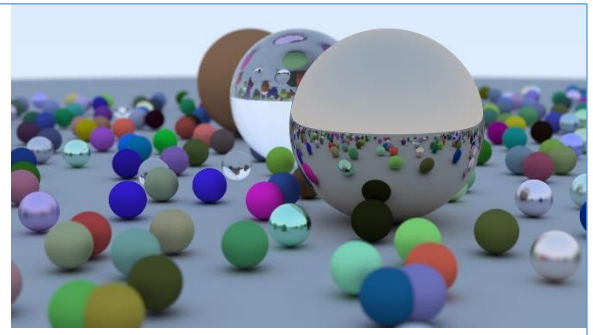
Comp4422



Computer Graphics

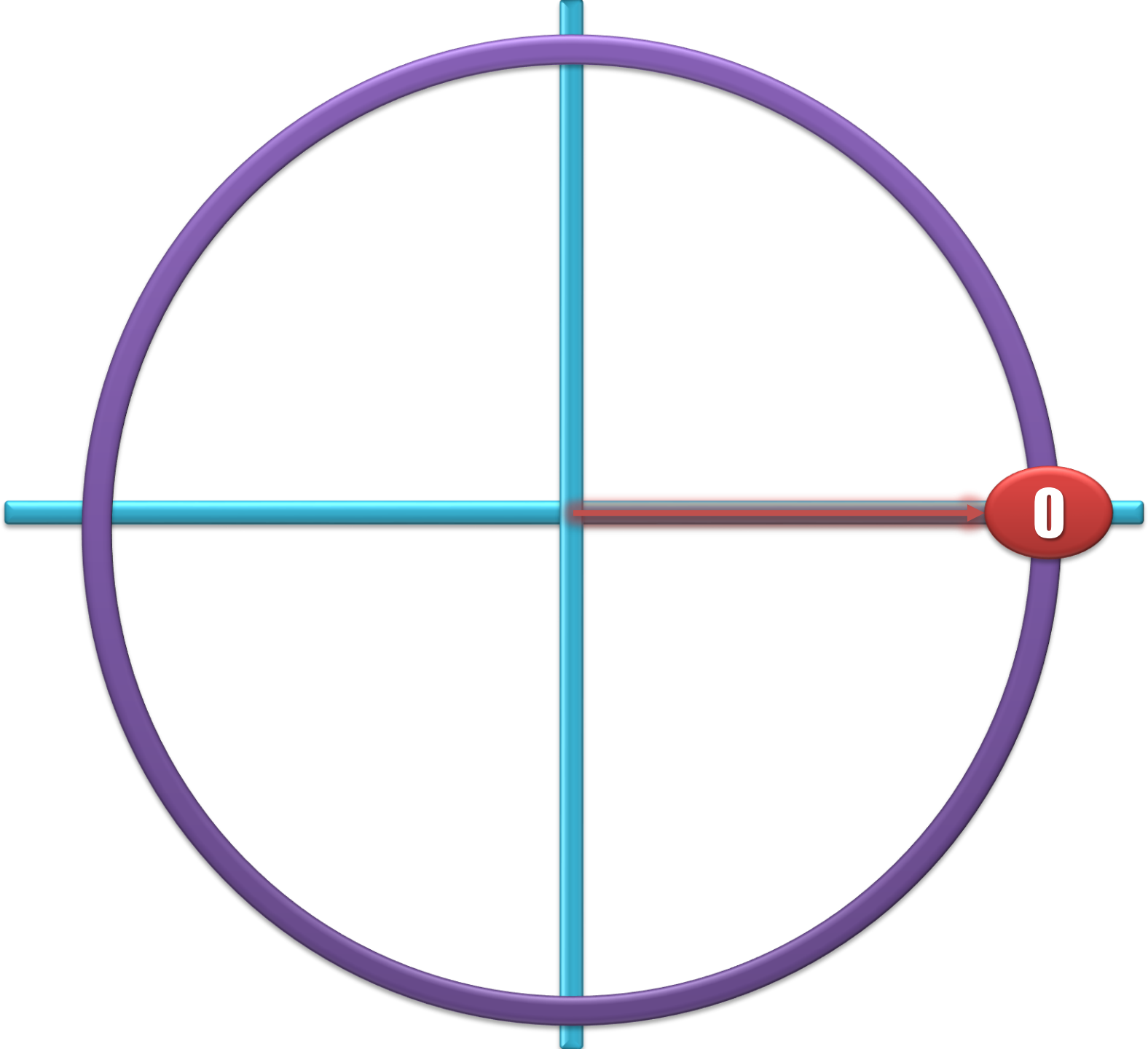
Lecture 04:

Shaders

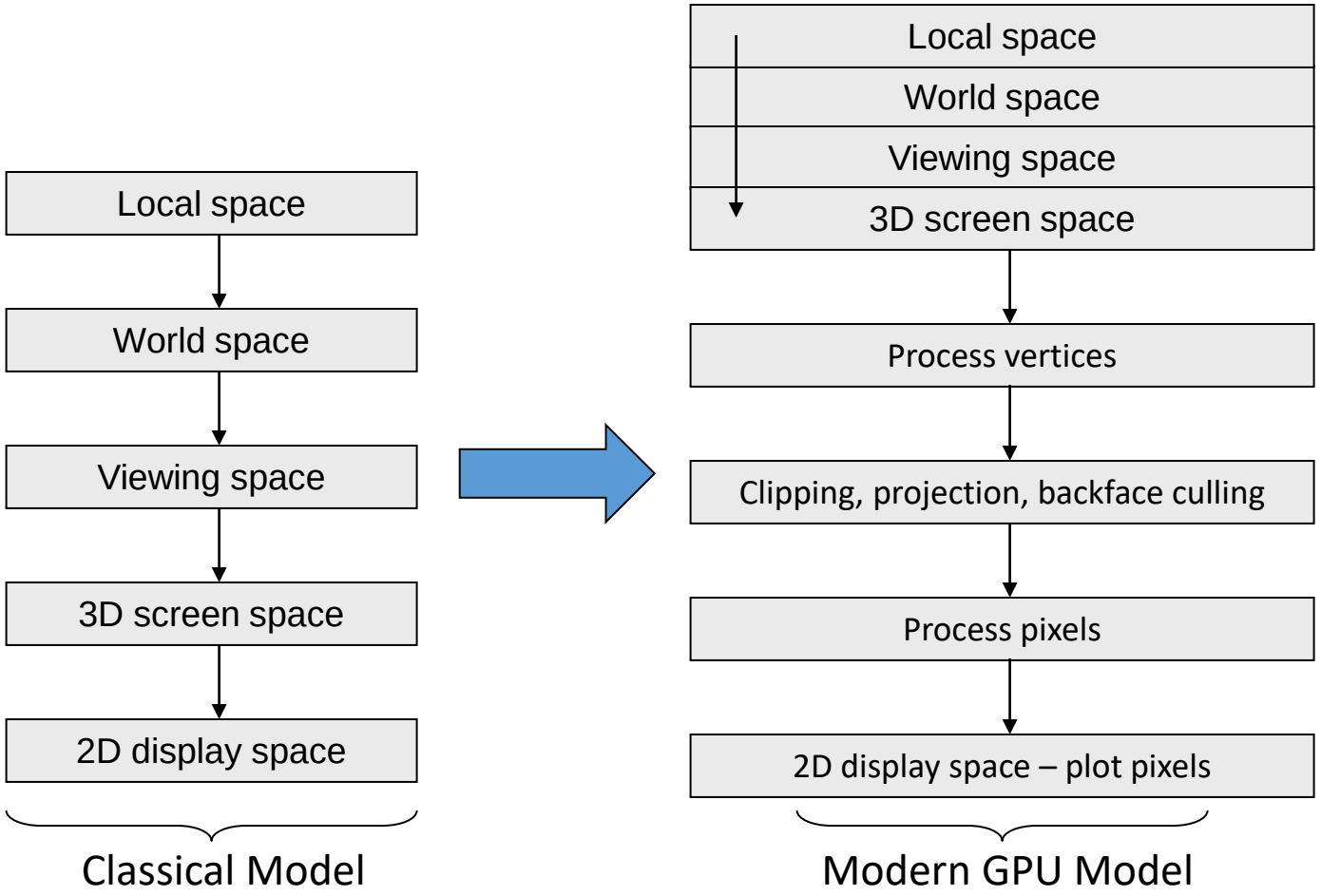
Shaders

What you will learn?

