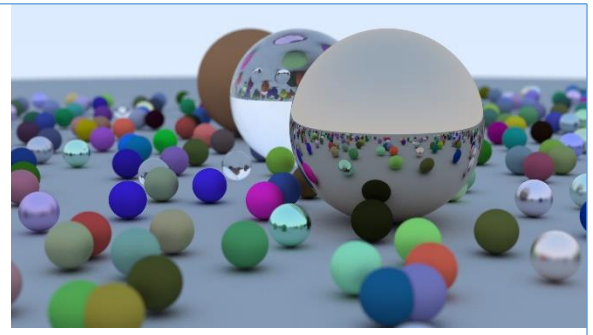
Comp4422



Computer Graphics

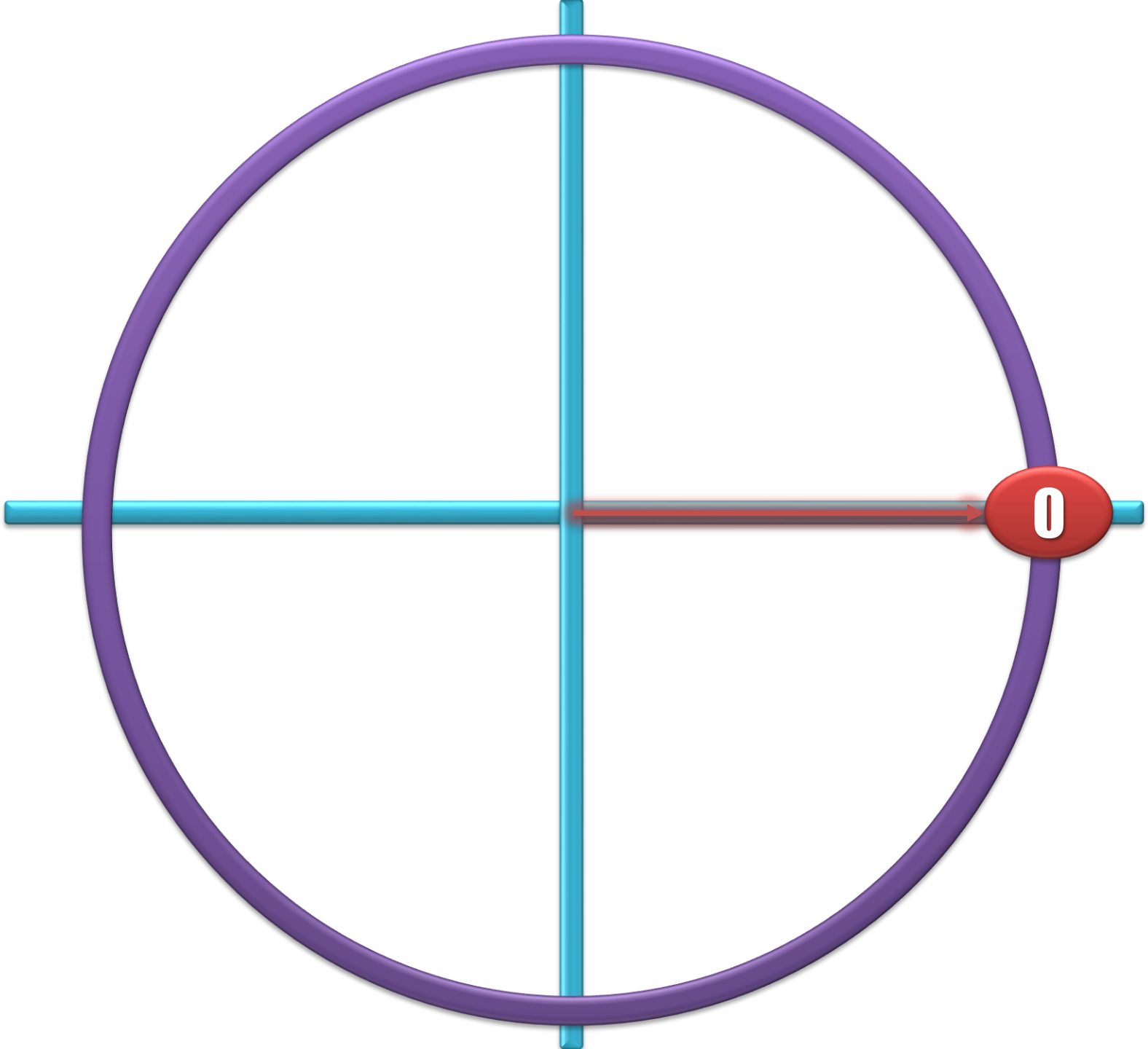
Lecture 05:

Data Flow

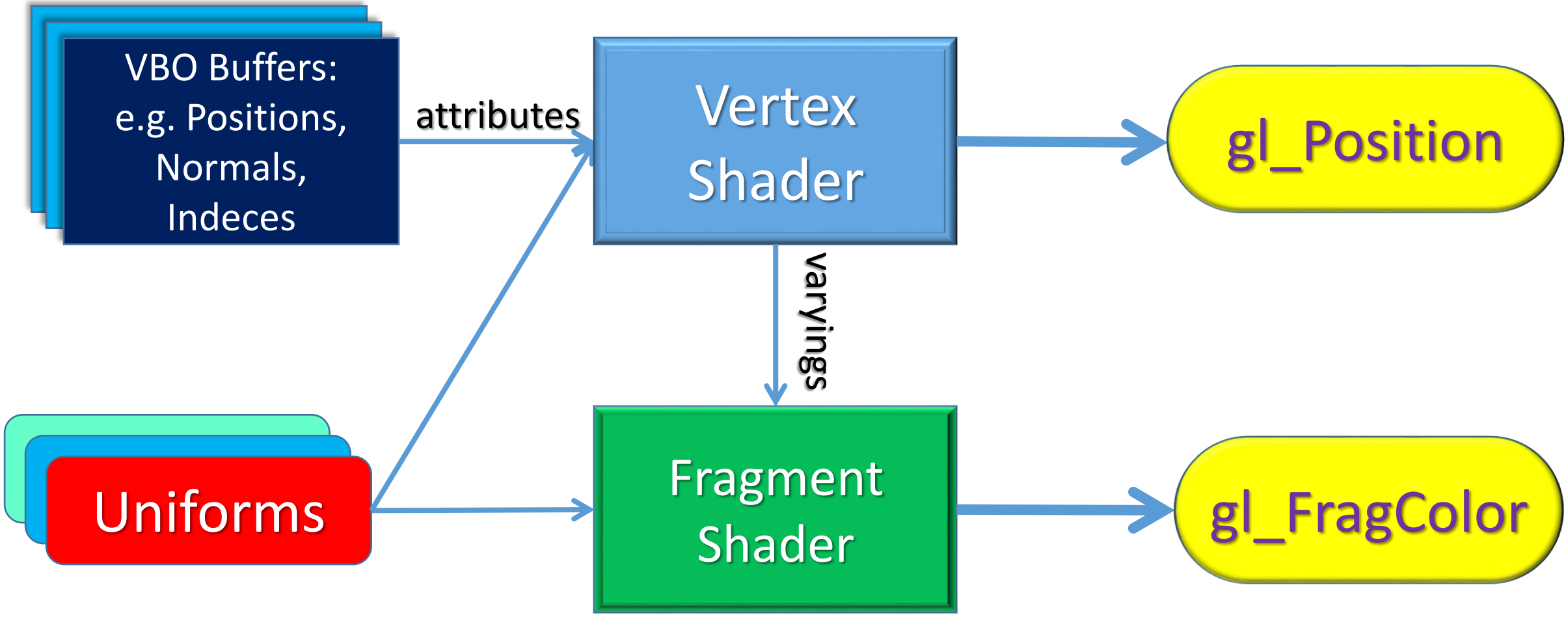
WebGL Data Flow

What you will learn?

