

Working with Graphics Programming

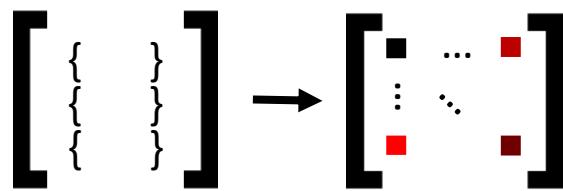
Patrick SARDINHA

What's a shader?

Small programs that run on the GPU

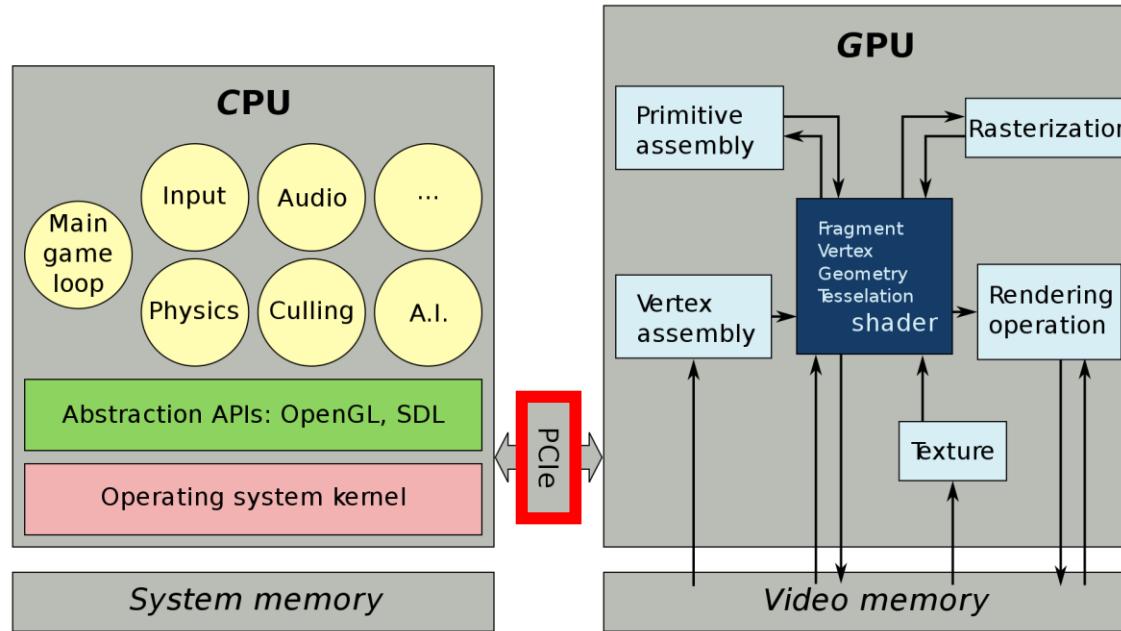
Executed for each specific section of the graphics pipeline

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

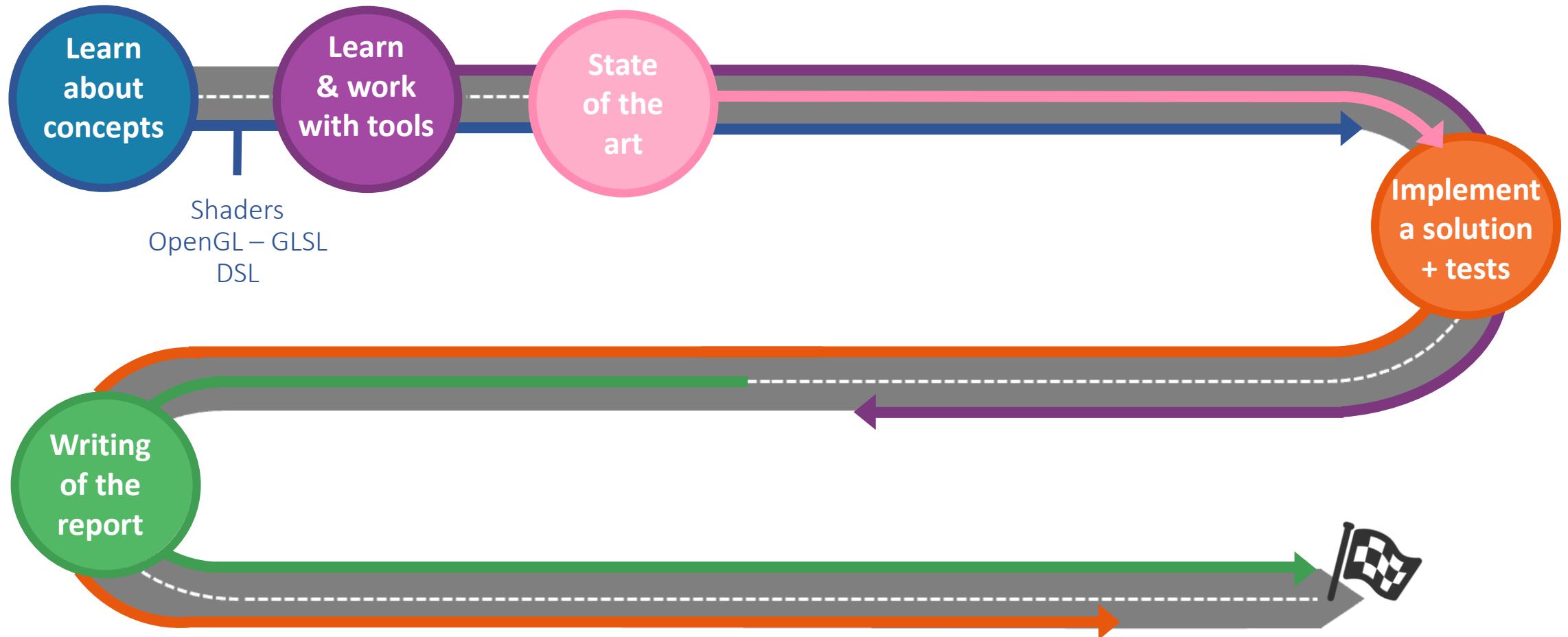


Work with the representation of the data
& abstract the types



Construct a DSL for shaders

Road map



3D space to 2D screen space

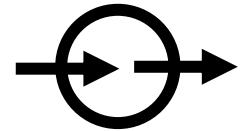


The process of transforming 3D coordinates to 2D pixel is done by the [graphics pipeline](#)

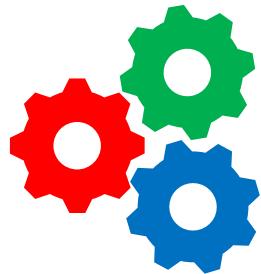
First big part: transforms 3D coordinates into 2D coordinates

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

Graphics pipeline



Input & Output Data

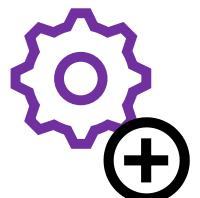


3 different shaders processing units

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 (RGB, ...)

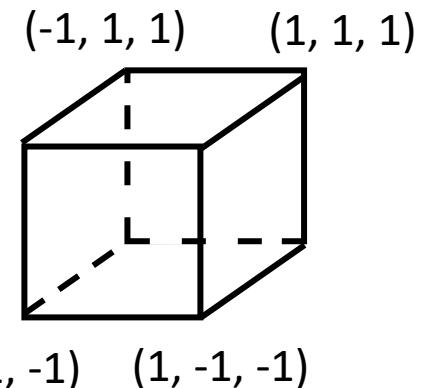


Texture coordinates



Example

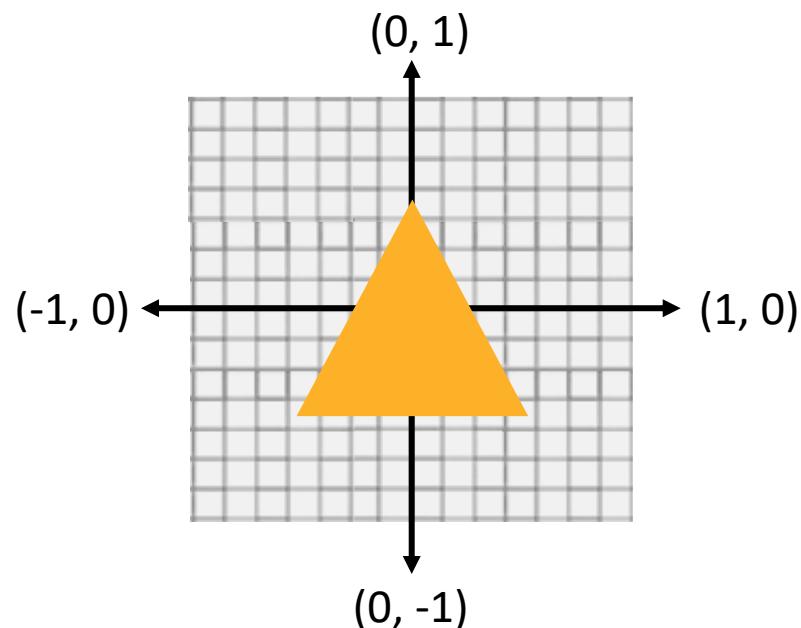
In OpenGL, only the Normalize Device Coordinates (NDC) are visible on the screen



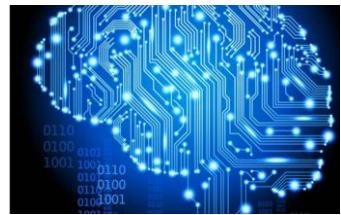
To render a single 2D triangle:

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

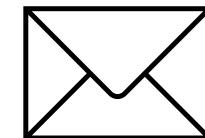


The input data will compose a [Vertex Buffer Object \(VBO\)](#) which can store a large number of vertices in the GPU memory

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

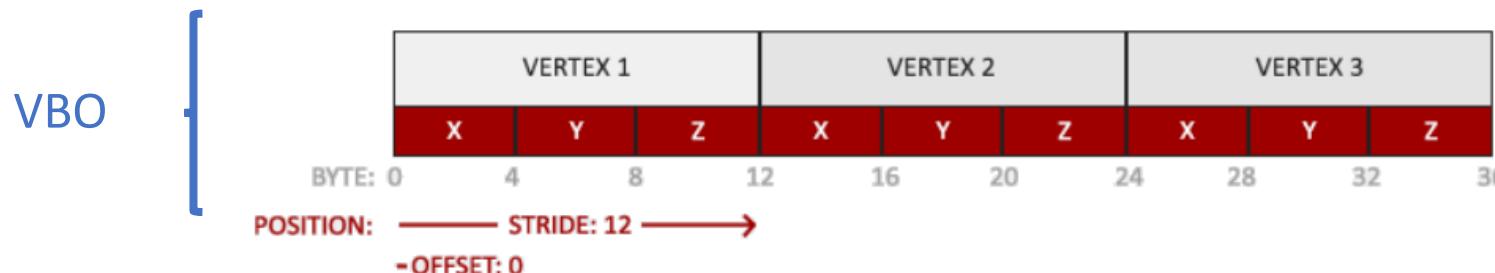
ID of the buffer which must be bind

Specifies the target buffer object

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

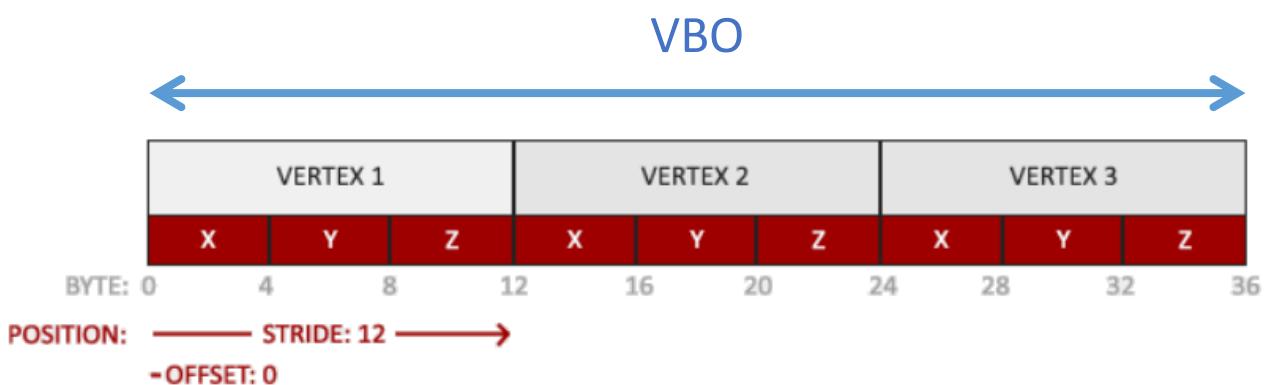
Size of the buffer object

Pointer to data



Example (Cont.)

Define how the vertex data should be interpreted



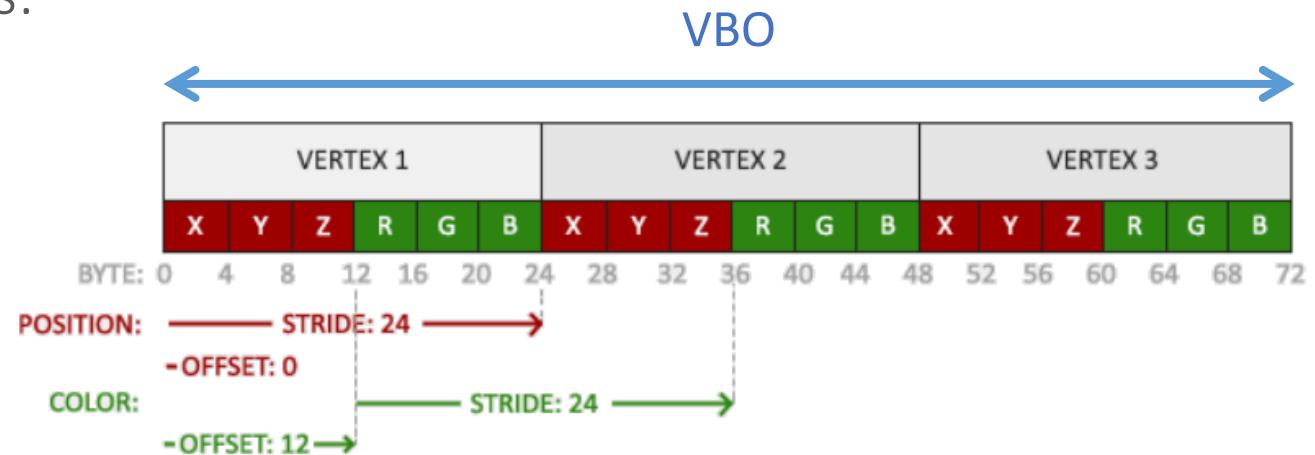
```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*) 0);  
glEnableVertexAttribArray(0);
```

The size of the vertex attribute
Specifies which vertex attribute
Type of the data
Normalized data or not
Stride: Space between consecutive vertex attributes
Enable the vertex attribute
Offset of where the position data begins in the buffer

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

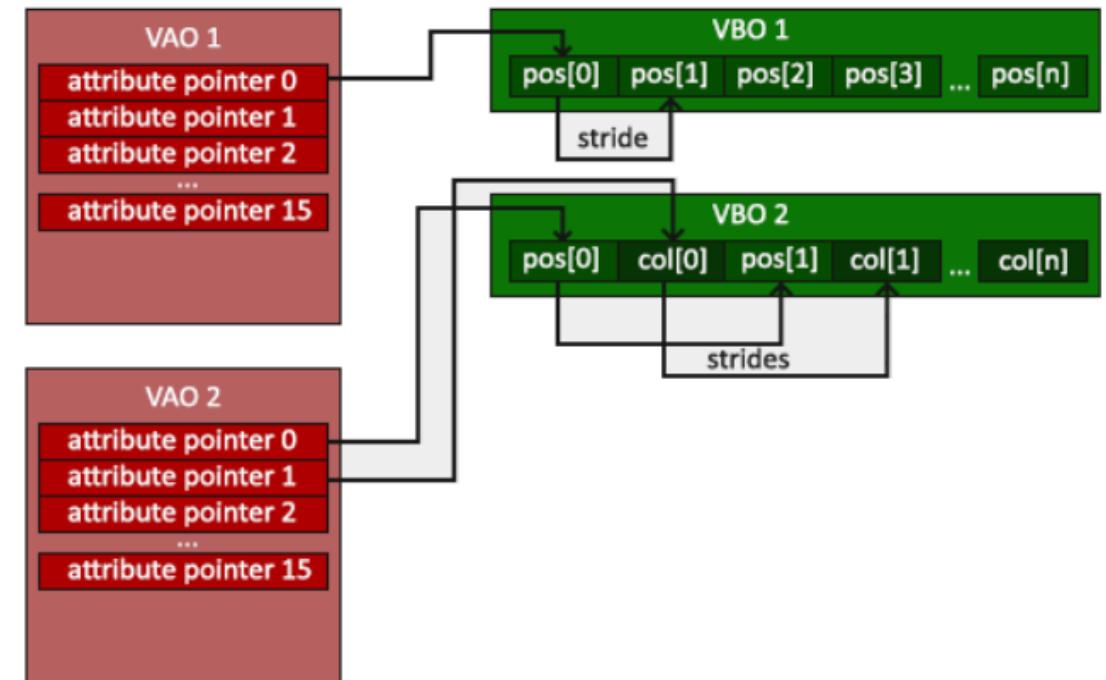
Color offset

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

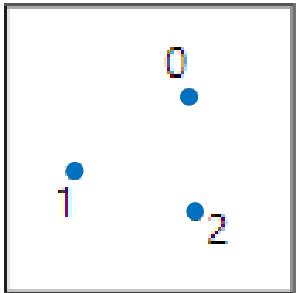


The idea now is to render and draw an object.
To do that we will have to:



- 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

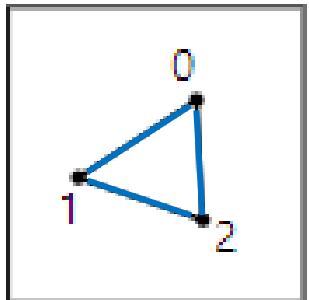
The diagram shows a vertex shader code block with several annotations:

- A red arrow points to the `#version 330 core` directive, labeled "Shader's version (here 3.3)".
- A blue arrow points to the `layout (location = 0)` declaration, labeled "Specifically set the location of the input variable".
- A pink box highlights the `in vec3 aPos;` declaration, labeled "All the input vertex attributes".
- A green bracket underlines the `gl_Position` assignment, labeled "(x, y, z) + w component".
- A red bracket underlines the `aPos.x`, `aPos.y`, and `aPos.z` components.
- A red bracket underlines the `1.0` constant.
- A red arrow points from the bottom left to the `gl_Position` line, labeled "Output of the vertex shader".

```
#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);
```