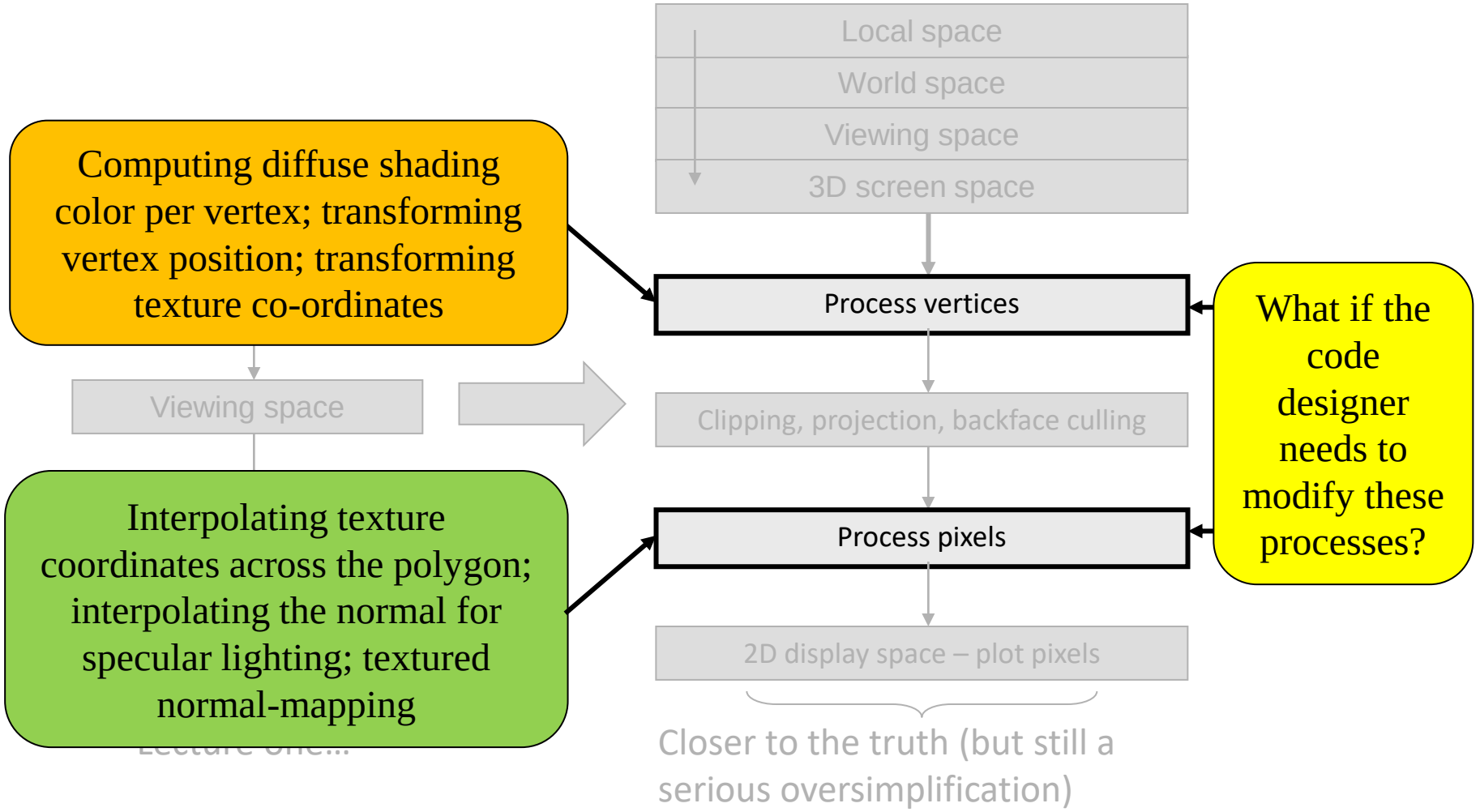
- An overview of WebGL **pipeline**
- Difference between fixed and modern programmable shaders
- **GLSL**: vertex and fragment shaders
- How to create and use shaders within WebGL applications
- Procedural and fragment shaders


0

Graphics Pipeline Model



Graphics Pipeline Model

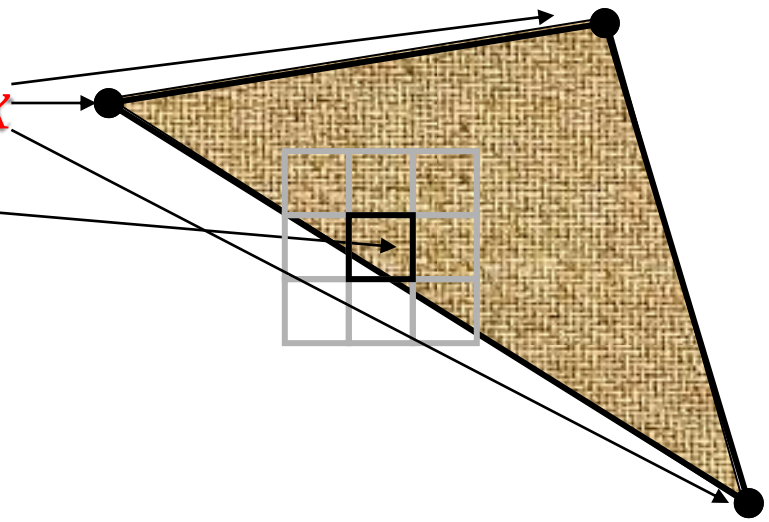


Programmable Shaders

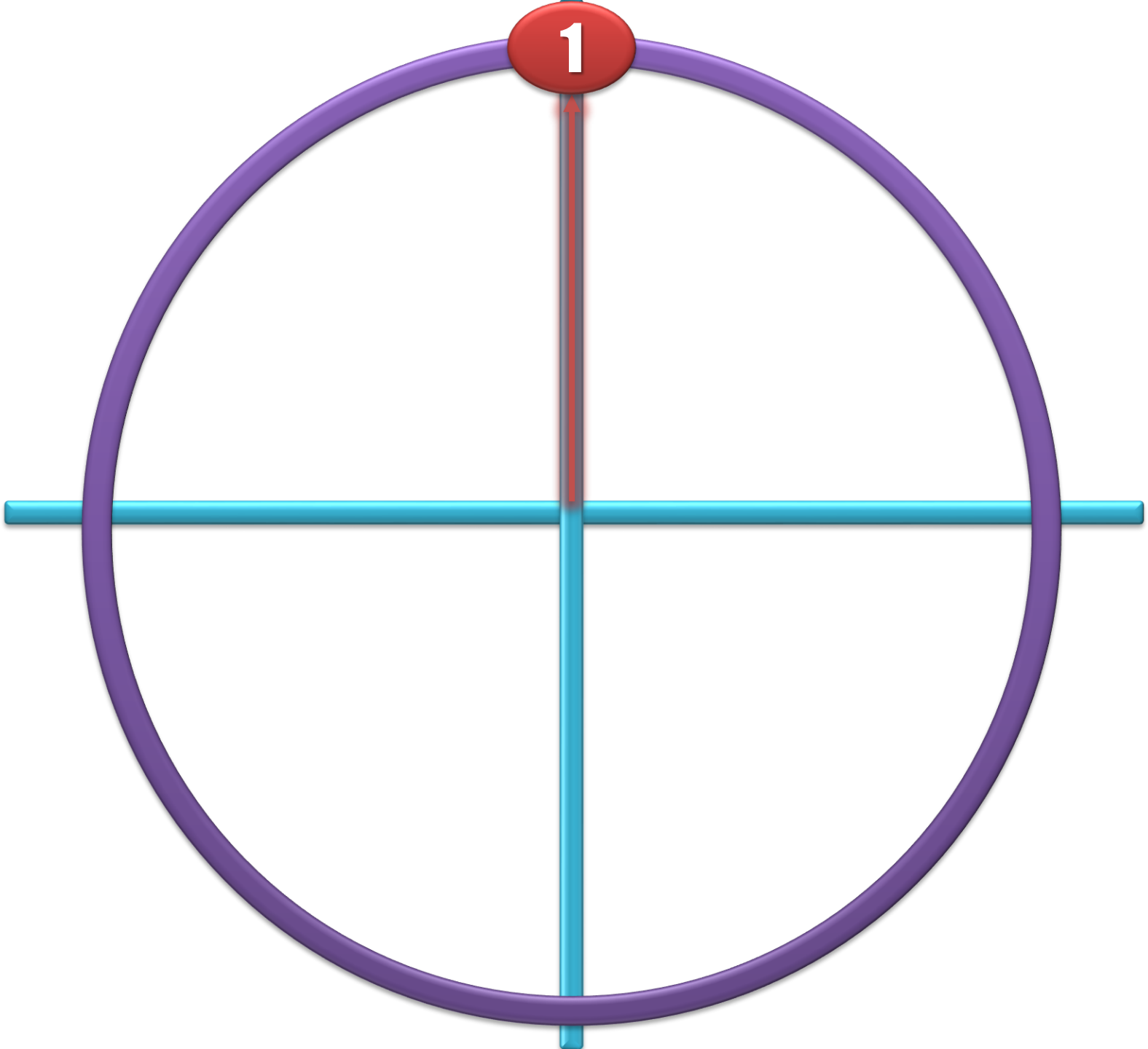
- Introduce programmable shaders:
  - programmable logical units on the GPU which can replace the “fixed” functionality of OpenGL API with user-generated code.
- Active installing custom shaders:
  - the application designer can completely override the existing implementation of core per-vertex and per-pixel behavior.

Why programmable?