- Separate the shaders into files
- Loading data from files
- Rendering data from files
- Linking multiple buffers together
- Complete data flow


0

WebGL Shader Model



Attributes and Uniforms

- VBOs: Vertex Buffer Objects
  - Store positions, normals, indices
- Vertices have attributes that can be passed down to shaders
- Uniforms: values that will be constant for each vertex during the processing of the VBOs
- Varyings: values that can be changed during the processing of the VBOs

WebGL Data Flow

1. Take vertex data and place it into vertex buffer objects (VBOs)
2. Stream the VBO data to the vertex shader (VS)
3. Send vertex index information using a call to:
   - drawArrays() with implicit index ordering, or
   - drawElements() and an index array

WebGL Data Flow

3. The VS runs, minimally setting the screen position of each vertex and optionally, performing additional calculations, which are then passed on to the fragment shader (FS).
4. Output data from VS continues down the fixed portion of the pipeline.

WebGL Data Flow

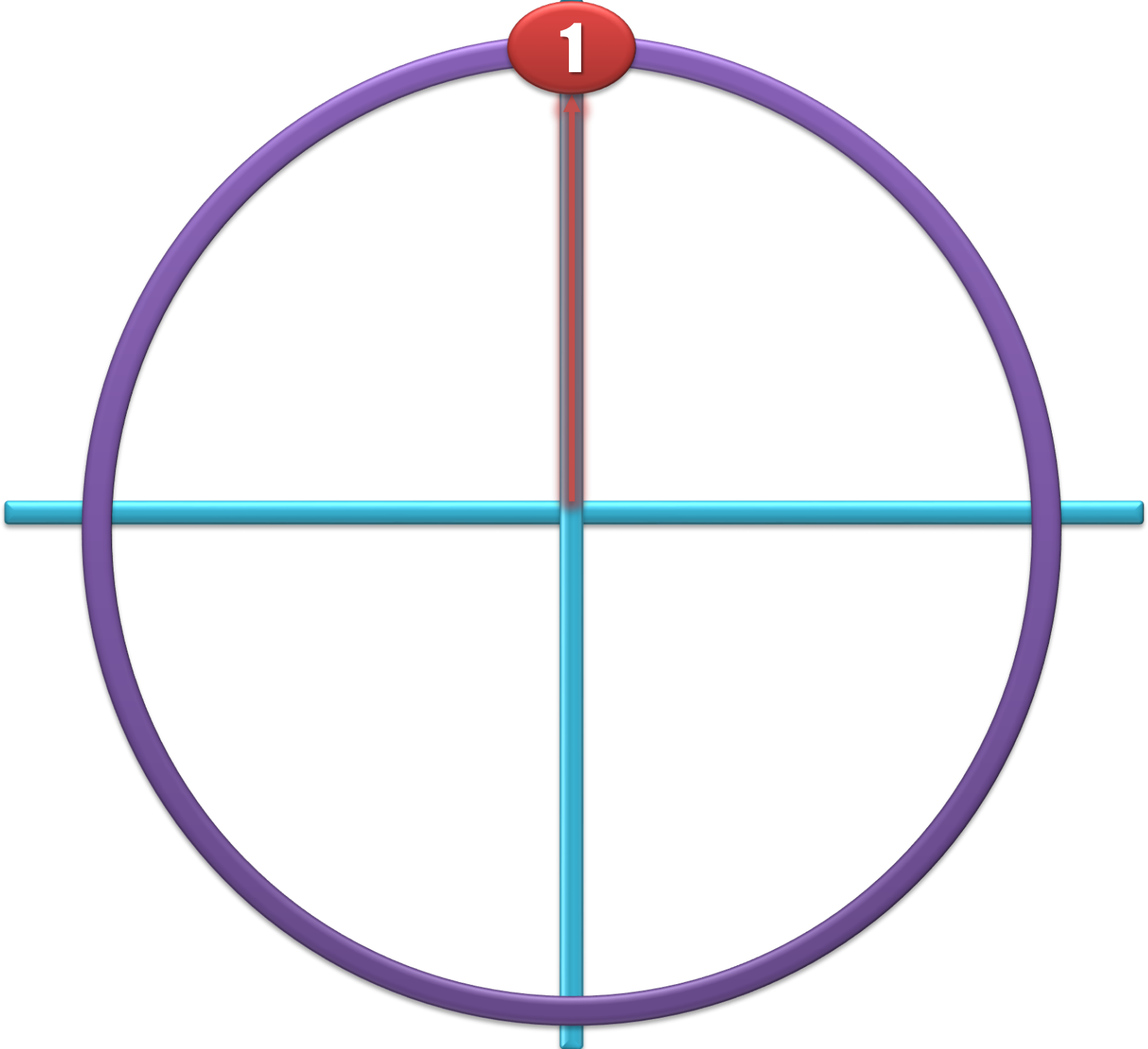
5. The GPU produces primitives using vertices and indices – primitive assembly
6. The rasterizer discards the primitive part that lies outside the viewport – clipping
7. The parts inside the viewport are then broken down into pixel fragments
8. Vertex values are interpolated across each fragment - rasterization

