POLITECNICO
MILANO 1863

Introduction to GLSL ES

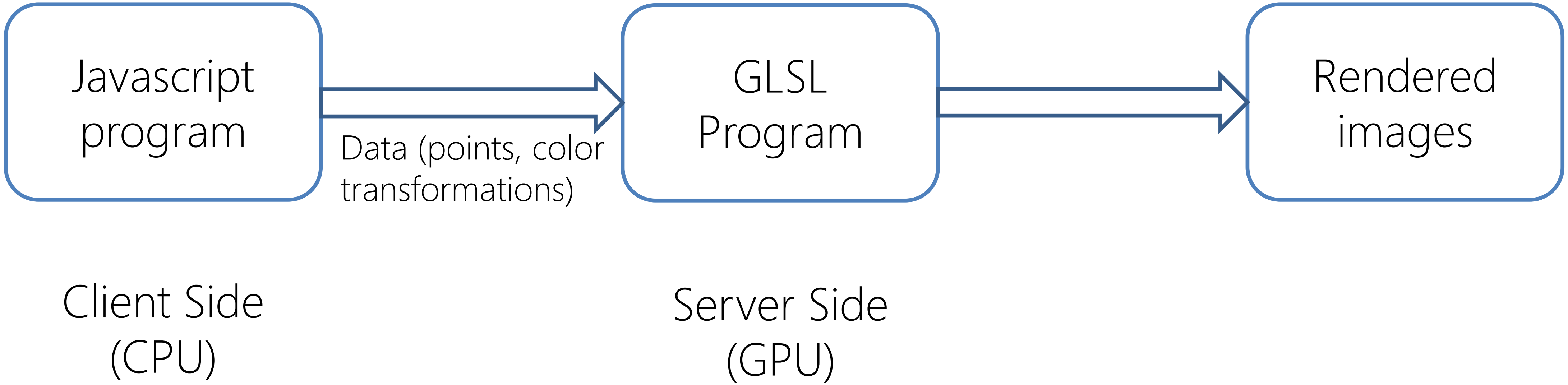
Computer Graphics 2021

Erica Stella (erica.stella@polimi.it)

WebGL pipeline

GLSL program (similar to C/C++) defines how the GPU handles the data

Follows the specification of OpenGL ES Shading Language (only slightly different from the OpenGL Shading Language)



Javascript program

Data (points, color transformations)

GLSL Program

Rendered images

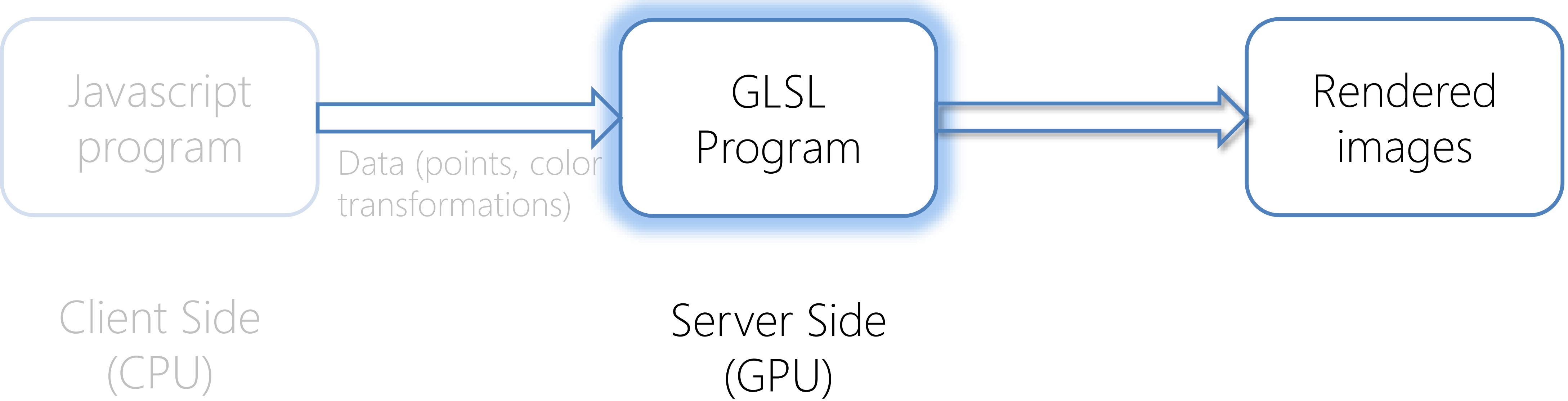
Client Side (CPU)

Server Side (GPU)

GLSL ES Program

GLSL ES Program = Vertex Shader + Fragment Shader

- Vertex Shader-> defines the position of the 3D primitives in the webGL space coordinates and other vertex attributes
- Fragment Shader-> defines the color of the pixels where privitives are rendered



Javascript program

Data (points, color transformations)

GLSL Program

Rendered images

Client Side (CPU)

Server Side (GPU)

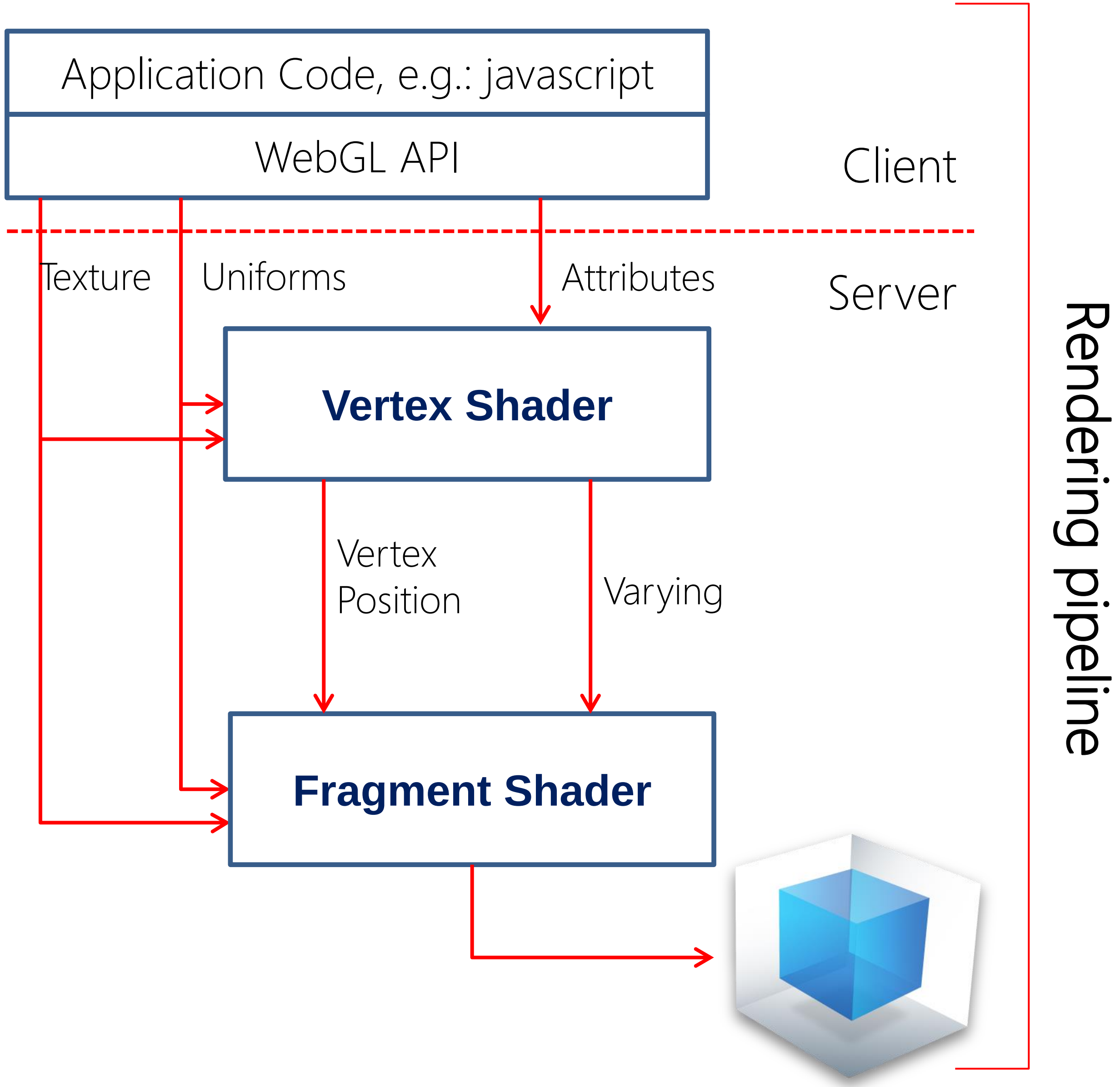
Rendering pipeline

The sequence of steps that go from the 3D scene's definition to the final 2D output image is called **Rendering Pipeline**

Shaders are programs executed by the graphics hardware to

- Process vertices
- Define how to rasterize the primitives
- Define the colour of the pixels

