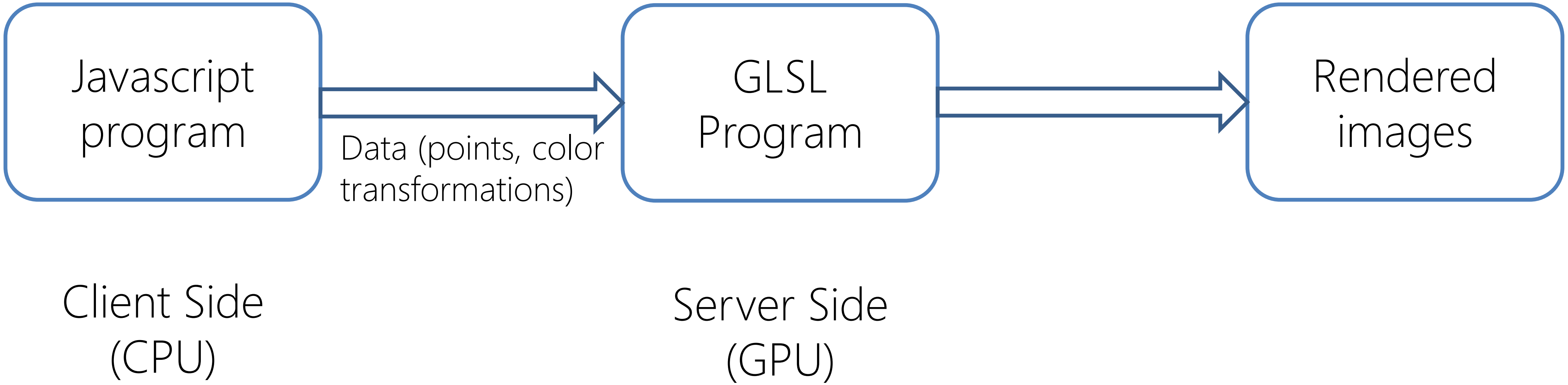
POLITECNICO
MILANO 1863

WebGL Fundamentals

Computer Graphics 2021

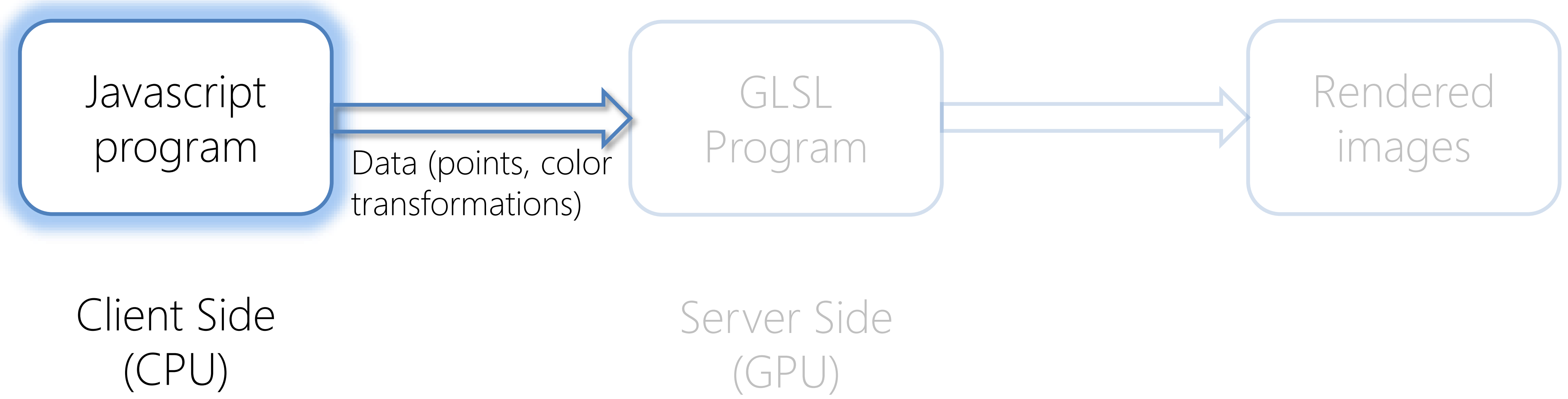
Erica Stella (erica.stella@polimi.it)

