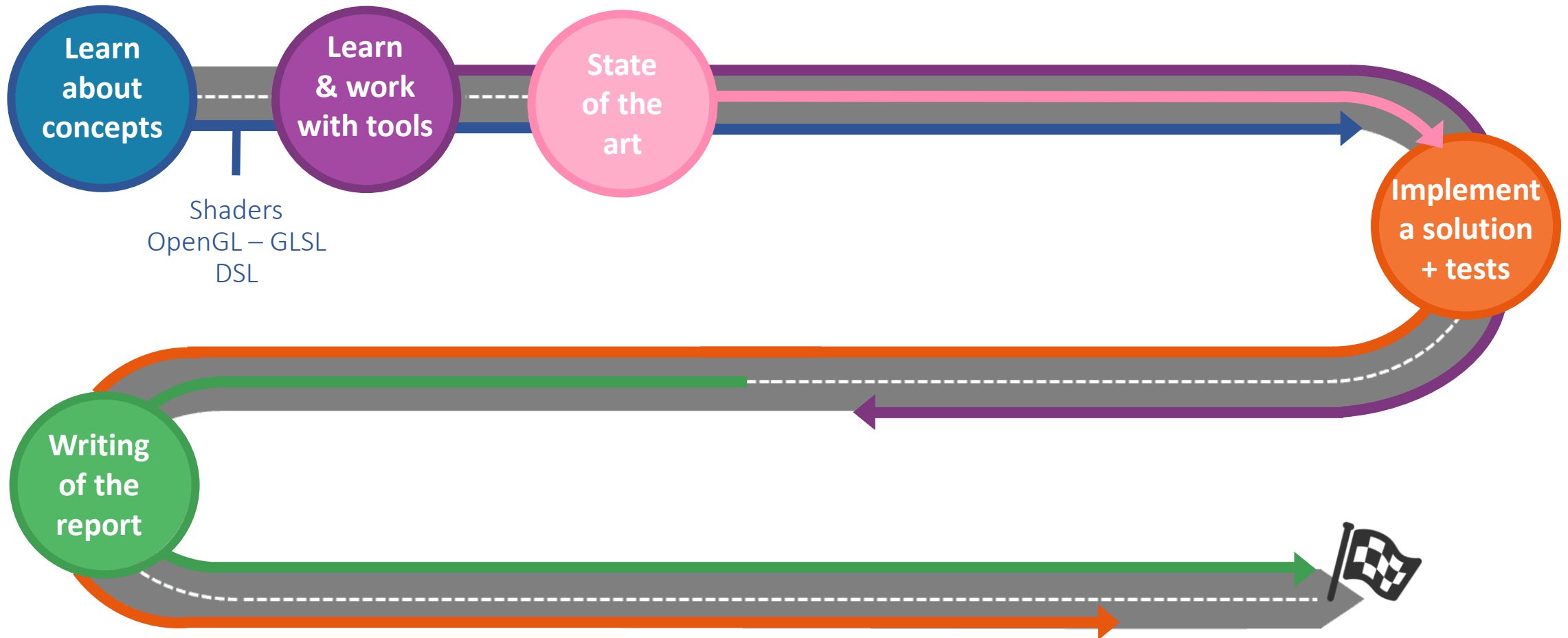


Work with the representation of the data
& abstract the types



Construct a DSL for shaders

Road map



3D space to 2D screen space

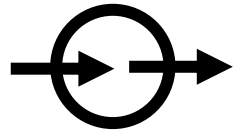


The process of transforming 3D coordinates to 2D pixel is done by the [graphics pipeline](#)

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

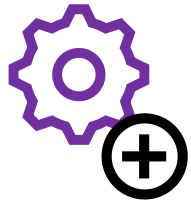


3 different shaders processing units

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**, ...)

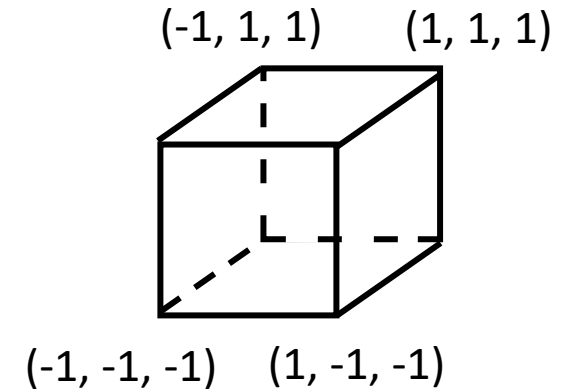


Texture coordinates



Example

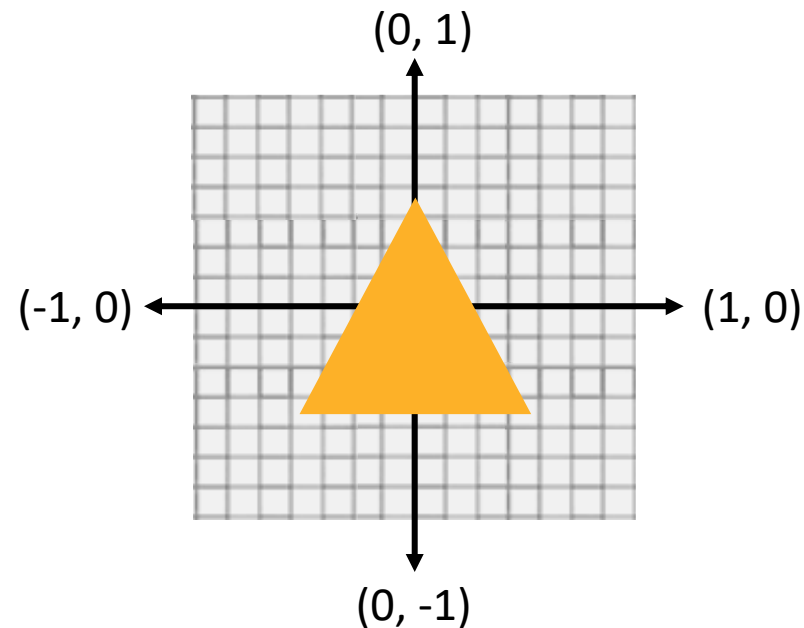
In OpenGL, only the Normalize Device Coordinates (NDC) are visible on the screen



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

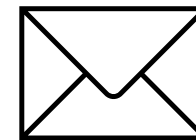


The input data will compose a **Vertex Buffer Object (VBO)** which can store a large number of vertices in the GPU memory