To write the shaders, WebGL uses the **GLSL ES Shading Language**



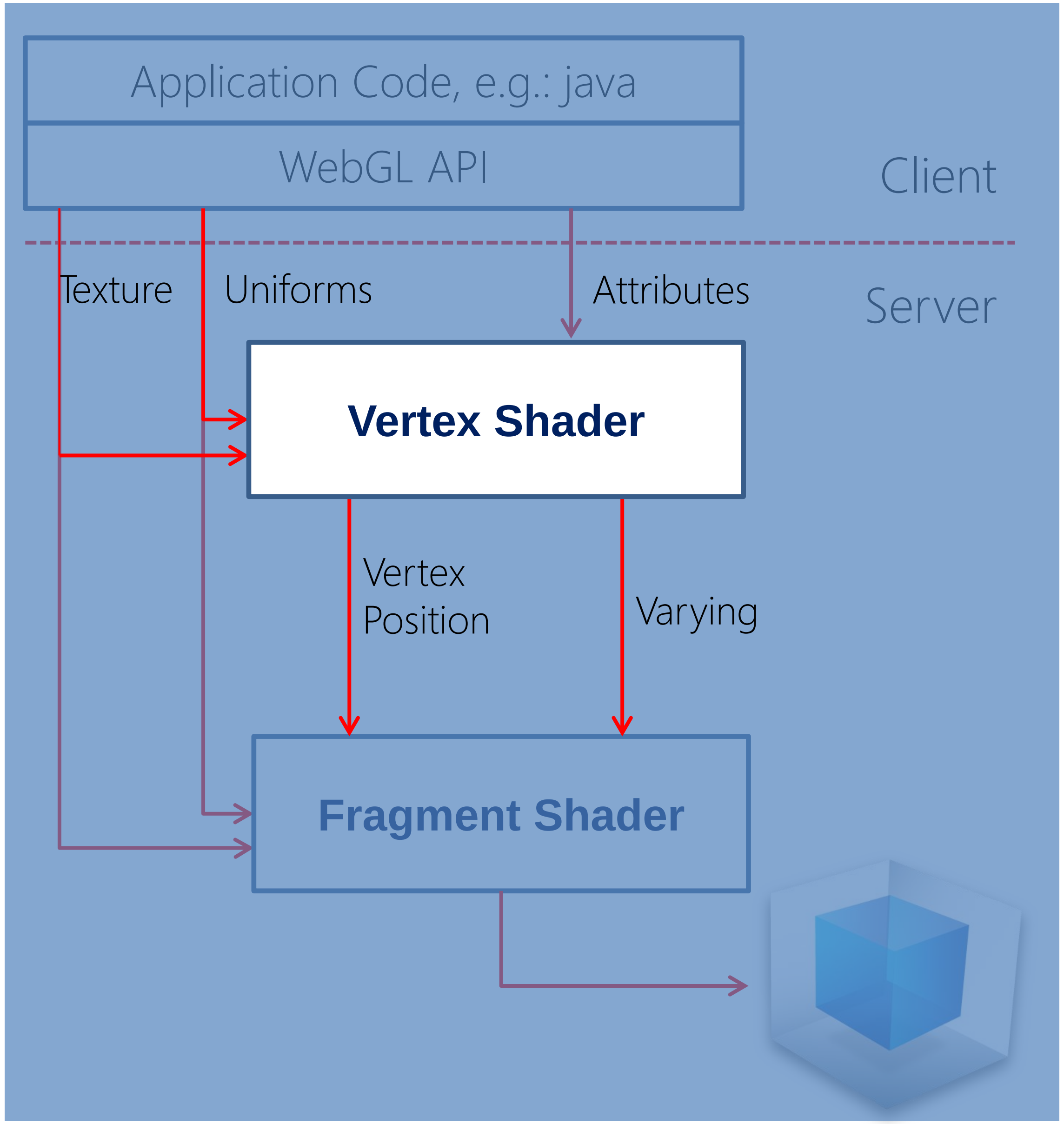
Vertex Shader

Vertex Shaders contains source code for the operations that are meant to occur on **each** vertex that is processed

Typical per-vertex operations include:

- World-View-Projection computations
- Vertex colour computation
- Light colour when using the Gouraud method.

- **Note:** To render a triangle with three vertices, the vertex shader is executed three times, once for each vertex.



Rendering Pipeline Step by Step – Vertex Shader

CPU (WebGL in our case)

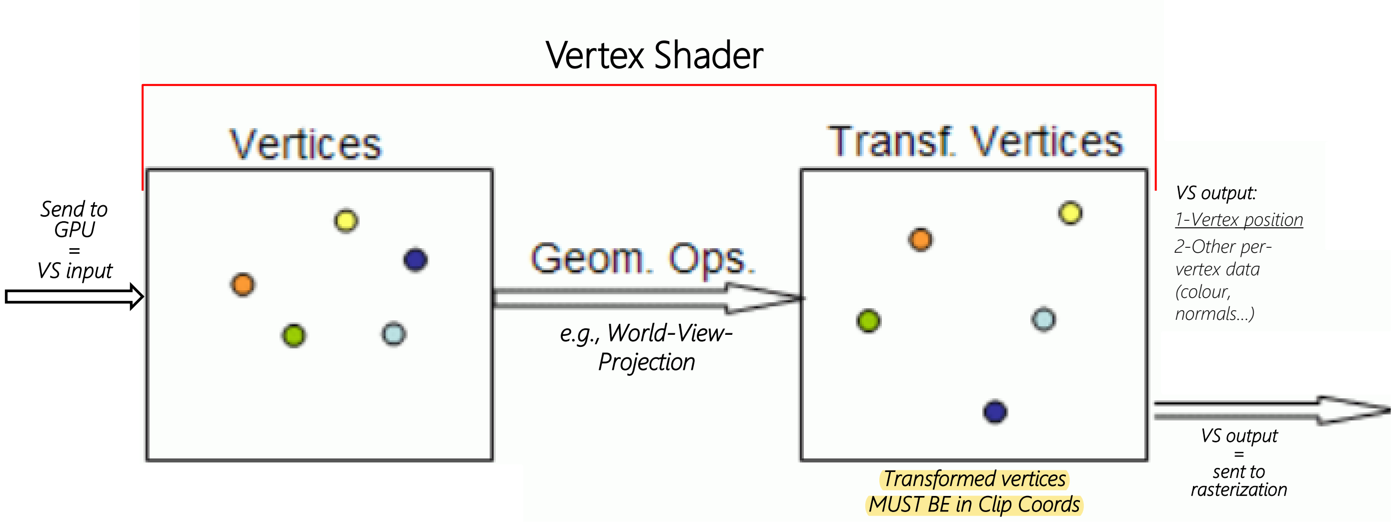
Vertices:

V_1 $(x_1,y_1,z_1,n_{x1},n_{y1},n_{z1},r,g,b\ldots)$

V_2 $(x_2,y_2,z_2,n_{x2},n_{y2},n_{z2},r,g,b\ldots)$

V_3 $(x_3,y_3,z_3,n_{x3},n_{y3},n_{z3},r,g,b\ldots)$

…



Send to GPU = VS input

Vertex Shader

e.g., World-View-Projection

VS output:

1-Vertex position

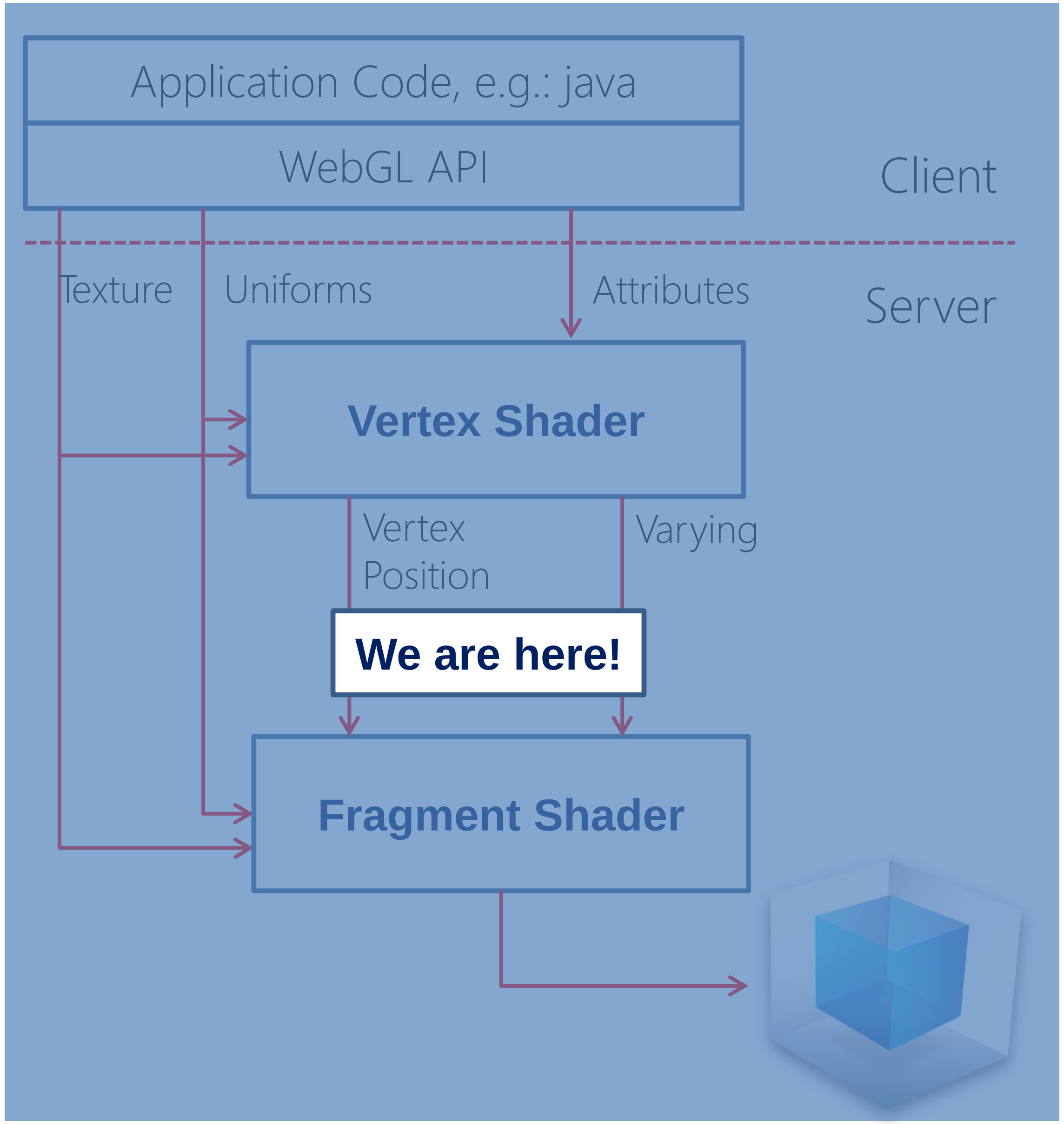
2-Other per-vertex data (colour, normals…)