WebGL pipeline



WebGL pipeline

Today we assume that the GLSL program is already available, so that we can focus on the first step of the pipeline



HTML5 <canvas> element

WebGL takes advantage of the ***<canvas>*** element provided by HTML5:

- Allows the scriptable rendering of graphics within the browser
- Supports different APIs (Including of course WebGL)
- The area within a canvas element can be manipulated with the **JavaScript** language
- Provides the default **frameBuffer**: a region of physical memory in the GPU used to temporarily store an image for rendering

```html
<canvas id="my-canvas" width="600" height="400">
       Write here something to show if browser does not support the
       HTML5 canvas element.
</canvas>
```

WebGL Context

From the *canvas*, the first element a program must create to draw graphics is a **context**:

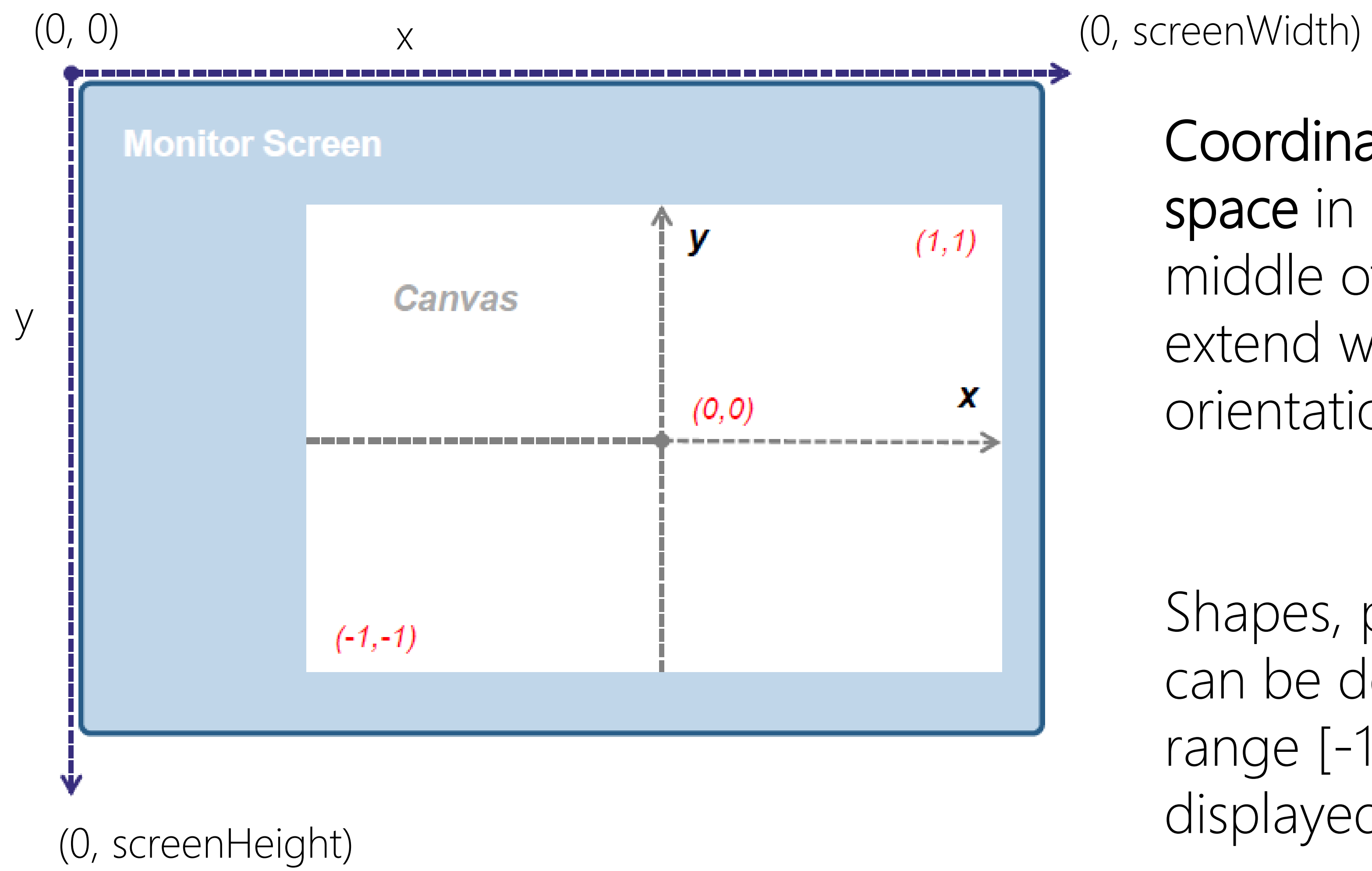
- Provides an internal data structure for keeping track of state settings and operations.
- Can be requested using the **javascript** function on the canvas element:

```
.getContext(contextId, *args…)
```