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

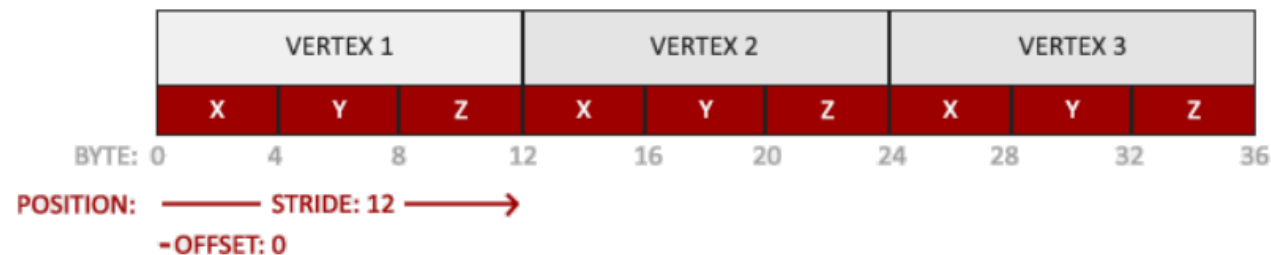
ID of the buffer which must be bind

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

Size of the buffer object

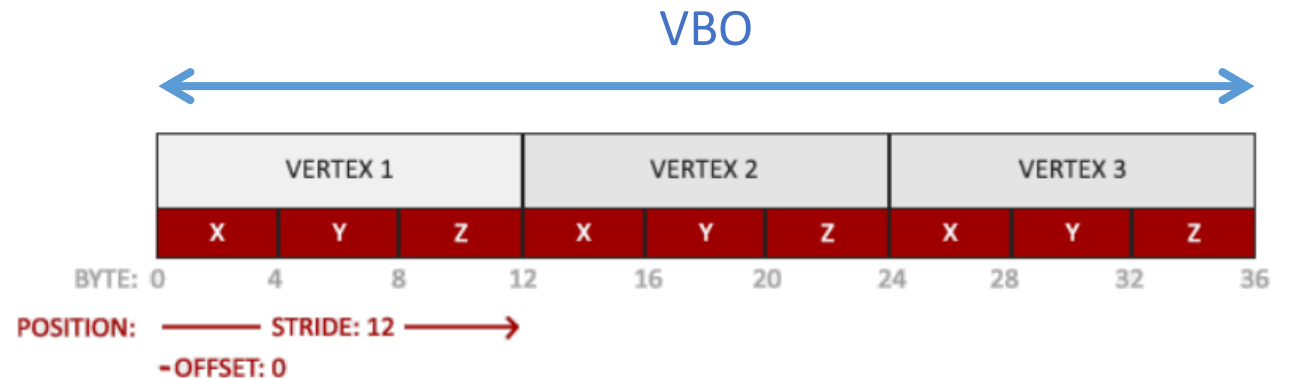
Pointer to data

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

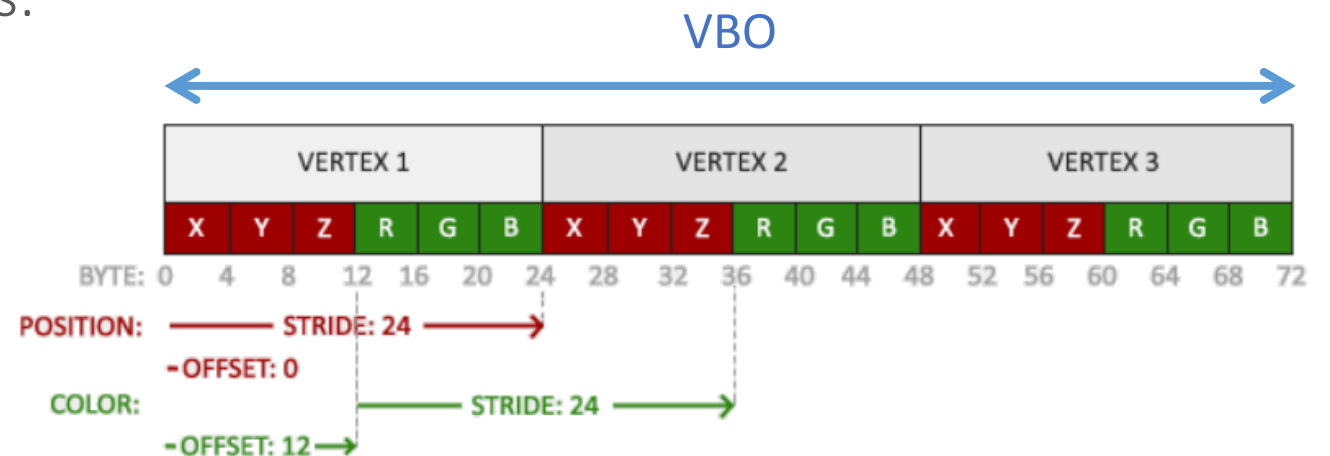
Type of the data

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);
```

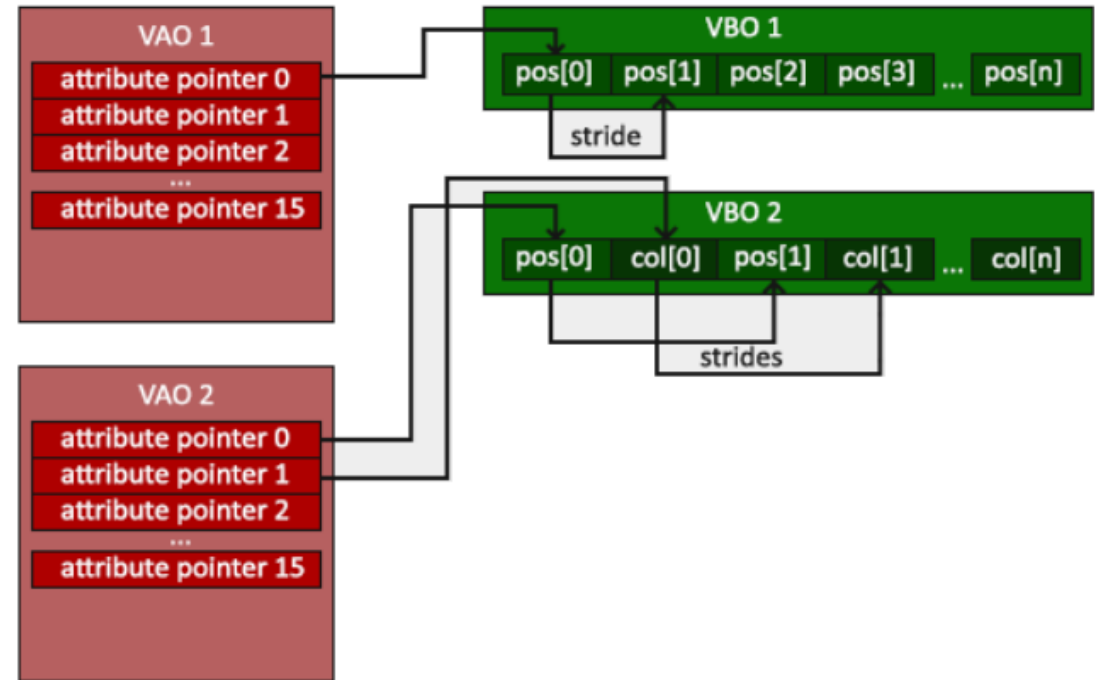
Stride

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:

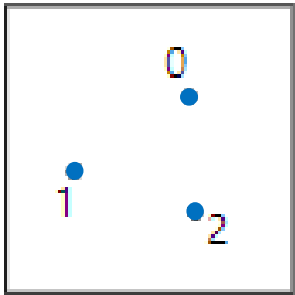


Set up a Vertex & a Fragment Shader

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

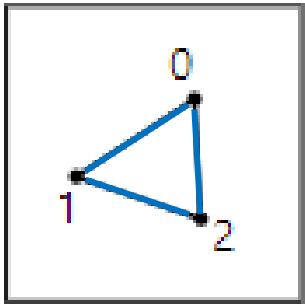
The diagram shows a GLSL vertex shader code snippet with several annotations:

- 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);
```

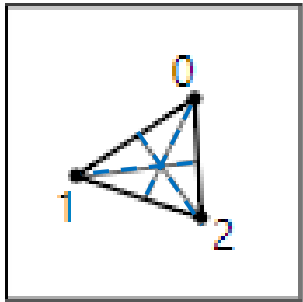
OpenGL function
that draws a shape

Kind of primitive to render

Starting index in the array

Number of vertices to render

Tessellation



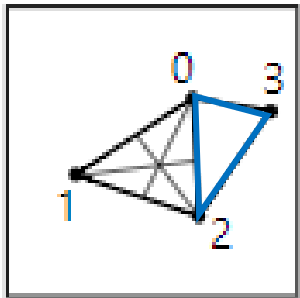
In 3D, the surfaces are built with triangular tiles

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

Geometry Shader



An unnecessary step



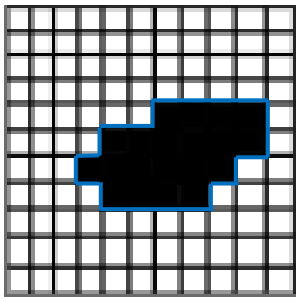
Allows to modify the geometry of each polygon and allows to create new polygons by emitting new vertices

Input data: data of a geometric primitive

Output data: data of one or more geometric primitive

Rasterization

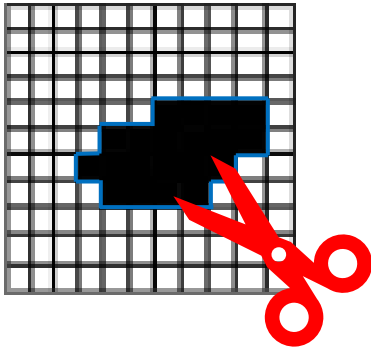
Method of converting a vector image into a raster image to be displayed on a screen



Vector image
composed of geometric objects

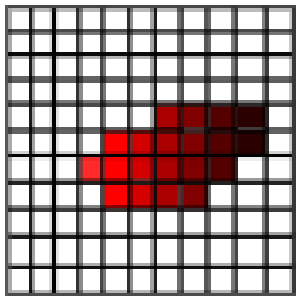
Raster image or Bitmap
composed of pixels

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);
```

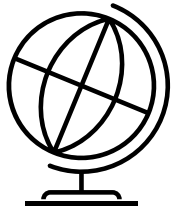
We can now activate this program to render and draw an object

```
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;

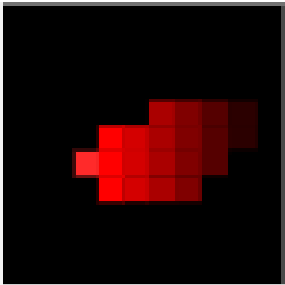
uniform vec4 ourColor;