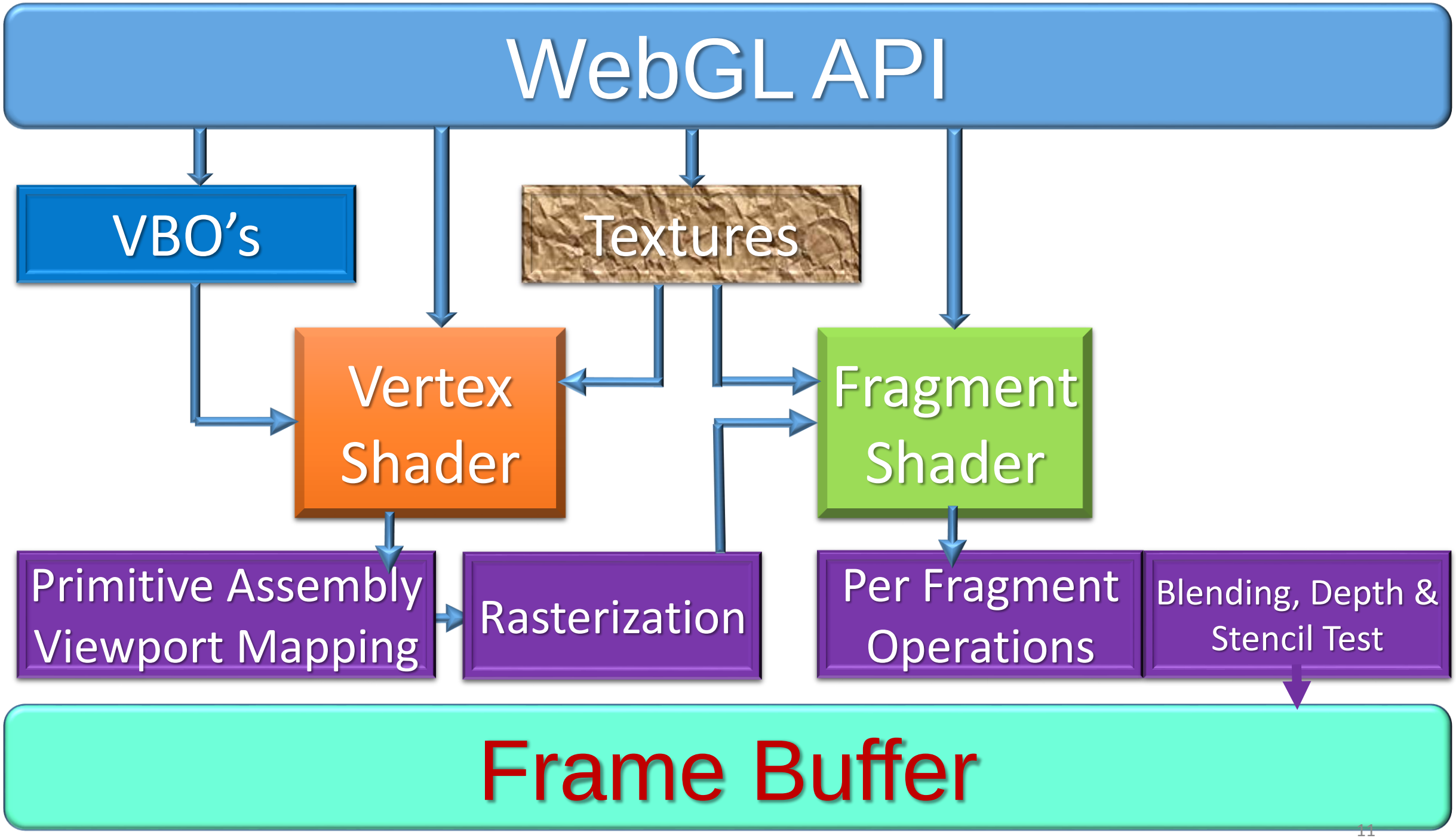
- Shaders give the user control over each *vertex* and each *fragment*
- Each pixel or partial pixel is interpolated between vertices.



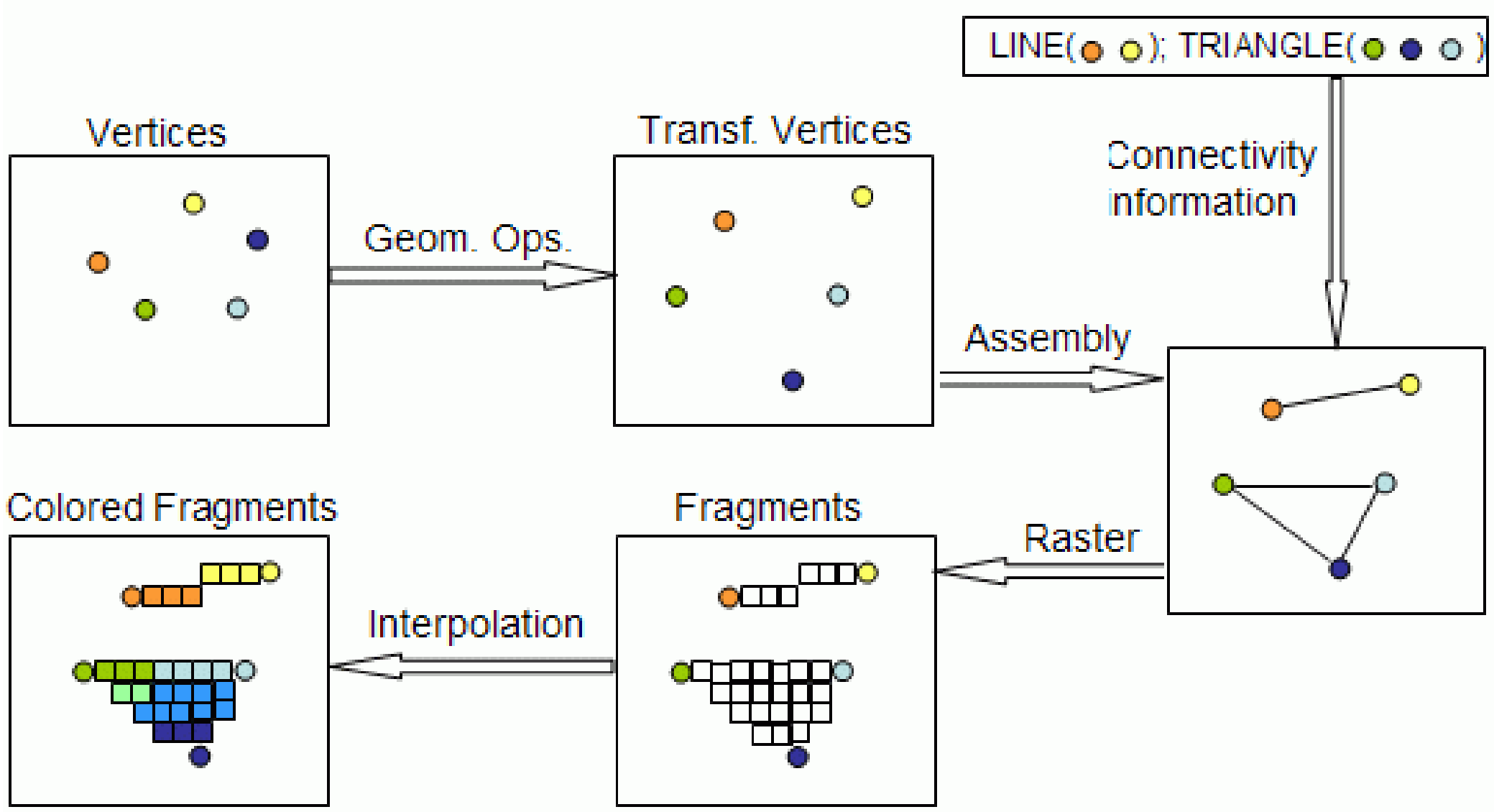
- After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and others are interpolated across the polygon. The interpolated values are passed to each pixel fragment.


1

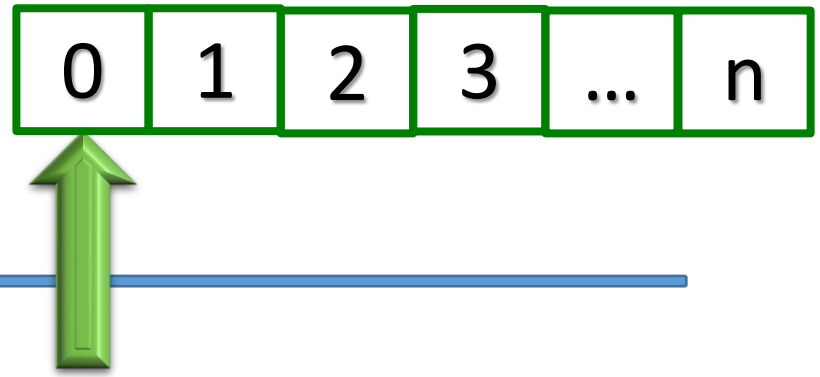
WebGL Model


WebGL API
VBO's
Textures
Vertex Shader
Fragment Shader
Primitive Assembly Viewport Mapping
Rasterization
Per Fragment Operations
Blending, Depth & Stencil Test
Frame Buffer