- The first argument is the context name such as `'2d'` or `'webgl2'` and the second is optional.
- Returns an object that exposes the API we want to use for drawing on the canvas.
  - We can access WebGL functions and attributes through the context

```
<script>
        var canvas = document.getElementById(”my-canvas”);
        var context = canvas.getContext(“webgl2”);
</script>
```

WebGL Coordinates System

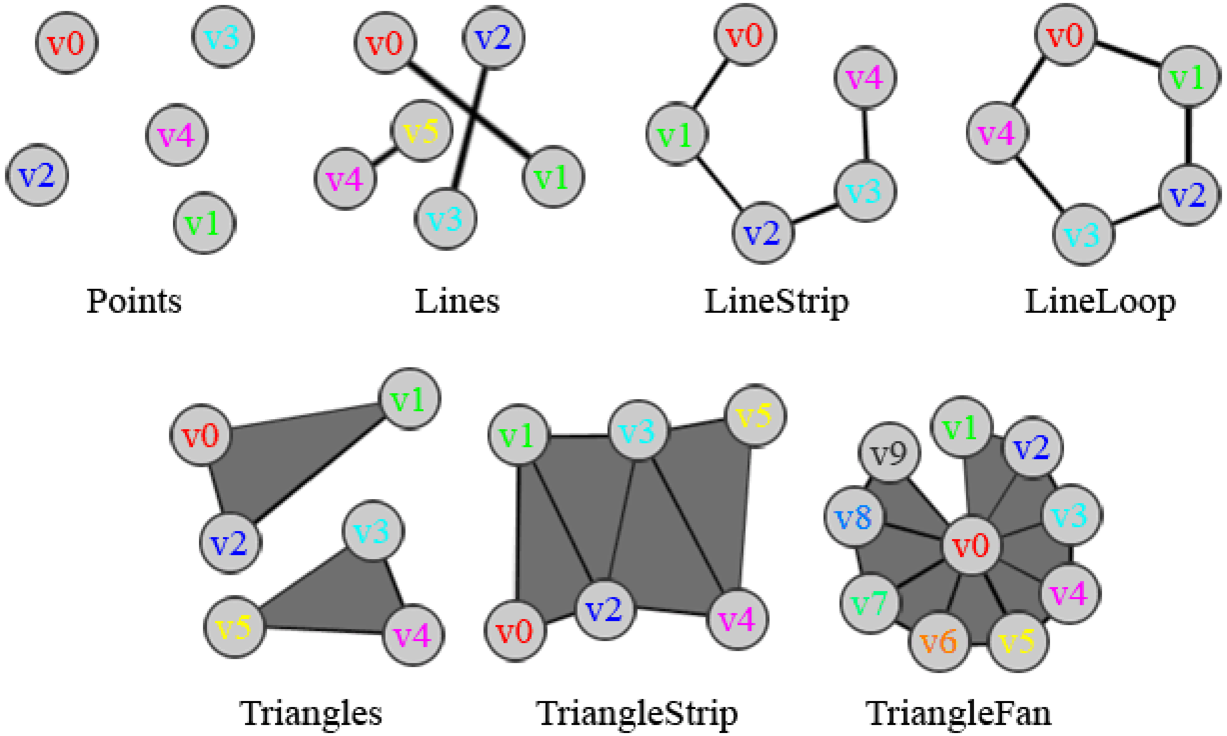


Coordinates of the drawing space in WebGL start in the middle of the canvas object and extend with Cartesian orientation.