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 

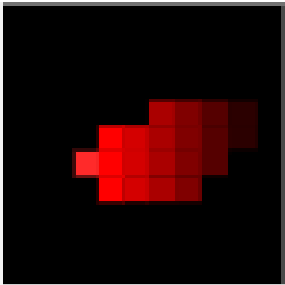


Done with the depth testing using a **Z-buffer** (in which the depth value of the fragments is stored)

```
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:

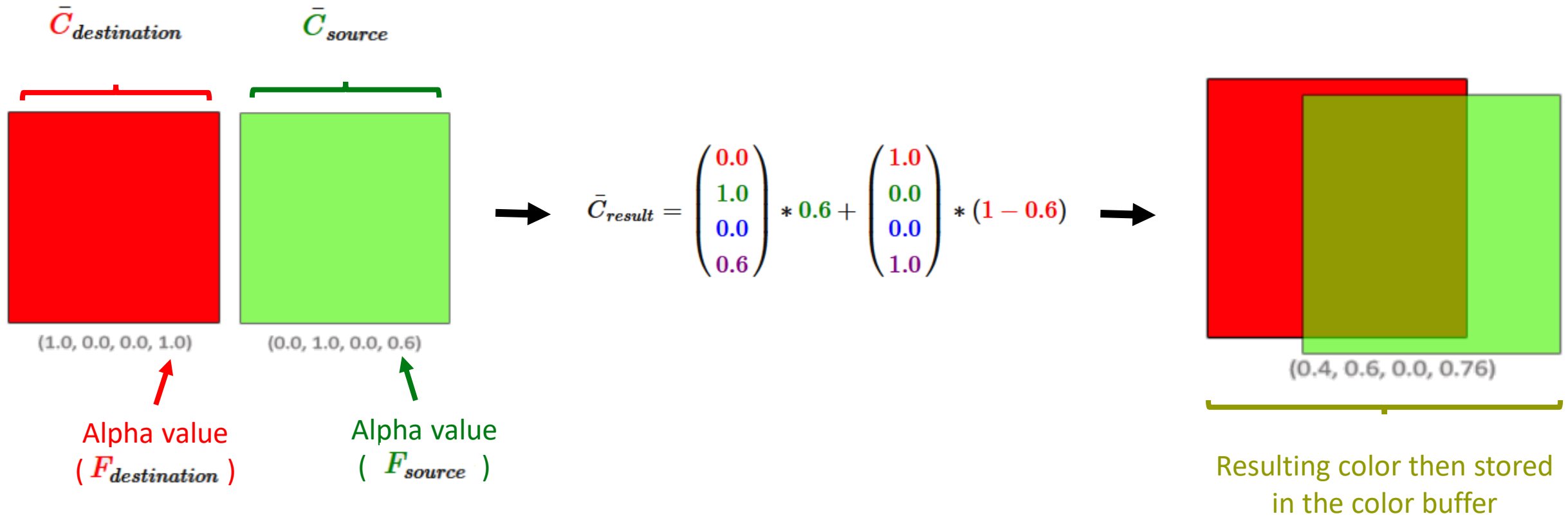
$$\bar{C}_{result} = \bar{C}_{source} * F_{source} + \bar{C}_{destination} * F_{destination}$$

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

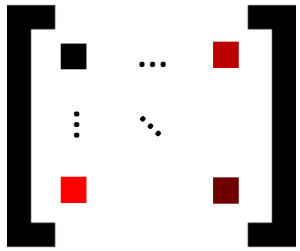
- \bar{C}_{result} : Final color of the fragment (indicated by a blue arrow)
- \bar{C}_{source} : Color output of the fragment shader (indicated by a green arrow)
- F_{source} : Impact of the alpha value (indicated by a green arrow)
- $\bar{C}_{destination}$: Color currently stored in the color buffer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

Example (Cont.)

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



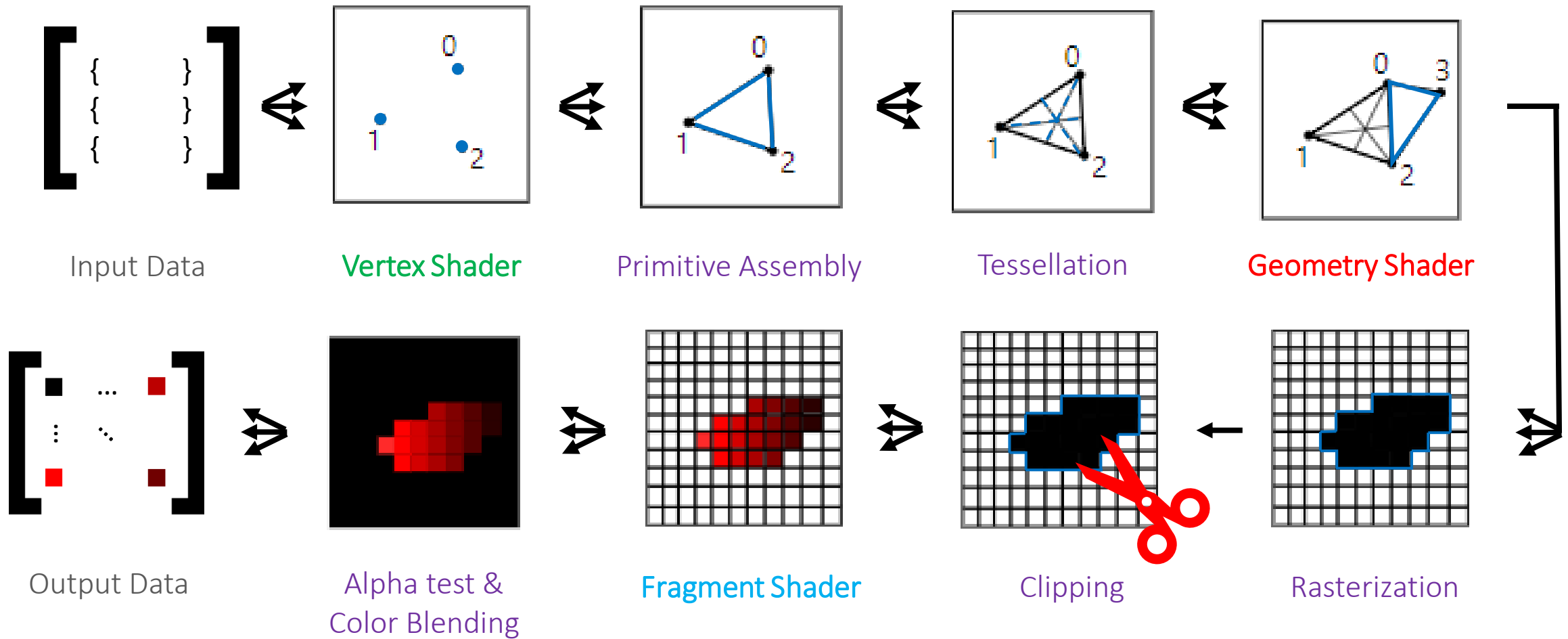
Output Data



Return a **Framebuffer**

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

Overall view



Textures

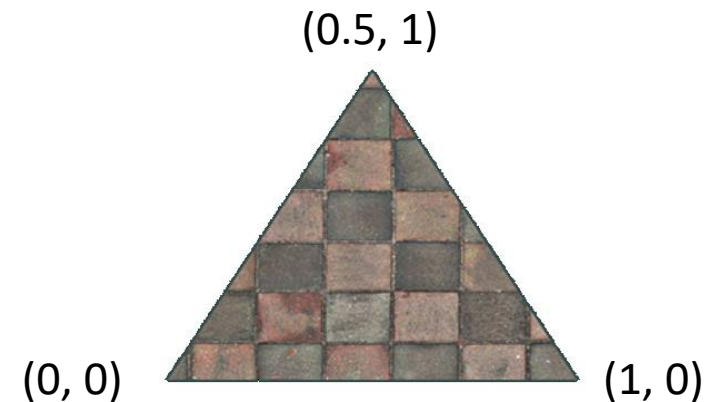
Allows to give the illusion the object is detailed
without having to specify vertices

Associate each vertex to a texture coordinate

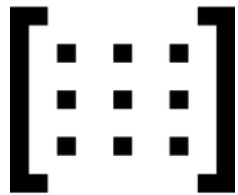
A fragment interpolation is then done for the other fragments

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:

- The matrix to transform (identity Matrix4)**: A green arrow points to the `trans` parameter in the `glm::translate` function call.
- The translation vector to apply to the matrix**: A blue arrow points to the `glm::vec3(1.0f, 1.0f, 0.0f)` argument in the `glm::translate` function call.
- A translate function**: A blue arrow points to the `glm::translate` function name.
- Multiply vec by the translation matrix**: A purple arrow points to the `vec = trans * vec;` line.

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


Coordinates system

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



Local Space



World Space



View Space

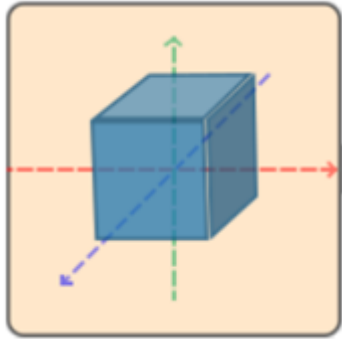
Clip Space



Screen Space



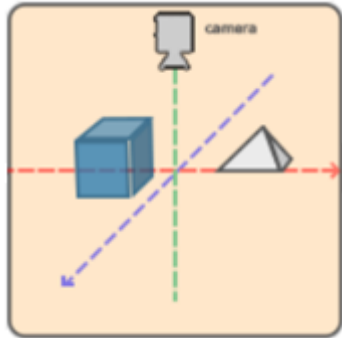
Local Space



Coordinates of the object relative to its local origin

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