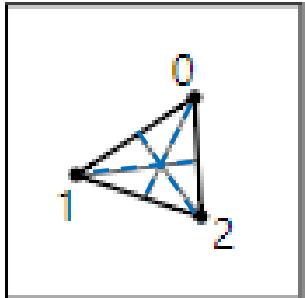
OpenGL function
that draws a shape

Starting index in the array

Kind of primitive to render

Number of vertices to render

Tessellation



In 3D, the surfaces are built with triangular tiles

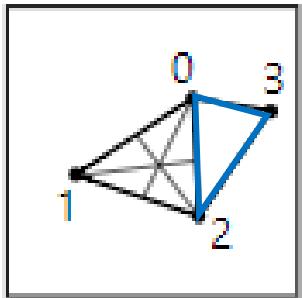
Tessellation allows to double triangles on a given surface
and therefore increase the level of details

Geometry Shader



An unnecessary step

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

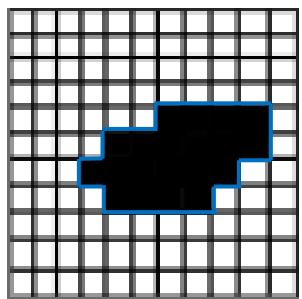


Input data: data of a geometric primitive

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

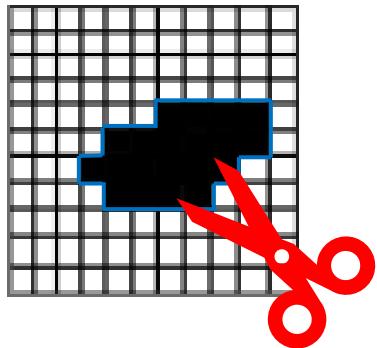


Vector image
composed of geometric objects



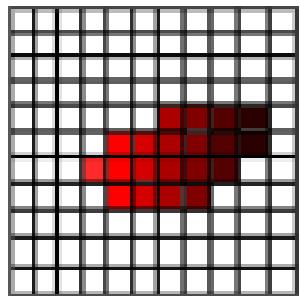
Raster image or Bitmap
composed of pixels

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)

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"
    "}\n";
```

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);
```

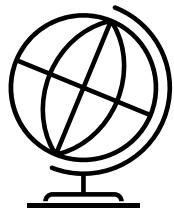
We can now activate this program to render and draw an object

```
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;

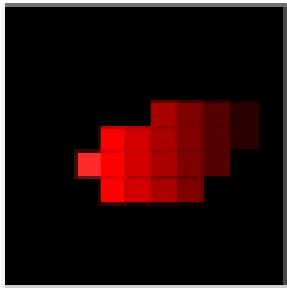
uniform vec4 ourColor;

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

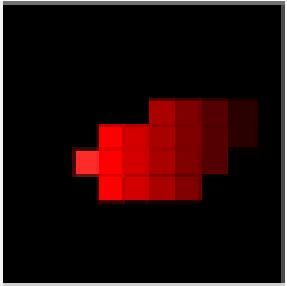


Done with the depth testing using a **Z-buffer** (in which the depth value of the fragments is stored)

```
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:

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

Final color of the fragment

Impact of the alpha value

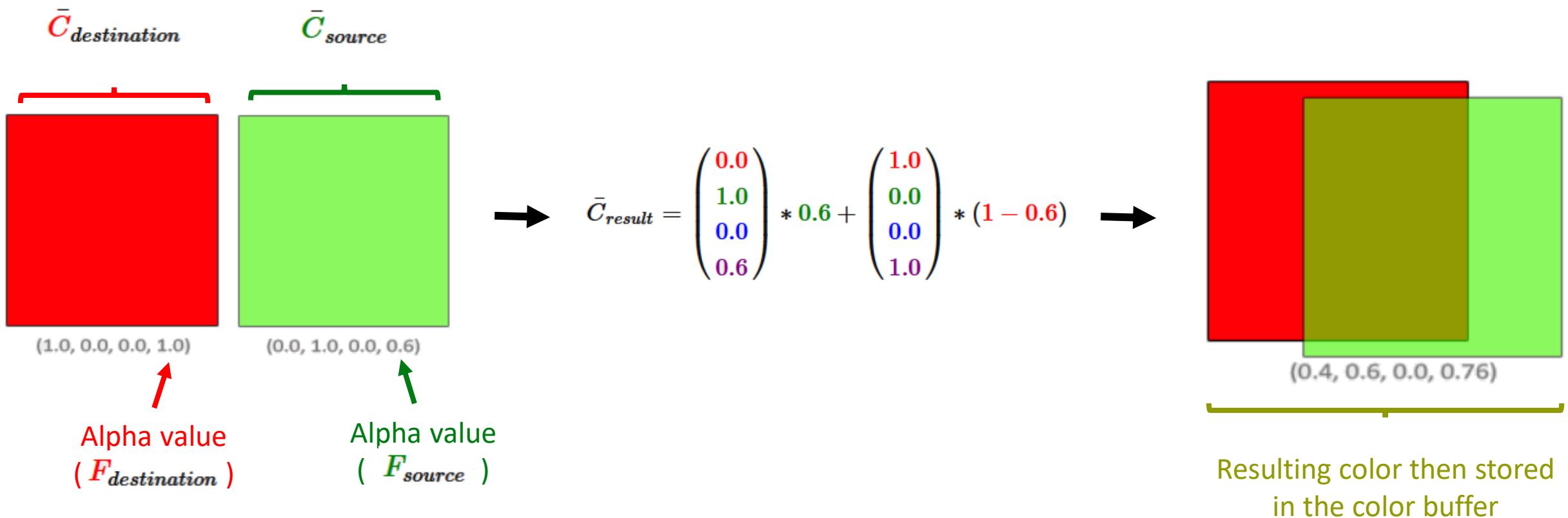
Color output of the fragment shader

Impact of the alpha value

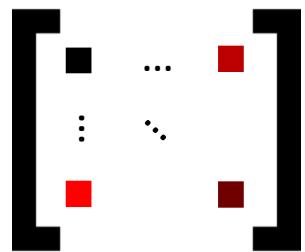
Color currently stored in the color buffer

Example (Cont.)

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



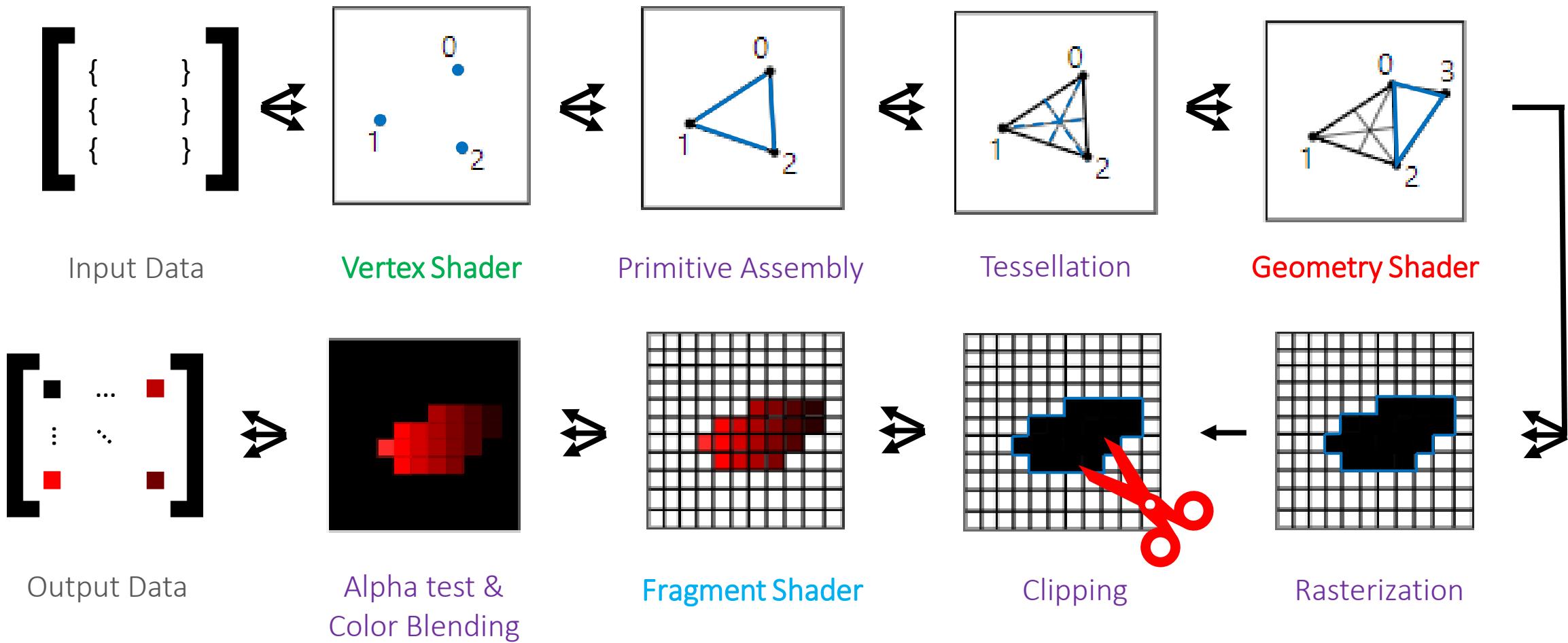
Output Data



Return a **Framebuffer**

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

Overall view



Textures

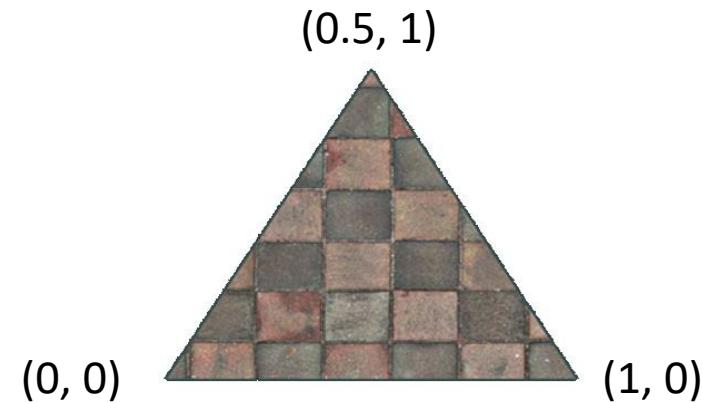
Allows to give the illusion the object is detailed without having to specify vertices

Associate each vertex to a texture coordinate

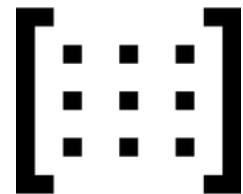
A fragment interpolation is then done for the other fragments

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)

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

Multiply vec by the translation matrix

The matrix to transform (identity Matrix4)

A translate function

The translation vector to apply to the matrix

Coordinates system

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



Local Space



World Space



View Space

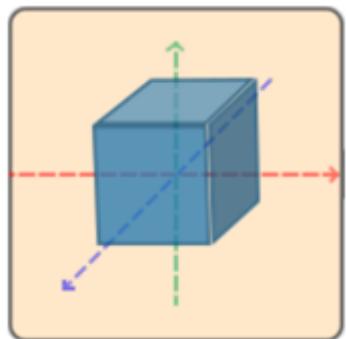
Clip Space



Screen Space



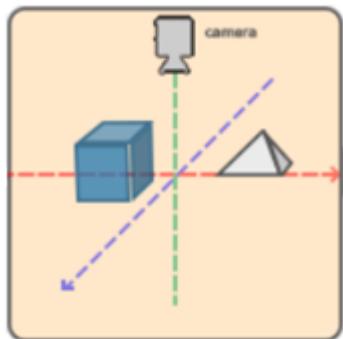
Local Space



Coordinates of the object relative to its local origin

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

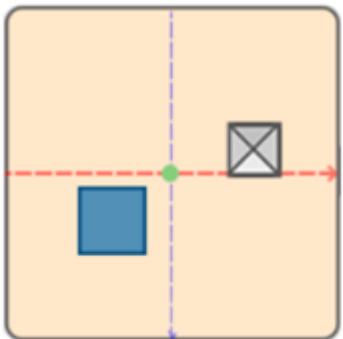
World Space



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

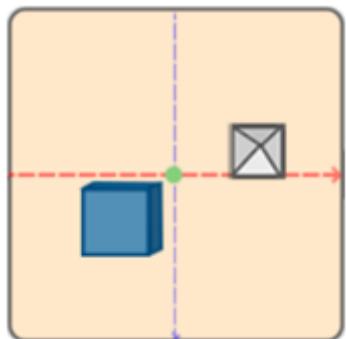
View Space



Each coordinate 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

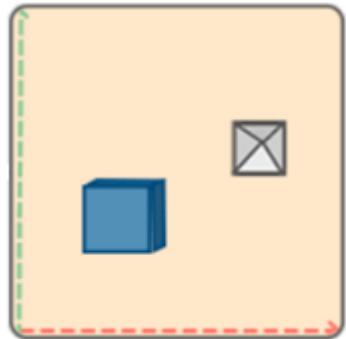
For this step, we use a projection matrix which transform the coordinates into NDC

Example:

Specified range [-1000, 1000] for each dimension

$(1250, 500, 750)$	\rightarrow	Not visible
$(900, 500, 750)$	\rightarrow	Visible

Screen Space

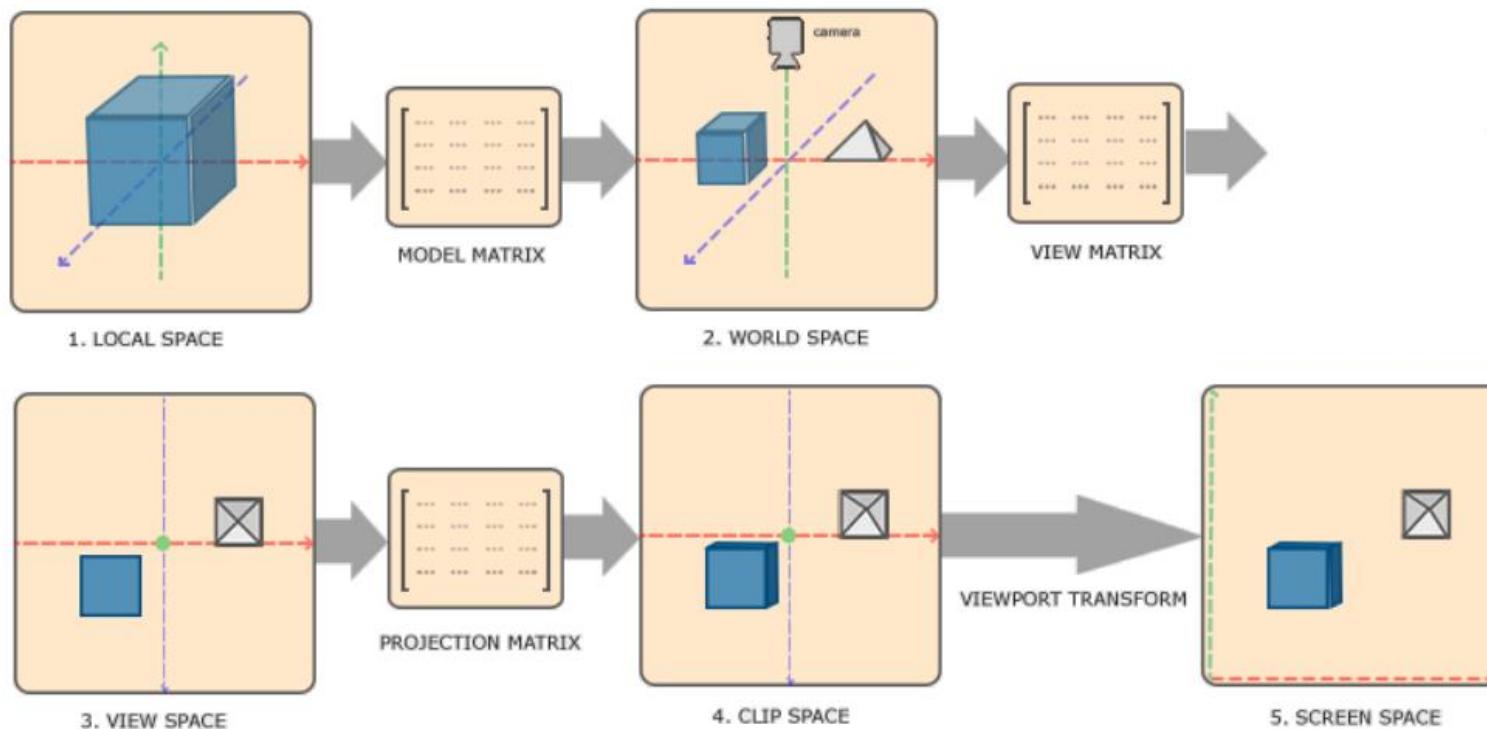


Transforms the NDC coordinates to the window coordinates with the `glViewport()` function

Resulting coordinates are then sent to the rasterizer

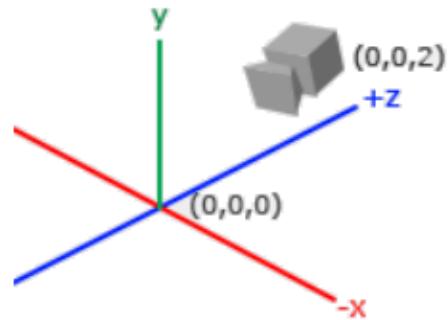
Overall view

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

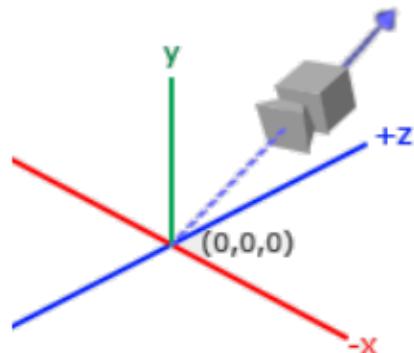


Camera

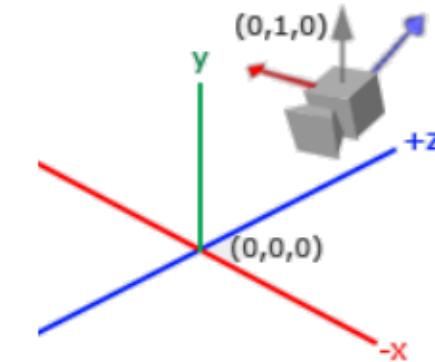
To define a camera we need 4 pieces of information