WebGL Data Flow

9. Fragments with these interpolated values are passed into FS
10. FS minimally sets the color values, but can also add texture and lighting operations
11. Fragments can be discarded or passed into the frame buffer

WebGL Data Flow

12. FS optionally uses the stencil buffer or the depth buffer to choose which fragments to write onto the final image
13. The image is passed into the drawing buffer or alternatively saved onto an off-screen buffer for later usage as texture data.


1

SHADER STRUCTURE

VS: Vertex Shader

- Responsible for all vertex coordinate transformations:
  - Final vertex position
  - Per vertex normal, texture, lighting, color
  - Passing values to the FS
  - Minimally sets gl_Position

Minimal VS

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

FS: Fragment Shader

- FS operates on each pixel
- A pixel is a rasterized portion of a primitive
- FS:
  - Computes the final color of each pixel
  - Performs texture lookup
  - Discards fragments
  - Minimally, needs to set the fragment color

Minimal FS

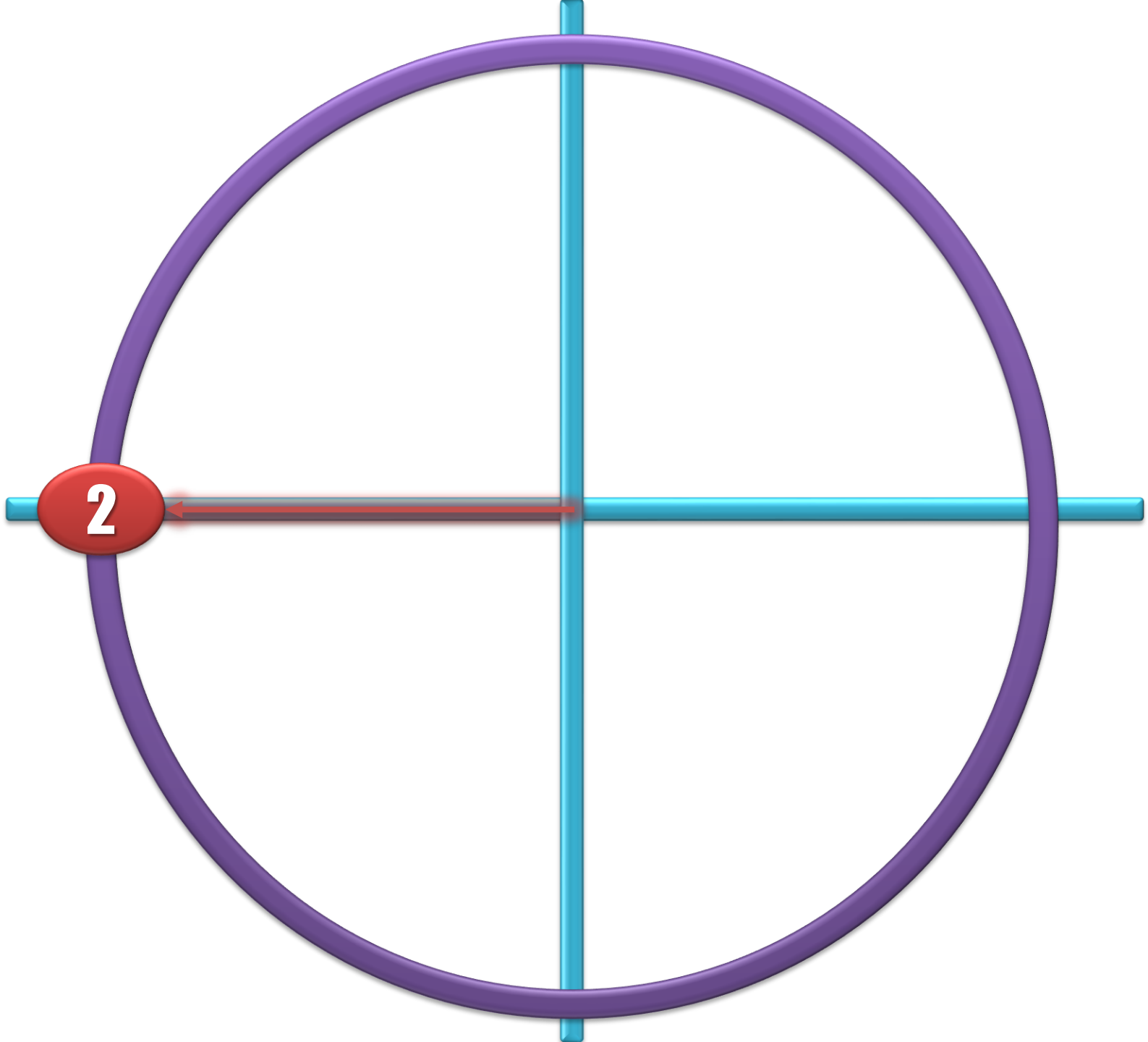
```
<script id="shader-fs" type="x-shader/x-fragment">
    void main(void) {
        gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
    }
</script>
```