Transformed vertices MUST BE in Clip Coords

VS output = sent to rasterization

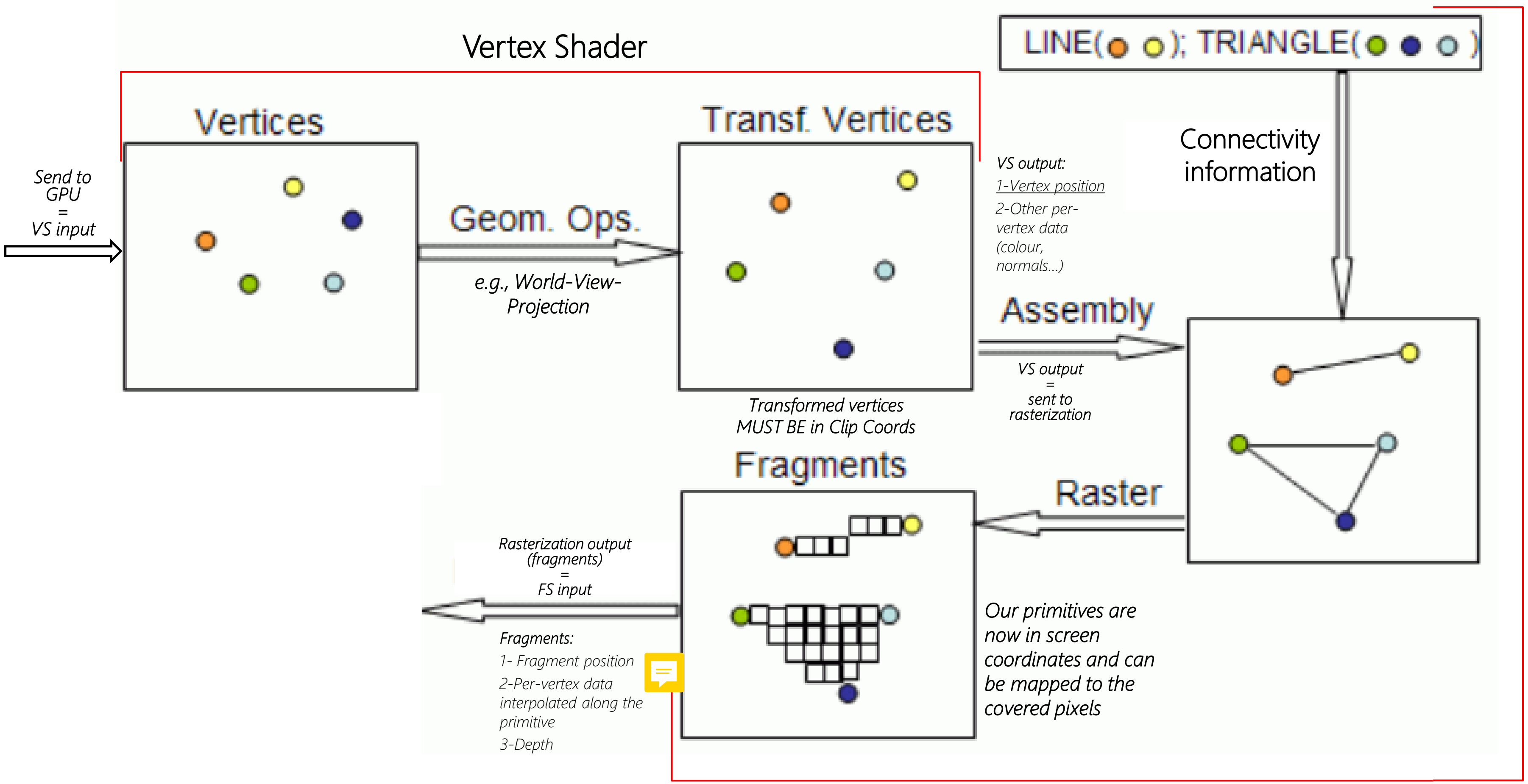
After the Vertex Shader

- Vertices are assembled into primitives according to the **mode** argument of the drawing command (TRIANGLE,POINTS,LINES,LINE_STRIP)
- **Primitive Clipping**, **Perspective Divide** (to Normalized Device Coordinates), and **Viewport Transform** (transparent to the client)
- **Rasterization:** the primitive is converted to a 2d image. Each point (fragment) of this image contains such information as **color** and **depth**.



Rendering Pipeline Step by Step – After the Vertex Shader

CPU (WebGL in our case)
Vertices:
V_1 (x_1,y_1,z_1,n_{x1},n_{y1},n_{z1},r,g,b...)
V_2 (x_2,y_2,z_2,n_{x2},n_{y2},n_{z2},r,g,b...)
V_3 (x_3,y_3,z_3,n_{x3},n_{y3},n_{z3},r,g,b...)
...



Send to GPU = VS input

Vertex Shader

e.g., World-View-Projection

Transformed vertices MUST BE in Clip Coords

VS output:
1-Vertex position
2-Other per-vertex data (colour, normals...)

Connectivity information

VS output = sent to rasterization

Our primitives are now in screen coordinates and can be mapped to the covered pixels

Rasterization output (fragments) = FS input

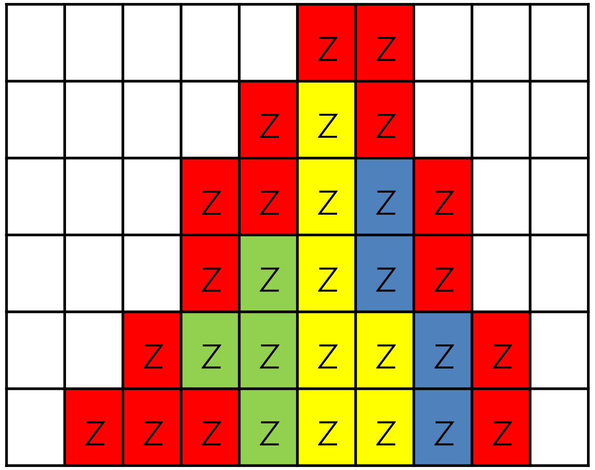
Fragments:
1- Fragment position
2-Per-vertex data interpolated along the primitive
3-Depth

Automatically done by the GPU after the VS and before the FS

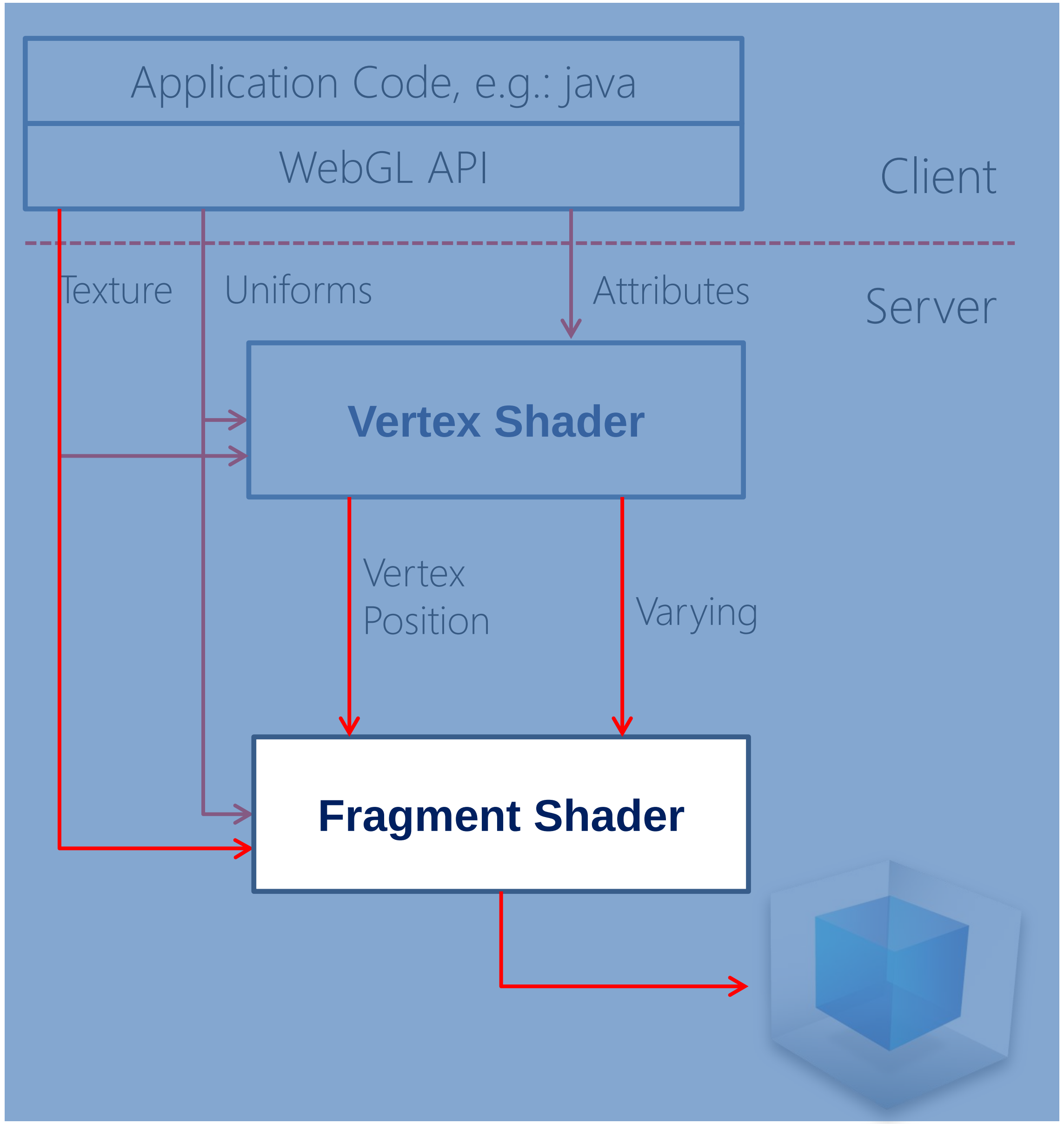
Fragment Shaders

The **Fragment Shader** contains source code for the operations that are meant to occur on each fragment that results from vertex shader rasterization.

A **fragment** is one square of the 2d image grid along with its parameters of depth and other data (e.g. color data)



					🟥 Z	🟥 Z			
				🟥 Z	🟨 Z	🟥 Z			
			🟥 Z	🟥 Z	🟨 Z	🟦 Z	🟥 Z		
			🟥 Z	🟩 Z	🟨 Z	🟦 Z	🟥 Z		
		🟥 Z	🟩 Z	🟩 Z	🟨 Z	🟨 Z	🟦 Z	🟥 Z	
	🟥 Z	🟥 Z	🟥 Z	🟩 Z	🟨 Z	🟨 Z	🟦 Z	🟥 Z	



Fragment Shaders

The **Fragment Shader** contains source code for the operations applied on each fragment that results from rasterization.

Fragment Shader **operations**:

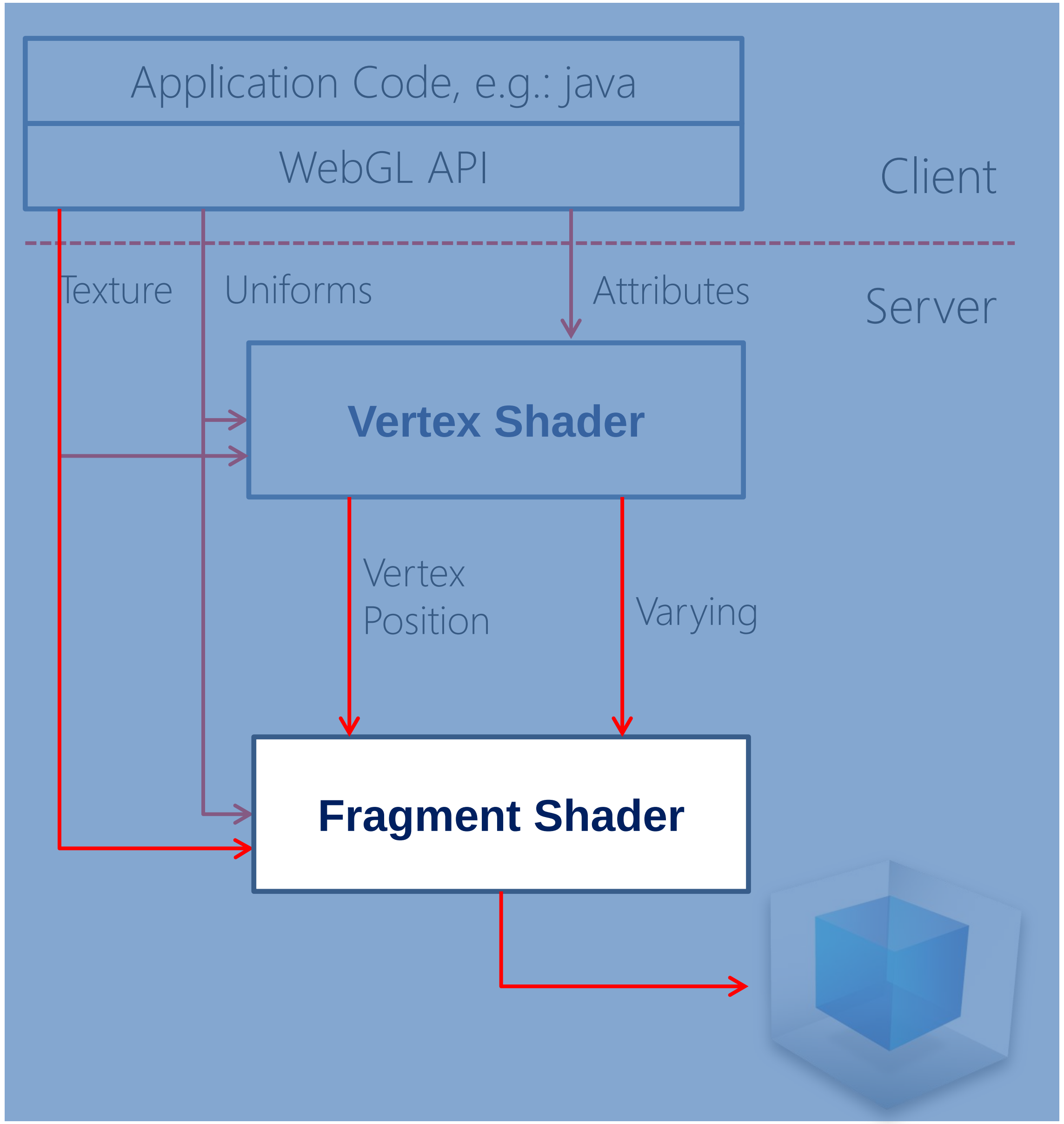
Anti-aliasing
Depth-test
Color blending
Dithering

If *enabled* in the client code, performed automatically.