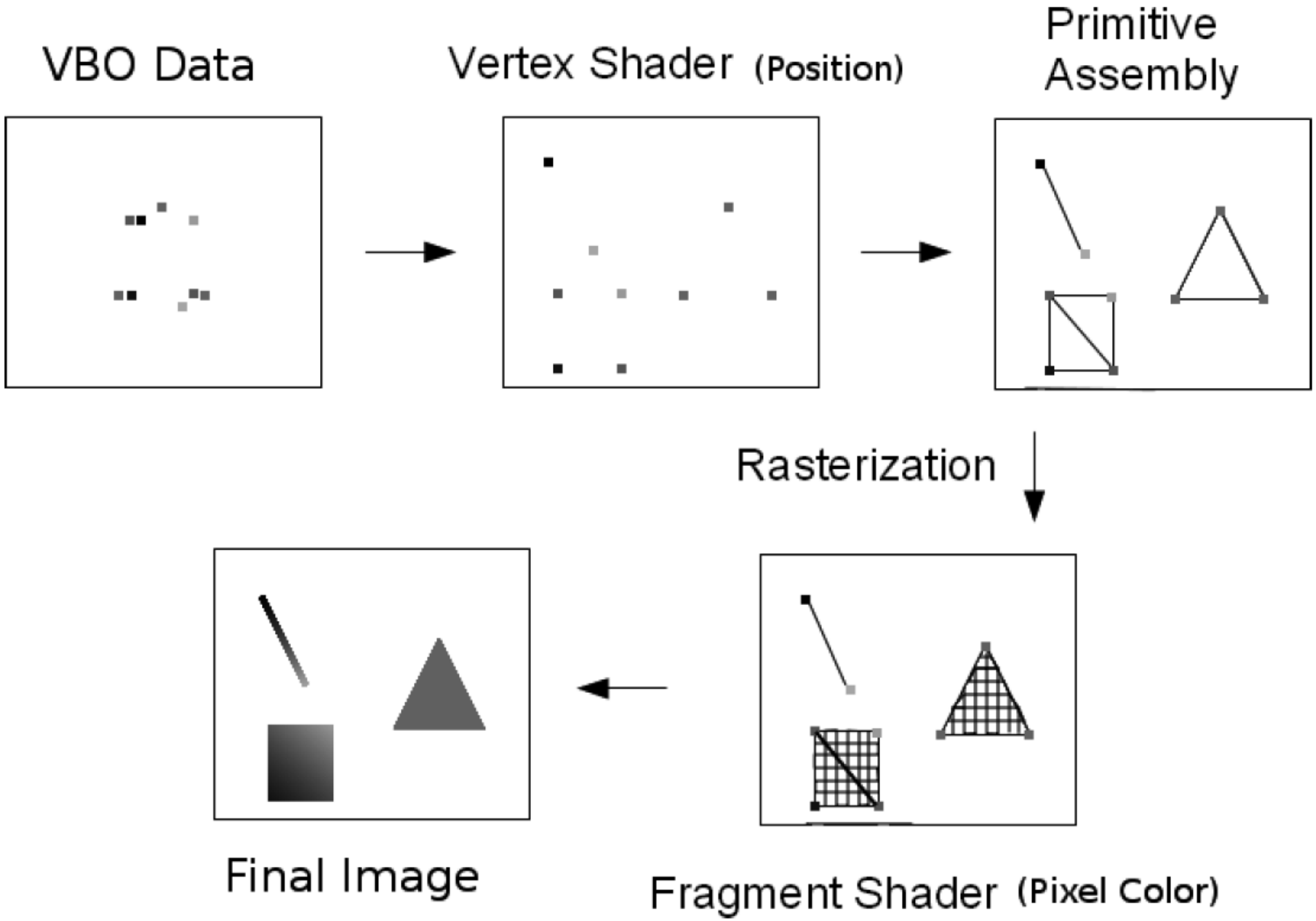
Data Flow Visualization



Recall: Shaders




```
<script id="shader-vs" type="x-shader/x-vertex">
        attribute vec3 aVertexPosition;
        void main(void) {
                gl_Position = vec4(aVertexPosition, 1.0);
        }
</script>
<script id="shader-fs" type="x-shader/x-fragment">
        void main(void) {
                gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
        }
</script>
```


VBO Data
Vertex Shader (Position)
Primitive Assembly
Rasterization
Final Image
Fragment Shader (Pixel Color)

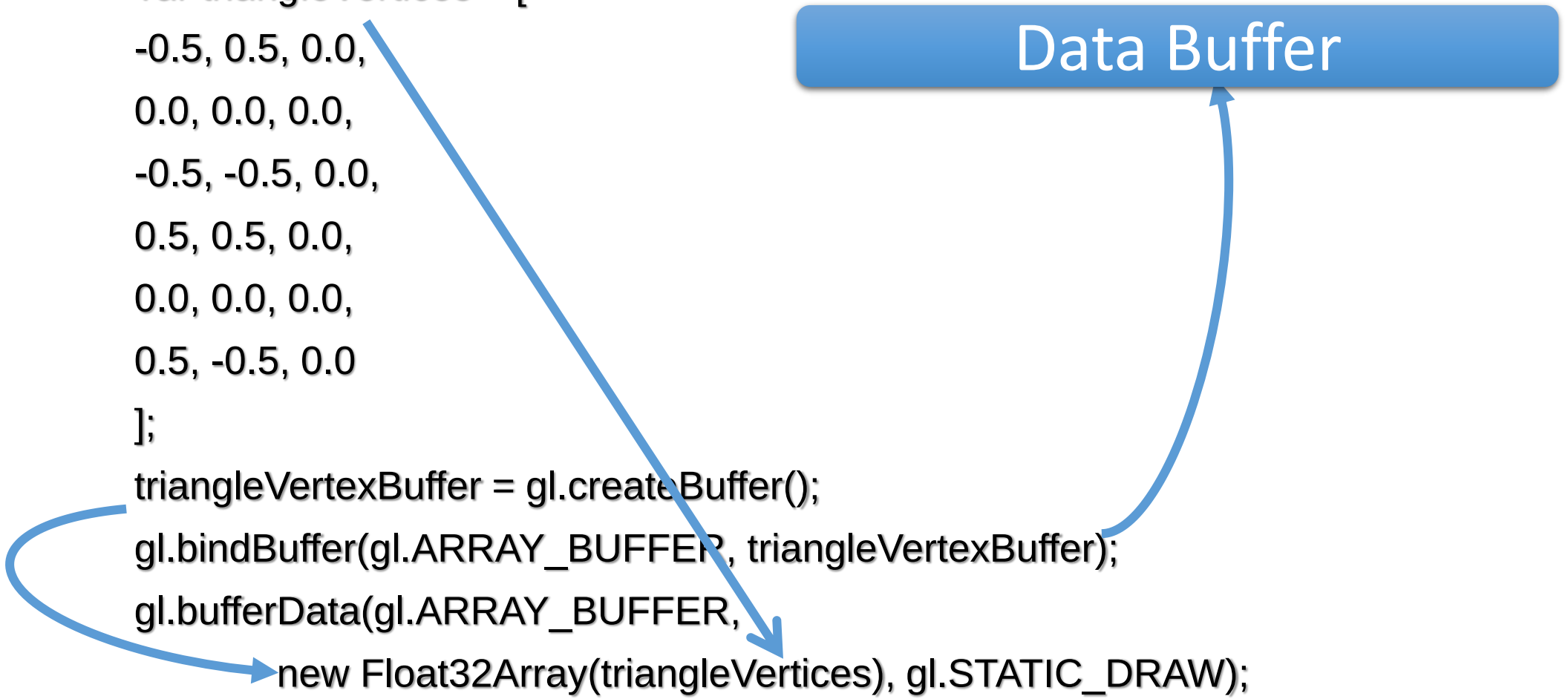
Set up linkages

Two Steps

1. Create and load buffers with data
2. Bind each buffer to a shader attribute

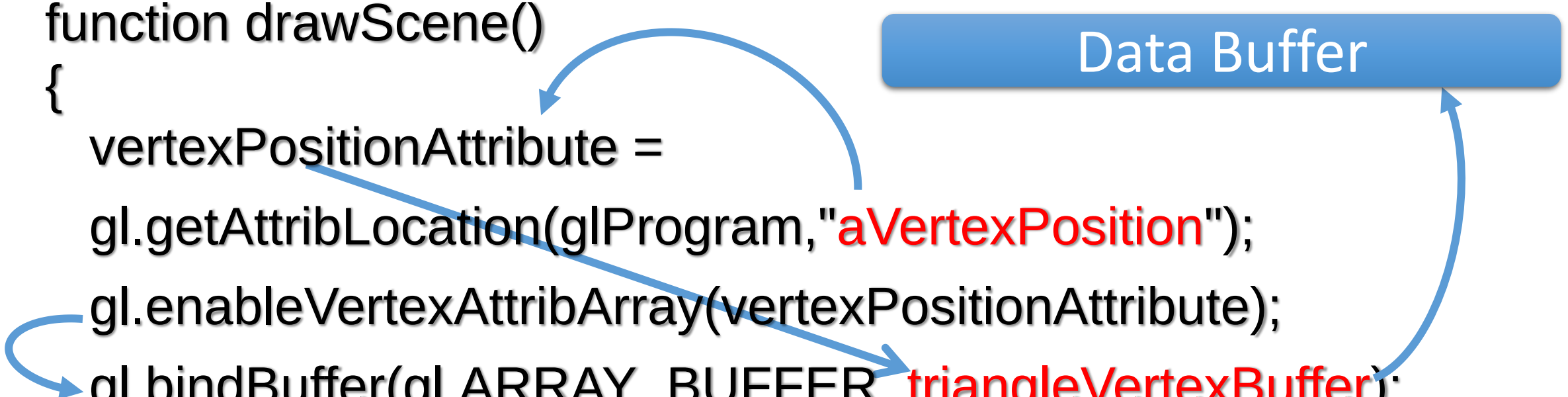
1. In setupBuffers function

```
function setupBuffers() {
        var triangleVertices = [
        -0.5, 0.5, 0.0,
        0.0, 0.0, 0.0,
        -0.5, -0.5, 0.0,
        0.5, 0.5, 0.0,
        0.0, 0.0, 0.0,
        0.5, -0.5, 0.0
        ];
        triangleVertexBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER,
                new Float32Array(triangleVertices), gl.STATIC_DRAW);
}
```