Shapes, points and primitives can be defined outside the range [-1,1] but they are not displayed.

WebGL Primitives

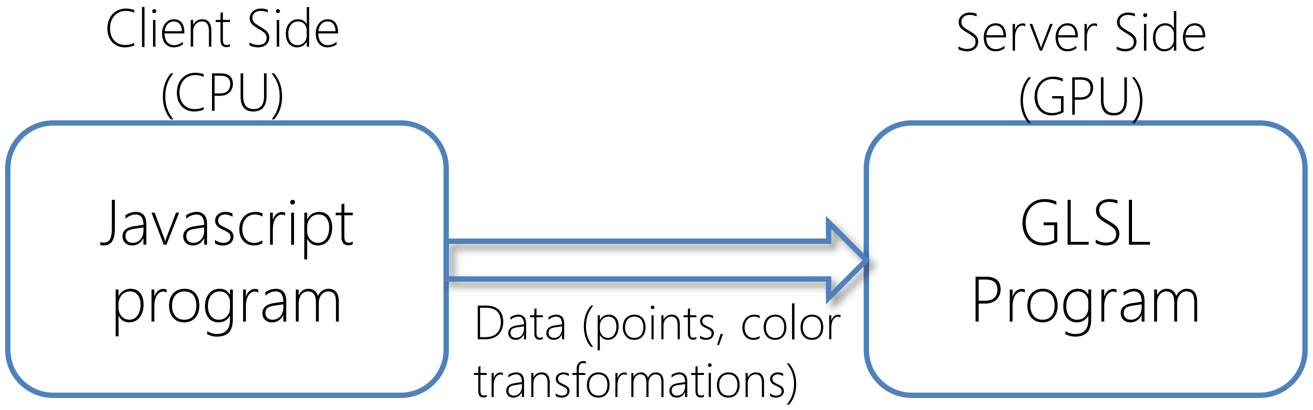
- A **primitive** in WebGL is simply a collection of <u>vertices</u>, that can be assembled in several ways.
- All primitives in WebGL are defined as one of the following:
  - POINTS, LINES, LINE_STRIP, LINE_LOOP, TRIANGLES, TRIANGLE_STRIP, and TRIANGLE_FAN
- They specify how we want to use the vertices
- All WebGL can do is to visualize these primitives and apply colors and textures to them.



Vertex Buffer Objects (VBO) and Vertex Array Objects (VAO)

Before drawing anything on the screen:

All data associated to vertices needs to be streamed from the JavaScript API to the GPU



To pass vertex data (`attributes` in GLSL), in WebGL you have to create **vertex buffer objects (VBOs)** that will hold vertex attributes such as position, normals, texture coordinates, etc…

Vertex Buffer objects can be grouped by means of **vertex array objects (VAO)** *(we will see them in later lessons…)*

WebGL: Draw a triangle pt1: VBO

- Define an array holding the vertices of our shape (in our case, 3 vertices each with x,y coordinates):

```
var positions = [ -0.5, -0.5, 0.5, -0.5, 0.0, 0.5 ];
```

- Create a *buffer*, which is a block of memory that can be written to or read from. The *handle* to it is stored in the VBO variable *positionBuffer*:

```
var positionBuffer = gl.createBuffer();
```

- The VBO buffer is set as the active one, and the type of data it will hold specified
  - ARRAY_BUFFER means it will hold vertex coords

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
```

- Vertex data are finally placed inside the buffer that is now ready to be used. WebGL implicitly uses the currently bound buffer as the receiving buffer.

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

WebGL: Draw a triangle pt2: towards the GLSL program

- In WebGL to specify any rendering operations you must use GLSL.
- Assume the GLSL program defines a variable: `vec4 a_position` representing the position of the primitives
- We retrieve the handle to the *location* of GPU memory where the GLSL program expects to find its input data:

```
var positionAttributeLocation = gl.getAttribLocation(program, "a_position");
```

- Activate the communication from Client to the Server input location specified in positionAttributeLocation