Texturing
Colors and Light computation in Phong model

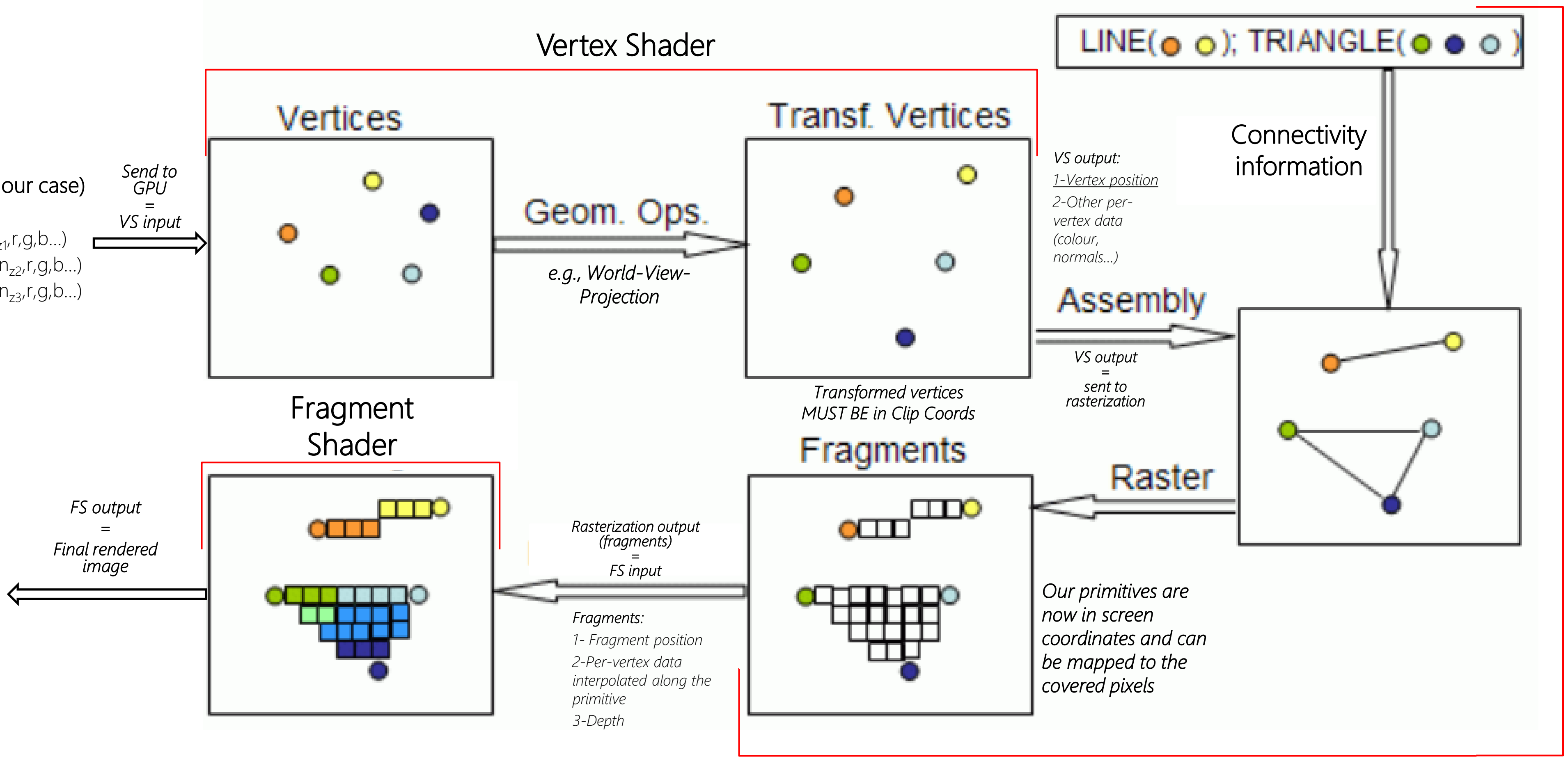
Must be explicitly programmed.

At the end of the pipeline the image is complete and ready to be displayed.



Rendering Pipeline Step by Step – Fragment Shader

CPU (WebGL in our case)
Vertices:
V_1 (x_1,y_1,z_1,n_{x1},n_{y1},n_{z1},r,g,b…)
V_2 (x_2,y_2,z_2,n_{x2},n_{y2},n_{z2},r,g,b…)
V_3 (x_3,y_3,z_3,n_{x3},n_{y3},n_{z3},r,g,b…)
…



Send to GPU
=
VS input

Vertex Shader

e.g., World-View-Projection

VS output:
1-Vertex position
2-Other per-vertex data (colour, normals…)

Connectivity information

VS output
=
sent to rasterization

Transformed vertices MUST BE in Clip Coords

Fragment Shader

Our primitives are now in screen coordinates and can be mapped to the covered pixels

Rasterization output (fragments)
=
FS input

Fragments:
1- Fragment position
2-Per-vertex data interpolated along the primitive
3-Depth

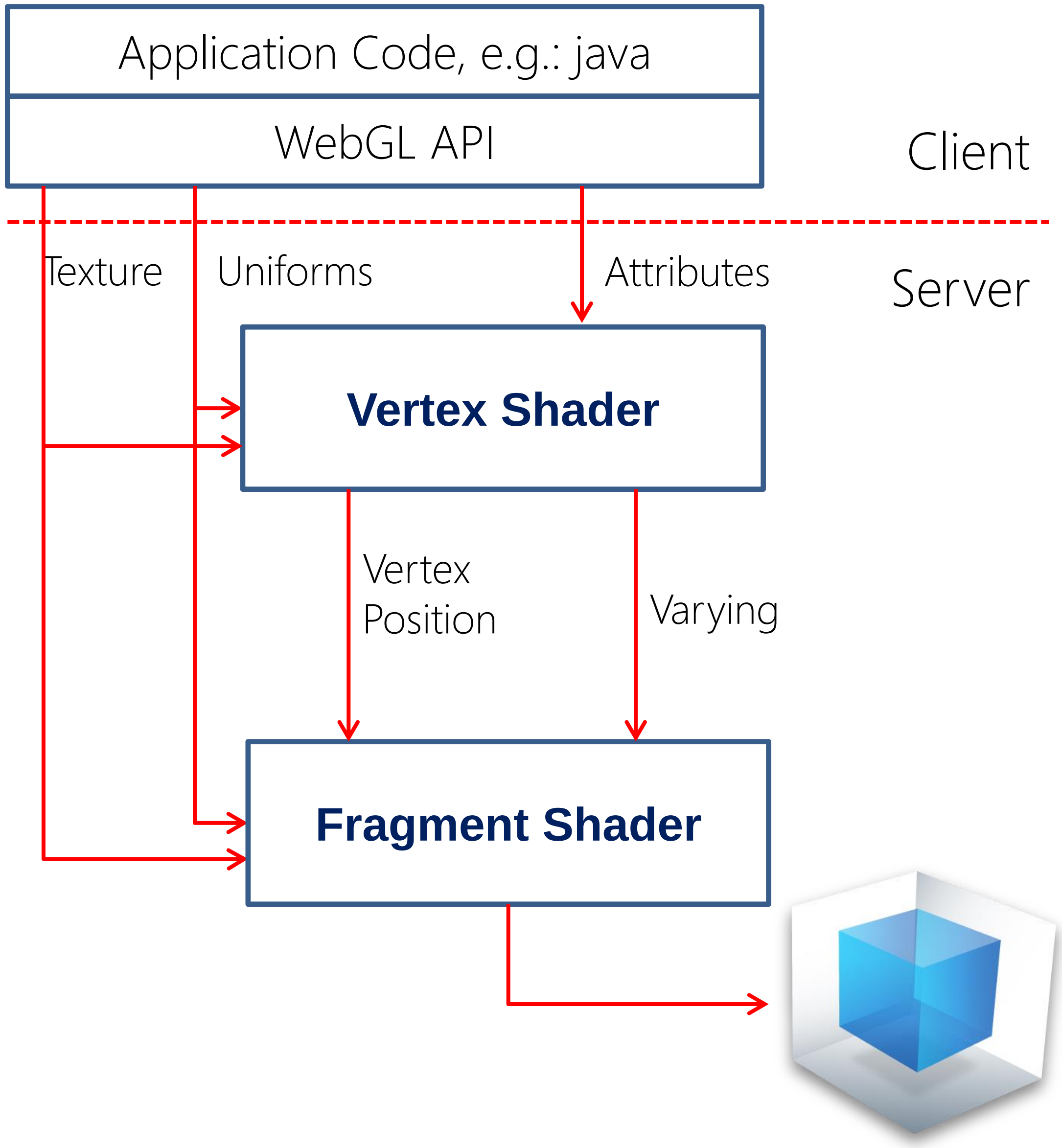
FS output
=
Final rendered image

Automatically done by the GPU after the VS and before the FS

Rendering pipeline

How to pass data through the pipeline?