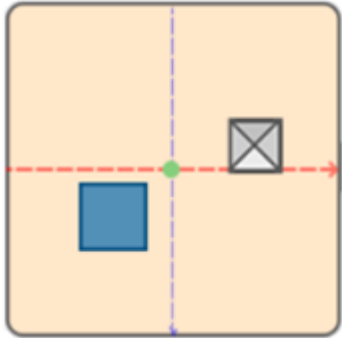
World Space



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

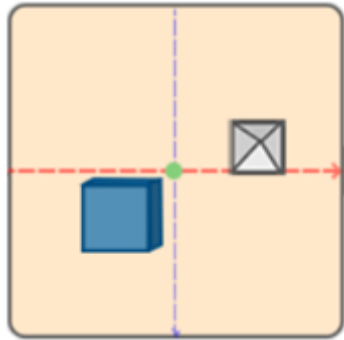
View Space



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

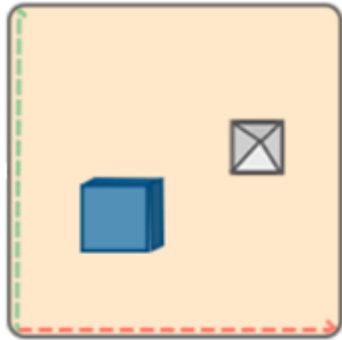
For this step, we use a projection matrix which transform the coordinates into NDC

Example:

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

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

Screen Space

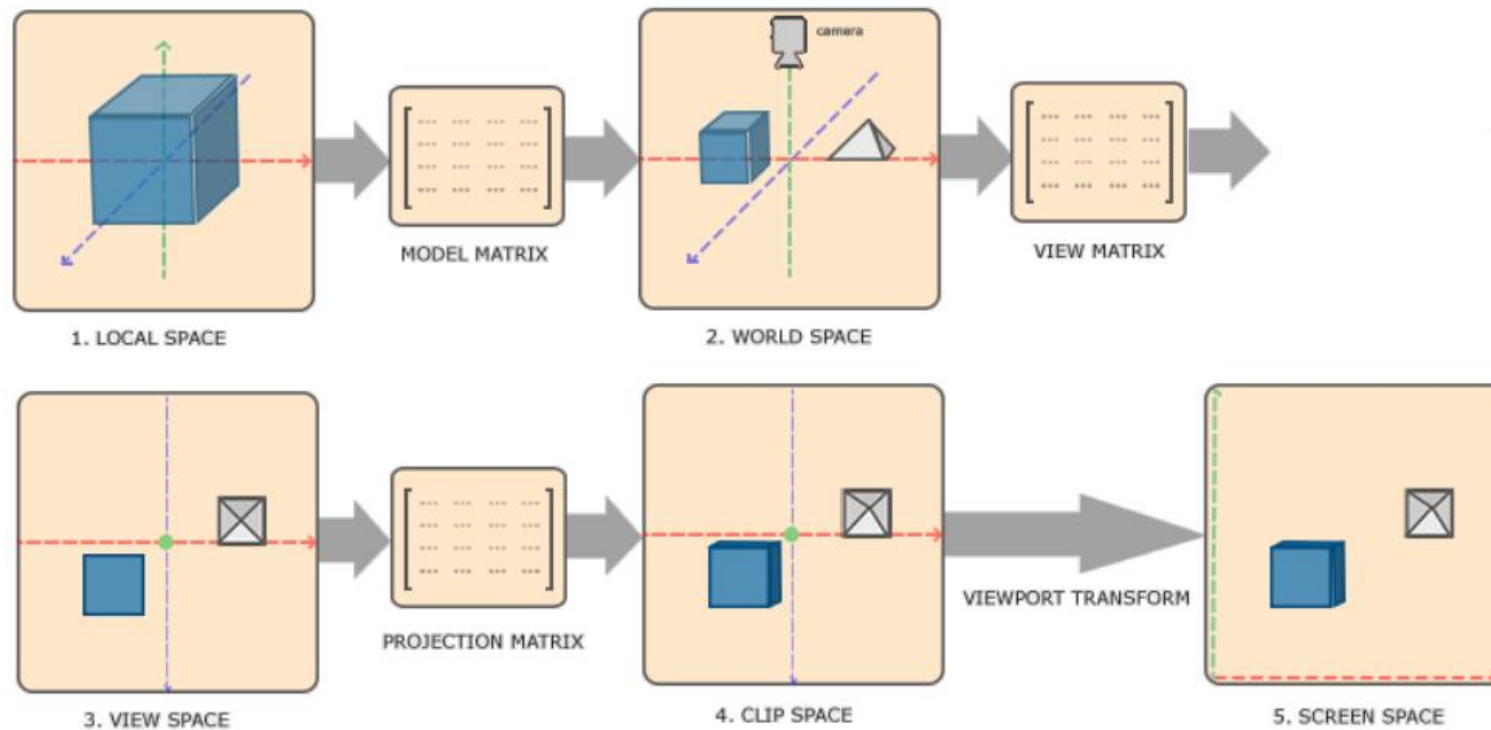


Transforms the NDC coordinates to the window coordinates with the *glViewport()* function

Resulting coordinates are then sent to the rasterizer

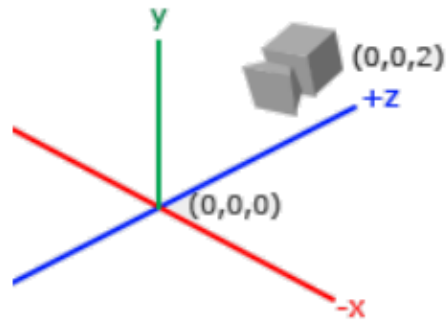
Overall view

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

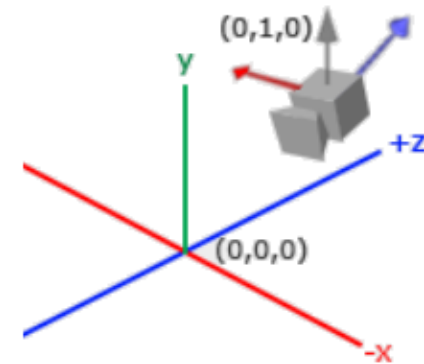


Camera

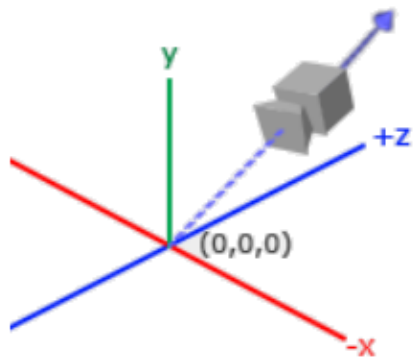
To define a camera we need 4 pieces of information



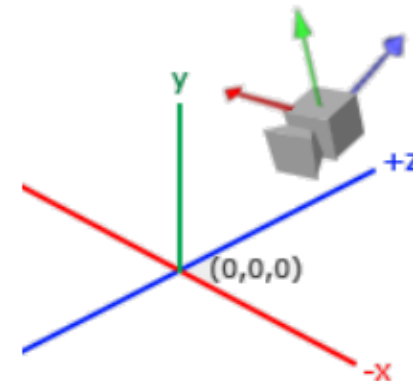
1. Its position in the world space



3. A vector pointing to the right

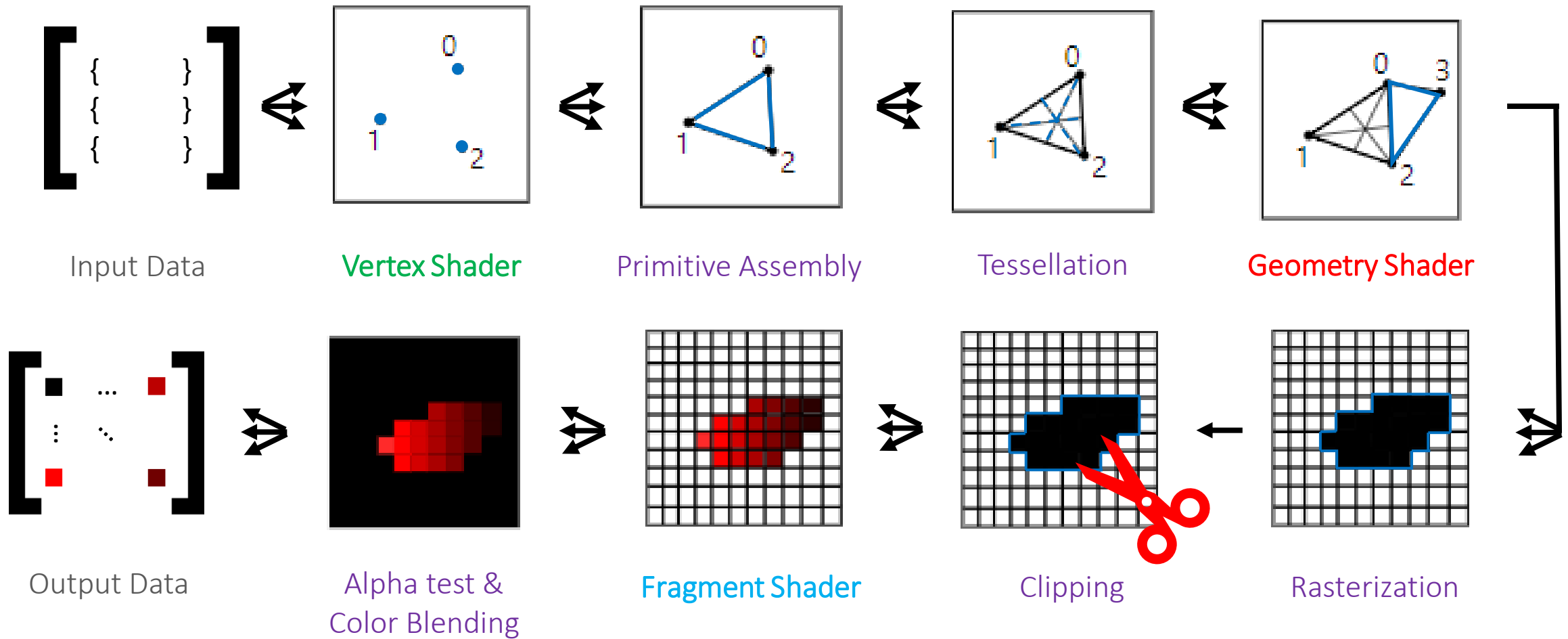


2. Its direction



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

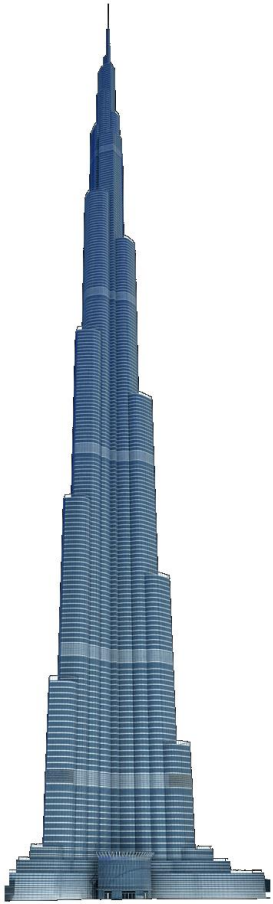
Context

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

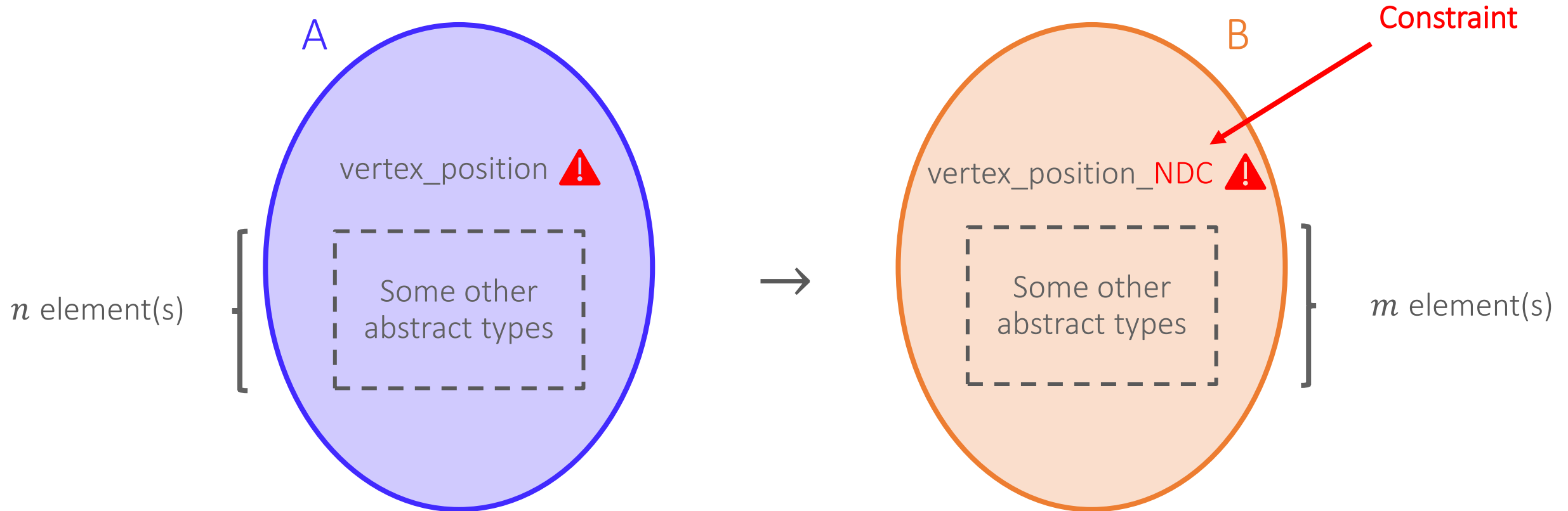
Ex: ~~vec3, vec4, ivec4, mat4, ...~~



But with abstract type objects

Ex: color, position, textures, ...

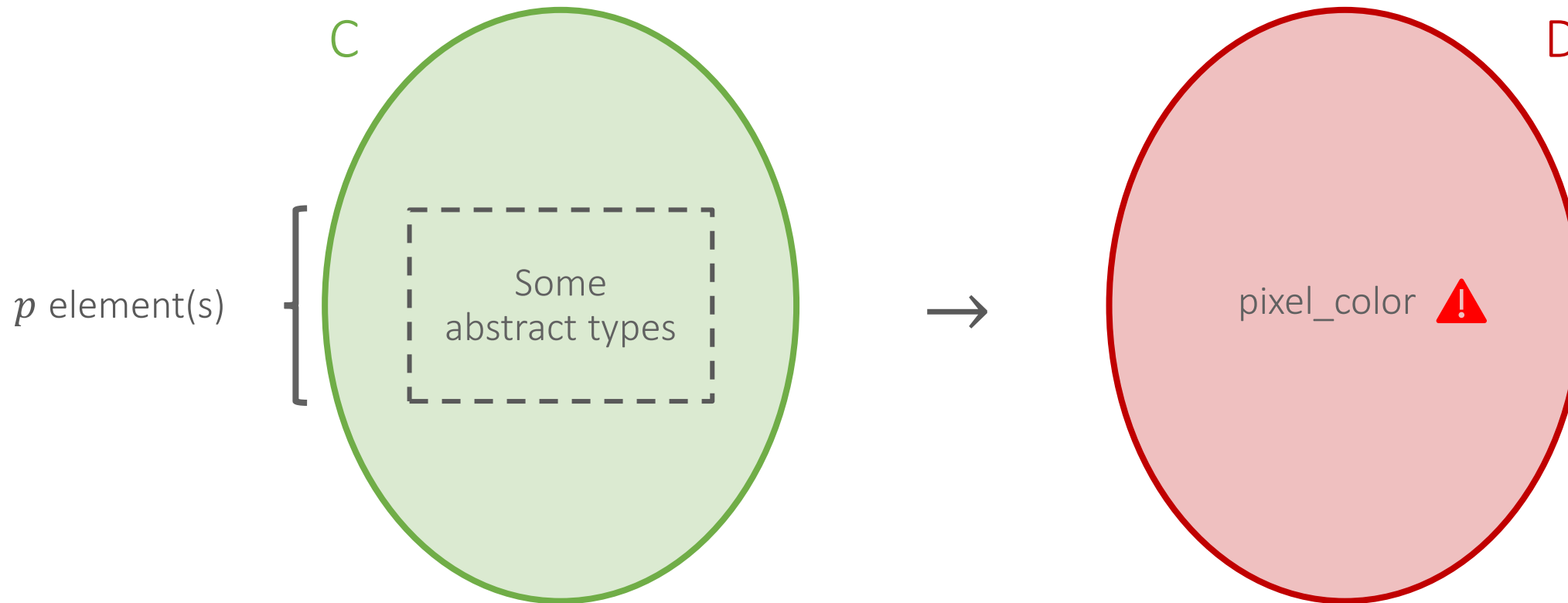
Vertex Shader function $vs : A \rightarrow B$



$vs(_ : vertex_position, \cup _ : (A_i \setminus vertex_position))$

Fragment Shader function

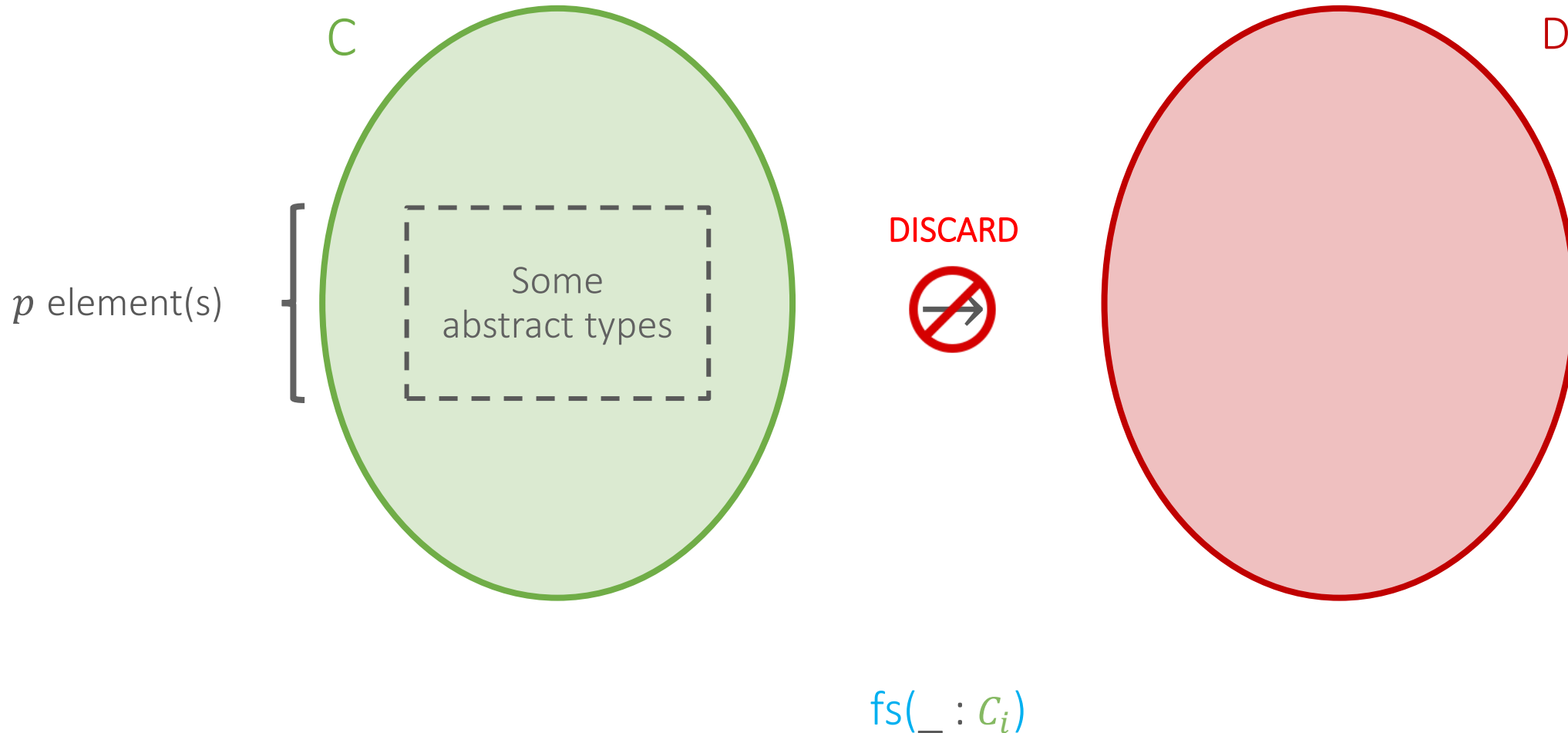
$fs : C \rightarrow D$



$fs(_ : C_i)$

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()`

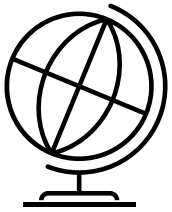
`vs(_ : position, _ : color)`

`fs(_ : fragment, _ : light:)`

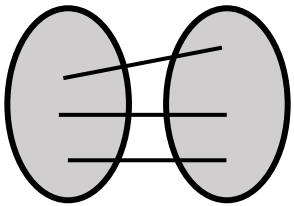
`vs(_ : position, _ : color, _ : texture:)`

`fs(_ : fragment, _ : light, _ : texture:)`

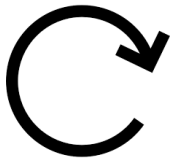
Uniforms



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;
};

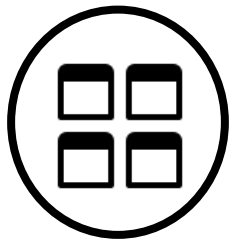
// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    {
        OUT.pos      = mul(modelViewProj, IN.pos);
        OUT.color    = IN.color;
    }
    return OUT;
}
```