```
gl.enableVertexAttribArray(positionAttributeLocation);
```

- Remember: attributes CANNOT be used unless enabled

WebGL: Draw a triangle pt2: towards the GLSL program

- Now we let the GLSL program know how to interpret our data.

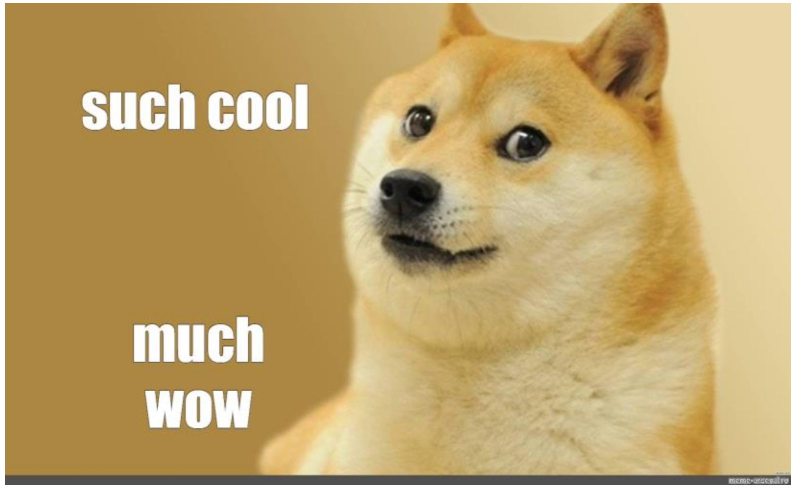
```
var size = 2;            // how many values define the vertex [2: (x,y)]
var normalize = false; // don't normalize the data between 0 and 1
var stride = 0;          // 0 = move forward size * sizeof(type) each iteration to
get the next position