- Attributes
- Uniforms
- Textures
- Varying



Application Code, e.g.: java

WebGL API

Client

Server

Texture

Uniforms

Attributes

Vertex Shader

Vertex Position

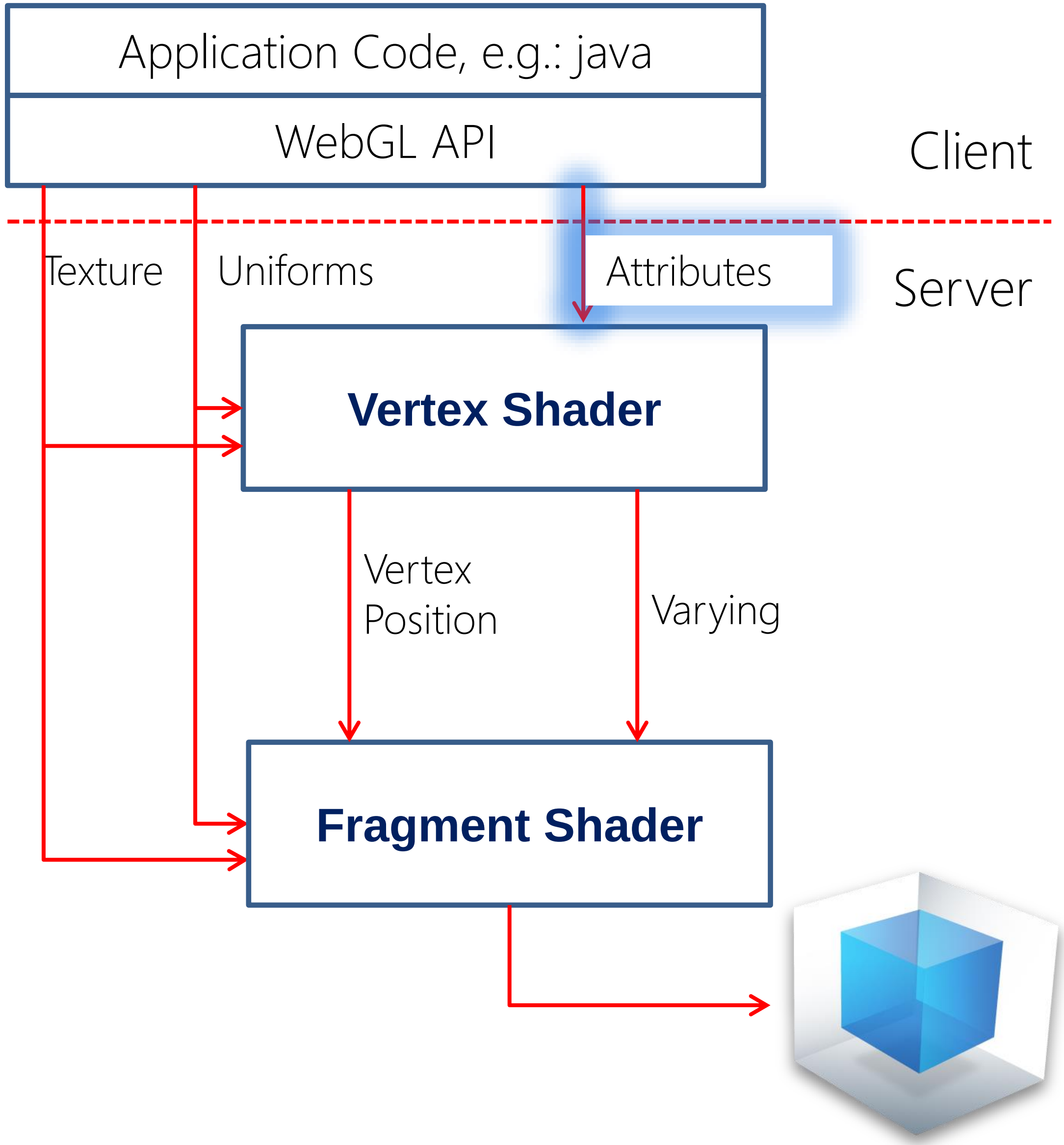
Varying

Fragment Shader

Attributes

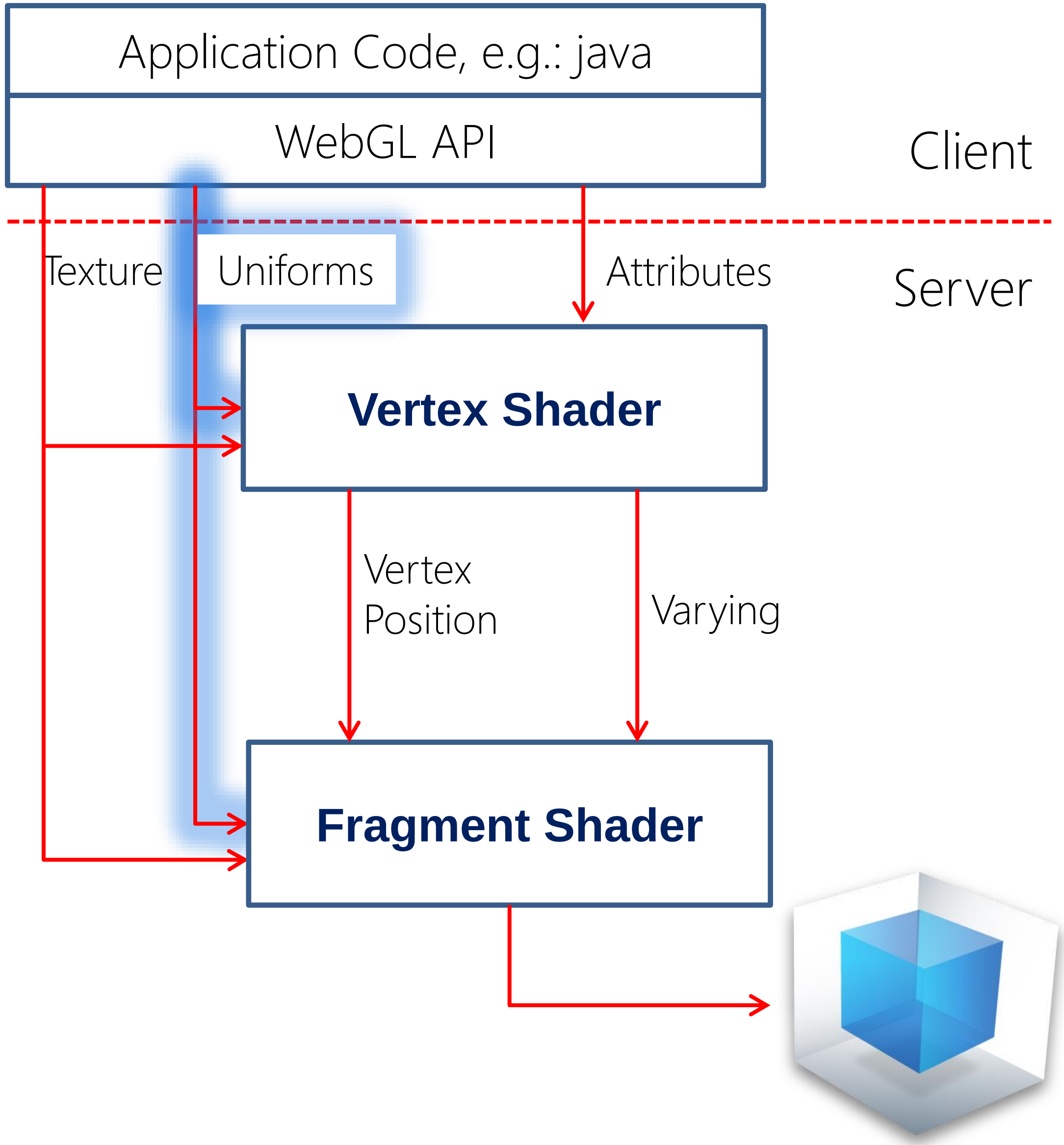
- **Attributes** are values that change for each vertex
- The most obvious example of attributes is a vertex (x, y, z) position, but more complex vertex attributes can be defined
- The data we decide to include to describe a vertex determines the **Vertex Format**

(x, y, z)	12 Bytes
(x, y, z, r, g, b)	24 Bytes
(x, y, z, nx, ny, nz)	24 Bytes
(x, y, z, nx, ny, nz, u, v)	32 Bytes



Uniforms

- **Uniforms** are per-program variables that are constant during the program execution
- Transformation matrices are usually uniforms
- Uniforms can be passed both to the vertex and the fragment shader

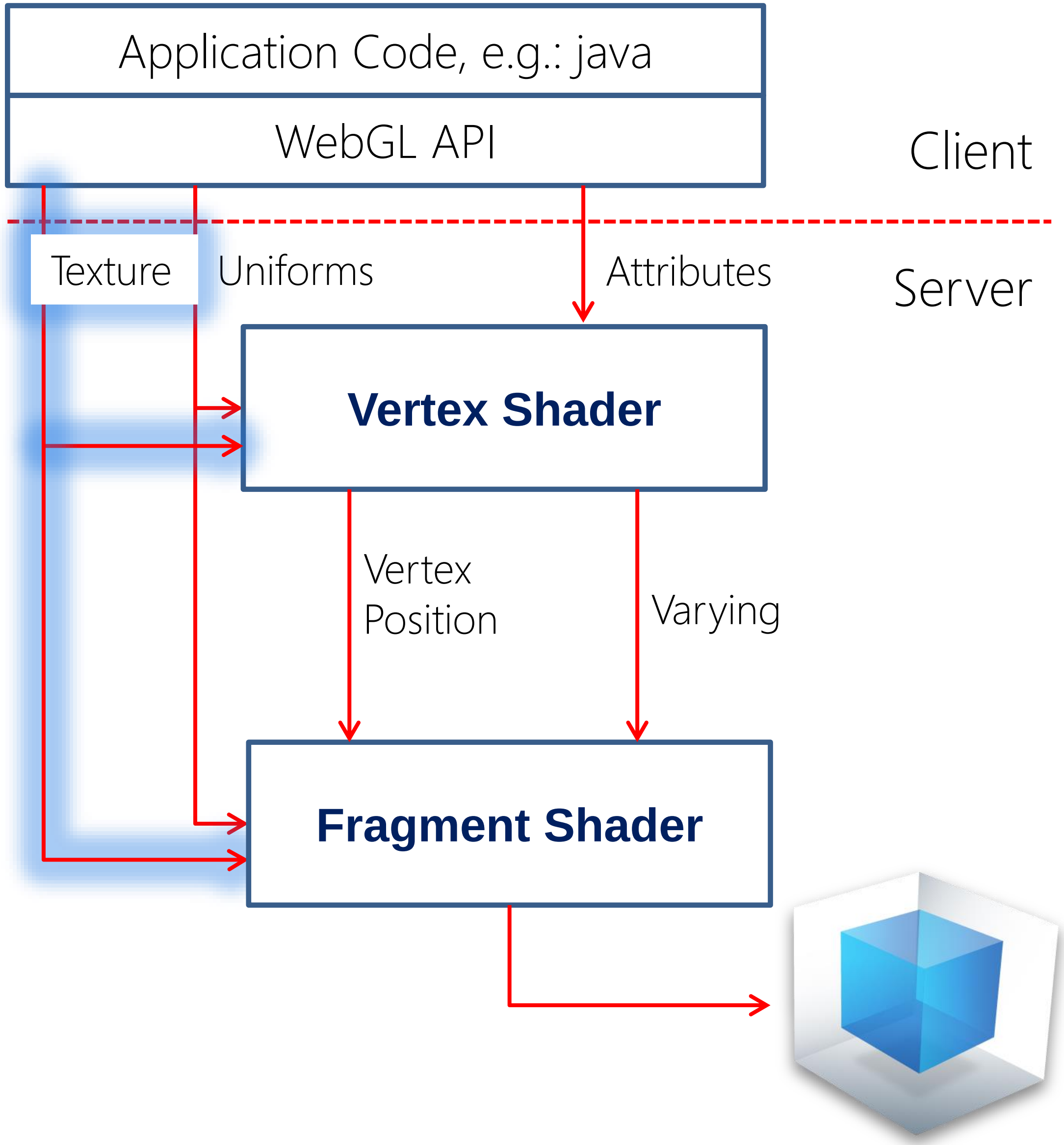


Textures (Sampler)