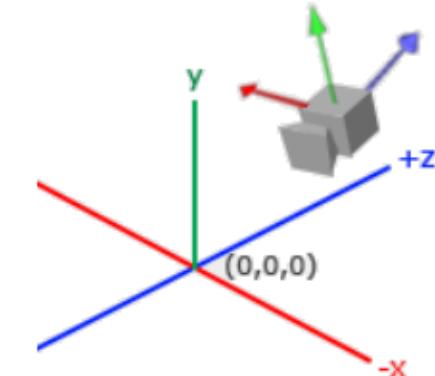
1. Its position in the world space



2. Its direction

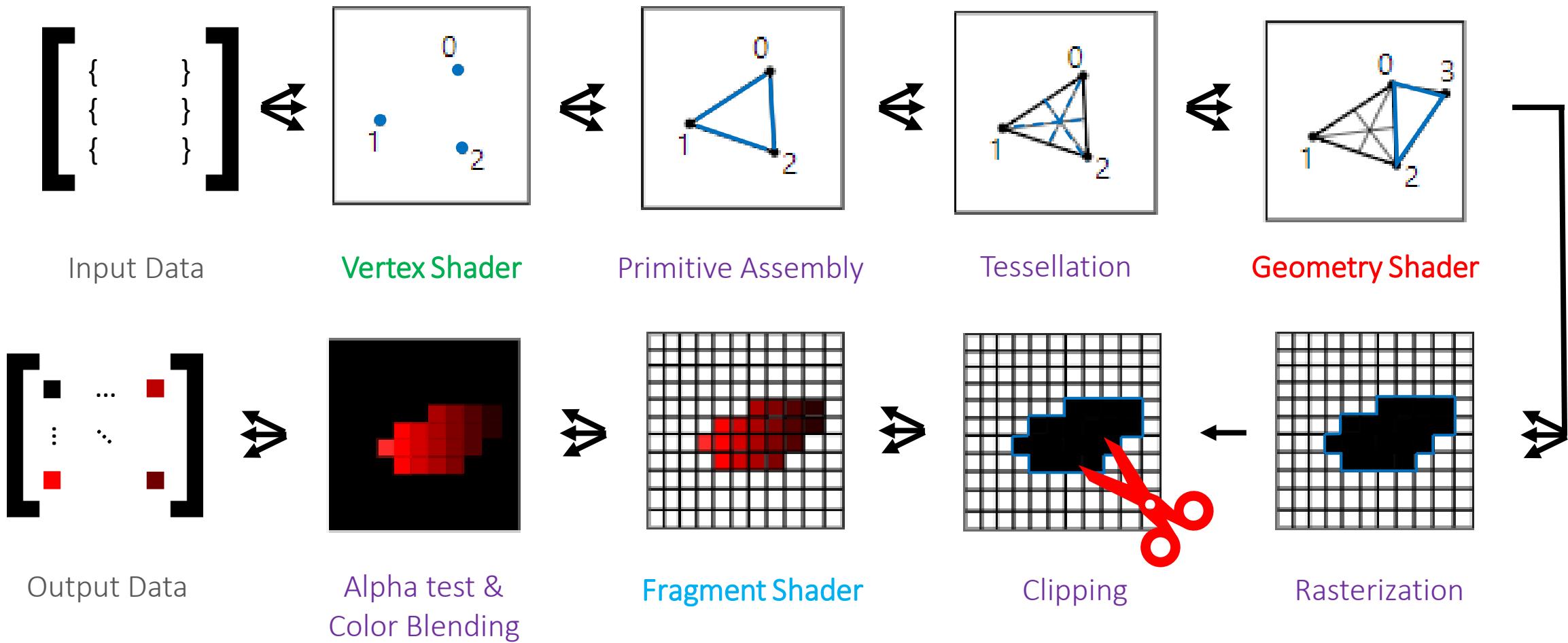


3. A vector pointing to the right



4. A vector pointing upwards

Recall : Graphics pipeline



Pipeline abstraction

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

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

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, ...

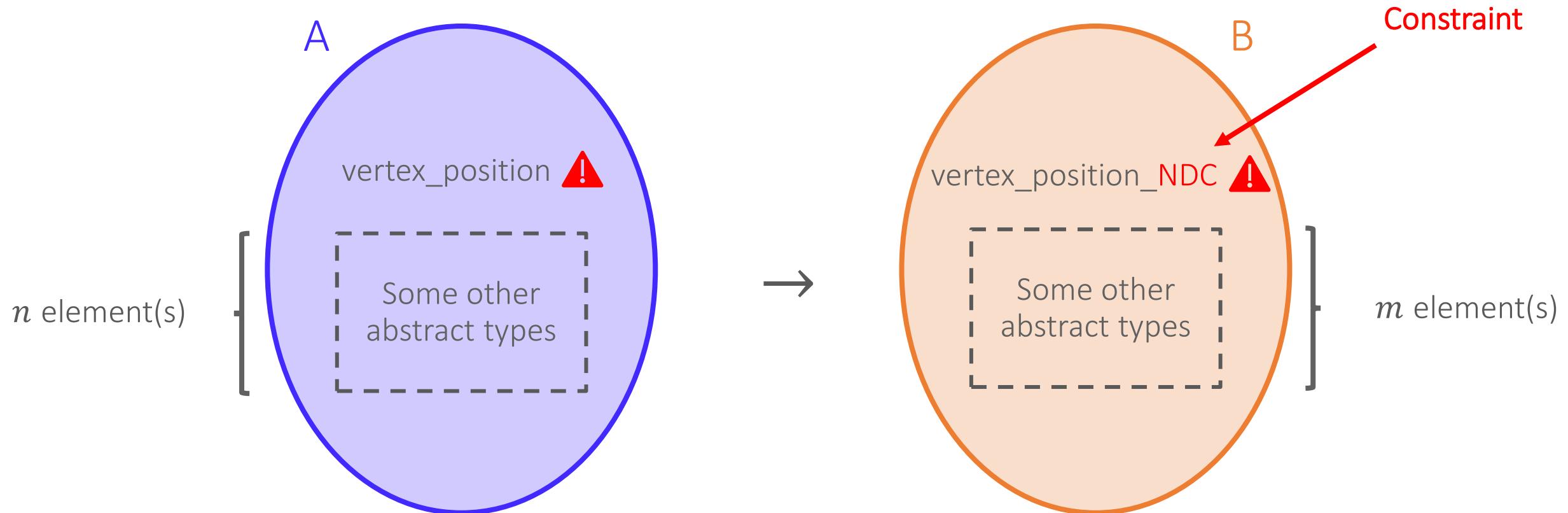


But with abstract type objects

Ex: color, position, textures, ...

Vertex Shader function

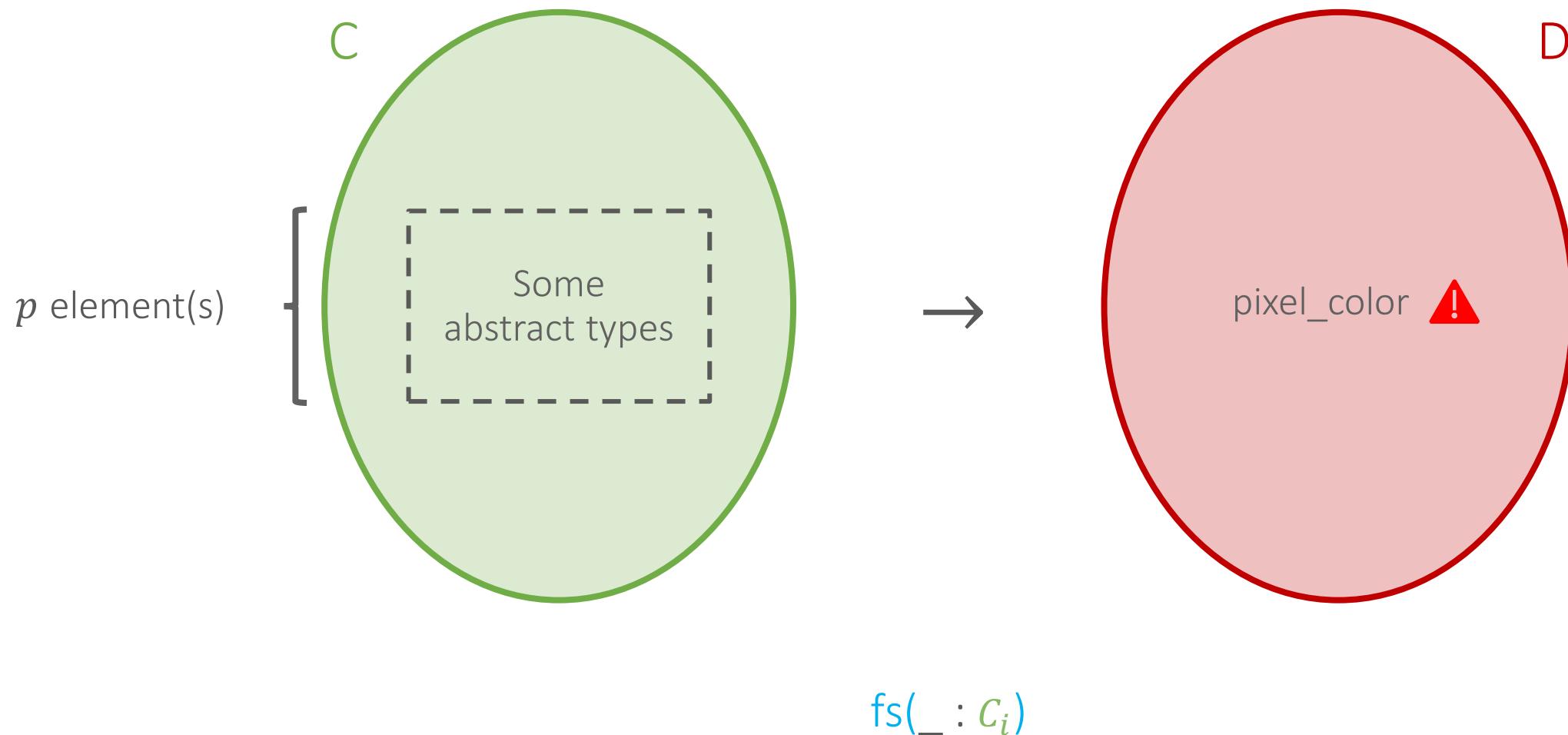
$\text{vs} : A \rightarrow B$



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

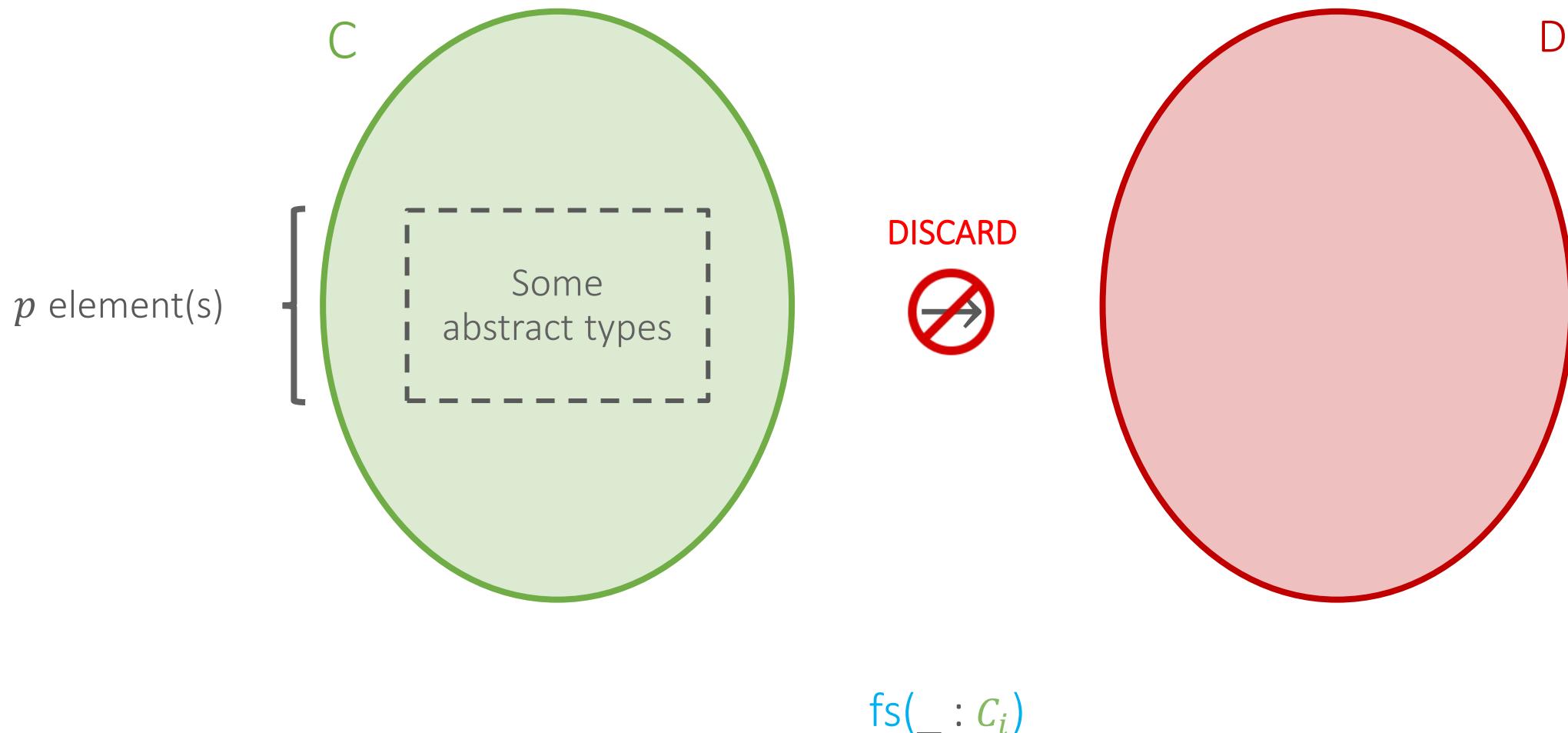
Fragment Shader function

$\text{fs} : \mathcal{C} \rightarrow \mathcal{D}$

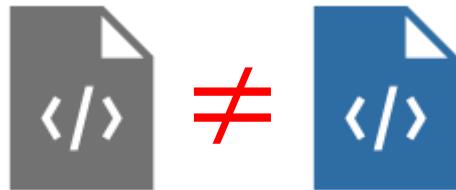


Fragment Shader function Alternative

$fs : C \rightarrow D$



Several signatures



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

Examples:

`vs(_ : position)`

`fs()`

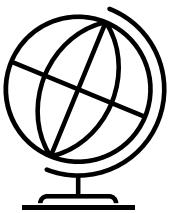
`vs(_ : position, _ : color)`

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

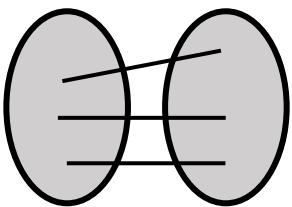
`vs(_ : position, _ : color, _ : texture:)`

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

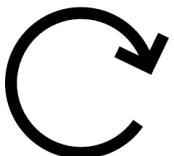
Uniforms



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;
};