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

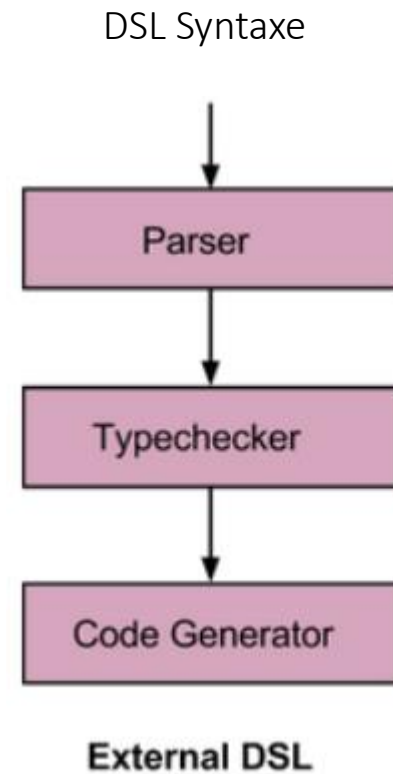
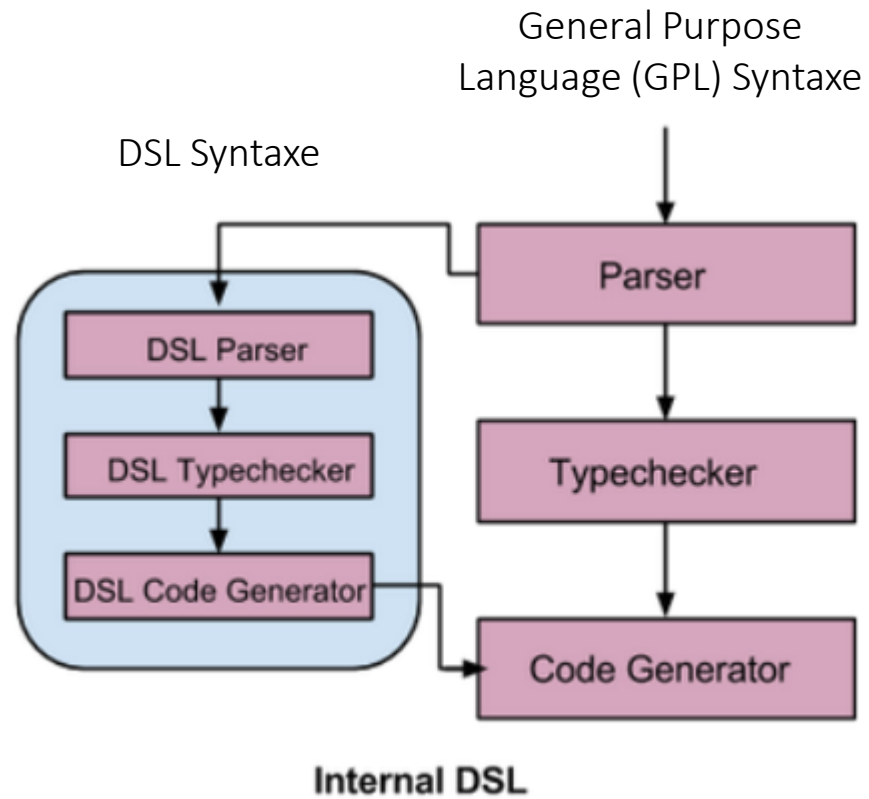
DSL can be reused for other purposes



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



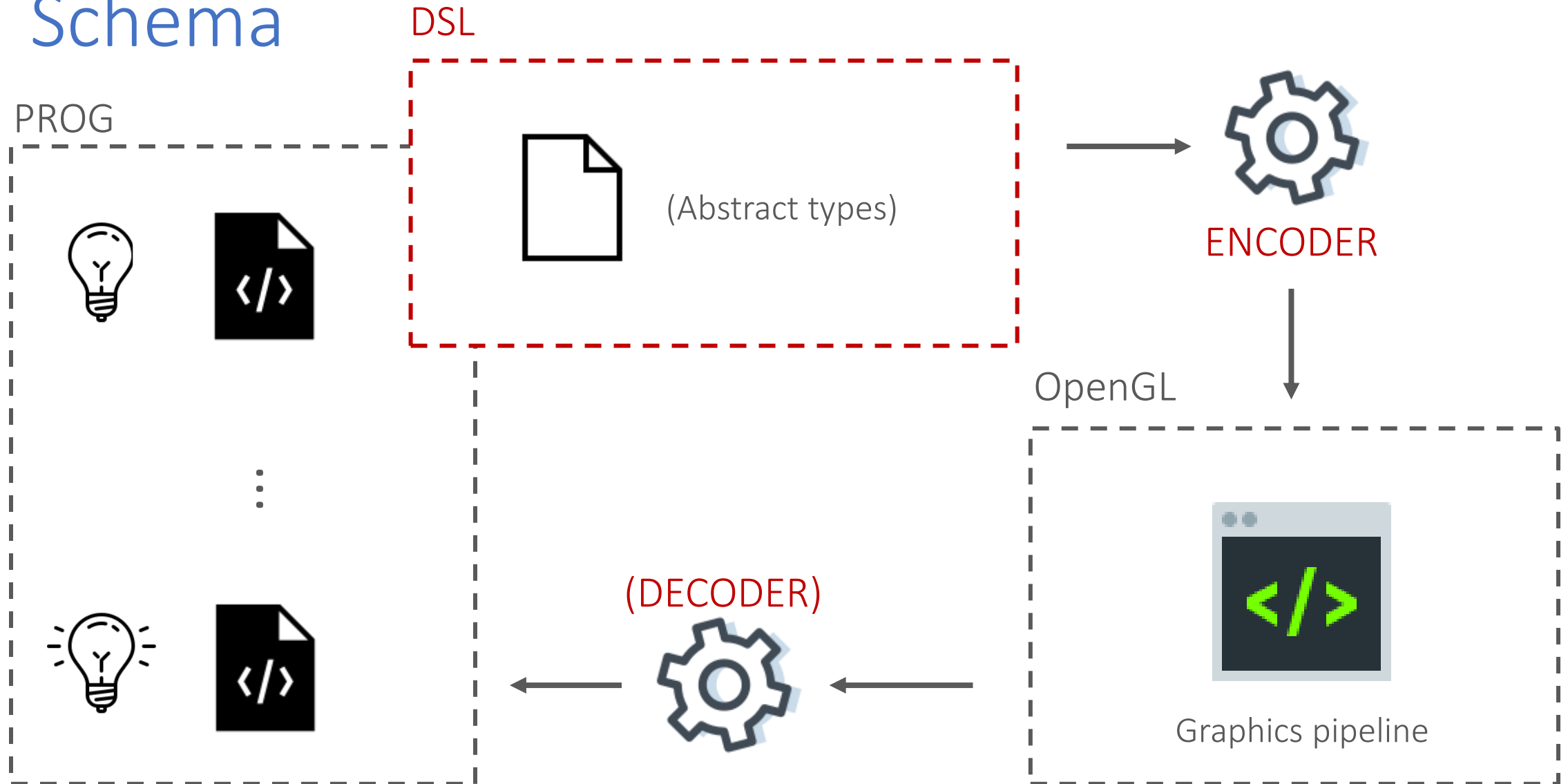
This translation can then be evaluated by the
graphics pipeline of OpenGL



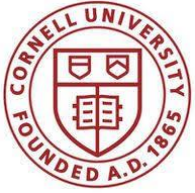
(Then a decoder allows us to get the results
with the desired abstract types)



Schema



Gator



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

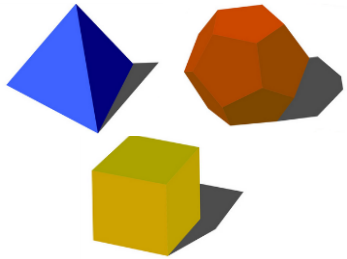


Higher level programming model that allows focus on the geometric semantics of programs

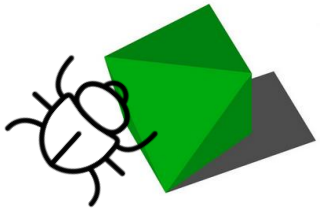


Gator is a **surface language** with an extended type system
based on a **target language** with a type set (GLSL)
A type-directed translation allows to compile Gator to GLSL

Problem & ideas



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

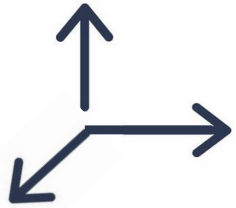


Geometry bugs are difficult to detect



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: $[x, y, z, w] \rightarrow [x/w, y/w, z/w]$

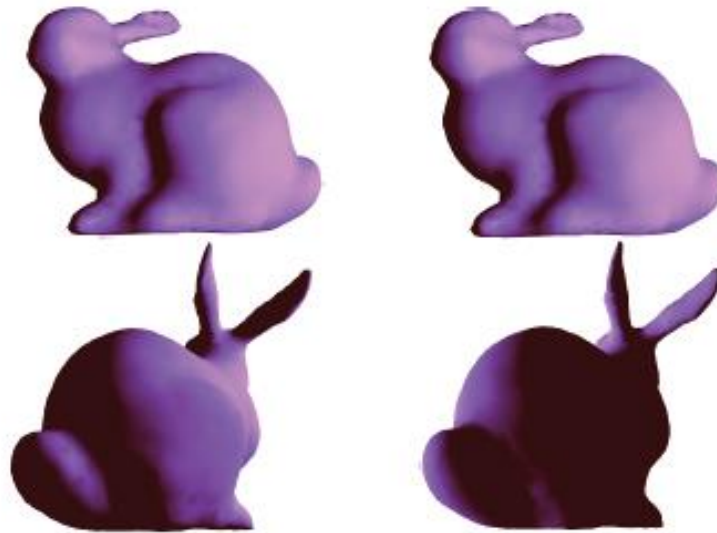
GLSL implementation (Cont.)

```
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

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);  
}
```

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

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

Example:

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

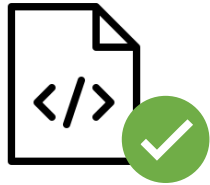
Subtype of float



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

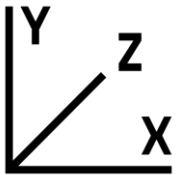


Gator helps to limit the number of geometry bugs

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



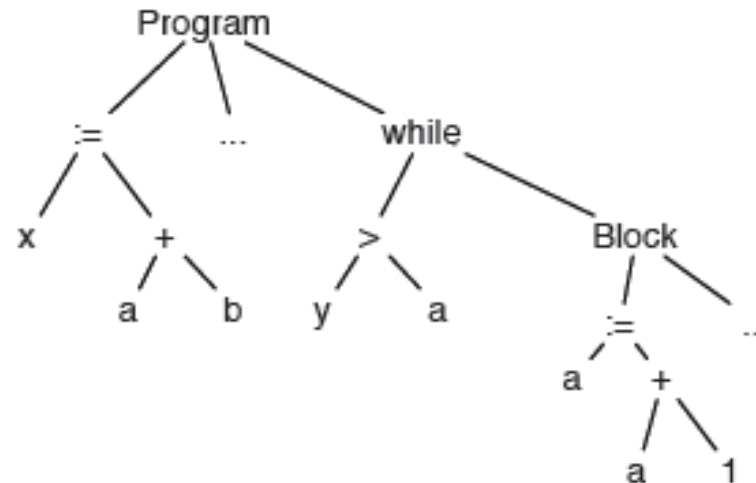
Abstract Syntax Tree (AST)



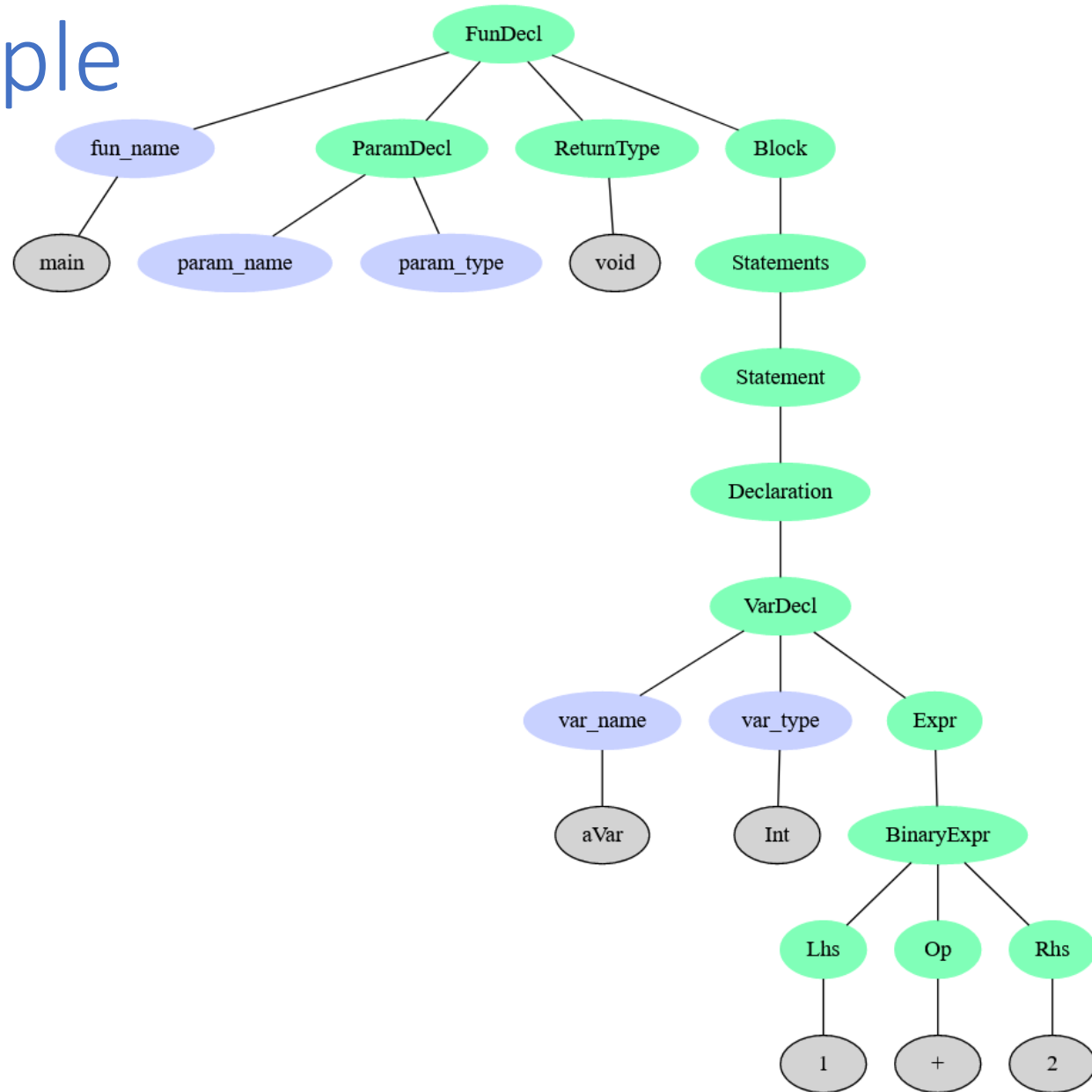
Tree in which, internal nodes are marked by operators and where external ones represent the operands