- **Texture data (Samplers)** are a special form of uniform used for texturing

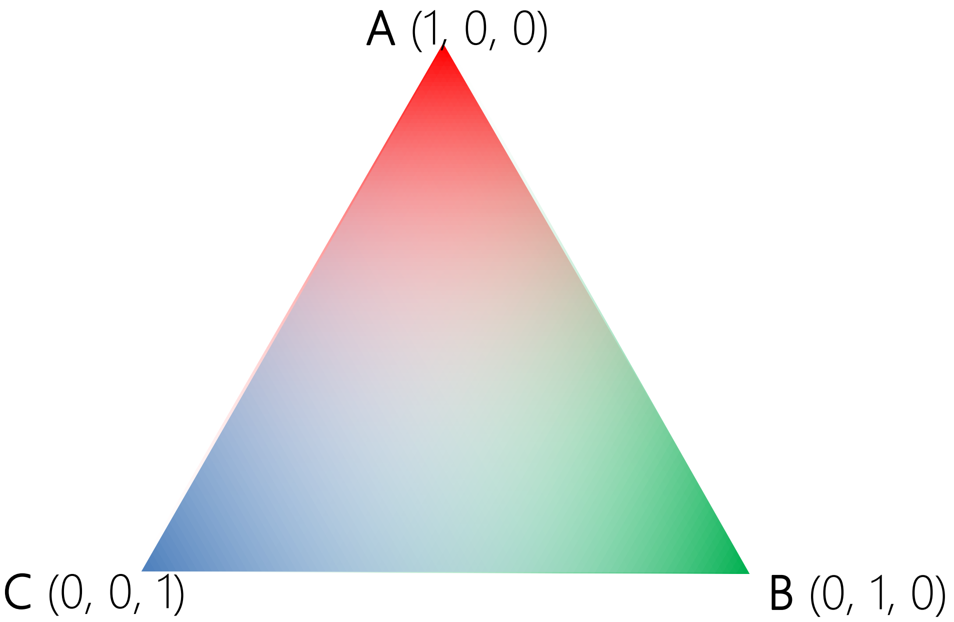
- Are used to identify the *texture object* used for each texture lookup.

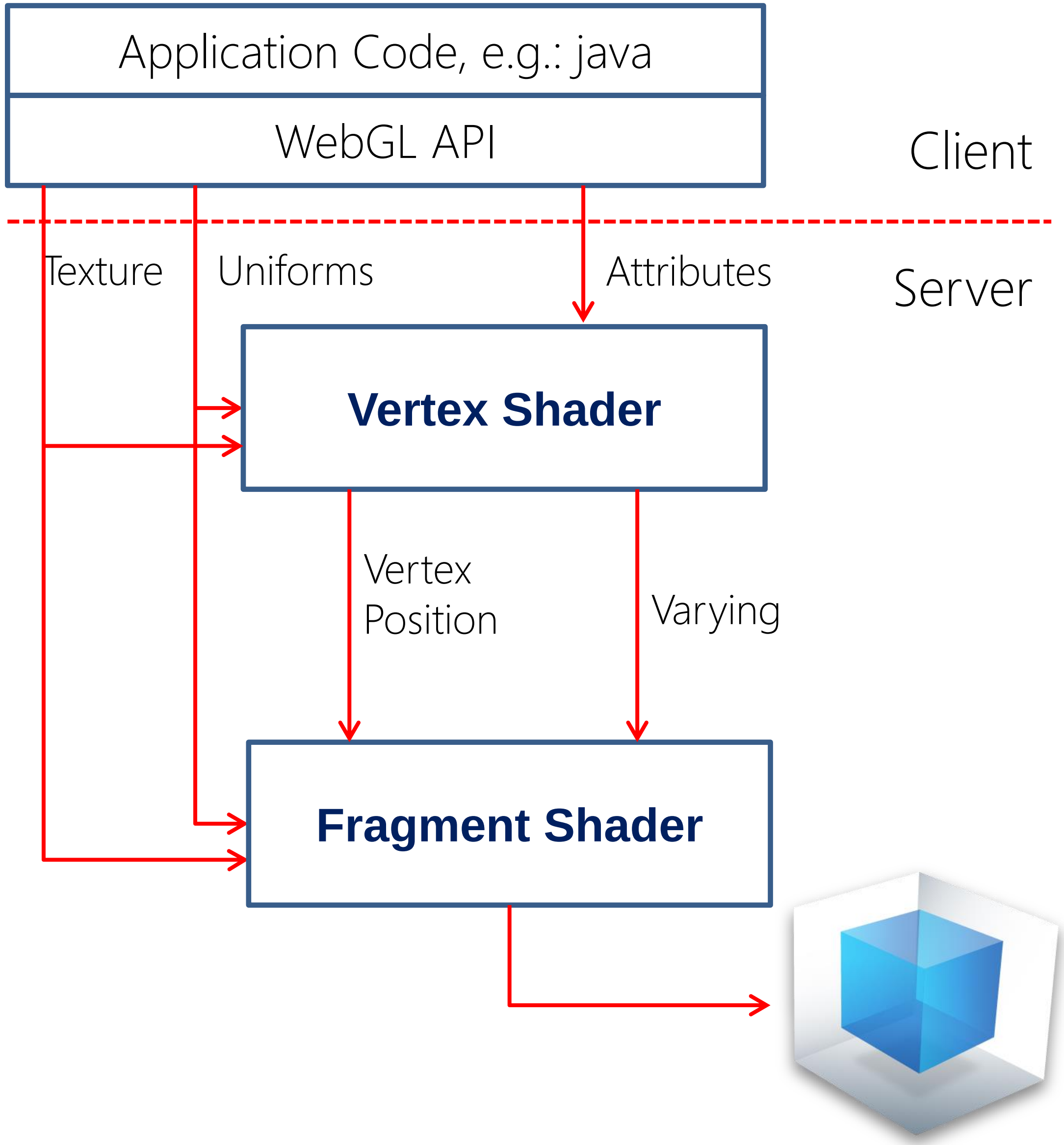
- *We will see more on GLSL textures later in the course*



Varyings

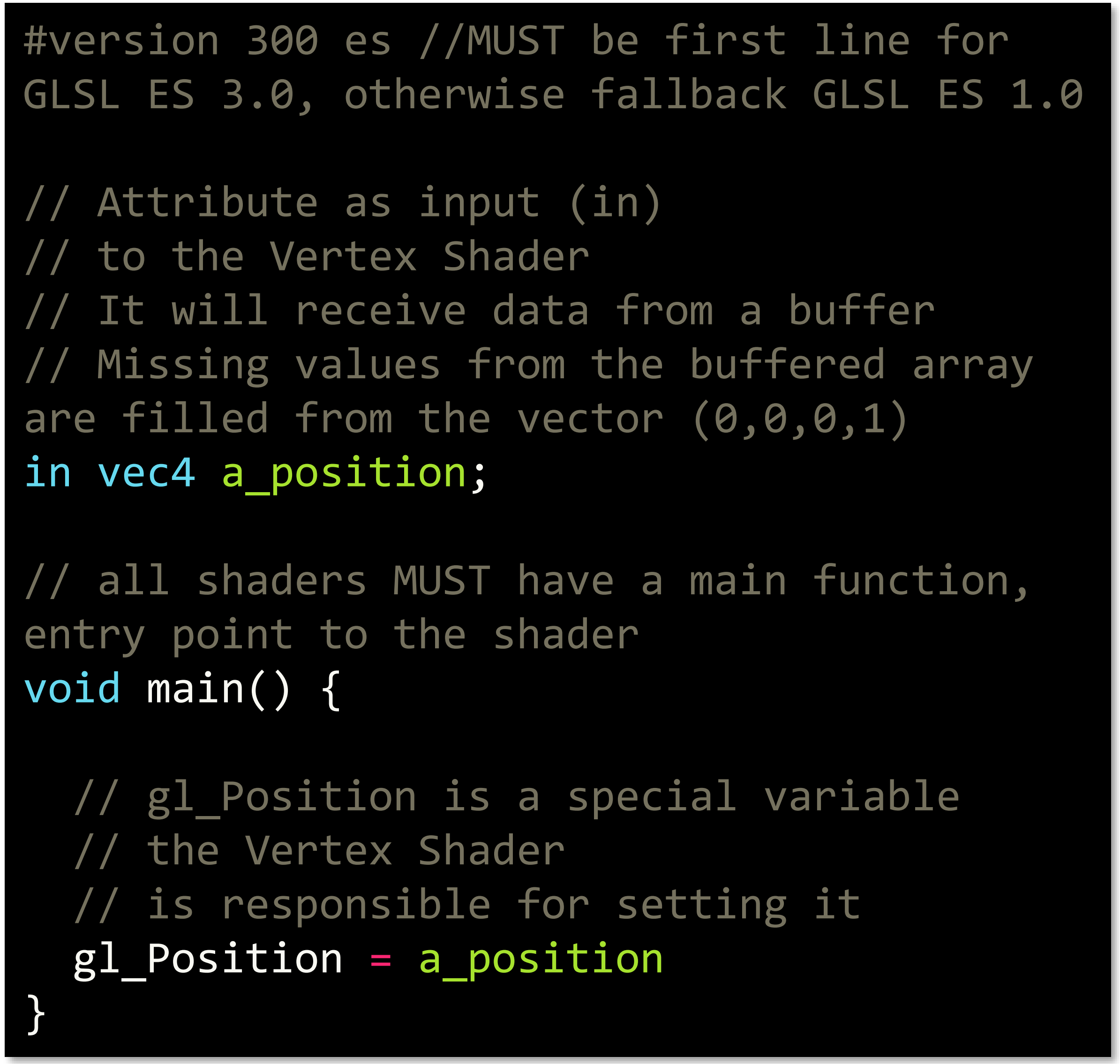
- **Varying variables** hold the results of vertex shader execution that are used later in the pipeline
- These values are expected to be **interpolated** across the primitive being rendered.