var offset = 0;          // start at the beginning of the buffer
gl.vertexAttribPointer(positionAttributeLocation, size, gl.FLOAT, normalize,
                      stride, offset);
```

- This step binds the **currently-bound VBO** to the specified attribute!
  - Now the attribute in *positionAttributeLocation* pulls data from it

WebGL: Draw a triangle pt3: at long last, we draw it

- Let's draw our triangle

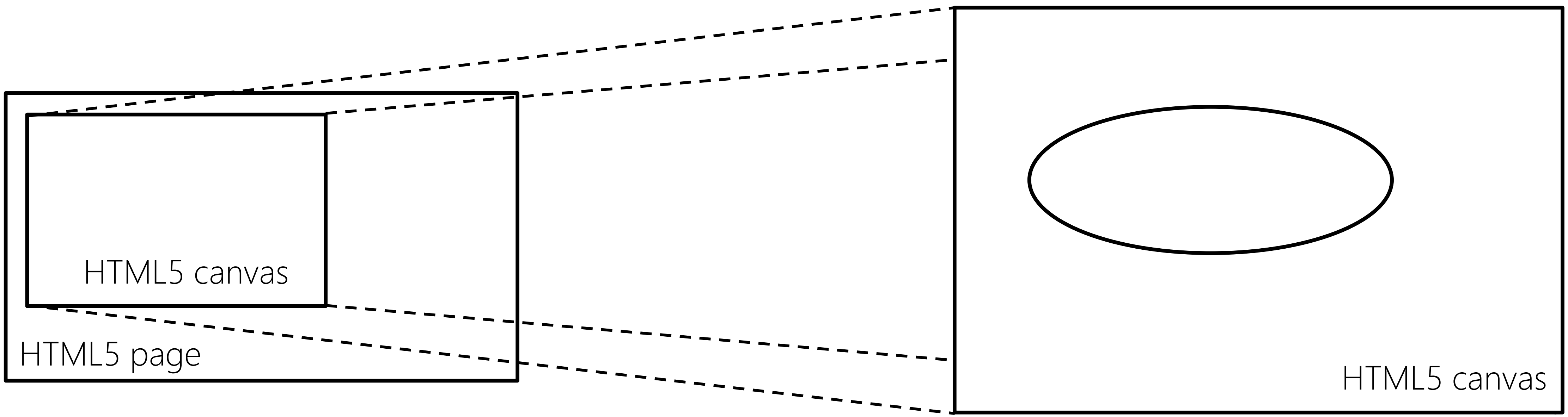
```
var primitiveType = gl.TRIANGLES; //primitive to be used
var offset = 0; //index of array elements to start from