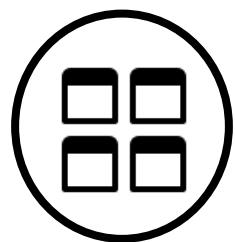
// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos     = mul(modelViewProj, IN.pos);
    OUT.color   = IN.color;
    return OUT;
}
```

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

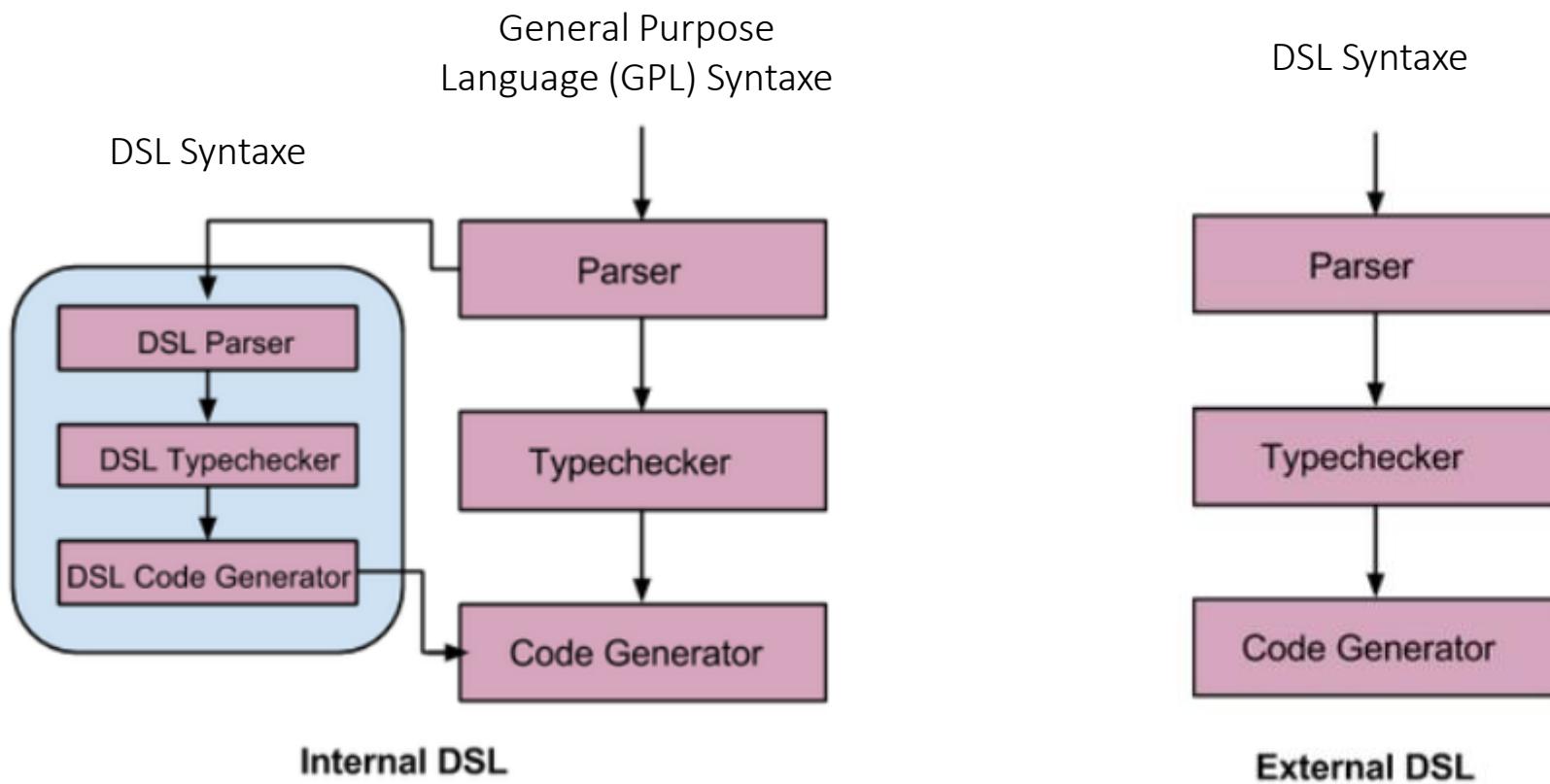
DSL can be reused for other purposes



DSL maintenance is complicated

The cost of a DSL is expensive

Different types of DSL



Our way



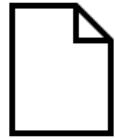
First, we will go on an Internal DSL based
on the Swift language

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



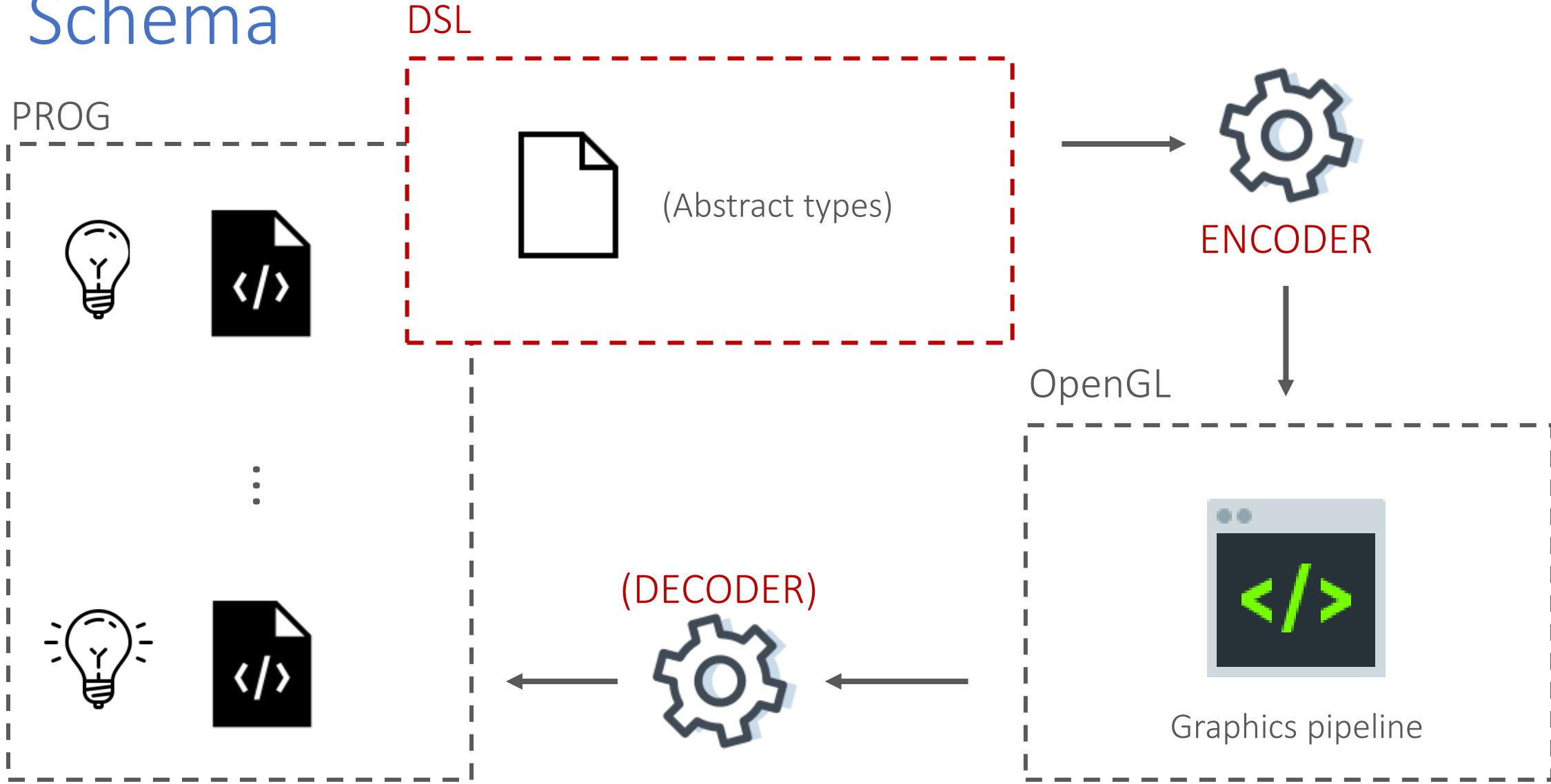
This translation can then be evaluated by the graphics pipeline of OpenGL



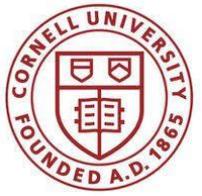
(Then a decoder allows us to get the results with the desired abstract types)



Schema



Gator



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

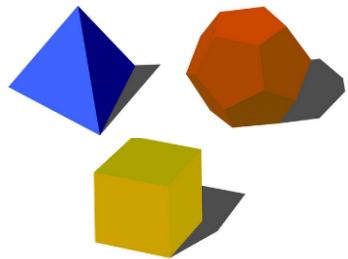


Higher level programming model that allows focus on the geometric semantics of programs

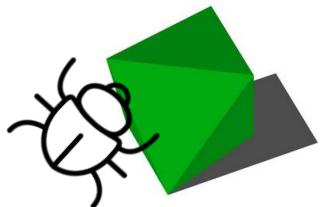


Gator is a **surface language** with an extended type system
based on a **target language** with a type set (GLSL)
A type-directed translation allows to compile Gator to GLSL

Problem & ideas



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

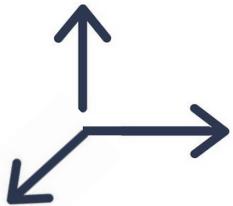


Geometry bugs are difficult to detect



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”

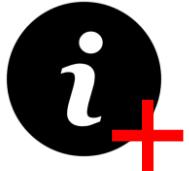
A geometry type is made up of 3 components:

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

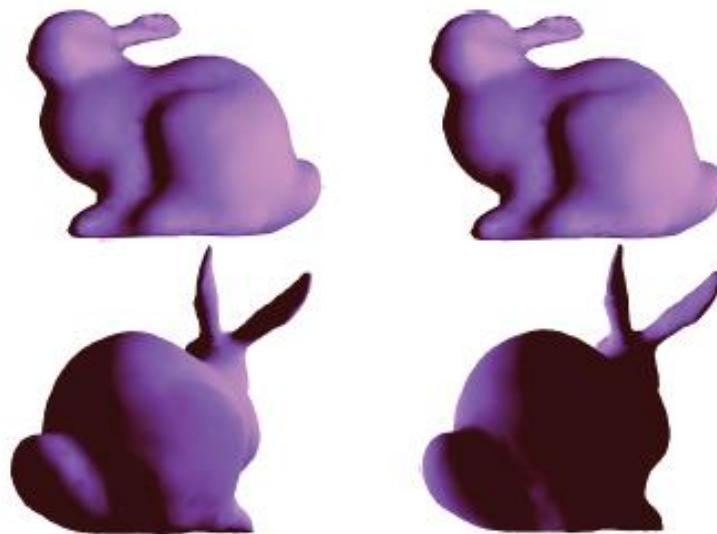
GLSL implementation (Cont.)

```
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

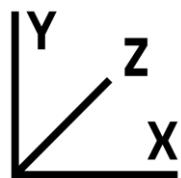


Gator helps to limit the number of geometry bugs

Limitations

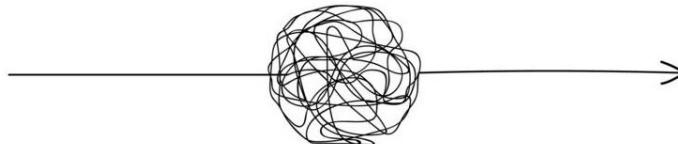


The created abstraction remains low level



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



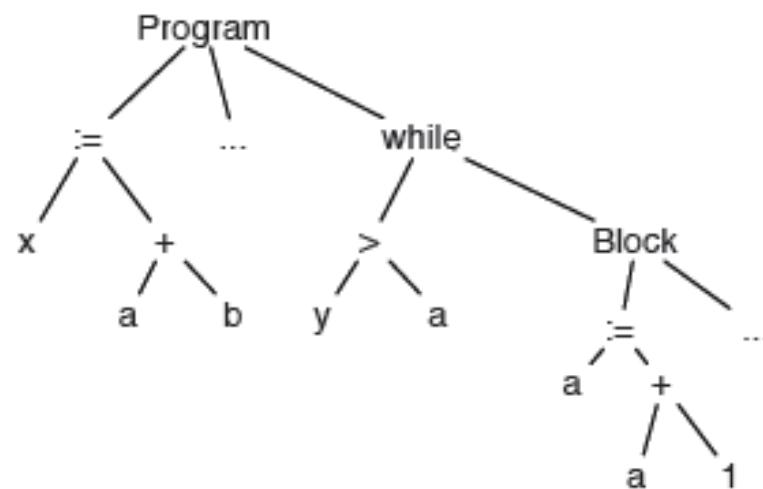
Abstract Syntax Tree (AST)



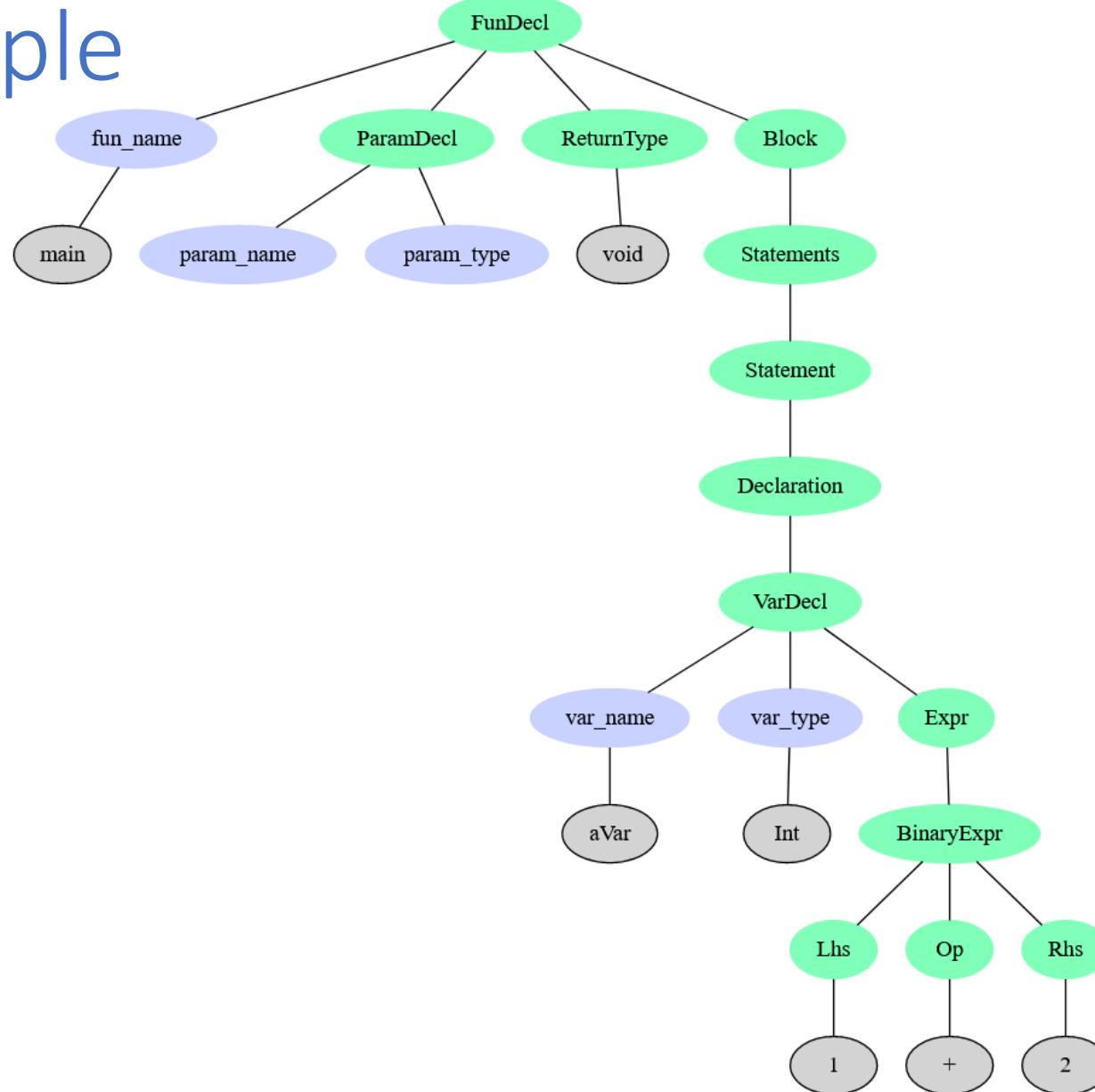
Tree in which, internal nodes are marked by operators
and where external ones represent the operands

Example:

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



AST example



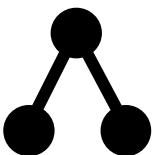
Some tools



Swift-AST-Explorer

Only supports Swift language

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



AST-Explorer

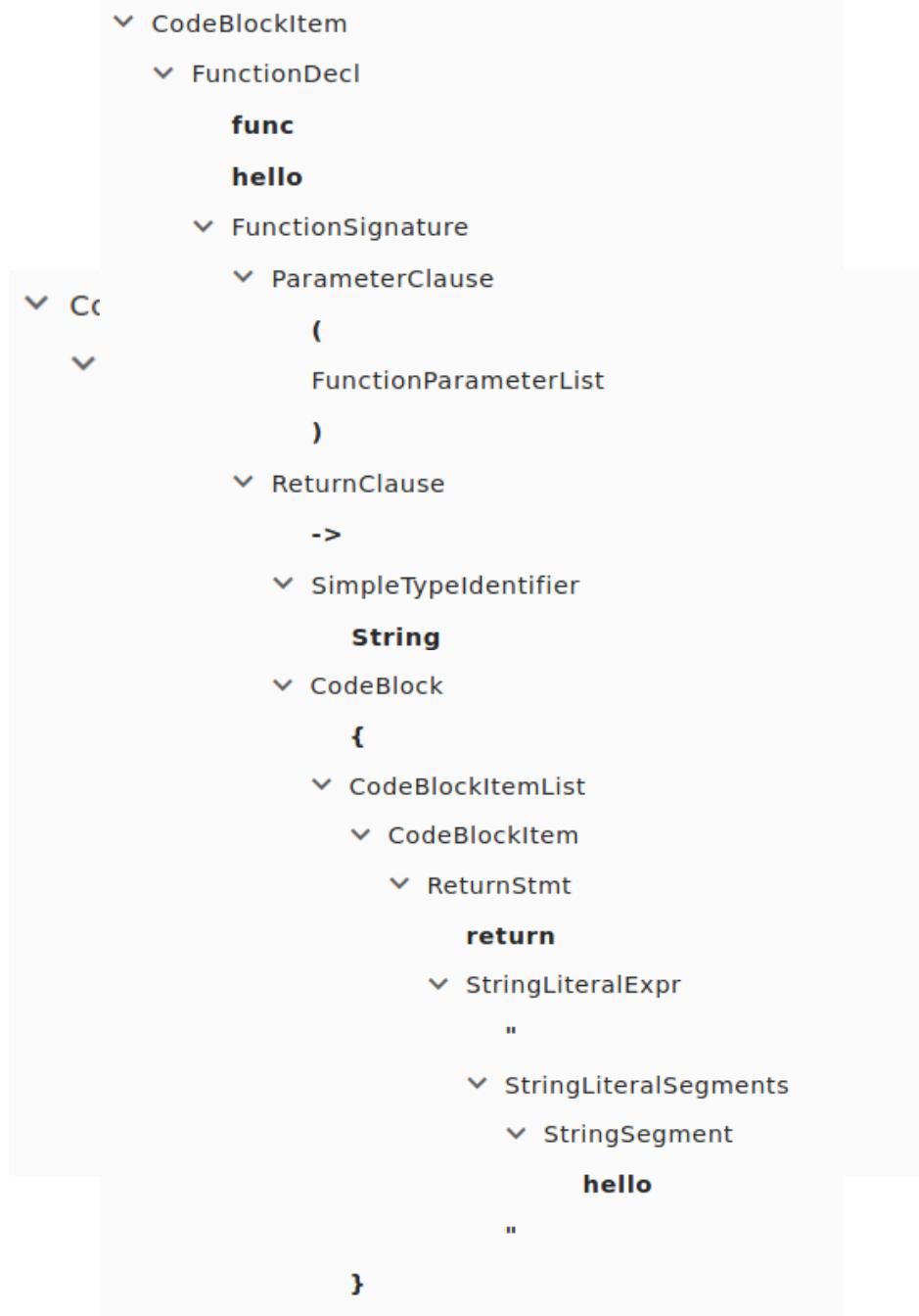
Supports a lot of languages like Java, HTML, Python, ... & GLSL

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}  
    + 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:

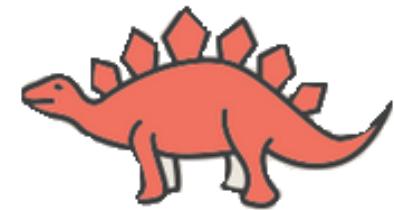
function_identifier:
 type_specifier
 IDENTIFIER
 FIELD_SELECTION

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

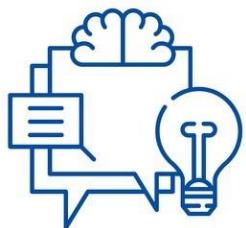


We want to have the perfect language for our use-cases

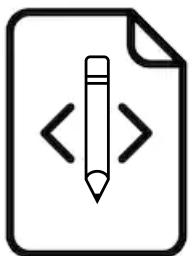


We want to have high-level objects that can be handled in a simple way

Development



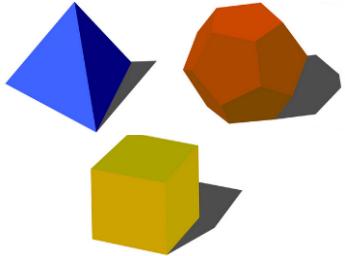
First, we must define our language and the concepts we want



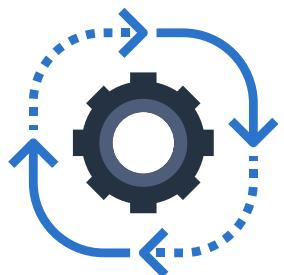
Then, we must specify the syntax of this language

Concepts

Use-cases



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



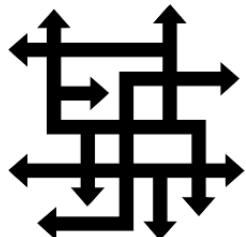
We want to call methods on them, do calculations, etc.