Where is gl_Position defined?

Where is gl_FragColor defined?

Answer:

In the GLSL


2

MORE on GLSL

GLSL Specifications

- ASCII character files with \n and \lf
- Subset of C++ language
- Case sensitive
- There are no char or string types
- Variable can be any names except "**gl_...**"

GLSL Primitive Types/C++

void	specify a function with no return value
bool	boolean true or false
int	signed integers
float	floating point number

GLSL Extended Types

vec2	float vector of size 1 x 2
vec3	float vector of size 1 x 3
vec4	float vector of size 1 x 4

GLSL Extended Types

ivec2	integer vector of size 1 x 2
ivec3	integer vector of size 1 x 3
ivec4	integer vector of size 1 x 4
bvec2	bool vector of size 1 x 2
bvec3	bool vector of size 1 x 3
bvec4	bool vector of size 1 x 4

GLSL Extended Types

mat2	2 x 2 matrix of floating point values
mat3	3 x 3 matrix of floating point values
mat4	4 x 4 matrix of floating point values

sampler2D	2D mapped textures
samplerCube	Cube mapped textures

GLSL Qualifiers

storage	describe variable scope
parameter	used for function parameters
precision	min. precision requirements
invariant	read-only variables

GLSL Storage Qualifiers

[none]	default local variable storage
const	constant read-only values
uniform	constant across an entire primitive
attribute	VS per vertex information
varying	VS write, FS read qualifier

GLSL Parameter Qualifiers

[none]	default local variable storage
in	parameters passed into a function
out	parameters passed out of a function
inout	initialized parms that will also be passed out of a function

Ex. Parameter Qualifiers

```
vec3 a = (0, 1, 0);
vec3 c;
void myFunction(a, out c) {
        c = a * 2;
}
```

GLSL Precision Qualifiers

highp	high precision (i.e. 64-bit floats)
mediump	medium precision (i.e. 32-bit floats)
lowp	low precision (i.e. 32-bit floats)

GLSL Order of Qualifiers

For variables:

1. invariant
2. storage
3. precision

Example:

```
invariant uniform highp mat4 m;
```

For parameters:

1. storage
2. parameter
3. precision

Example:

```
void myFunc(const in lowp c) { ; }
```

GLSL built-in shader vars

Variable	Type	Description	Used in	Input/Output
gl_Position	vec4	Vertex position	VS	output
gl_PointSize	float	Point size	FS	output
gl_FragCoord	vec4	Fragment position in frame buffer	FS	input
gl_FrontFacing	bool	True if fragment is part of a front facing primitive	FS	input
gl_PointCoord	vec2	Fragment position in a point	FS	input
gl_FragColor	vec4	Final fragment color	FS	output
gl_FragData[n]	**vec4**	**Final fragment color for a color attachment, n**	**FS**	**output**

GLSL Built-in Constants

```
const mediump int gl_MaxVertexAttribs = 8;
const mediump int gl_MaxVertexUniformVectors = 128;
const mediump int gl_MaxVaryingVectors = 8;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits=8;
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```

GLSL Vectors

- Coordinate positions and normals {x,y,z,w}
- Colors {r,g,b,a}
- Textures {s,t,p,q}
- Example:
  - vec4 green = vec4(0.0, 1.0, 0.0, 1.0);
  - vec4 blue = vec4(0.0, 0.0, 1.0, 1.0);
  - final_color = vec4(green.rg, blue.ba);

GLSL Vector Matrix Ops

```
vec4 a = vec2(1.0, 2.0);
vec2 b = vec2(3.0, 4.0);
mat2 m = mat2(a, b); // column major
// produces a matrix with values:
// [ 1.0  3.0 ]
// [ 2.0  4.0 ]
//
// and stored in memory as [1.0 2.0 3.0 4.0]
```

GLSL Matrices

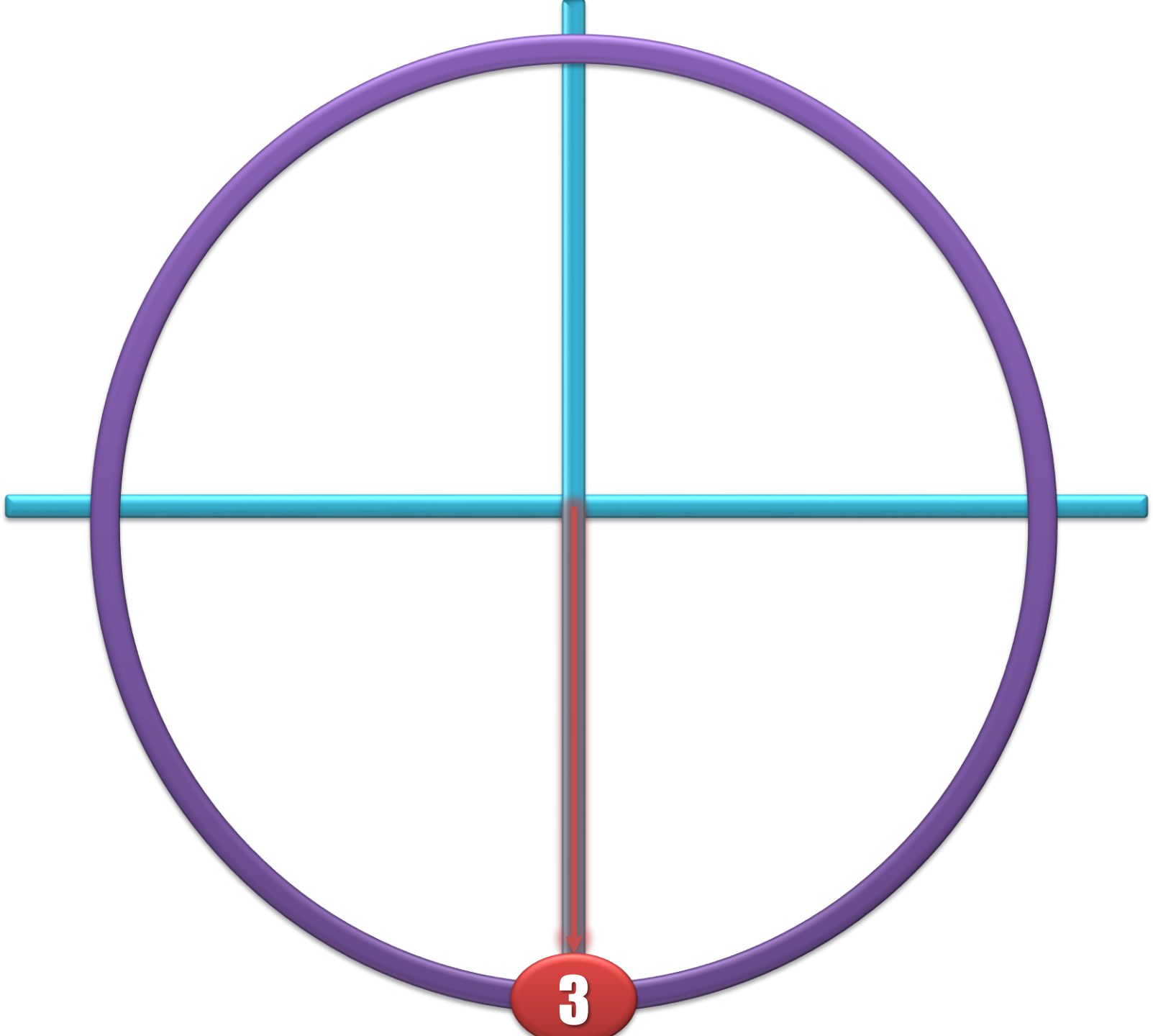
```
mat2 m = mat2(1.0, 0.0, 0.0, 1.0);
mat2 m = mat2(1.0); // same 2x2 identity
// vector functions
vec3 u, v, w;
w = cross(u, v);
w = dot(u, v);
```

GLSL Example