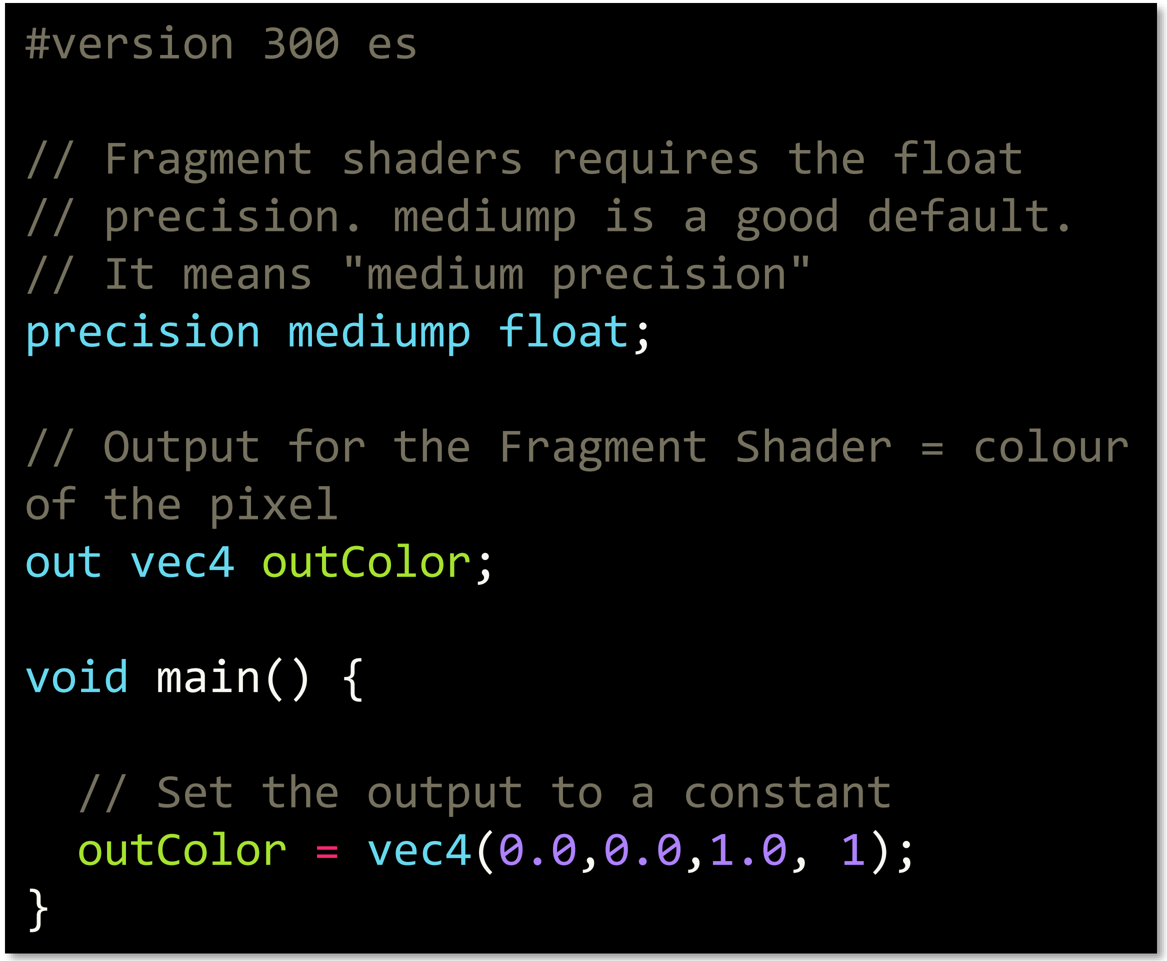
A simple example

Vertex Shader



```
#version 300 es //MUST be first line for
GLSL ES 3.0, otherwise fallback GLSL ES 1.0

// Attribute as input (in)
// to the Vertex Shader
// It will receive data from a buffer
// Missing values from the buffered array
are filled from the vector (0,0,0,1)
in vec4 a_position;

// all shaders MUST have a main function,
entry point to the shader
void main() {

  // gl_Position is a special variable
  // the Vertex Shader
  // is responsible for setting it
  gl_Position = a_position
}
```

Fragment Shader



```
#version 300 es

// Fragment shaders requires the float
// precision. mediump is a good default.
// It means "medium precision"
precision mediump float;

// Output for the Fragment Shader = colour
of the pixel
out vec4 outColor;

void main() {

  // Set the output to a constant
  outColor = vec4(0.0,0.0,1.0, 1);
}
```

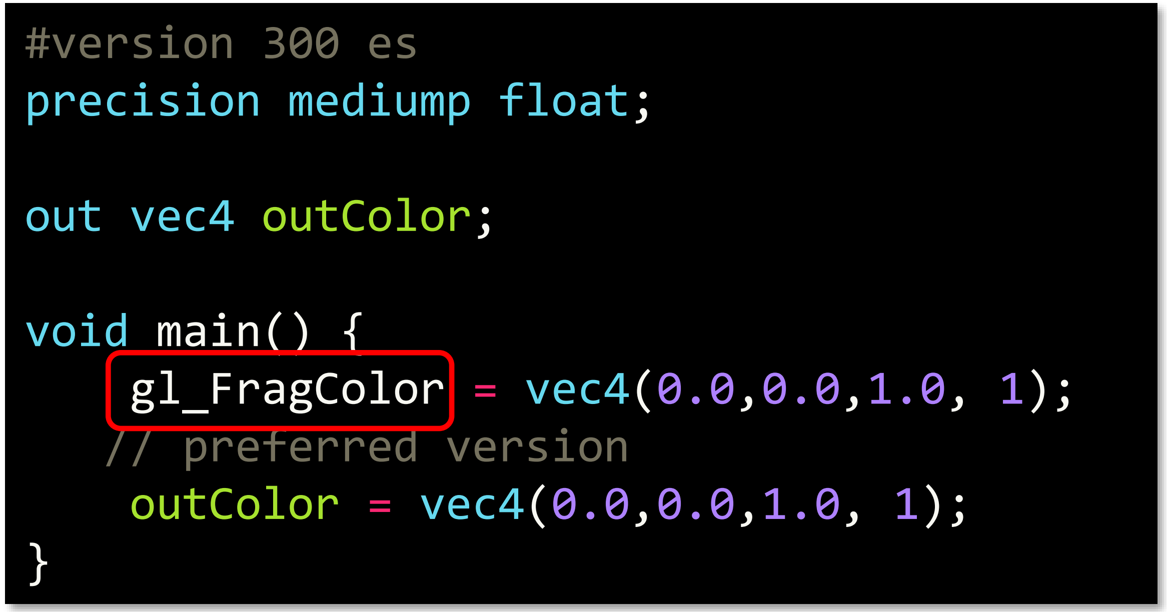
Special Variables

Shader programs use **Special Variables** (global) to communicate with fixed-function parts of the pipeline.

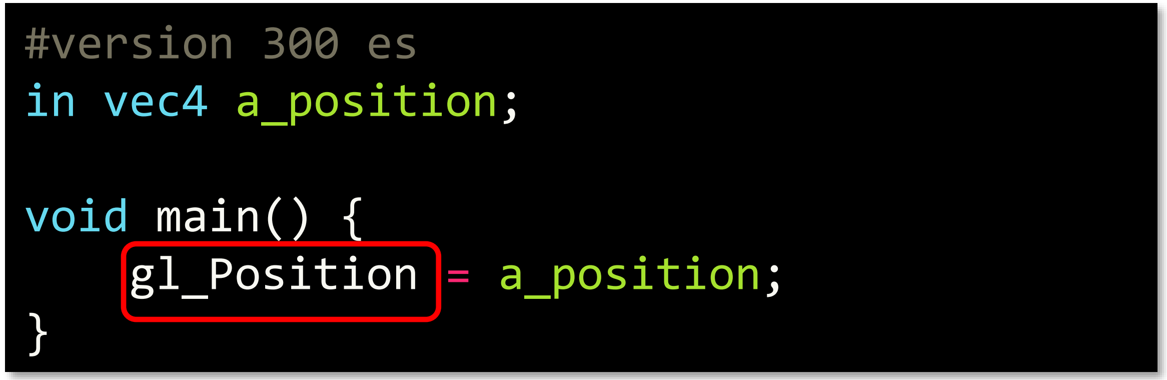
Built-in, so no need to declare them.

`vec4 gl_Position;` (VS) Final transformed vertex position, computed in clip space coordinates.

`vec4 gl_FragColor;` (FS) Final fragment color output, in RGBA (from WebGL2 can be avoided)



```
#version 300 es
precision mediump float;

out vec4 outColor;

void main() {
    gl_FragColor = vec4(0.0,0.0,1.0, 1);
    // preferred version
     outColor = vec4(0.0,0.0,1.0, 1);
}
```

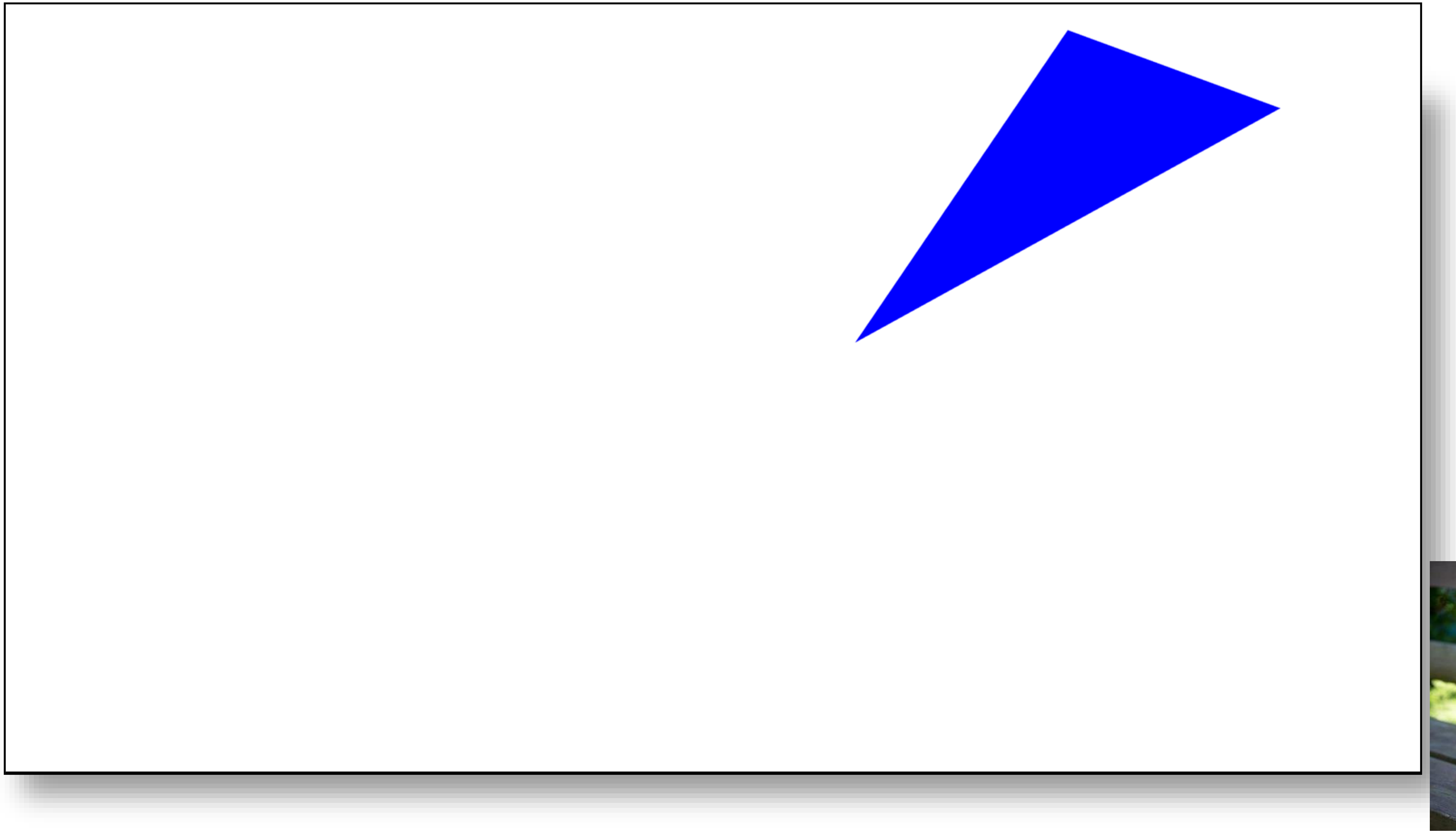


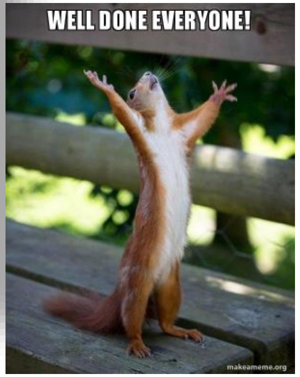
```
#version 300 es
in vec4 a_position;

void main() {
    gl_Position = a_position;
}
```