Example:

```
x := a + b;  
y := a * b;  
while (y > a) {  
    a := a + 1;  
    x := a + b  
}
```



AST example



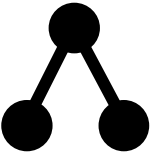
Some tools



Swift-AST-Explorer

Only supports Swift language

Structure representation with pop-up details
for each element of the code



AST-Explorer

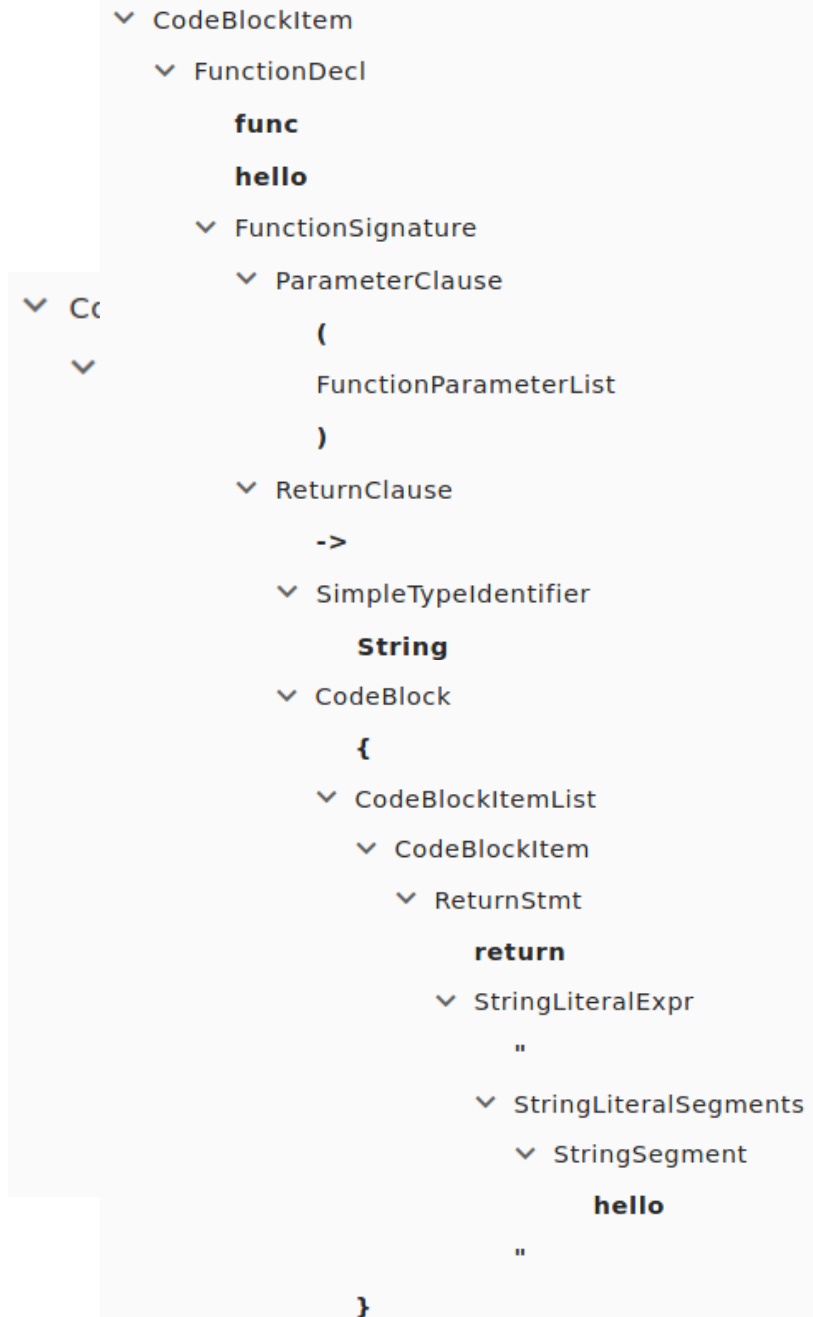
Supports a lot of languages like Java, HTML, Python, ... & GLSL

Tree or JSON representation

Examples

Swift-AST-Explorer

```
1 import Foundation
2
3 let num: Int = 5
4
5 func hello() -> String {
6     return "hello"
7 }
```



Examples (Cont.)

AST-Explorer

```
1 varying vec3 ourColor;  
2  
3 void main() {  
4     FragColor = vec4(ourColor, 1.0f);  
5 }  
6
```

```
- children: [  
  + placeholder {mode, token, children, type, id}  
  + placeholder {mode, token, children, type, id}  
  + placeholder {mode, token, children, type, id}  
  + placeholder {mode, token, children, type, id}  
  + keyword {mode, token, children, type, id}  
  - function {  
    mode: 4  
    + token: operator {type, data, position, line, column}  
    - children: [  
      + ident {mode, token, children, type, id, ... +2}  
      + functionargs {mode, token, children, type, id}  
      - stmtlist {  
        mode: 2  
        + token: ident {type, data, position, line, column, ... +2}  
        - children: [  
          - stmt {  
            mode: 1  
            + token: ident {type, data, position, line, column, ... +2}  
            - children: [  
              - expr {  
                mode: 11  
                + token: ident {type, data, position, line, column, ... +2}  
                + children: [1 element]  
                type: "expr"  
                id: "399d8c18.17fc04"  
                + expecting: [1 element]  
                + tokens: [8 elements]  
                parenlevel: 0  
                bracelevel: 0  
              }  
            ]  
          ]  
        ]  
      ]  
    ]  
  ]  
]
```

GLSL 3.0 Grammar



Examples:

function_identifier:

type_specifier

IDENTIFIER

FIELD_SELECTION

parameter_declarator:

type_specifier IDENTIFIER

type_specifier IDENTIFIER LEFT_BRACKET constant_expression RIGHT_BRACKET

expression:

assignment_expression

expression COMMA assignment_expression

DynaSOAr

Here, we will be based on the idea of DynaSOAr



Which is an object-oriented language for manipulating objects

General idea

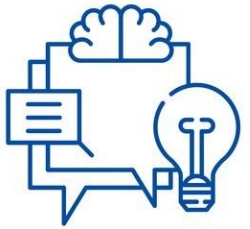


We want to have the perfect language for our use-cases

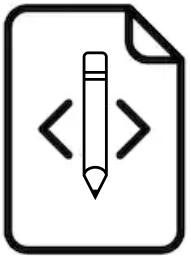


We want to have high-level objects that can be handled in a simple way

Development



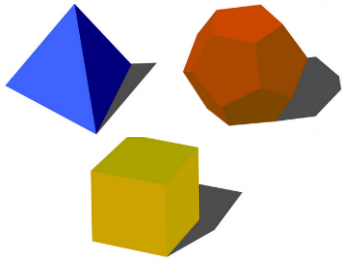
First, we must define our language and the concepts we want



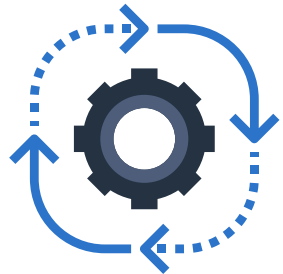
Then, we must specify the syntax of this language

Concepts

Use-cases



Our needs are to write graphics applications
in which we manipulate 3D objects



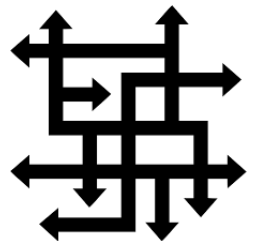
We want to call methods on them, do calculations, etc.

Concepts

Limitations



Here, we want to define everything that it would be possible to do for users ...



... but it's complicated to be complete

Concepts

The abstractions that we need to write an application with 3D objects are:



The position of the objects



The type of the objects



The color of the objects



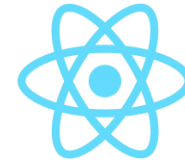
The operations linked to these objects



The physics of the objects

Declarative Language

We are moving towards a **declarative** language inspired by React



React is based on the notions of:



Components, Properties and States



Render function, Update functions

React - Component



The component system allows us to consider each piece of code independently



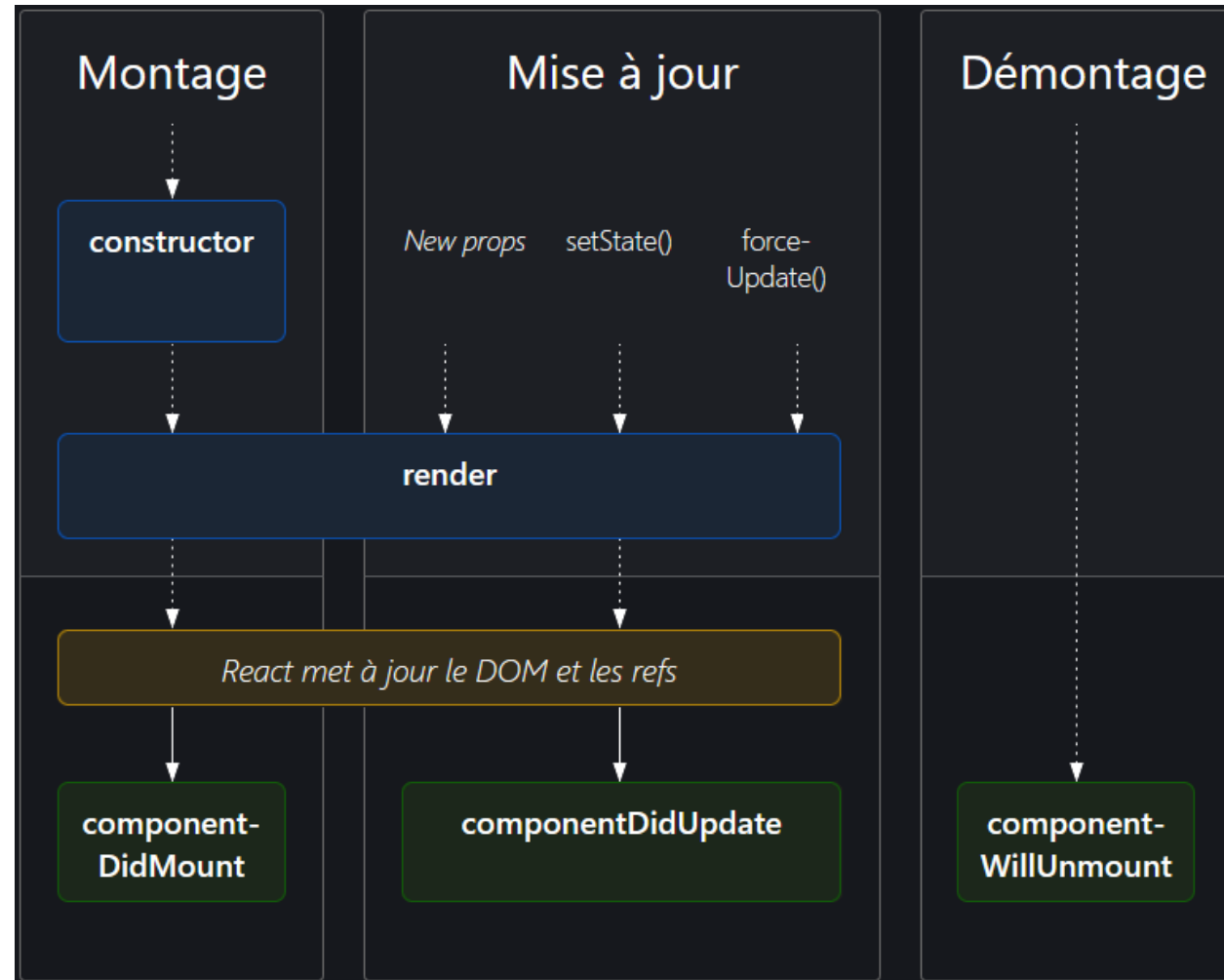
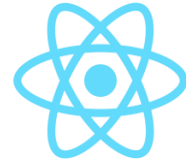
We can then combine different components to get something more complex