```
varying vec2 uv;
void main(void)
{
        vec3 blue = vec3(0.0, 0.0, 1.0);
        vec3 green = vec3(0.0, 1.0, 0.0);
        gl_FragColor = vec4(mix(blue, green, uv.s), 1.0);
}
```

GLSL Example

```
varying vec2 uv;
void main(void) {
        float repetition = 15.0;
        vec3 black = vec3(0.0, 0.0, 0.0);
        vec3 white = vec3(1.0, 1.0, 1.0);
        bool color = (mod(uv.s * repetition, 1.0) > 0.5);
        if(color){
                gl_FragColor = vec4(black, 1.0);
        }else{
                gl_FragColor = vec4(white, 1.0);
        }
}
```


3

Loading Data

Loading Shaders

1. From HTML DOM scripts
2. From files:
   a) Via XMLHttpRequestObject
   b) Via AJAX jQuery

XMLHttpRequestObject

- Synchronous:
  - block and wait for each loading to finish
- Asynchronous:
  - No need to wait for each loading
  - However, we need proper order of events

VS:Synchronous Request

```
var fs_source = null, vs_source = null;
var xhr = new XMLHttpRequest();
xhr.open('GET', './shader.vs', false); // synchronous request requires a false third parameter
xhr.overrideMimeType('text/xml');     // overriding the mime type is required
xhr.send(null);
if (xhr.readyState == xhr.DONE) {
        if (xhr.status === 200) {
                vs_source = xhr.responseXML.documentElement.firstChild.data;
        } else {
                console. error("Error: " + xhr.statusText);
        }
}
```

FS:Synchronous Request