Vertex shader

Fragment shader

Result

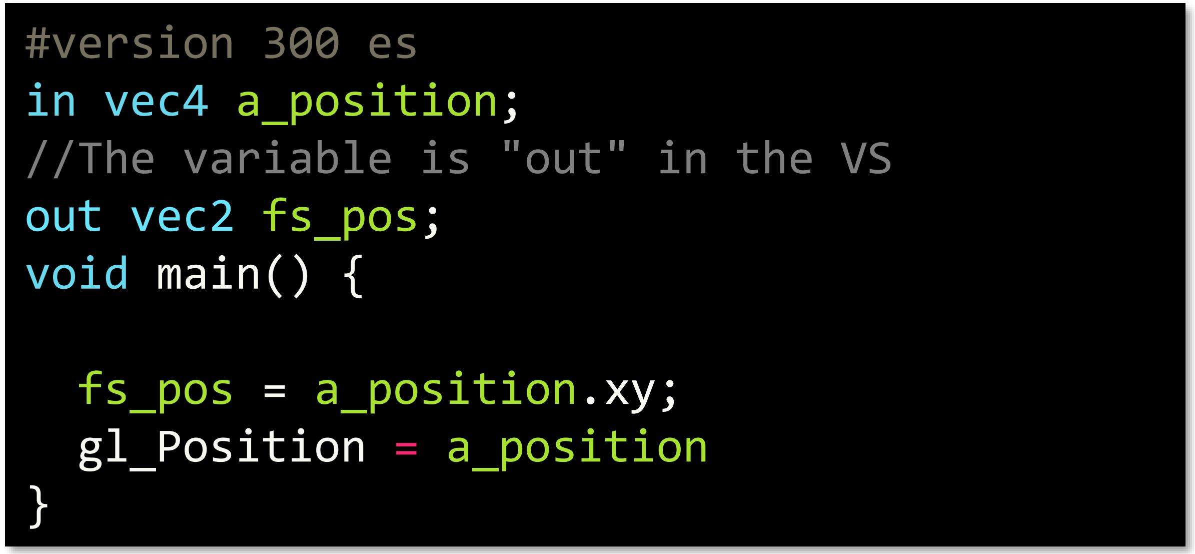




Varyings

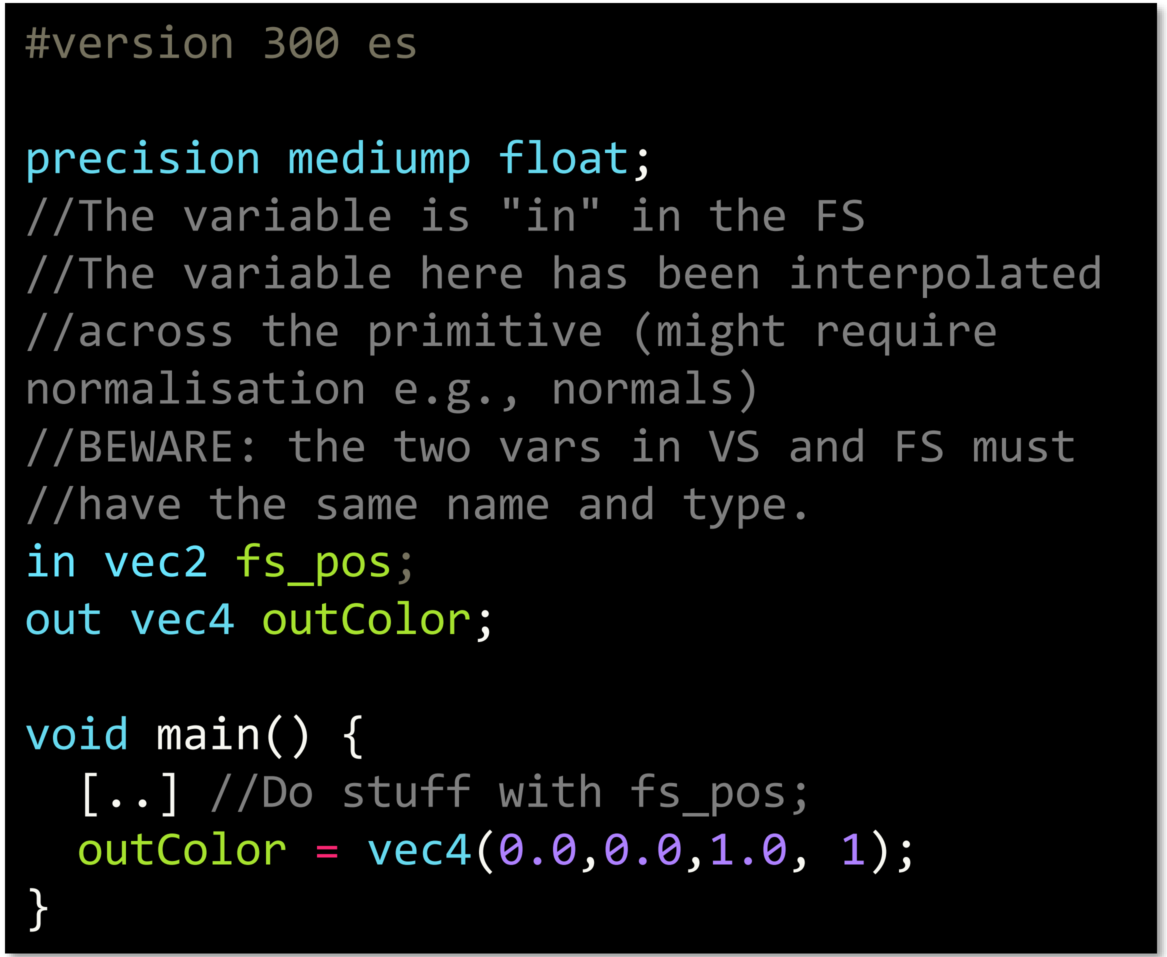
- What if we want to pass a value from VS to FS?

Vertex Shader



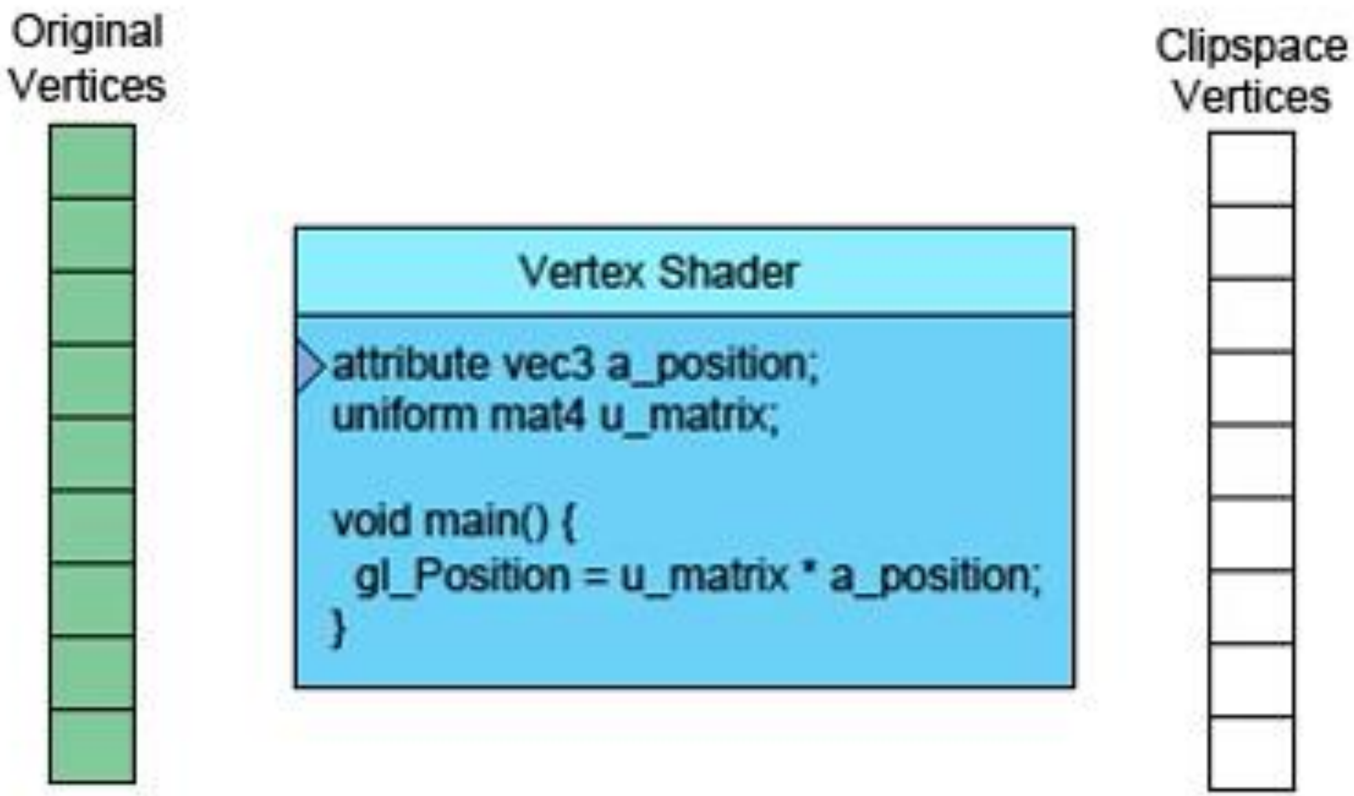
```
#version 300 es
in vec4 a_position;
//The variable is "out" in the VS
out vec2 fs_pos;
void main() {

  fs_pos = a_position.xy;
  gl_Position = a_position
}
```

Fragment Shader



```
#version 300 es

precision mediump float;
//The variable is "in" in the FS
//The variable here has been interpolated
//across the primitive (might require
normalisation e.g., normals)
//BEWARE: the two vars in VS and FS must
//have the same name and type.
in vec2 fs_pos;
out vec4 outColor;

void main() {
  [..] //Do stuff with fs_pos;
  outColor = vec4(0.0,0.0,1.0, 1);
}
```

How vertex shader works



```
attribute vec3 a_position;
uniform mat4 u_matrix;

void main() {
  gl_Position = u_matrix * a_position;
}
```