Example:

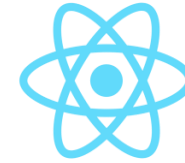
```
function Welcome(props) {  
  return <h1>Bonjour, {props.name}</h1>;  
}
```

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Bonjour, {this.props.name}</h1>;  
  }  
}
```

React - Life cycle



React - Constructor, Props & State



`this.props` contains the properties defined by the caller of the component

A component should never modify its own properties



Local state contains component-specific data, which may change over time

To modify the local state of a component we use `setState()`

React - Render()



The render() method takes data as input and returns what should be displayed.

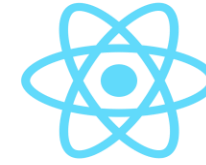


When the local state of a component changes, its display is updated by calling render()



We use a tick() function to update every second / frame

React - componentDidMount() & componentDidUpdate()



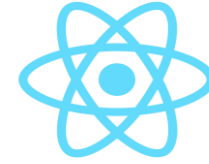
`componentDidMount()` is called immediately after the component is mounted

This is where we make the subscriptions



`componentDidUpdate()` is called immediately after the update has taken place.

React - componentWillUnmount()



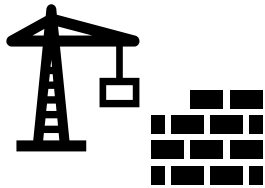
`componentWillUnmount()` is called immediately before a component is unmounted

Here we delete the subscriptions made in `componentDidMount()`

Idea for our language



We give some basic geometric shapes



The idea is to use the notion of **mesh (primitive)**, which is a complete object, composed of several shapes

Example:

A cube is a particular mesh constructed with two triangles per face



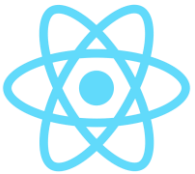
Syntax



We define in the syntax how we would like to define and manipulate the objects



Our goal is to have a simple and meaningful syntax



The syntax will be based on React

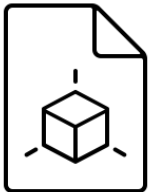
React to our language



We use this notion of components to declare and define a geometric object



We use the same principle of global state, property, and render function



Use of the tick function to update every second / frame

Components (highest level)

Defines the global state of the application

⚠ It is only used in the highest component

Component declaration

```
Component <name_component> {  
  
    var state: [String: AnyObject] = [:]  
  
    init() {  
        self.state = ["key": value,  
                      ...      ]  
    }  
  
    var <var_name> = <value>  
  
    function <function_name>([<arg_name>: <type>]) -> <return_type> {  
        <body>  
        return <return_value>  
    }  
}
```

Variables or constants
declaration

A way to sub and unsub to Tick()

Declares the functions relating to the component

Components (highest level)

Function given in the standard library
to update the global state

Tick() function
declaration

```
function tick() {  
  state = updateState(lastState: state,  
                      newState: ["key": value,  
                                ...    ]  
)  
}
```

Component rendering
function

```
function render() {  
  return (  
    <container>  
  
      <<component_name> ([<arg_name>=value])/>  
  
    </container>  
  
  )  
}
```

Components (lowest level)

Component declaration

Get the properties passed to the component

```
Component <name_component> {  
  
    init() {  
        super.init()  
    }  
  
    var <var_name> = <value>  
  
    function <function_name>([<arg_name>: <type>]) -> <return_type> {  
        <body>  
        return <return_value>  
    }  
}
```

Components (lowest level)

Component rendering
function

```
function render() {  
  return (  
    <mesh type= <value>  
      position= <value>  
      properties= [<value>] />  
  )  
}
```

Call a subcomponent or
render a mesh to graphically represent an object



Render a mesh is never in the
highest component

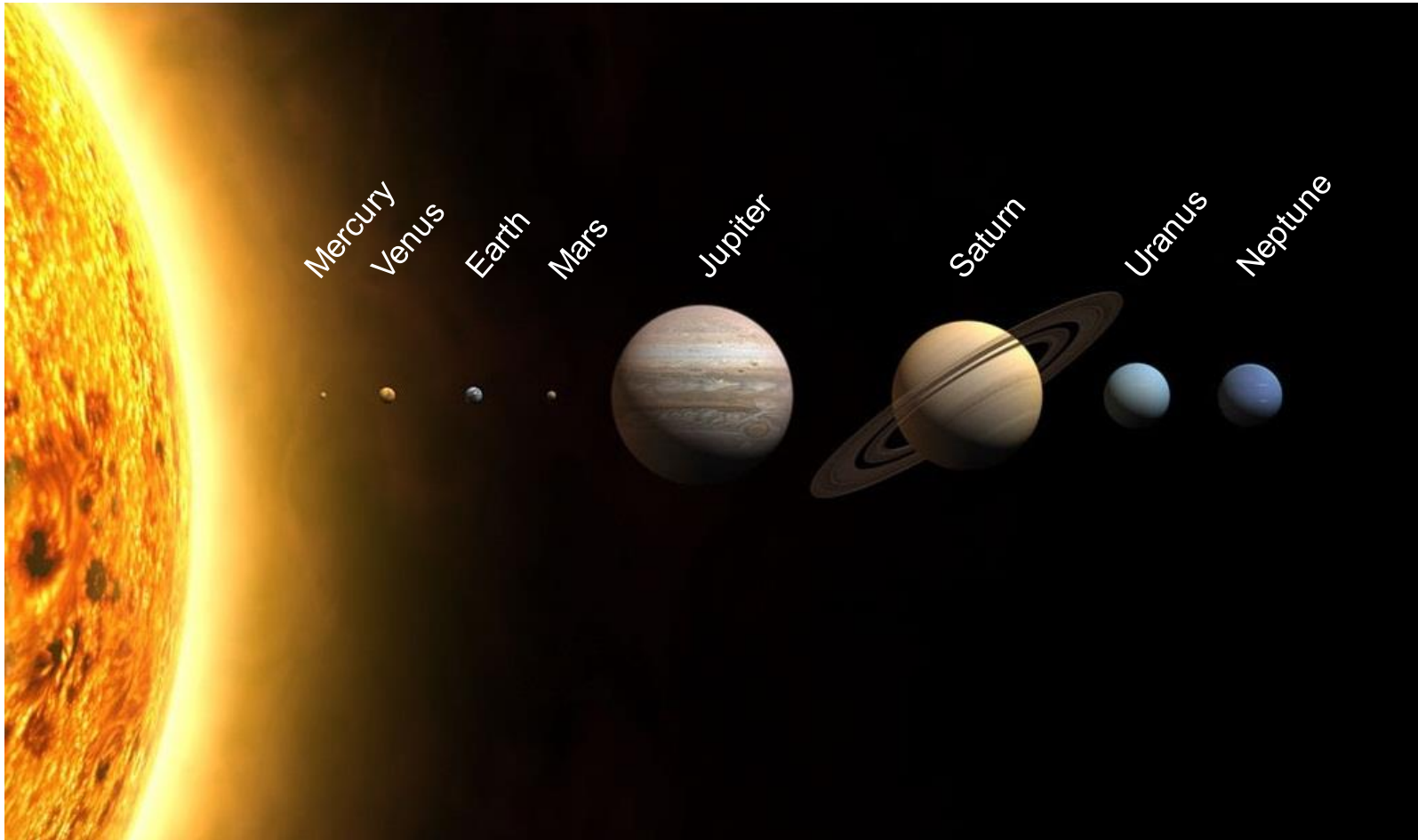
Entry point

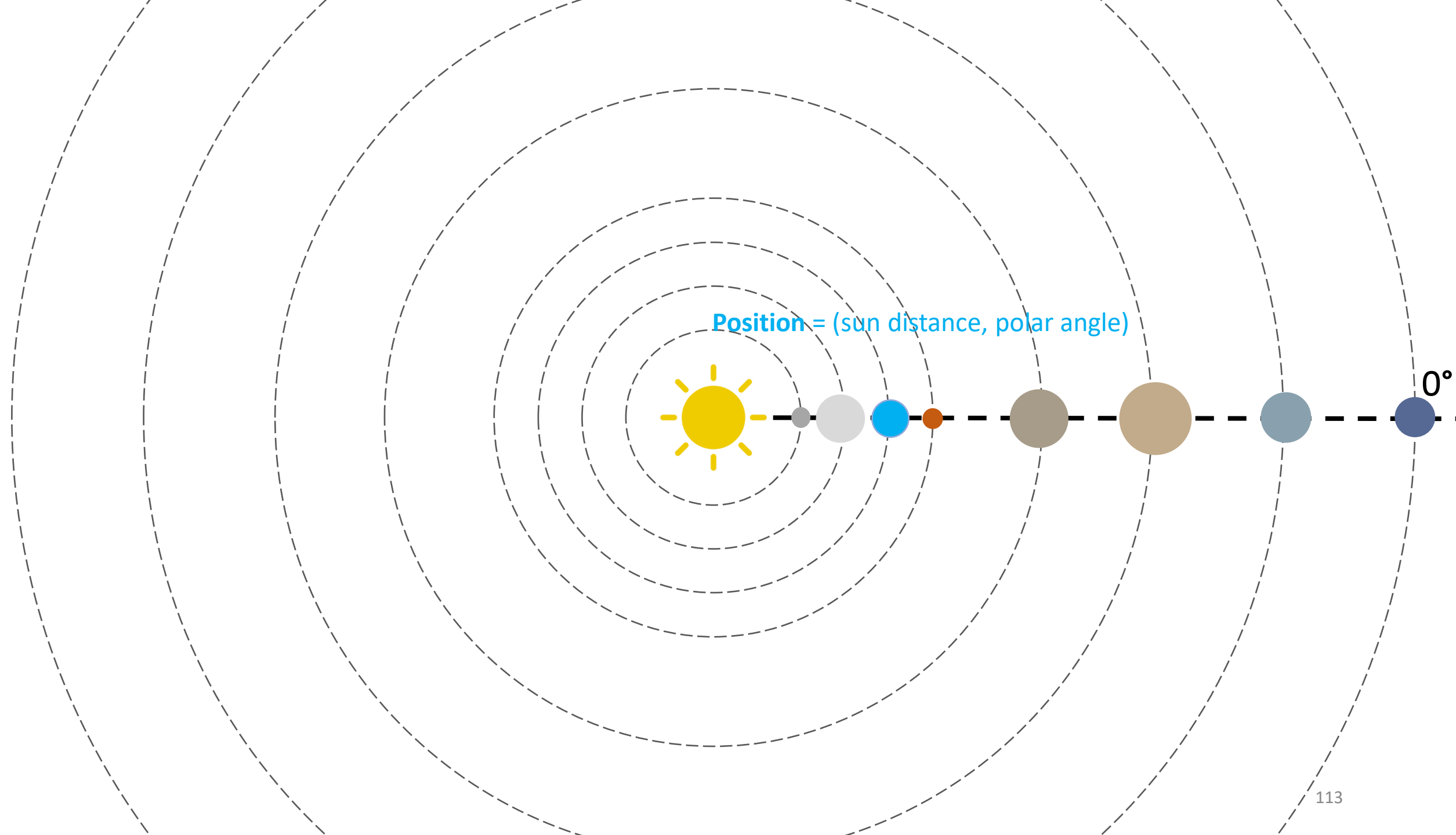
Users can define their own types in addition to what is given in the standard library

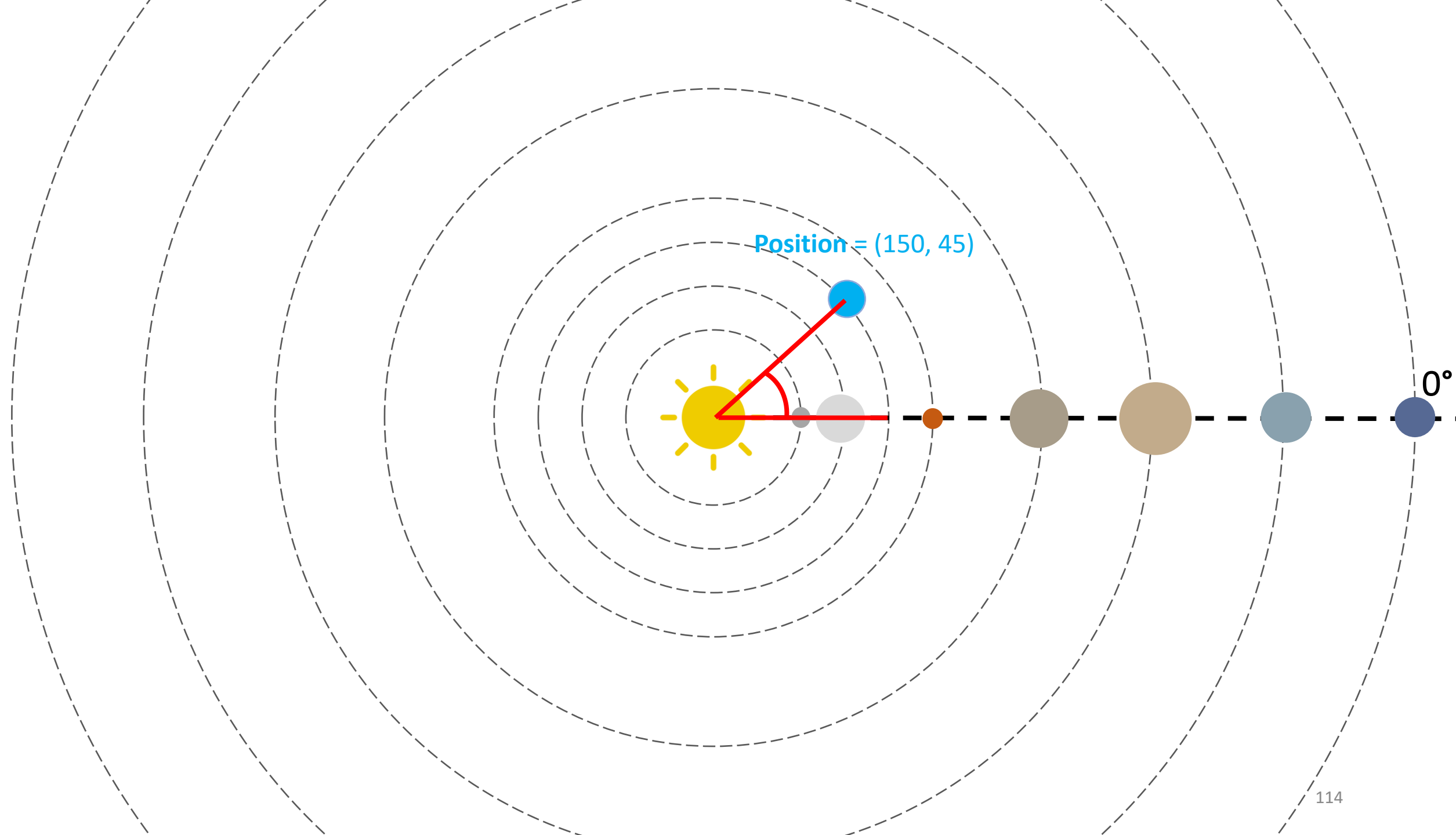
```
structure <name_structure> {  
    var <var_name>: <type>  
    ...  
    init(<args> ) {  
    }  
  