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



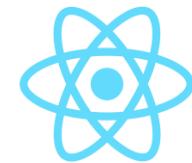
The operations linked to these objects



The physics of the objects

Declarative Language

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



React is based on the notions of:

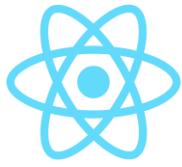


Components, Properties and States



Render function, Update functions

React - Component



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



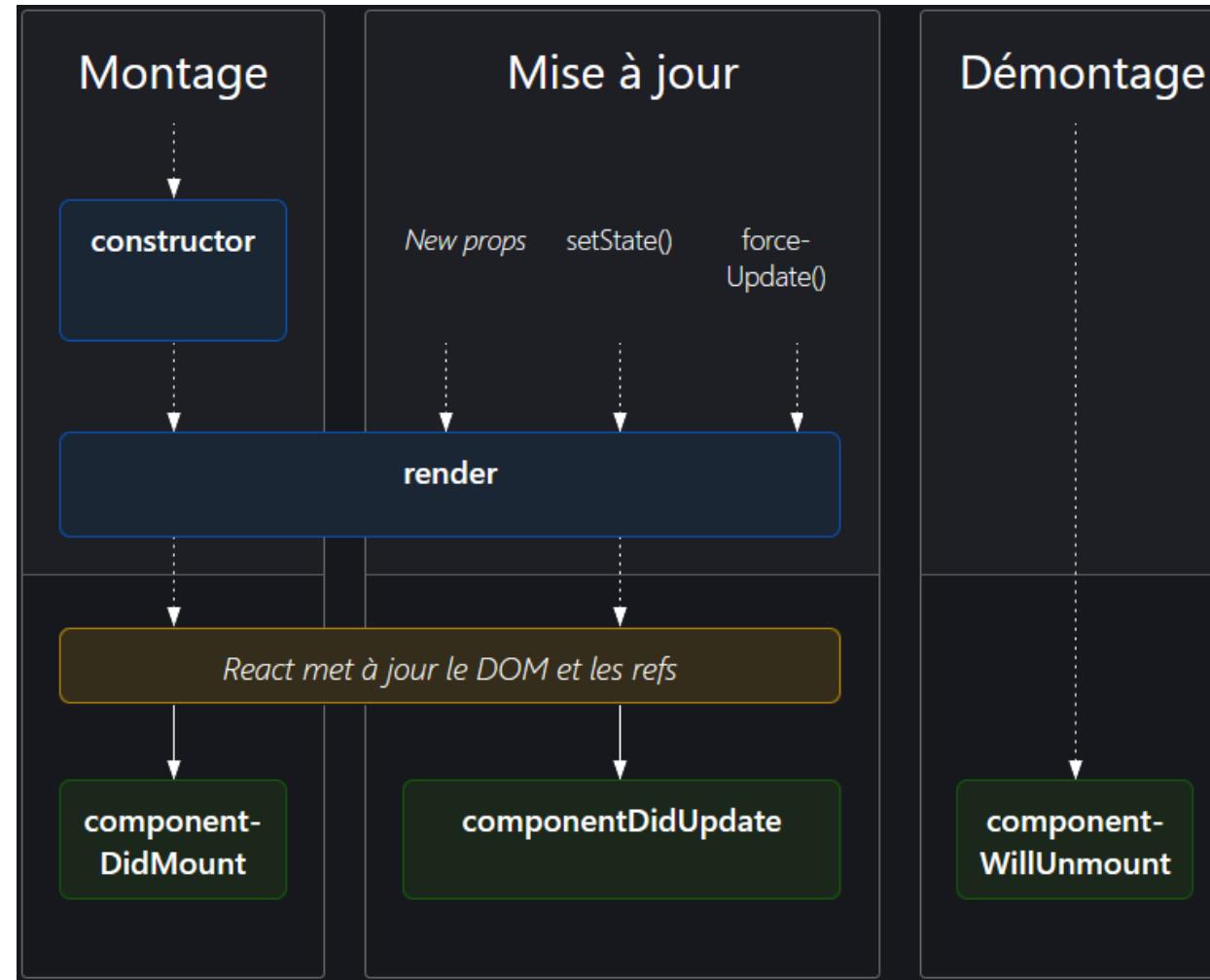
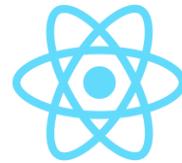
We can then combine different components to get something more complex

Example:

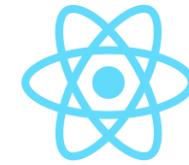
```
function Welcome(props) {
  return <h1>Bonjour, {props.name}</h1>;
}
```

```
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

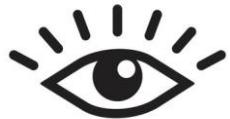
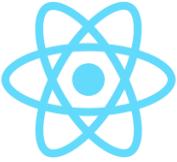
A component should never modify its own properties



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.

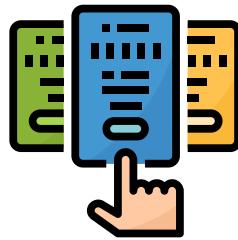
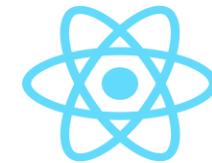


When the local state of a component changes, its display is updated by calling render()



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

React - componentDidMount() & componentDidUpdate()



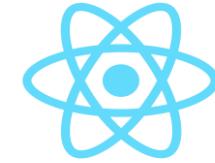
componentDidMount() is called immediately after
the component is mounted

This is where we make the subscriptions



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

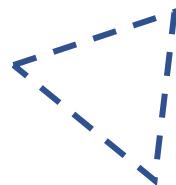
React - componentWillMount()



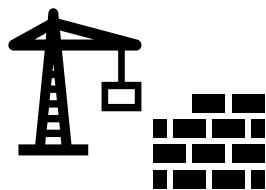
componentWillUnmount() is called immediately before
a component is unmounted

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



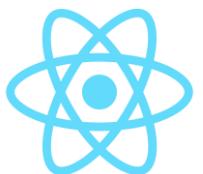
Syntax



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

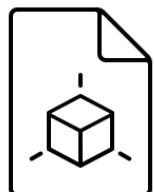
React to our language



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)

Tick() function declaration

Function given in the standard library
to update the global state

```
function tick() {  
    state = updateState(lastState: state,  
                        newState: ["key": value,  
                                   ...]  
    )  
}  
  
function render() {  
    return (  
        <container>  
  
        <><component_name> ([<arg_name>=value])</>  
  
        </container>  
    )  
}
```

Component rendering function

Components (lowest level)

Component declaration

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

Get the properties passed to
the component

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

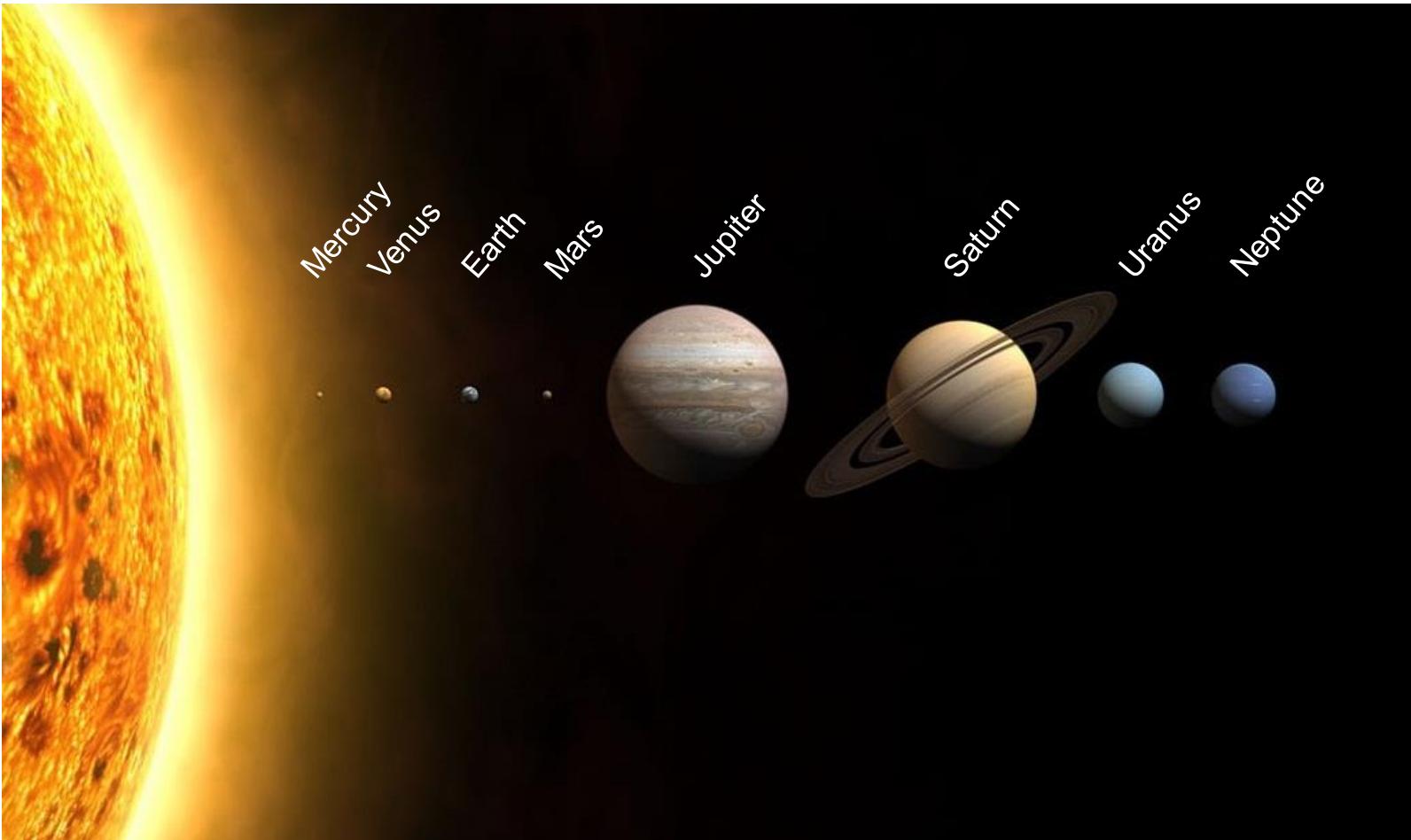
Entry point

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

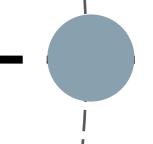
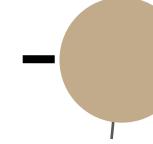
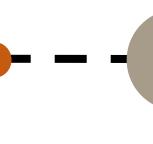
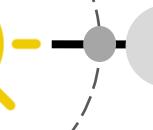
```
structure <name_structure> {  
    var <var_name>: <type>  
    ...  
  
    init(<args>) {  
        ...  
  
        function <function_name>([<arg_name>: <type>]) -> <return_type> {  
            <body>  
            return <return_value>  
        }  
    }  
  
    <><highest_level_component>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

Then, we specify the top-level component

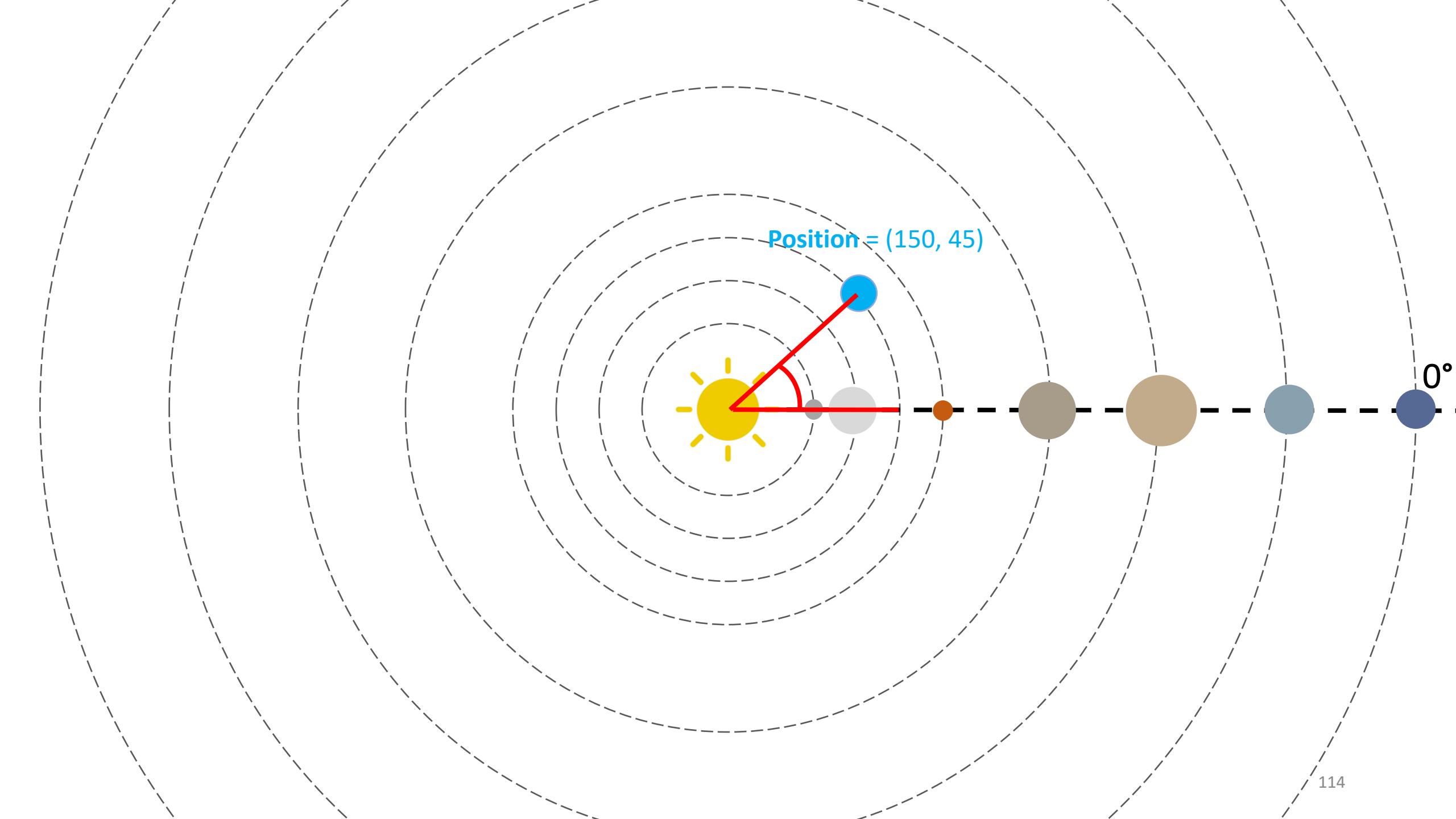
Example - Solar system



Position = (sun distance, polar angle)



0°



A diagram illustrating the positions of celestial bodies relative to the Sun. The Sun is represented by a yellow circle with radiating lines. A blue circle, labeled "Position = (150, 45)", is shown in its orbit. A grey circle is also visible. A red arrow points from the Sun to the blue circle, indicating its position. To the right, a series of dashed circles represent other orbits. A horizontal dashed line extends from the Sun through several grey dots, which are labeled with numbers: 0°, 114, and 114 again. The background consists of dashed elliptical orbits.

0°

Position = (150, 45)

```
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.plist

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
```

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

bibli.patl

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

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

bibli.patl

```
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.pat

```
Component Sphere {
    init() {
        super.init()
    }

    let meshPosition = pos.polarToCart()

    function render() {
        return (
            <mesh type="sphere"
                position={meshPosition}
                properties={[radius]}/>
        )
    }
}
```

solarSystem.patl

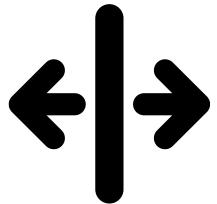
Local states are avoided in low-level components

We get the information with `super.init()`

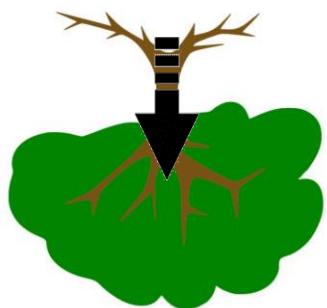
```
// Entry point  
<SystemSolar ()/>
```

solarSystem.pat

In summary



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



All properties descend from the highest component to the lowest component

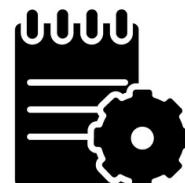
Recall: Embedded DSL



The idea is to use Swift as the host language



This implies making some sacrifices on the syntax of our language ...



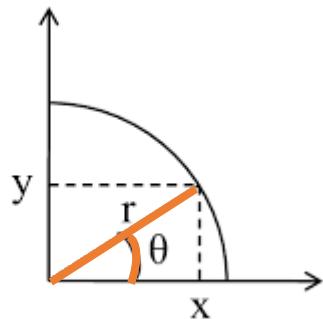
... but allows us to use the swift compiler

Example - Solar system: Using Swift

```
class SystemSolar {
    var state: Dictionary<String, Any> = [:]
    var callTick: Bool = true

    init() {
        self.state = ["solarCoord": Coord(polar: (0, Angle(deg: 0))), "solarRevo": 0.0, "solarRadius": 696340.0,
                     "mercuryCoord": Coord(polar: (58, Angle(deg: 0))), "mercuryRevo": 88.0, "mercuryRadius": 2440.0,
                     "venusCoord": Coord(polar: (108, Angle(deg: 0))), "venusRevo": 225.0, "venusRadius": 6050.0,
                     "earthCoord": Coord(polar: (150, Angle(deg: 0))), "earthRevo": 365.0, "earthRadius": 6370.0,
                     "marsCoord": Coord(polar: (228, Angle(deg: 0))), "marsRevo": 687.0, "marsRadius": 3390.0,
                     "jupiterCoord": Coord(polar: (779, Angle(deg: 0))), "jupiterRevo": 4380.0, "jupiterRadius": 69910.0,
                     "saturnCoord": Coord(polar: (1434, Angle(deg: 0))), "saturnRevo": 10585.0, "saturnRadius": 58230.0,
                     "uranusCoord": Coord(polar: (2871, Angle(deg: 0))), "uranusRevo": 30660.0, "uranusRadius": 25360.0,
                     "neptuneCoord": Coord(polar: (4495, Angle(deg: 0))), "neptuneRevo": 60225.0, "neptuneRadius": 24600.0
                ]
        self.subTick(interval: 1)
    }
}
```

main.swift



```
"solarCoord": Coord(polar: (0, Angle(deg: 0)))
```

main.patl

```
struct Coord {
    var polar: (Double, Angle)?
    var cart: (Double, Double)?
}

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

func polarToCart() -> (Double, Double) {
    let x = polar!.0 * cos(polar!.1.deg!)
    let y = polar!.0 * sin(polar!.1.deg!)
    return (x,y)
}

func cartToPolar() -> (Double, Angle) {
    let r = sqrt(pow(cart!.0, 2.0) + pow(cart!.1, 2.0))
    let t = Angle(deg: atan(cart!.1 / cart!.0))
    return (r,t)
}
```

bibli.patl

```
"solarCoord": Coord(polar: (0, Angle(deg: 0)))
```

main.patl

```
struct Angle {
    var deg: Double?
    var rad: Double?