```
xhr.open('GET', './shader.fs', false);
xhr.send(null);
if (xhr.readyState == xhr.DONE) {
        if(xhr.status === 200) {
                fs_source = xhr.responseXML.documentElement.firstChild.data;
        } else {
                console. error("Error: " + xhr.statusText);
        }
}
```

Using AJAX jQuery

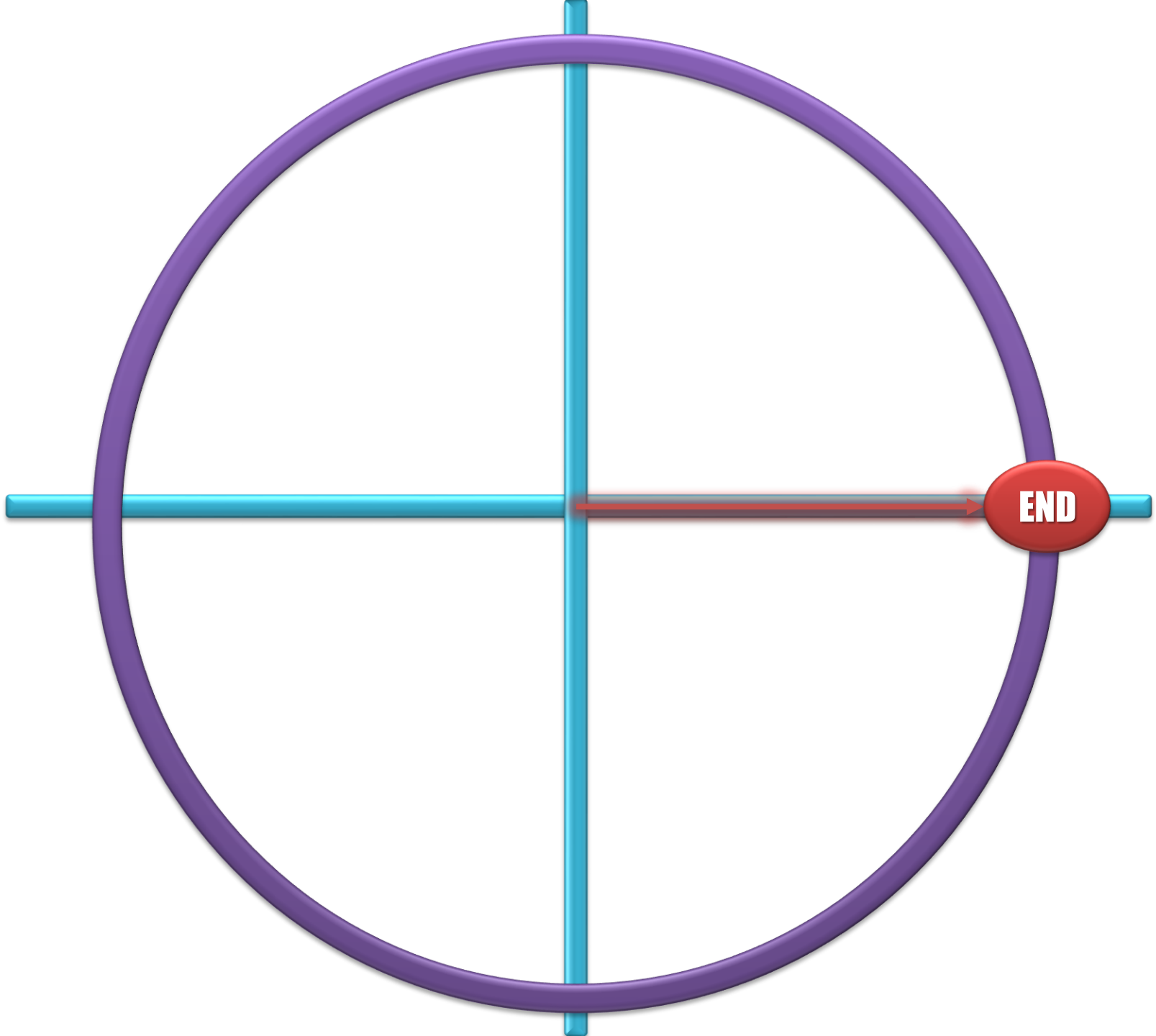
- Using JavaScript API is better, why?
  - low level code is hidden from us
  - it is actually more browser compatible
  - but, ajax requires jQuery library link
    http://code.jquery.com/jquery-latest.js

VS: Using AJAX jQuery

```
$.ajax({
        async: false,
        url: './shader.vs',
        success: function (data) {
                vs_source = data.firstChild.textContent;
        },
        dataType: 'xml'
});
```

FS: Using AJAX jQuery

```
$.ajax({
        async: false,
        url: './shader.fs',
        success: function (data) {
                fs_source = data.firstChild.textContent;
        },
        dataType: 'xml'
});
```


END