2. In drawScene function

```
function drawScene()
{
   vertexPositionAttribute =
   gl.getAttribLocation(glProgram,"aVertexPosition");
   gl.enableVertexAttribArray(vertexPositionAttribute);
   gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexBuffer);
   gl.vertexAttribPointer(vertexPositionAttribute, 3, gl.FLOAT, false,0,0);
   gl.drawArrays(gl.TRIANGLES, 0, 6);
}
```



Programmable Fragments

Programmable Vertex Shader

```
<script id="shader-vs" type="x-shader/x-vertex">
        attribute vec3 aVertexPosition;
        attribute vec3 aVertexColor;
        varying highp vec4 vColor;
        void main(void) {
            gl_Position = vec4(aVertexPosition, 1.0);
            vColor = vec4(aVertexColor, 1.0);
        }
</script>
```

Programmable Fragments

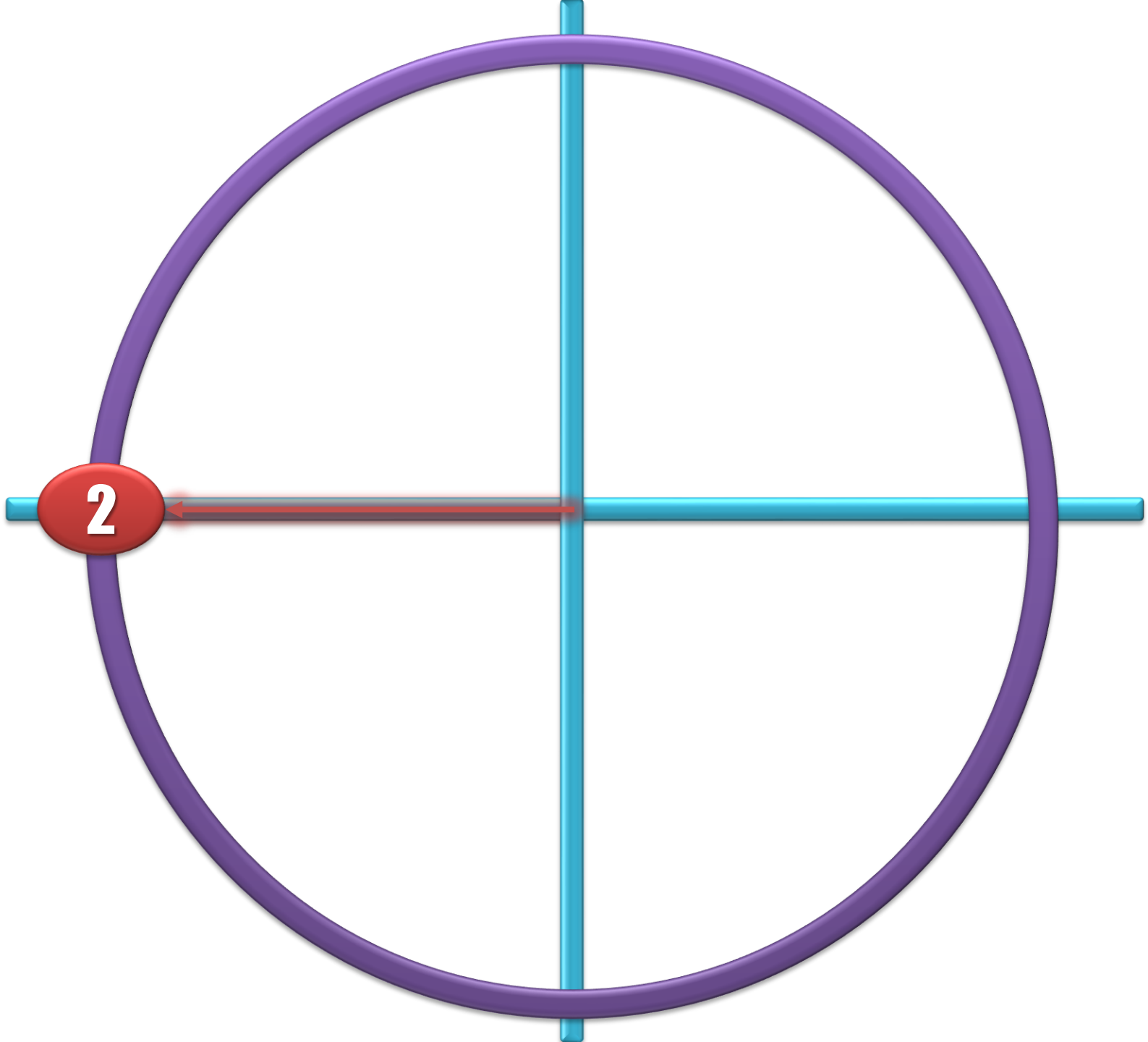
```
<script id="shader-fs" type="x-shader/x-fragment">
        varying highp vec4 vColor;
        void main(void) {
            gl_FragColor = vColor;
        }
</script>
```

Compare Shaders

Fixed	Programmed

<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
 	
void main(void) {
gl_Position = vec4(aVertexPosition, 1.0);
 	
}
</script>
	<script id="shader-vs" type="x-shader/x-vertex">
	attribute vec3 aVertexPosition;
	attribute vec3 aVertexColor;
	varying highp vec4 vColor;
	void main(void) {
	gl_Position = vec4(aVertexPosition, 1.0);
	vColor = vec4(aVertexColor, 1.0);
	}
	</script>

<script id="shader-fs" type="x-shader/x-fragment">
 	
void main(void) {
gl_FragColor = ~~vec4(1.0, 1.0, 1.0, 1.0);~~
}
</script>
	<script id="shader-fs" type="x-shader/x-fragment">
	varying highp vec4 vColor;
	void main(void) {
	gl_FragColor = vColor;
	}
	</script>


2

Shader APIs

Totally Different Model

- Main difference: entirely parallel code
- Shaders are compiled from within your code
  - Used to be written in assembler
  - Now written in high-level languages (C++ like)
- They execute on the GPU
- GPUs have thousands of processing units
- Multiple shaders execute in parallel!
- Nonlinear code linkages
- Must pay attention to variable names, buffers, bindings…

What can be overwritten?

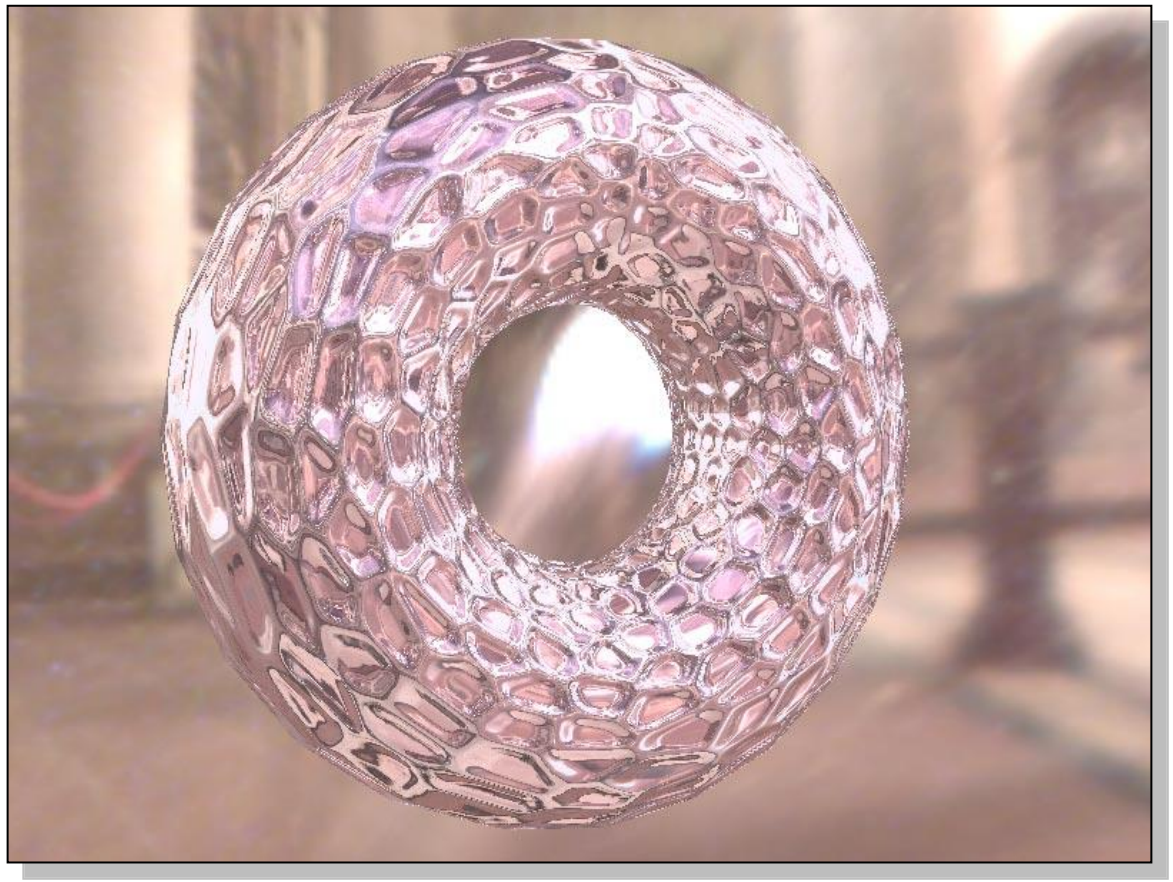
Per vertex:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