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

    func degToRad() -> Double {
        return deg! * (Double.pi/180)
    }

    func radToDeg() -> Double {
        return rad! * (180/Double.pi)
    }
}
```

bibli.patl

```
// Directly call when the component is created
func subTick(interval: UInt32) {
    self.callTick = true
    while(callTick) {
        self.tick()
        sleep(interval)
    }
}
```

main.patl

```
// Call when the component is removed
func unsubTick() {
    self.callTick = false
}
```

main.patl

```
func updatePlanetPos(currentAngle: Angle, revolution: Double) -> Double {  
    let newAngle = currentAngle.deg!+(360/revolution)  
    return (newAngle.truncatingRemainder(dividingBy: 360.0))  
}
```

main.patl

```
func tick() {  
    self.state = updateState(currentState: state,  
        nextState: ["mercuryCoord": Coord(polar: ((state["mercuryCoord"] as! Coord).polar!.0,  
            Angle(deg: self.updatePlanetPos(currentAngle: (state["mercuryCoord"] as! Coord).polar!.1,  
                revolution: (state["mercuryRevo"] as! Double)))),  
        "venusCoord": Coord(polar: ((state["venusCoord"] as! Coord).polar!.0,  
            Angle(deg: self.updatePlanetPos(currentAngle: (state["venusCoord"] as! Coord).polar!.1,  
                revolution: (state["venusRevo"] as! Double)))),  
        ...  
    ]  
}  
    let _ = self.render()  
}
```

main.patl

Coord(polar: (58, Angle(deg: 0)))

(58, Angle(deg: 0))

58

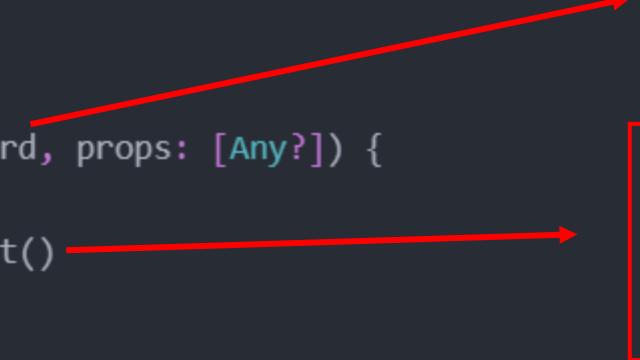
```
func updateState(currentstate: [String: Any], nextState: [String: Any]) -> [String: Any] {
    var tmpState = currentstate
    for (key, _) in nextState {
        tmpState.updateValue(nextState[key]!, forKey: key)
    }
    return tmpState
}
```

bibli.patl

```
func render() -> (Sphere, Sphere, Sphere, Sphere, Sphere, Sphere, Sphere, Sphere, Sphere) {
    return (
        Sphere(name: "Solar", pos: (state["solarCoord"] as! Coord), props: [state["solarRadius"]]),
        Sphere(name: "Mercury", pos: (state["mercuryCoord"] as! Coord), props: [state["mercuryRadius"]]),
        Sphere(name: "Venus", pos: (state["venusCoord"] as! Coord), props: [state["venusRadius"]]),
        Sphere(name: "Earth", pos: (state["earthCoord"] as! Coord), props: [state["earthRadius"]]),
        Sphere(name: "Mars", pos: (state["marsCoord"] as! Coord), props: [state["marsRadius"]]),
        Sphere(name: "Jupiter", pos: (state["jupiterCoord"] as! Coord), props: [state["jupiterRadius"]]),
        Sphere(name: "Saturn", pos: (state["saturnCoord"] as! Coord), props: [state["saturnRadius"]]),
        Sphere(name: "Uranus", pos: (state["uranusCoord"] as! Coord), props: [state["uranusRadius"]]),
        Sphere(name: "Neptune", pos: (state["neptuneCoord"] as! Coord), props: [state["neptuneRadius"]])
    )
}
```

main.patl

```
class Sphere {  
    // Here we work with 2D cartesian and 2D polar coordinates  
    var name: String  
    var pos: (Double, Double)  
    var props: [Any?]  
  
    init(name: String, pos: Coord, props: [Any?]) {  
        self.name = name  
        self.pos = pos.polarToCart()  
        self.props = props  
  
        let _ = self.render()  
    }  
  
    func render() -> Mesh {  
        log(str: "render a Sphere for [\\"(name) at position: \\"(pos) with props: \\"(props)]")  
        return (  
            Mesh(meshType: "Sphere", meshPosition: pos, meshProps: props)  
        )  
    }  
}
```



```
Coord(polar: (58, Angle(deg: 0)))  
  
func polarToCart() -> (Double, Double) {  
    let x = polar!.0 * cos(polar!.1.deg!)  
    let y = polar!.0 * sin(polar!.1.deg!)  
    return (x,y)  
}
```

main.path

```
struct Mesh {  
    var meshType: String  
    var meshPosition: (Double, Double)  
    var meshProps: [Any?]  
  
    init(meshType: String, meshPosition: (Double, Double), meshProps: [Any?]) {  
        self.meshType = meshType  
        self.meshPosition = meshPosition  
        self.meshProps = meshProps  
    }  
}
```

bibli.pat

```
// Entry: call the highest component
let _ = SystemSolar()
```

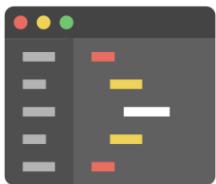
main.patl

Syntax decoration

So, to get a different syntax from Swift we will go through:



The definition of custom operators



The creation of a template language to define our own keywords

Formalization

A Semantics for the Essence of React

Magnus Madsen 

Aarhus University, Denmark

<https://www.cs.au.dk/~magnusm/>

magnusm@cs.au.dk

Ondřej Lhoták 

University of Waterloo, Canada

<https://plg.uwaterloo.ca/~olhotak/>

olhotak@uwaterloo.ca

Frank Tip 

Northeastern University, USA

<https://www.franktip.org>

f.tip@northeastern.edu

GLOBAL_SYSTEM

COMPONENT_TOPLEVEL

(STATE)

INIT

UN/SUBSCRIPTIONS

RENDER

COMPONENT_LOWLEVEL

(PROPS)

INIT

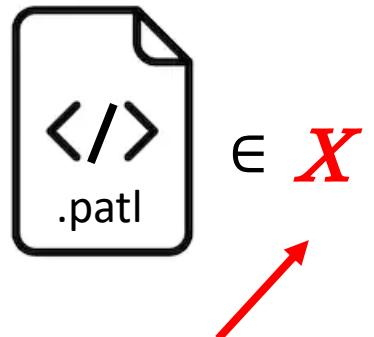
RENDER

COMPONENT_LOWLEVEL | MESH

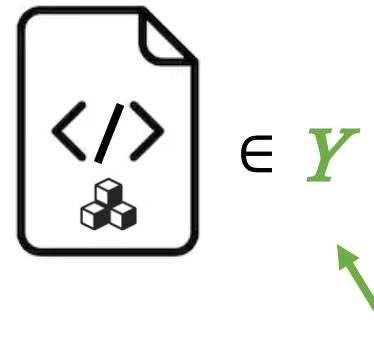
Static typing semantics

GLOBAL_SYSTEM

$$h : X \rightarrow Y$$



$\in X$



$\in Y$

Set of .patl programs which are built with the concept of components top/low level

Set of Rendery programs composed of sequences of Rendery (Swift) instruction

Static typing semantics

COMPONENT_TOPLEVEL

Can be considered as a 4-tuple:

(name , state , tick , render_top)

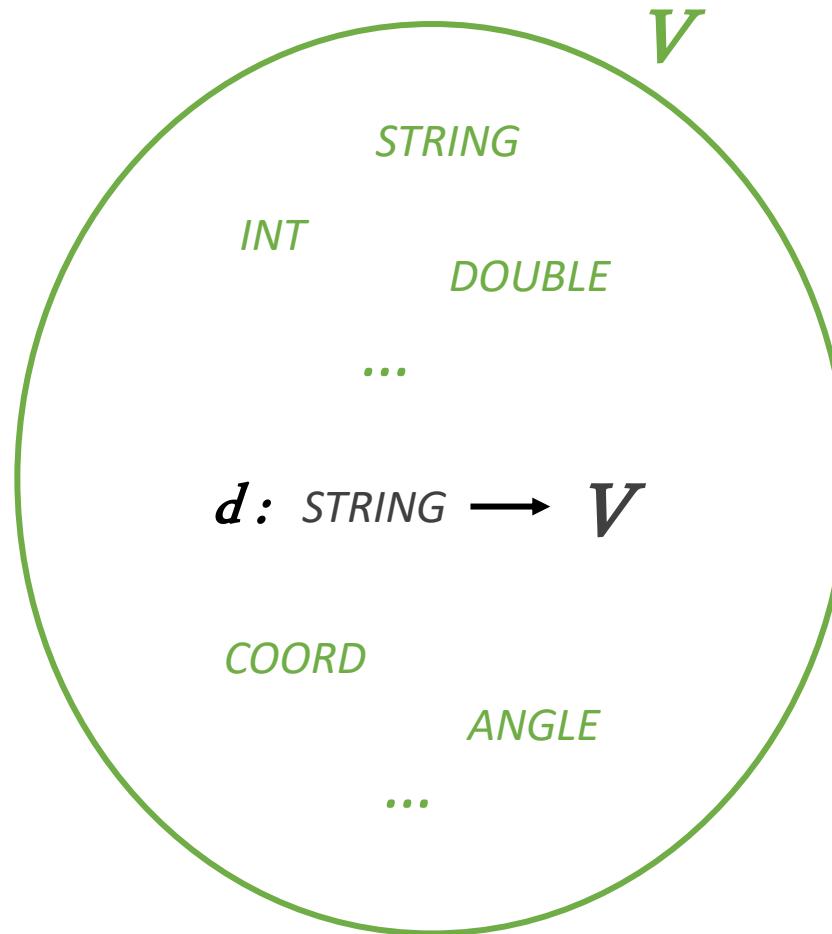
name \in STRING

Static typing semantics

state $\in V$

Where V is a set of some elements:

- Several primitives
- Dictionaries
- Abstract objects



Static typing semantics

`render_top` $\in \textcolor{violet}{RT}$

Where $\textcolor{violet}{RT}$ is a set of partial function : $\textcolor{violet}{rt} : \textcolor{violet}{V} \rightarrow (\textcolor{violet}{STRING} \times \textcolor{violet}{P} \times \textcolor{violet}{RL})^*$

A `COMPONENT_LOWLEVEL` can be considered as a 3-tuple:

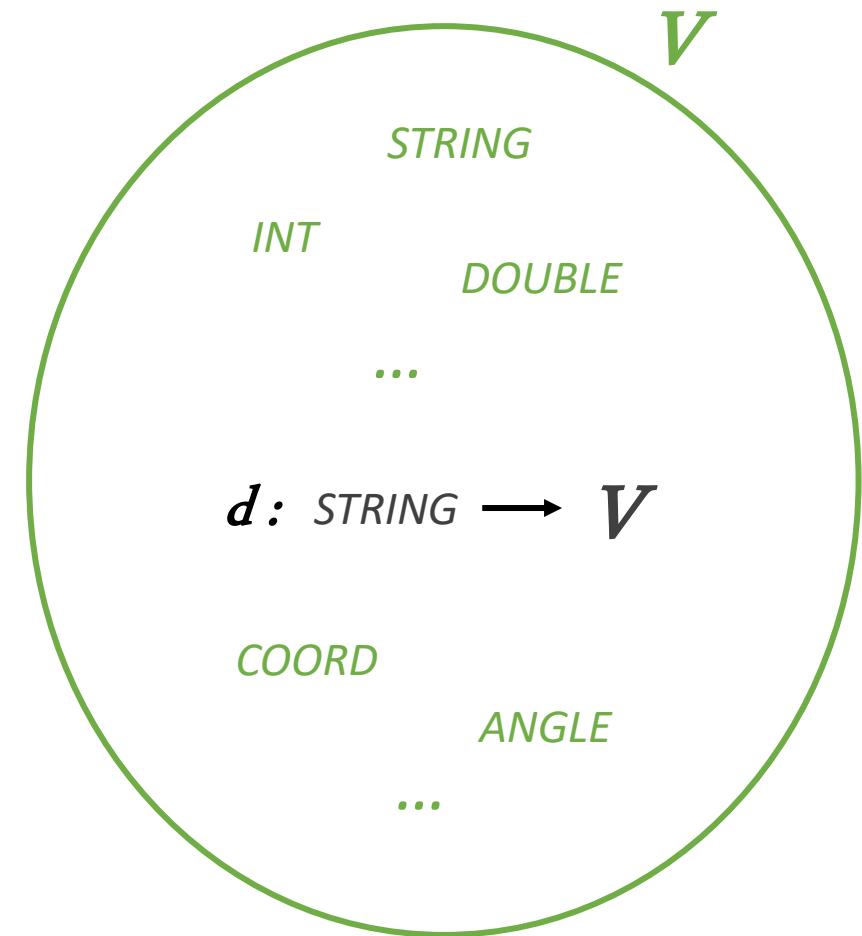
(`name` , `props` , `render_low`)

`name` $\in \textcolor{violet}{STRING}$

Static typing semantics

props $\in P$

Where P is a set of partial function : $p: STRING \rightarrow V$



Static typing semantics

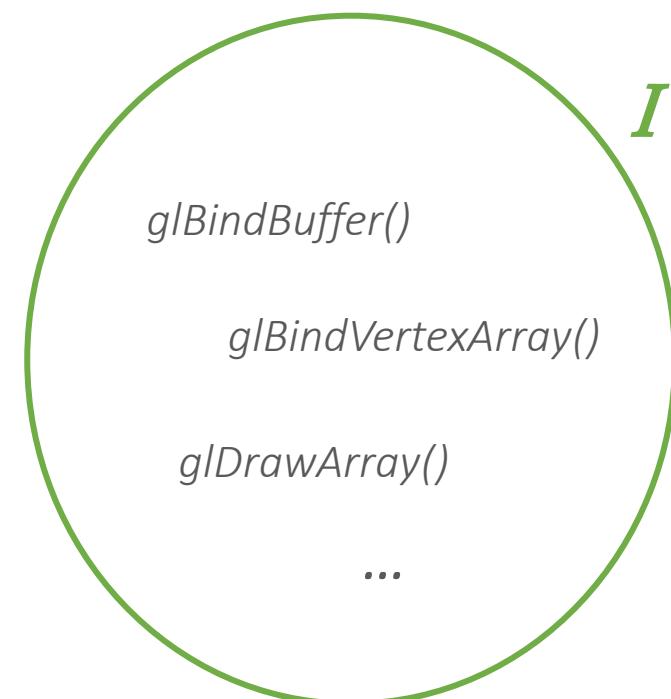
`render_low` $\in \textcolor{violet}{RL}$

Where $\textcolor{violet}{RL}$ is a set of partial function : $\textcolor{violet}{rl}: P \rightarrow ((\textcolor{brown}{STRING} \times P \times \textcolor{violet}{RL}) \cup \textcolor{violet}{M})^*$

`MESH` $\in \textcolor{violet}{M}$

Where $\textcolor{violet}{M}$ is a set of partial function : $\textcolor{violet}{m}: P \rightarrow \textcolor{violet}{I}$

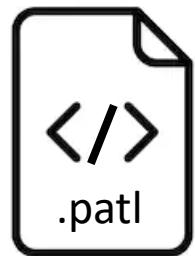
$\textcolor{violet}{I}$ is a set of OpenGL instructions



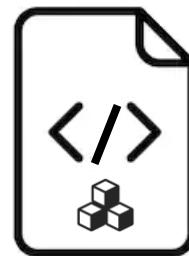
Static typing semantics

GLOBAL_SYSTEM

$$h : X \rightarrow Y$$



$\in X$



$\in Y$

Set of .patl programs which are built with the concept of components top/low level

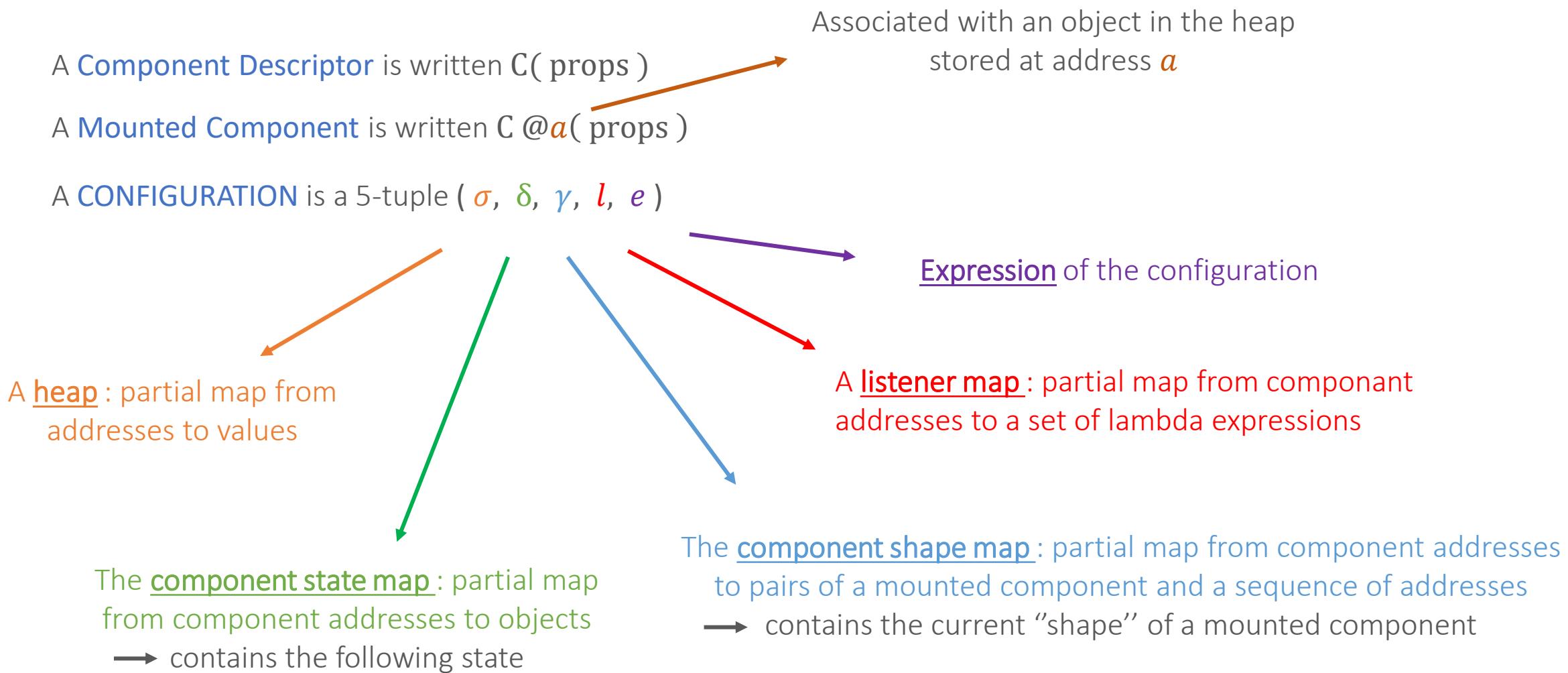


Set of Rendery programs composed of sequences of Rendery (Swift) instruction



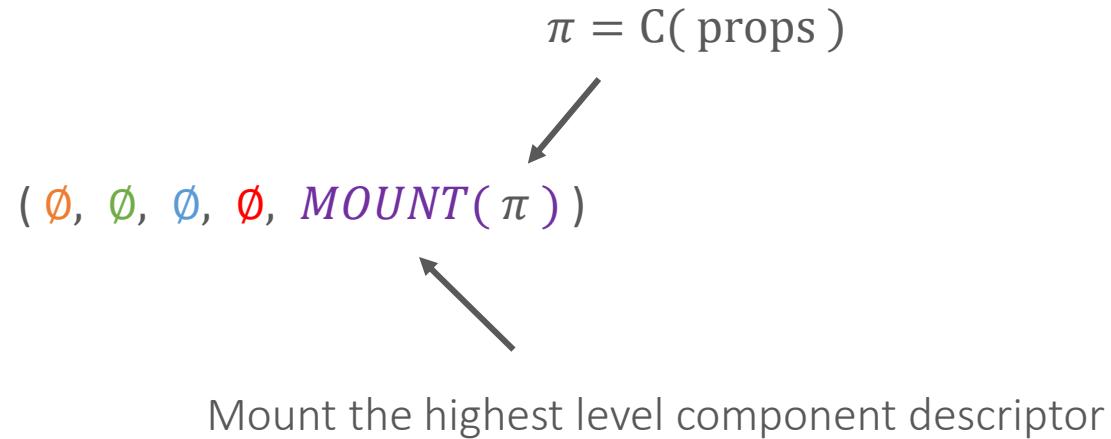
→ Then we send it to an interpreter

Operational semantics



Operational semantics

INITIAL STATE



Operational semantics

MOUNT (Initialization step)

We want to mount component a addressed with π and update the present state map l

$$\begin{array}{ccccc} \swarrow & \searrow & \downarrow & \downarrow & \downarrow \\ \pi = C(\text{props}) & a \notin \text{dom}(\sigma) & \sigma' = \sigma[a \rightarrow \{\text{props}\}] & \delta' = \delta[a] & l' = l[a] \end{array}$$

$$(\sigma, \delta, \gamma, l, MOUNT(\pi)) \longrightarrow (\sigma', \delta', \gamma, l', MOUNTED(C @ a(\text{props}), MOUNTSEQ(RENDER(\pi))))$$

Update the **component shape map** to reflect the current “shape”
of component a and its current subcomponents

Recursively mount its subcomponents

$$\gamma' = \gamma[a \rightarrow (C @ a(\text{props}), \bar{a})]$$

$$(\sigma, \delta, \gamma, l, MOUNTED(C @ a(\text{props}), \bar{a})) \longrightarrow (\sigma, \delta, \gamma', l, a)$$

Operational semantics

UNMOUNT

Subcomponents of a are known from γ

$$\gamma(a) = (\mathcal{C} @a(\text{props}), \bar{a})$$

First, unmount subcomponents of a

$$(\sigma, \delta, \gamma, l, \text{UNMOUNT}(a)) \longrightarrow (\sigma, \delta, \gamma, l, \text{UNMOUNTSEQ}(\bar{a}); \text{UNMOUNTED}(a))$$

Once the subcomponents of a have been unmounted, remove all listener (event)

$$l' = l - a$$

$$(\sigma, \delta, \gamma, l, \text{UNMOUNTED}(a)) \longrightarrow (\sigma, \delta, \gamma, l', \text{NIL})$$

Operational semantics

OBJECT EQUALITY

Compare the values of primitive types by equality

They share the same keys

$\forall k \in \text{keys}(o_1) \ o_1(k) \text{ is primitive} \Rightarrow o_1(k) == o_2(k)$

$$\text{keys}(o_1) = \text{keys}(o_2)$$

References are compared using reference equality

$$o_1 \equiv o_2$$

Operational semantics

STATE MERGES

Take all keys and values from the left object

$$\forall k_i \in \text{keys}(o_1), o_1(k_i) = v_i$$

Take all keys and values from the right object that did not appear in the left one

$$\forall k'_i \in (\text{keys}(o_2) - \text{keys}(o_1)), o_2(k'_i) = v'_i$$

$$o_3 = \{k_1: v_1, \dots, k_n: v_n, k'_1: v'_1, \dots, k'_n: v'_n\}$$

$$\text{MERGE}(o_1, o_2) = o_3$$

Add all keys and value into a new object

Operational semantics

RECONCILIATION (update a mounted component)

First case: a mounted component is replaced by another one

The new component we want to replace the component at the address and descriptor and mounted components are different

$$\pi = C_1(nextprops) \quad \gamma(a) = (C_2 @a(prevprops), _) \quad C_1 \neq C_2$$

$$(\sigma, \delta, \gamma, l, RECONCILE(\pi, a)) \longrightarrow (\sigma, \delta, \gamma, l, UNMOUNT(a); MOUNT(\pi))$$

Mounted component is unmounted & we mount π

Operational semantics

Second case: update a mounted component with news props & state

The same component with the
news props we want

Update the component at the address a

Get the next state of the component

$$\pi = C(nextprops)$$

$$o = \sigma(a)$$

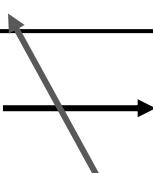
$$\gamma(a) = (C @a(props), \bar{a})$$

$$o' = o[props \rightarrow nextprops] [state \rightarrow nextstate]$$

$$nextstate = \delta(a)$$

$$\sigma' = \sigma[a \rightarrow o']$$

$$(\sigma, \delta, \gamma, l, RECONCILE(\pi, a))$$



$$(\sigma', \delta, \gamma, l, RECONCILED(C @a(nextprops), RECONCILESEQ(RERENDER(a, \bar{a}))))$$

Object o represents the component object

Update the props & state fields

Update σ at the address a

$$\gamma' = \gamma[a \rightarrow (C @a(props), \bar{a})]$$

Re render a which gives us $\bar{\pi}$ and apply
 $RECONCILE$ on the subcomponents

$$(\sigma, \delta, \gamma, l, RECONCILED(C @a(props), \bar{a})) \longrightarrow (\sigma, \delta, \gamma', l, a)$$

Update the component shape map to reflect
the current "shape" of component a
and its current subcomponents

Operational semantics

RENDERS

We want to render π

Invoke the render method of the component

$$\pi = C(\text{props}) \quad (\sigma, \delta, \gamma, l, \text{render}()) \rightarrow (\sigma, \delta, \gamma, l, \bar{\pi})$$

$$(\sigma, \delta, \gamma, l, \text{RENDER}(\pi)) \longrightarrow (\sigma, \delta, \gamma, l, \bar{\pi})$$

RERENDER (for a mounted component)

Return the sequence of remaining components

Use the address a to retrieve the component from γ

$$\gamma(a) = (C @ a (\text{props}), _) \quad (\sigma, \delta, \gamma, l, \text{render}()) \rightarrow (\sigma, \delta, \gamma, l, \bar{\pi})$$

$$(\sigma, \delta, \gamma, l, \text{RERENDER}(a)) \longrightarrow (\sigma, \delta, \gamma, l, \bar{\pi})$$

Operational semantics

UPDATERSTATE

Use the address a to retrieve the component from state of the component

$$\gamma(a) = (\text{C} @ a (\text{props}), \bar{a})$$

$$nextstate = \delta(a)$$

$$\delta' = \delta[a \rightarrow \text{MERGE}(newstate, nextstate)]$$

$(\sigma, \delta, \gamma, l, \text{UPDATERSTATE}(a, newstate)) \rightarrow (\sigma, \delta', \gamma, l, \text{RECONCILED}(a, \text{RECONCILESEQ}(\text{RENDER}(a), \bar{a})))$

The $newstate$ object is passed to the component

Changing the state of a component could change what is returned by its render method

Call OpenGL from Swift (Linux)

An OpenGL function loader in pure Swift

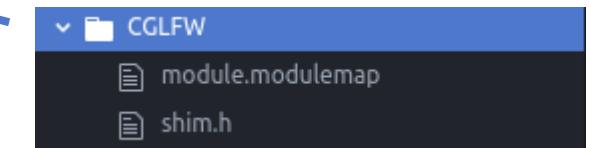
It can load any OpenGL function up to OpenGL 4.5