var count = 3; //number of vertices to draw
gl.drawArrays(primitiveType, offset, count);
```



WebGL: aspect ratio

When the size of a <canvas> element is set, its width and height values may not be equal.

In this case, the [-1,1] coordinates of the canvas are mapped non proportionally to screen coordinates:



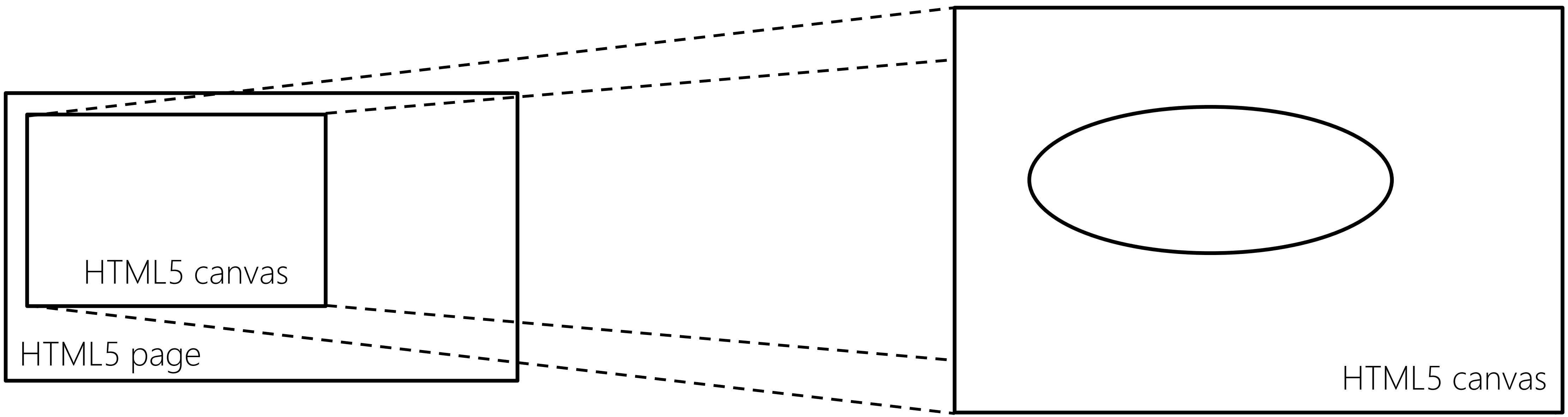
The intended circle looks more like an ellipse…

WebGL: aspect ratio

The solution is to consider the Aspect Ratio of the canvas that describes how the canvas width compares to the canvas height.