Per fragment (pixel):

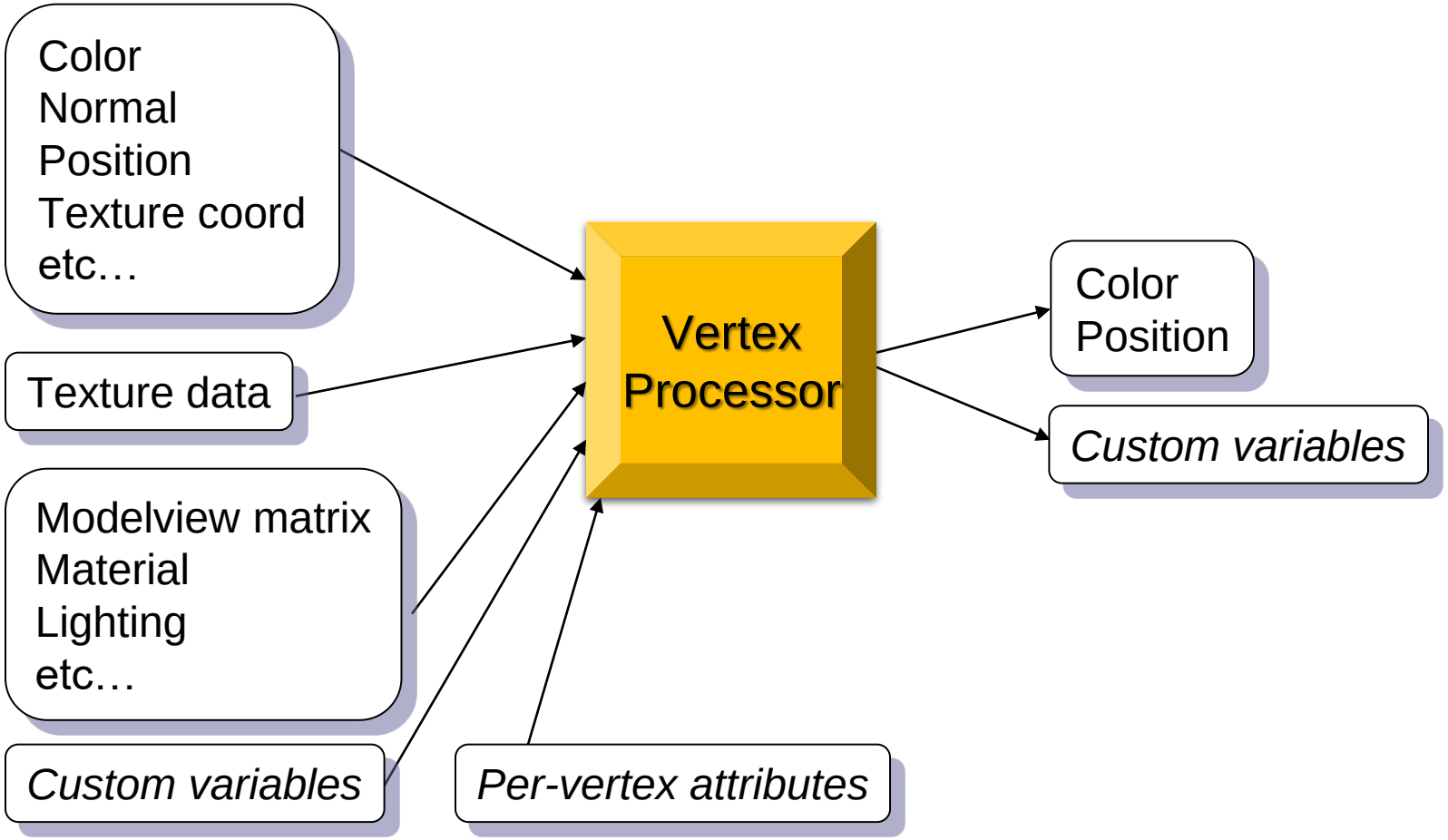
- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color summation
- Optionally:
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution

What can be accomplished?

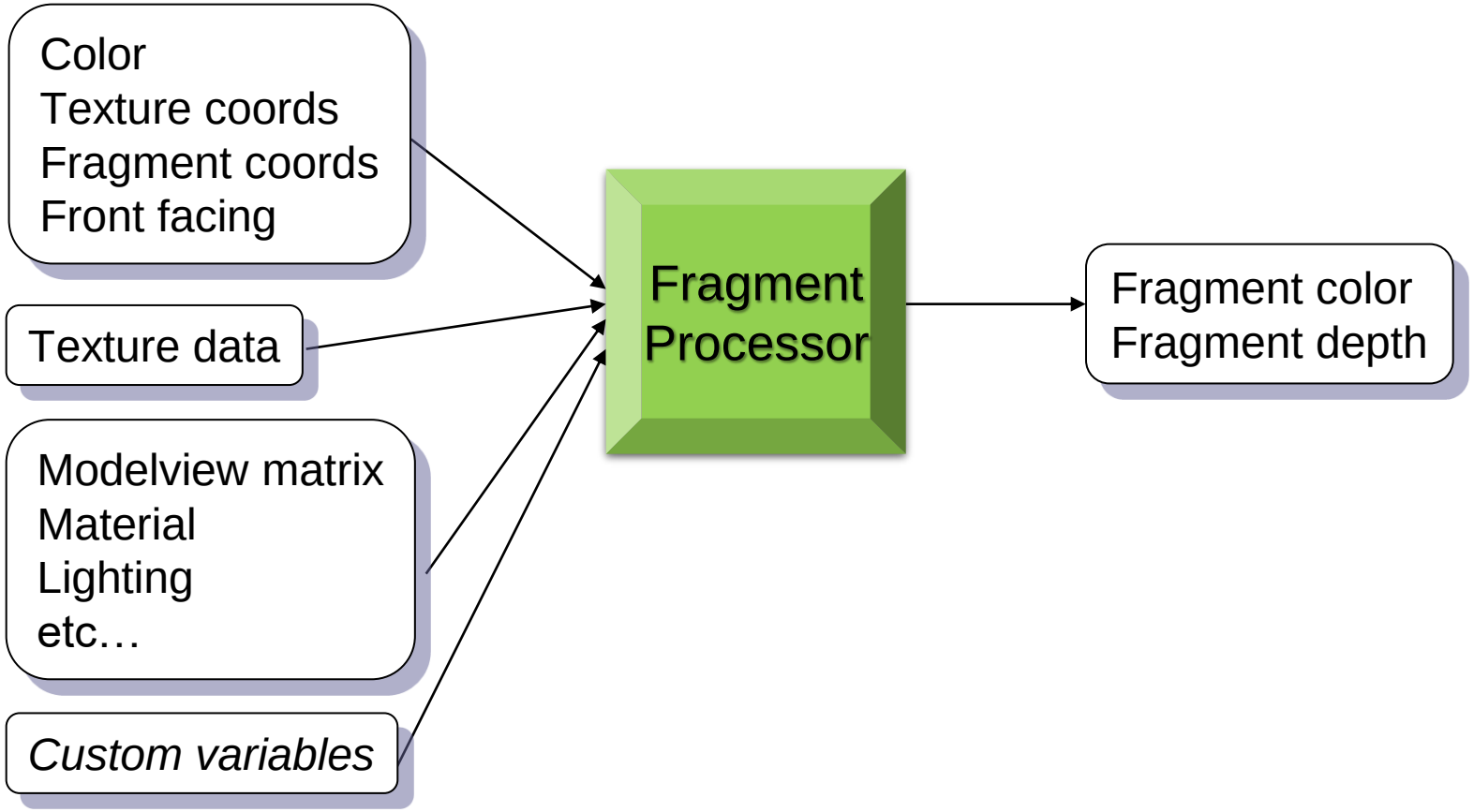


Above: Demo of Microsoft's XNA game platform
Right: Product demos by NVIDIA (top) and Radeon (bottom)