```
import PackageDescription

let package = Package(
    name: "test_OpenGL_swift_linux_SPM",
    products: [
        .executable(name: "UsageExample", targets: ["UsageExample"]),
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        // .package(url: /* package url */, from: "1.0.0"),
        .package(url: "https://github.com/apple/swift-numerics", from: "0.0.5"),
        .package(name: "GL", url: "https://github.com/kelvin13/swift-opengl.git", .branch("master"))
    ],
    targets: [
        // Targets are the basic building blocks of a package. A target can define a module or a test suite.
        // Targets can depend on other targets in this package, and on products in packages this package depends on.
        .target(name: "UsageExample", dependencies: ["test_OpenGL_swift_linux_SPM"]),
        .target(name: "test_OpenGL_swift_linux_SPM", dependencies: ["GL", "CGlad", "CGLFW", "CFreeType"]),
        .target(name: "CGlad", dependencies: []),

        .systemLibrary(name: "CFreeType", pkgConfig: "freetype2"),
        .systemLibrary(name: "CGLFW", pkgConfig: "glfw3"),

        .testTarget(
            name: "test_OpenGL_swift_linux_SPMTests",
            dependencies: ["test_OpenGL_swift_linux_SPM"]),
    ]
)
```



package.swift

Import the OpenGL function loader & the other important libraries



```
import GL
import CGLFW
import CFreeType
import Foundation
```

```
// Initialize GLFW
if(glfwInit() == 0) {
    print("Failed to initialize GLFW! I'm out!")
    exit(-1)
}
```

```
guard let window = glfwCreateWindow(600, 600, "OpenGL test - Swift", nil, nil)
else {
    print("Failed to open a window! I'm out!")
    glfwTerminate()
    exit(-1)
}

// Set the window context current
glfwMakeContextCurrent(window)
```

Open a window & attach an OpenGL context to the window surface



Red background



Draw a yellow square

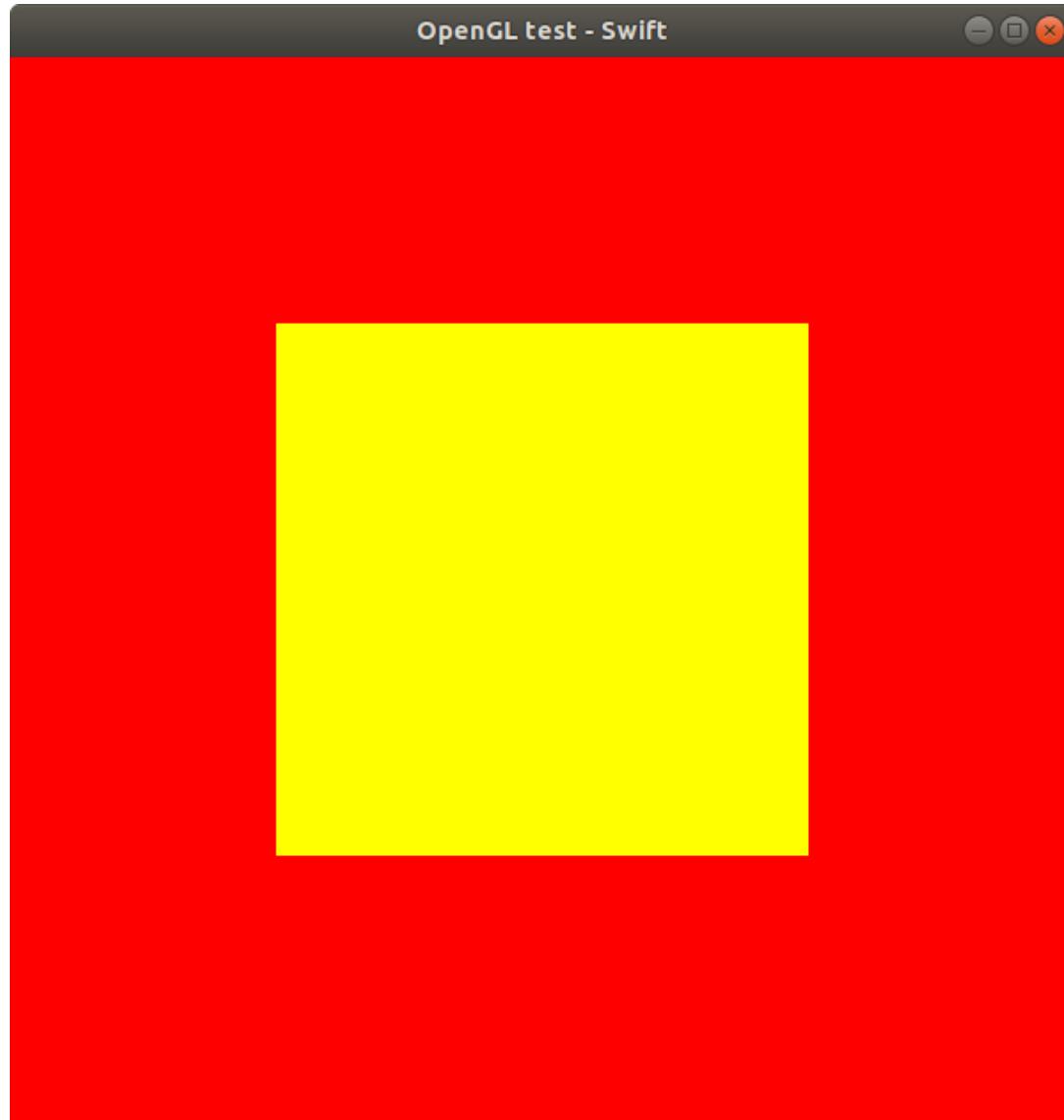
```
// Use red to clear the screen
glClearColor(1, 0, 0, 1)
while (glfwWindowShouldClose(window) == 0) {
    // Clear the screen (window background)
    glClear(UInt32(GL_COLOR_BUFFER_BIT))

    // Draw a square using the (deprecated) fixed pipeline functionality
    glColor3f(1.0, 1.0, 0.0)
    glBegin(GL_QUADS)
    glVertex2f(-0.5, -0.5)
    glVertex2f(0.5, -0.5)
    glVertex2f(0.5, 0.5)
    glVertex2f(-0.5, 0.5)
    glEnd()
    // Swap front and back buffers for the current window
    glfwSwapBuffers(window)
    // Poll for events
    glfwPollEvents()
}

// Destroy the window and its context
glfwDestroyWindow(window)