    function <function_name>([<arg_name>: <type>]) -> <return_type> {  
        <body>  
        return <return_value>  
    }  
}  
  
<<highest_level_component> ()/>
```

Then, we specify the top-level component

Example - Solar system







```

Component SystemSolar {

  var state: [String: AnyObject] = [:]

  init() {
    self.state = ["solarCoord": Coord(polar: (0, Angle(deg: 0)), "solarRevo": 0, "solarRadius": 696340,
      "mercuryCoord": Coord(polar: (58, Angle(deg: 0)), "mercuryRevo": 88, "mercuryRadius": 2440,
      "venusCoord": Coord(polar: (108, Angle(deg: 0)), "venusRevo": 225, "venusRadius": 6050,
      "earthCoord": Coord(polar: (150, Angle(deg: 0)), "earthRevo": 365, "earthRadius": 6370,
      "marsCoord": Coord(polar: (228, Angle(deg: 0)), "marsRevo": 687, "marsRadius": 3390,
      "jupiterCoord": Coord(polar: (779, Angle(deg: 0)), "jupiterRevo": 4380, "jupiterRadius": 69910,
      "saturnCoord": Coord(polar: (1434, Angle(deg: 0)), "saturnRevo": 10585, "saturnRadius": 58230,
      "uranusCoord": Coord(polar: (2871, Angle(deg: 0)), "uranusRevo": 30660, "uranusRadius": 25360,
      "neptuneCoord": Coord(polar: (4495, Angle(deg: 0)), "neptuneRevo": 60225, "neptuneRadius": 24600
    ]
  }
}

```

solarSystem.patI

The application has a **global state** which contains all the information
 It is defined in the highest-level component

```

structure Angle {
    var deg: Double
    var rad: Double

    init(deg: Double = nil, rad: Double = nil) {
        self.deg = deg
        self.rad = rad
    }

    function degToRad() -> Double {
        return (deg * (pi/180))
    }

    function radToDeg() -> Double {
        return (rad * (180/pi))
    }
}

```

bibli.patl

```

structure Coord {
    var polar: (Double, Angle) // Angle(deg: _)
    var cart: (Double, Double)

    init(polar: (Double, Angle) = nil, cart: (Double, Double) = nil) {
        self.polar = polar
        self.cart = cart
    }

    function polarToCart() -> (Double, Double) {
        cons r = polar[0]
        cons t = polar[1]
        return (r*cos(t), r*sin(t))
    }

    function cartToPolar() -> (Double, Angle) {
        cons x = cart[0]
        cons y = cart[1]
        return (sqrt(x^2 + y^2), atan(y/x))
    }
}

```

bibli.patl

```
// Directly call when the component is created  
let st = subTick(interval: 1)
```

solarSystem.patl

```
// Call when the component is removed  
let _ = unsubTick(sub: st)
```

solarSystem.patl

```
function subTick(interval: Double) {  
    while(1) {  
        sleep(interval)  
        tick()  
    }  
}
```

bibli.patl

```
function unsubTick(sub: _ ) {  
    destroy(sub)  
}
```

bibli.patl

```

function updatePlanetPos(currentAngle: Angle, revolution: Int) -> Angle {
    var newAngle = currentAngle+(360/revolution)
    return (newAngle % 360)
}

function tick() {
    state = updateState(lastState: state,
        newState: ["mercuryCoord": Coord(polar: (58,
            Angle(deg: this.updatePlanetPos(currentAngle: state["mercuryCoord"].polar[1],
                revolution: state["mercuryRevo"])))),
            "venusCoord": Coord(polar: (108,
                Angle(deg: this.updatePlanetPos(currentAngle: state["venusCoord"].polar[1],
                    revolution: state["venusRevo"])))),
            "earthCoord": Coord(polar: (150,
                Angle(deg: this.updatePlanetPos(currentAngle: state["earthCoord"].polar[1],
                    revolution: state["earthRevo"])))),
            ...
        ]
    )
}

```

solarSystem.patI

```
function updateState(lastState: [String: AnyObject], newState: [String: AnyObject]) -> [String: AnyObject] {  
    for key in newState {  
        lastState[key] = newState[key]  
    }  
    return lastState  
}
```

bibli.patI

```
function render() {  
  return (  
    <container>  
      <Sphere (name="Solar", pos={state["solarCoord"]}, radius={state["solarRadius"]})/>  
      <Sphere (name="Mercury", pos={state["mercuryCoord"]}, radius={state["mercuryRadius"]})/>  
      <Sphere (name="Venus", pos={state["venusCoord"]}, radius={state["venusRadius"]})/>  
      <Sphere (name="Earth", pos={state["earthCoord"]}, radius={state["earthRadius"]})/>  
  
      ...  
  
    </container>  
  );  
}
```

solarSystem.patl


```
Component Sphere {  
  init() {  
    super.init()  
  }  
  
  let meshPosition = pos.polarToCart()  
  
  function render() {  
    return (  
      <mesh type="sphere"  
        position={meshPosition}  
        properties=[{radius}]/>  
    )  
  }  
}
```

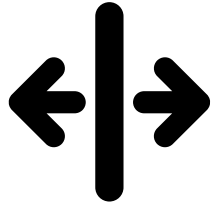
solarSystem.patl

Local states are avoided in low-level components
We get the information with `super.init()`

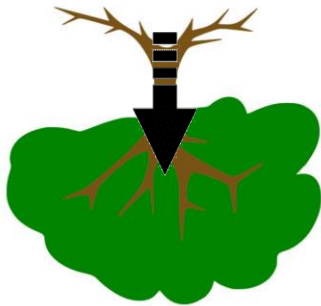
```
// Entry point  
<SystemSolar ()/>
```

solarSystem.patl

In summary

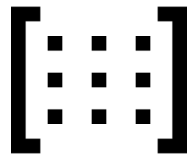


The application state is separated from the rendering (graphic representation)



All properties descend from the highest component to the lowest component

For rendering



See how to calculate the different matrices (model, view, projection) for the graphical representation



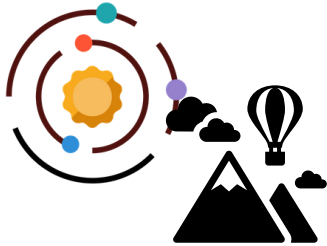
See how translate all the information in a model



And how to send it to the graphics card

Abstractions

See what abstractions we must set up:



Extend abstractions for all kinds of graphics applications

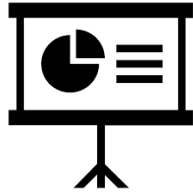


and concerning all the calculations to be made

Work incoming



Describe more our language



Work on the points of the 2 previous slides



Begin to define the grammar

References (Links)

<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/RenderEngine/Rendery>

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

References (Links)

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

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

<http://adv-r.had.co.nz/dsl.html>

<http://www.raywenderlich.com/1517-swift-tutorial-introducing-structures>

[https://fr.wikipedia.org/wiki/Arbre de la syntaxe abstraite](https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite)

[https://fr.wikipedia.org/wiki/Grammaire formelle](https://fr.wikipedia.org/wiki/Grammaire_formelle)

References (Links)

<https://thebookofshaders.com/>

<https://github.com/yanagiba/swift-ast/tree/master>

<https://medium.com/@DAloG/swift-ast-wrote-in-swift-part-1-of-e8768cae9cd3>

<https://ruslanspivak.com/lsbasi-part7/>

<https://docs.swift.org/swift-book/ReferenceManual/zzSummaryOfTheGrammar.html#>

<https://developer.apple.com/documentation/swift/mirror>

References (new links)

<https://craftinginterpreters.com/representing-code.html>

https://www.youtube.com/watch?v=bJ9ciH2XEqA&ab_channel=RWTH-Aachen-LehrstuhlSoftwareEngineering

<https://2020.splashcon.org/details/splash-2020-oopsla/49/Geometry-Types-for-Graphics-Programming>

<https://github.com/anzen-lang/anzen>

<http://sdz.tdct.org/sdz/les-shaders-en-glsl.html>

References (Tools)

<https://swift-ast-explorer.com/>

<https://github.com/kishikawakatsumi/swift-ast-explorer>

<https://astexplorer.net/>

<https://codepen.io>

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.

Joey de Vries. 2020. Learn OpenGL – Graphics programming. (Juin 2020), 523 pages.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop Domain-Specific Languages. ACM Comput. Surv. 37, 4, (December 2005), Pages 316-344.

Tomaž Kosar, Pablo E. Martí'nez López, Pablo A. Barrientos, Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, Volume 50, Issue 5, 2008, Pages 390-405.

References (Research)

Khronos Group. 2016. The OpenGL ES Shading Language. (Janvier 2016), 161 pages.

<https://arxiv.org/abs/1908.05845> (ex: game of life, ..) DynaSOAR

<https://fr.reactjs.org/>

<https://fr.reactjs.org/docs/hello-world.html>

Working with shaders

Patrick SARDINHA