Vertex Processor

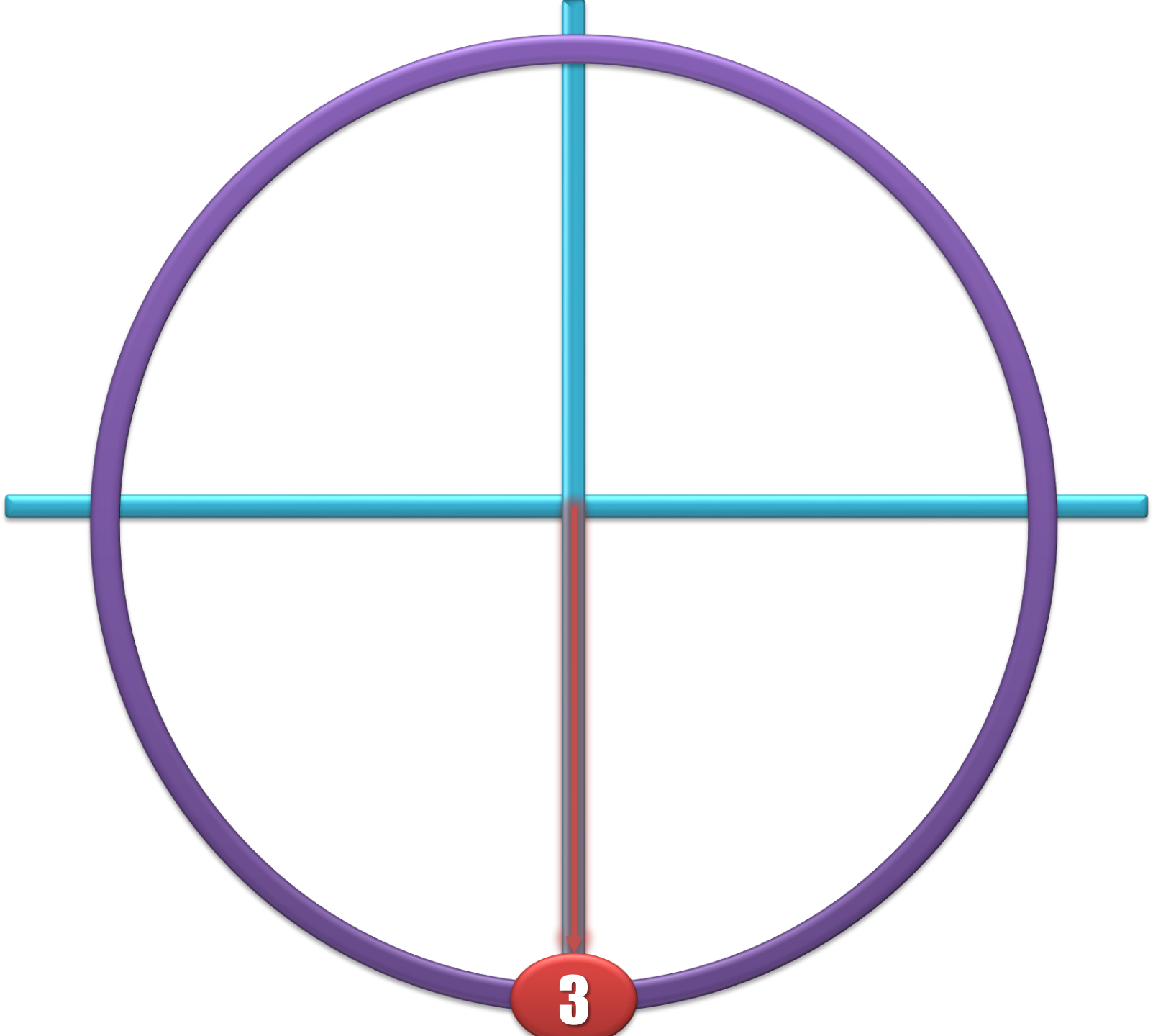


Fragment Processor



Shader Languages

- There are several languages for describing shaders:
  - *HLSL* the *High Level Shading Language*
    - Author: Microsoft
    - DirectX 8+
  - *Cg*
    - Author: NVIDIA
  - *GLSL*, the *OpenGL Shading Language*
    - Author: the Khronos Group
      - A self-sponsored group of industry affiliates (ATI, 3DLabs, etc)


3

GLSL

OpenGL Shading Language

http://glslsandbox.com/

GLSL Design

- GLSL was designed to:
  - Work well with OpenGL APIs
    - Shaders should be optional, not required.
    - Fit into the design model of

      "set the state first, then render the data in the context of the state"
  - Be extendable to future upgrades
  - Be hardware-independent
    - As a broad consortium, GLSL supports hardware-independence more than NVIDIA.
    - However, different platforms may have different compilers
  - Support inherent parallelization
  - Keep streamlined, small, simple programming pipeline

GLSL based on ANSI C

- The GLSL language is strongly based on ANSI C
- Some C++ added.
  - There is a preprocessor--**#define**
  - Basic types: int, float, bool
    - No double-precision float
  - Vectors and matrices are standard:
    - **vec2**, **mat2** = 2x2; **vec3**, **mat3** = 3x3; **vec4**, **mat4** = 4x4
  - Texture samplers:
    - **sampler1D**, **sampler2D**, etc are used to sample multidemensional textures
  - New instances are built with *constructors*, like in C++
  - Functions can be declared before they are defined
  - Operator overloading is supported.

Differences from C/C++

- Some differences from C/C++:
  - No pointers, strings, chars; no unions, enums;
  - No bytes, shorts, longs; no unsigned.
  - No switch() statements.
  - No implicit casting (type promotion):

    `float foo = 1;` // fails because you can't implicitly cast **int** to **float**.
  - Explicit type casts are done by constructor:

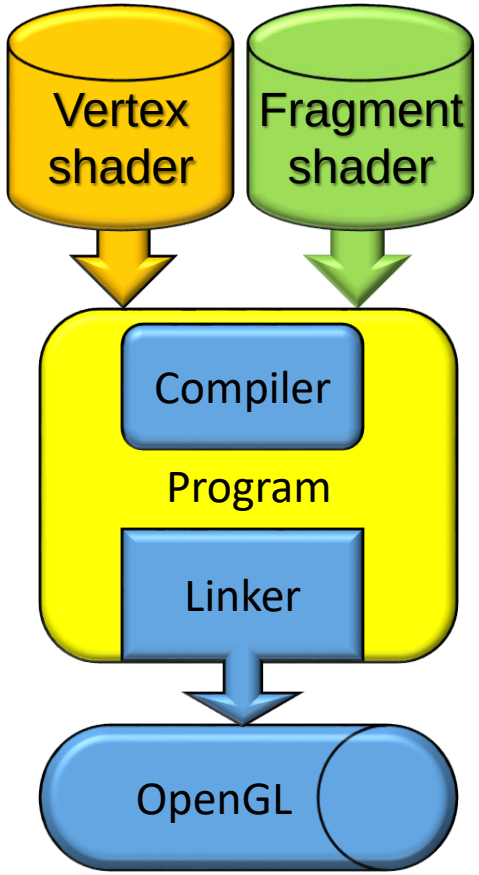
```
vec3 foo = vec3(1.0, 2.0, 3.0);
vec2 bar = vec2(foo);  // Drops foo.z
```

- Function parameters are labeled as **in** (default), **out**, or **inout**.
  - Functions are called by *value-return*, meaning that values are copied into and out of parameters at the start and end of calls.

Installing a GLSL API

To install and use a shader in OpenGL:

1. Create one or more empty *shader objects* with **glCreateShader**
2. Load source code, in text, into the shader with **glShaderSource**
3. Compile the shader with **glCompileShader**
   - Note: The compiler cannot detect every program that would cause a crash.
4. Create an empty *program object* with **glCreateProgram**.
5. Bind your shaders to the program with **glAttachShader**.
6. Link the program with **glLinkProgram**.
7. Register your program for use with **glUseProgram**.

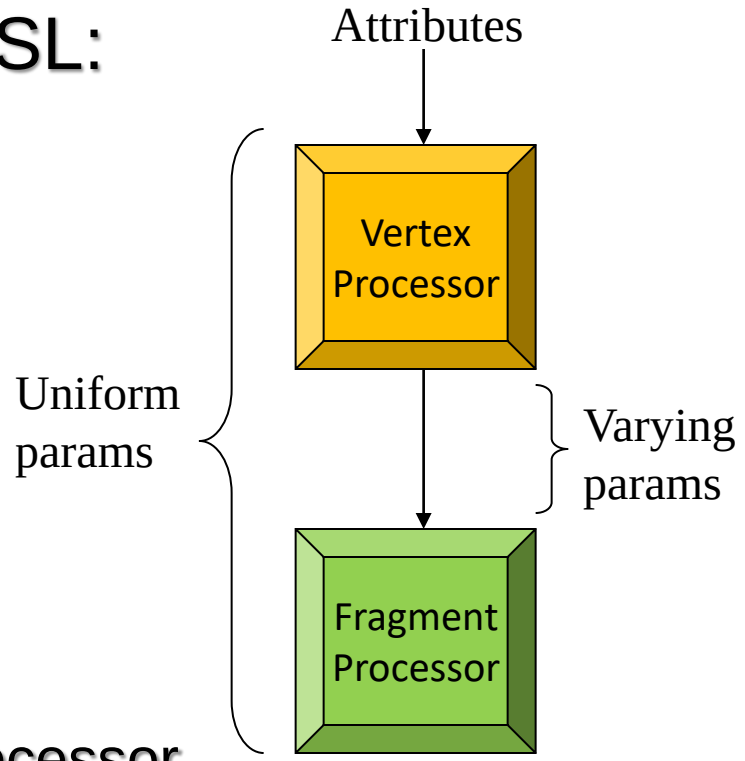


Example: See function initShaders() in webgl examples

GLSL Communication

Three types of *shader parameters* in GLSL:

- *Uniform parameters*
  - Set throughout execution
  - Example: surface color
- *Attribute parameters*
  - Set per vertex
  - Example: local tangent
- *Varying parameters*
  - Passed from vertex processor to fragment processor
  - Example: transformed normal



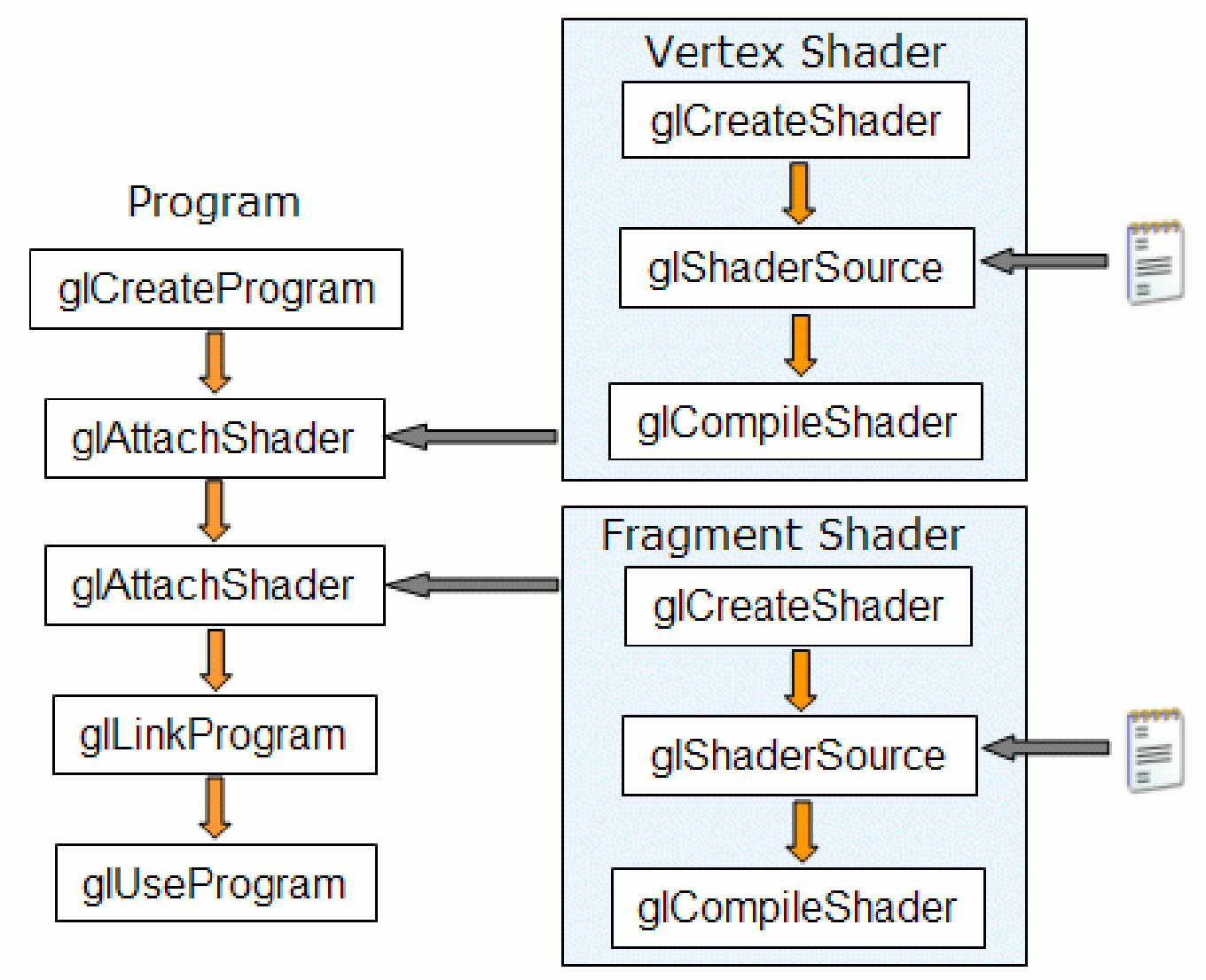
Compiling & Linking Shaders

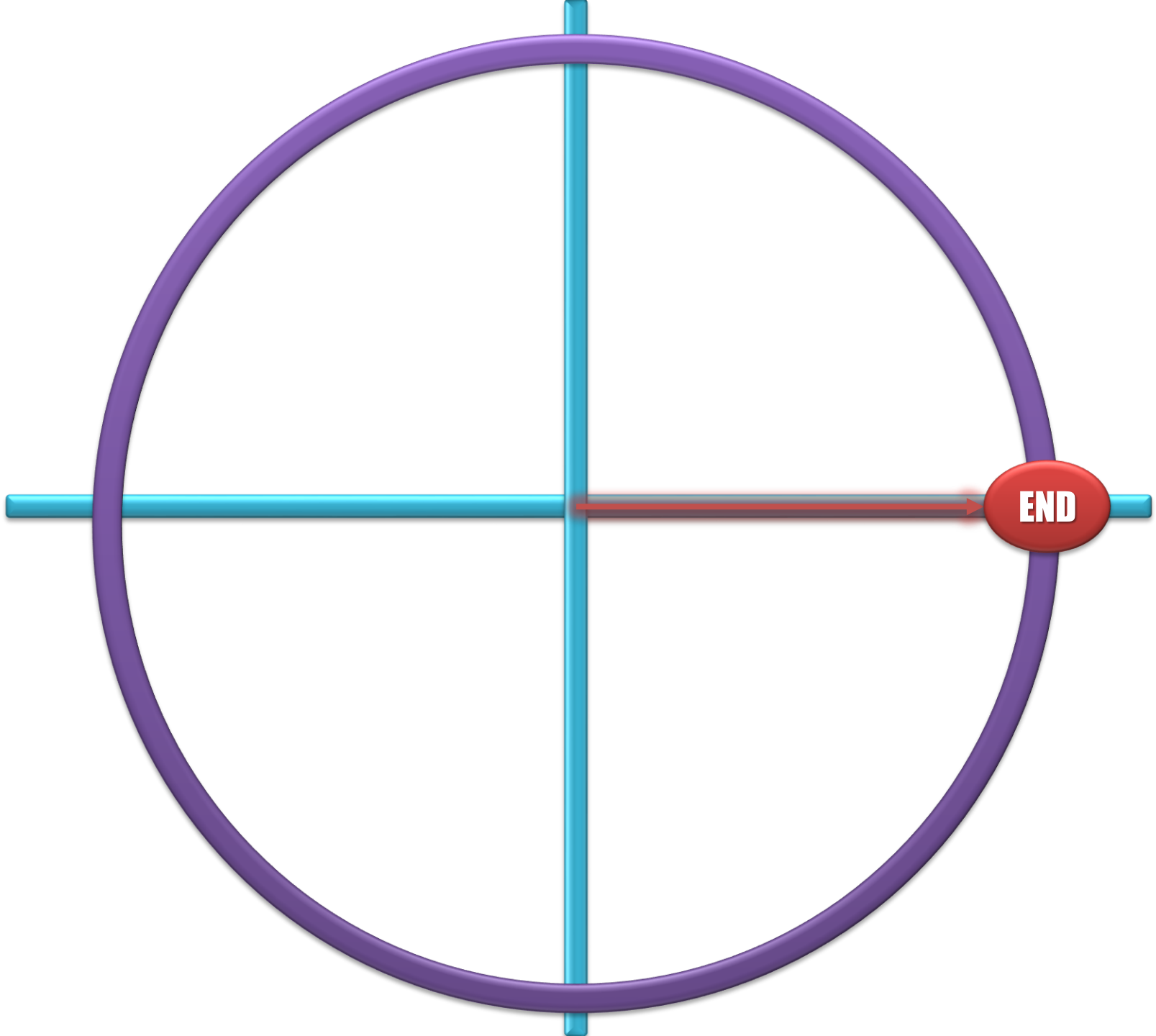
```
function initShaders() {
        var     fs_source = document.getElementById('shader-fs').innerHTML,
                vs_source = document.getElementById('shader-vs').innerHTML;
        vertexShader   = makeShader(vs_source, gl.VERTEX_SHADER);
        fragmentShader = makeShader(fs_source, gl.FRAGMENT_SHADER);
        glProgram = gl.createProgram();
        gl.attachShader(glProgram, vertexShader);    // attach the two shaders to the main program
        gl.attachShader(glProgram, fragmentShader);
        gl.linkProgram(glProgram);
        if (!gl.getProgramParameter(glProgram, gl.LINK_STATUS)) {
                alert("Unable to initialize the shader program.");
        }
        gl.useProgram(glProgram);
}
```

Utility function to make shader

```
function makeShader(src, type)
{
	// compile the vertex shader
	var shader = gl.createShader(type);
	gl.shaderSource(shader, src);
	gl.compileShader(shader);
	if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
		alert("Error compiling shader: " + gl.getShaderInfoLog(shader));
	}
	return shader;
}
```

In Summary


Vertex Shader
glCreateShader
glShaderSource
glCompileShader
Program
glCreateProgram
glAttachShader
glAttachShader
glLinkProgram
glUseProgram
Fragment Shader
glCreateShader
glShaderSource
glCompileShader


END