// Terminate GLFW
glfwTerminate()
```

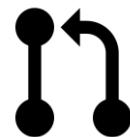
Swift build
Swift run



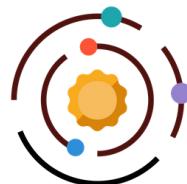
Rendery on Linux



Rendery is now operational on Linux

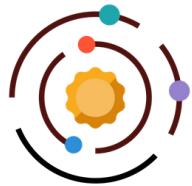


I sent a pull request



Then, the idea was to work on the example of the solar system

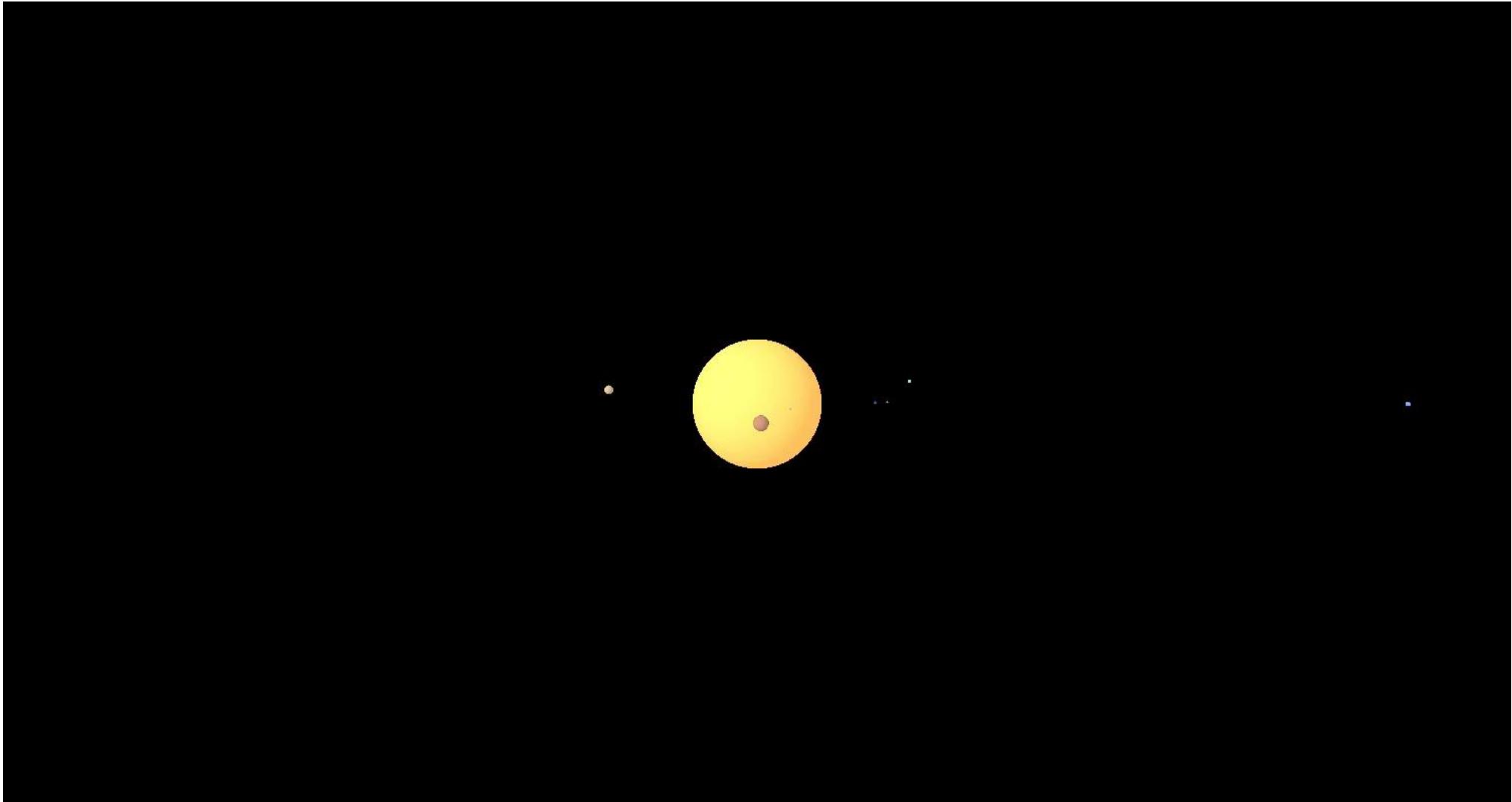
How it works



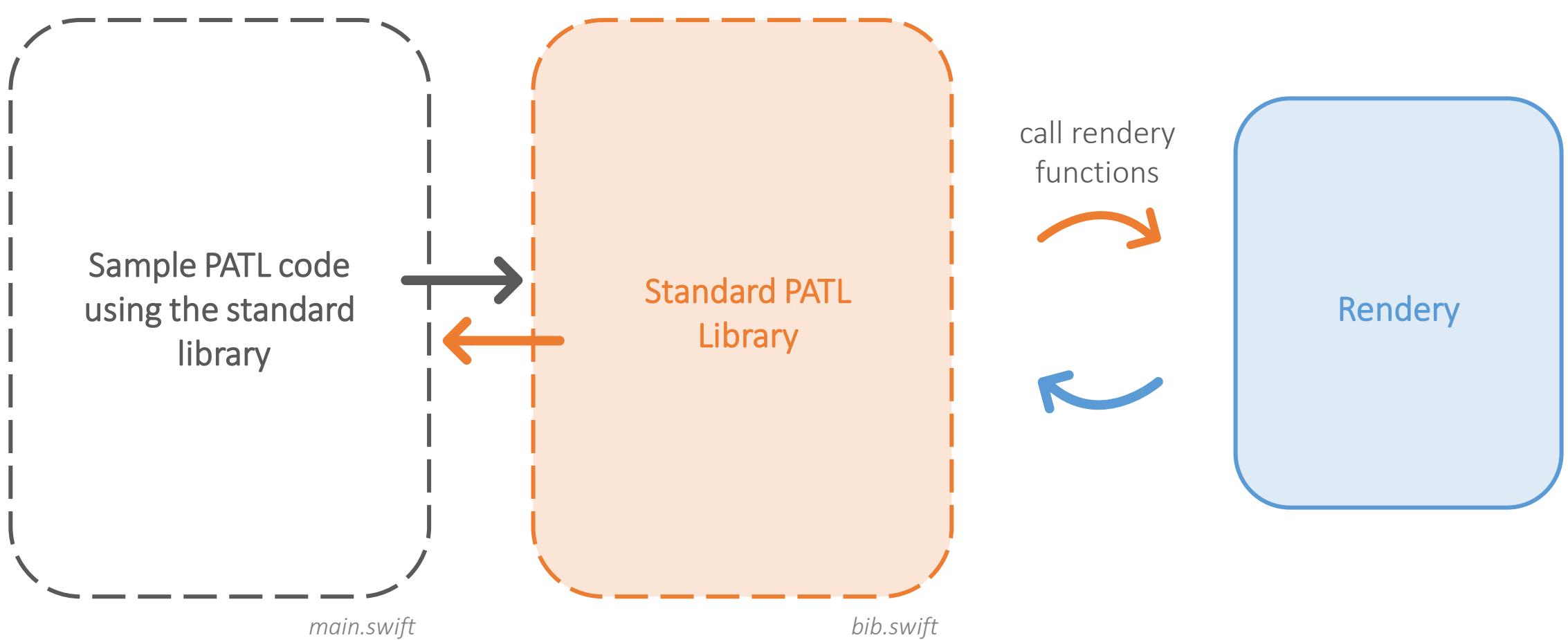
Create a context and attach a window to it

- └ Create a scene where we will place our 3D objects
 - └ Create child nodes that will define our objects
 - └ Define a camera that will be the point of view of the users
 - └ ...
 - └ Create spheres representing our planets
 - └ Update the position of the planets

Solar system with Rendery



Principal idea of PATL



Example with the solar system

```
class SystemSolar: ComponentTopLevel {  
  
    var state: [String: [String: Any]] = [:]  
    let solarRadius: Double = 50.0  
  
    override init() {  
        super.init()  
        // Define the background color as "#000000"  
        self.setBackgroundColor(colHexa: "#000000")  
        // Define the global state of the application: this is a dictionary of dictionaries.  
        // Each object in the scene has some properties.  
        // Here, each planet has a name, a radius, coordinates (polar: which are represented by the distance from the sun and the polar angle),  
        // a revolution and finally an image representing its texture.  
        self.state = ["root": ["name": "Solar",  
                            "radius": solarRadius,  
                            "coord": Coord(polar: (0.0, Angles(deg: 0.0), Angles(deg: 0.0))),  
                            "tex": "/img/solar_tex.jpg"],  
                    "obj0": ["name": "Mercury",  
                            "radius": self.sizePlanet(s: solarRadius, p: 11440),  
                            "coord": Coord(polar: (self.distPlanet(s: solarRadius, d: 35.791), Angles(deg: 0.0), Angles(deg: 0.0))),  
                            "revo": 88.0,  
                            "tex": "/img/mercury_tex.jpeg"],  
                    ...  
                    self.render()  
    }  
}
```

```
// Build the Scene related to ComponentTopLevel  
open class ComponentTopLevel: Scene {  
    public override init() {  
        // Set the background color of the scene  
        public func setBackgroundColor(colHexa: Color) -> Color {  
            backgroundColor = colHexa  
            return backgroundColor!  
        }  
        cameraNode.camera?.farDistance = 5000.0  
        cameraNode.translation = Vector3(x: 650.0, y: 100, z: 100)  
        cameraNode.constraints.append(LookAtConstraint(target: root))  
    }  
}
```

bib.swift

main.swift

163

```
public struct Coord {  
  
    public var polar: (Double, Angles, Angles) = (0.0, Angles(deg: 0.0), Angles(deg: 0.0))  
    public var cart: (Double, Double, Double) = (0.0, 0.0, 0.0)  
  
    public init(polar: (Double, Angles, Angles)) {  
        self.polar = polar  
    }  
  
    public init(cart: (Double, Double, Double)) {  
        self.cart = cart  
    }  
  
    public func polarToCart() -> (Double, Double, Double) {  
        let x = polar.0 * sin(polar.1.deg) * cos(polar.2.deg)  
        let y = polar.0 * sin(polar.1.deg) * sin(polar.2.deg)  
        let z = polar.0 * cos(polar.1.deg)  
        return (x,y,z)  
    }  
}
```

bib.swift

```
func render() {
    let solar = Sphere(name: (self.state["root"]!["name"] as! String),
                       scene: self,
                       coord: (self.state["root"]!["coord"] as! Coord),
                       props: [(self.state["root"]!["tex"] as! String), (self.state["root"]!["radius"] as! Double)]
                           ).render()
    let s0 = Sphere(name: (self.state["obj0"]!["name"] as! String),
                    scene: self,
                    coord: (self.state["obj0"]!["coord"] as! Coord),
                    props: [(self.state["obj0"]!["tex"] as! String), (self.state["obj0"]!["radius"] as! Double)]
                        ).render() // Call the low level render function

    ...
    self.tick(node: [s0 as! Node, s1 as! Node, s2 as! Node, s3 as! Node,
                    s4 as! Node, s5 as! Node, s6 as! Node, s7 as! Node])
}
```

main.swift

```

// Apply the "tex" texture to the node
public func applyTexture(node: Node, tex: String) {
    node.model?.materials[0].diffuse = .texture(ImageTexture(image:
        Image(contentsOfFile: "Sources"+tex)!, wrapMethod: .repeat))
}

// Render function is called the first time
func render() -> Any {
    // createChildNode() allows to create a child node in the scene (root)
    let n = scene.createChildNode(name: self.name!)
    // Call createSphere() to create a sphere in the current scene
    scene.createSphere(node: n, segments: 100, rings: 100, radius: self.props![1] as! Double)
    // Apply a texture from an image
    scene.applyTexture(node: n, tex: self.props![0] as! String)

    let x = coord.polarToCart().0
    let y = coord.polarToCart().1
    let z = coord.polarToCart().2
    scene.setNodePosition(node: n, x: x, y: y, z: z)
    return n
}

// Rerender function is called for a mounted camera
func rerender() {
    let x = coord.polarToCart().0
    let y = coord.polarToCart().1
    let z = coord.polarToCart().2
    print(x,y,z)
    scene.setNodePosition(node: node!, x: x, y: y, z: z)
}

```

bib.swift

```

// Define a node as sphere with some properties: (segments, rings, radius)
public func createSphere(node: Node, segments: Int, rings: Int, radius: Double) {
    node.model = Model(
        meshes: [.sphere(segments: segments, rings: rings, radius: radius)],
        // Set the position of a node
        public func setNodePosition(node: Node, x: Double, y: Double, z: Double) {
            node.translation.x = x
            node.translation.y = y
            public func setNodePosition(node: Node, x: Double, y: Double, z: Double) {
                node.translation.z = z
            }
            meshes: [.box(.init(centeredAt: .zero, dimensions: .unitScale))],
            materials: [Material()])
}

```

bib.swift

main.swift

```

func tick(node: [Node]) {
    ApplicationContext.shared.subscribe(frameListener: { _, delta in
        self.state = updateState(
            currentState: self.state,
            nextState: ["obj0": ["coord": Coord(polar: ((self.state["obj0"]!["coord"] as! Coord).polar.θ,
                Angles(deg: self.updatePlanetPos(currentAngle: (self.state["obj0"]!["coord"] as! Coord).polar.1,
                    revolution: (self.state["obj0"]!["revo"] as! Double))),
                Angles(deg: (self.state["obj0"]!["coord"] as! Coord).polar.2.deg)))),
            "obj1": ["coord": Coord(polar: ((self.state["obj1"]!["coord"] as! Coord).polar.θ,
                Angles(deg: self.updatePlanetPos(currentAngle: (self.state["obj1"]!["coord"] as! Coord).polar.1,
                    revolution: (self.state["obj1"]!["revo"] as! Double))),
                Angles(deg: (self.state["obj1"]!["coord"] as! Coord).polar.2.deg)))),
            ...
        ]
    )
    self.rerender(newState: self.state, node: node)
})
}

// Predefined function to update the global state of the application
public func updateState(currentState: [String: [String: Any]], nextState: [String: [String: Any]]) -> [String: [String: Any]] {
    var tmpState = currentState
    for (outerKey, _) in nextState {
        for (innerKey, _) in nextState[outerKey]! {
            tmpState[outerKey]![innerKey] = nextState[outerKey]![innerKey]
        }
    }
    return tmpState
}

func rerender(newState: [String: [String: Any]], node: [Node]) {
    Sphere(node: node[0],
        scene: self,
        coord: (self.state["obj0"]!["coord"] as! Coord)
    ).rerender() // Call the low level rerender function
    ...
}

```

main.swift

167

bib.swift

```
// Entry: Call the highest component
let _ = createScene(name: "Solar System", compTL: SystemSolar())
```

main.swift

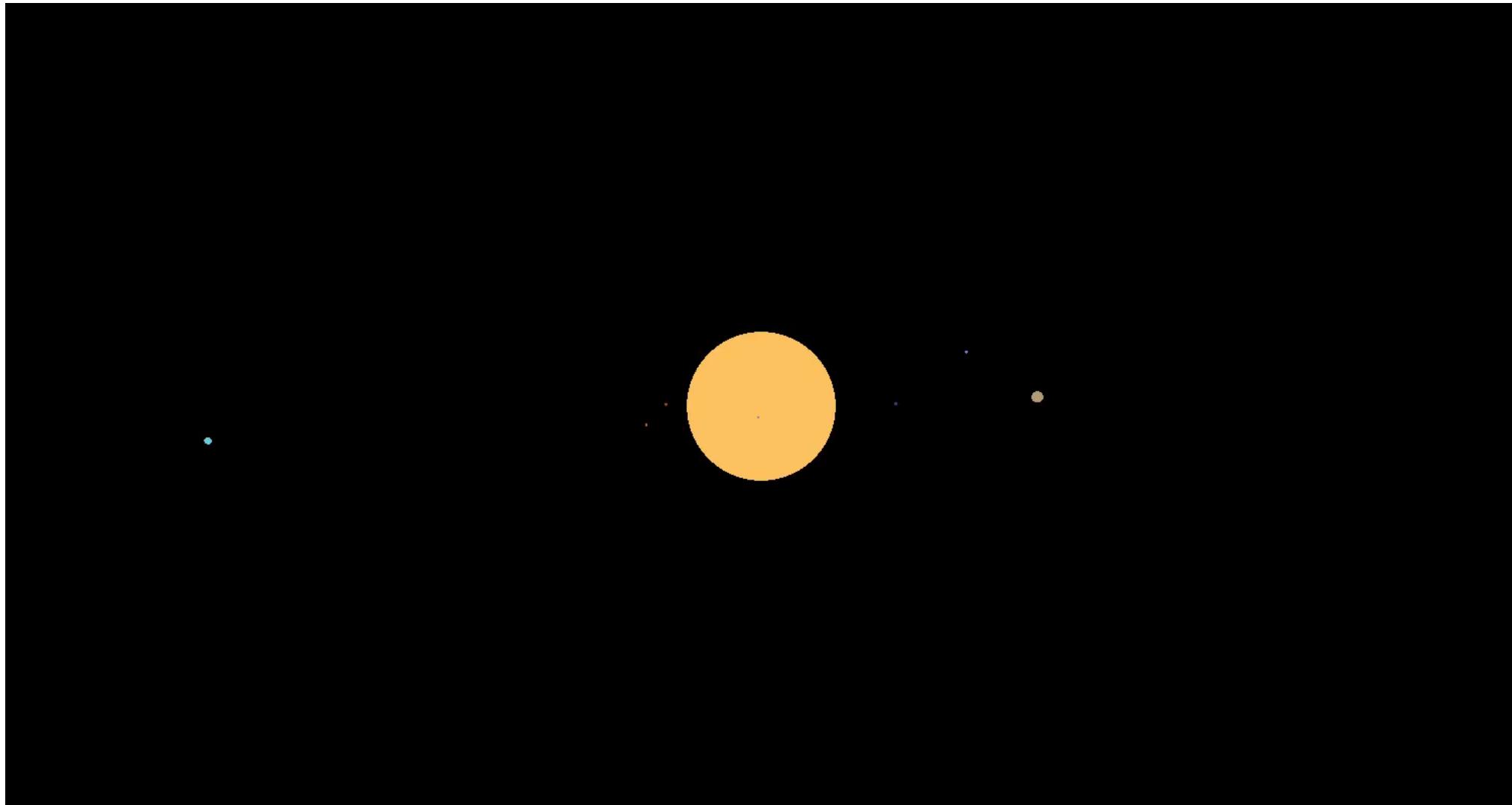
```
public func createScene(name: String, compTL: Scene) {
    // Initialize Rendery's engine.
    guard let window = ApplicationContext.shared.initialize(width: 1500, height: 800, title: name)
        else { fatalError() }
    //defer { ApplicationContext.shared.clear() } #TODO: bug under linux

    // Create the game scene and present it in the window's viewport.
    let scene = compTL
    window.viewports.first?.present(scene: scene)

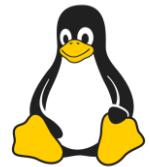
    // Run the rendering loop.
    ApplicationContext.shared.targetFrameRate = 60
    ApplicationContext.shared.render()
}
```

bib.swift

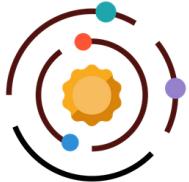
Solar system with PATL



Work done



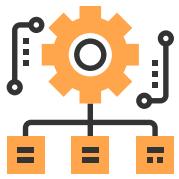
Successfully run Rendery on Linux



Implement the solar system example with Rendery



Advancement



Update and creation of new structures for given abstract types

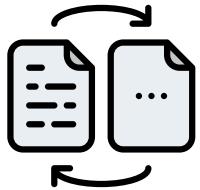


Allow to use those of Rendery



Debugging and coding new Rendery features (PATL library)

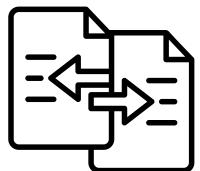
Shader part



Convert shaders with Swift syntax to GLSL shaders
→ Using a parser, etc.



See the side of [SpriteKit](#) & [SKShader](#)
→ Framework to create 2D games
→ Object class for applying custom fragment shaders

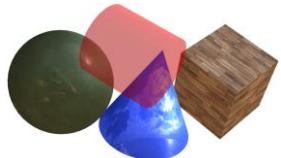


Simply write shaders in GLSL and pass them to Rendery

Work in progress



Implement the PATL to Rendery translation system
(PATL standard library)



Integrate the possibility of creating custom shaders
with PATL and translate them for 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://developer.apple.com/metal>

<https://github.com/RenderyEngine/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/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/lbasi-part7/>

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

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

References (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-gsl.html>

References (new links)

<https://cs.au.dk/~magnusm/papers/ecoop20/paper.pdf>

https://en.wikipedia.org/wiki/Partial_function

<https://solarianprogrammer.com/2016/11/19/swift-opengl-linux-macos-glfw/>

<https://engineeringbydoing.com/article/swift-opengl-application>

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. Martínez 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>

Add react semantics

Working with shaders

Patrick SARDINHA