Aspect Ratio *a* = <canvas>.width / <canvas>.height

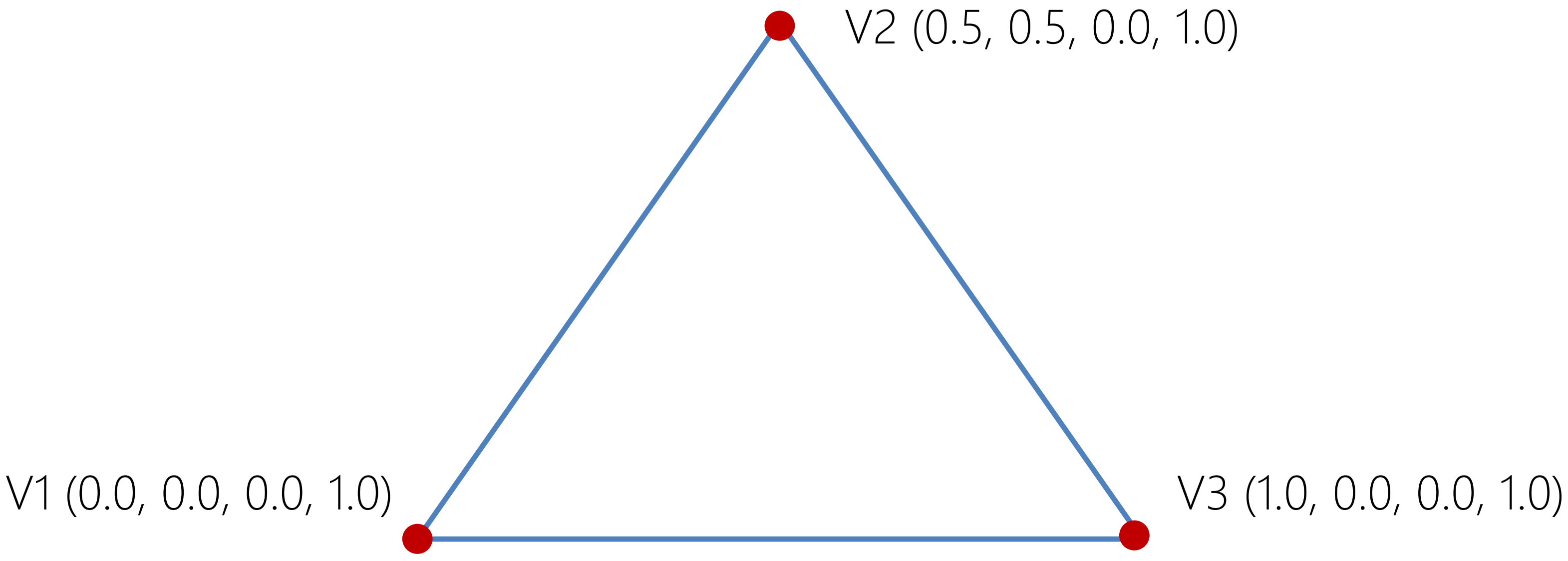
If the *y coordinate* of a point is *multiplied* by the aspect ratio or the *x coordinate* is *divided* by it, the shape will result as intended.



Drawing with Indexing

Up to now we drew a triangle by specifying the position of each vertex

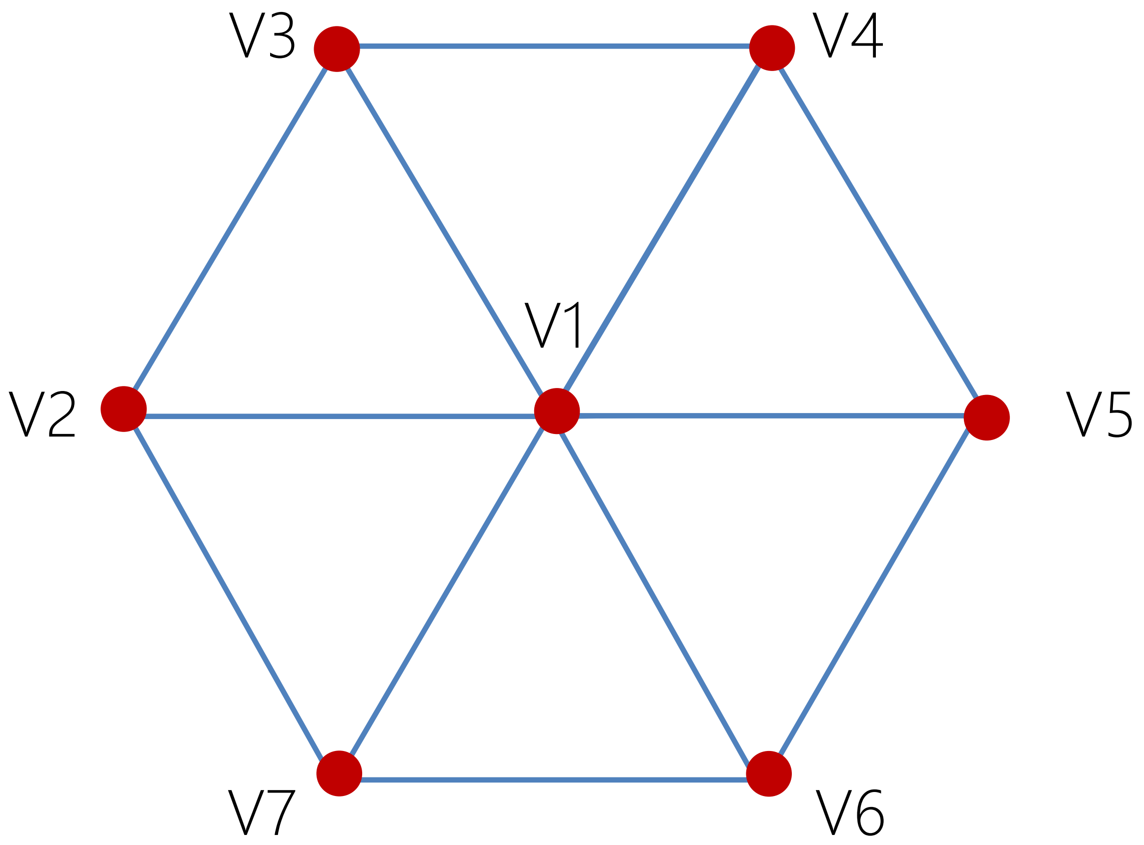
- draw V1, V2, V3



Drawing with Indexing

…but what if we have to draw?

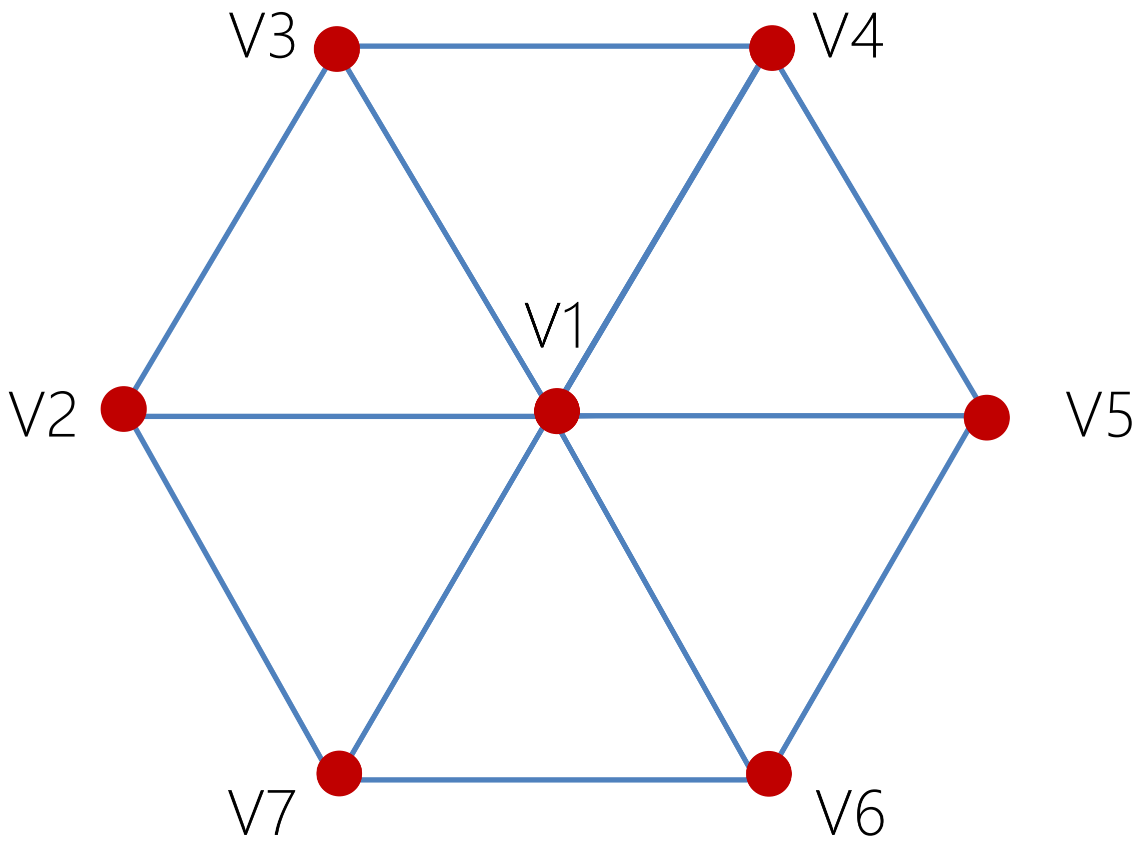
- draw V1, V2, V3, V1, V3, V4, V1, V4, V5, V1, V5, V6, V1, V6, V7, V1, V7, V2



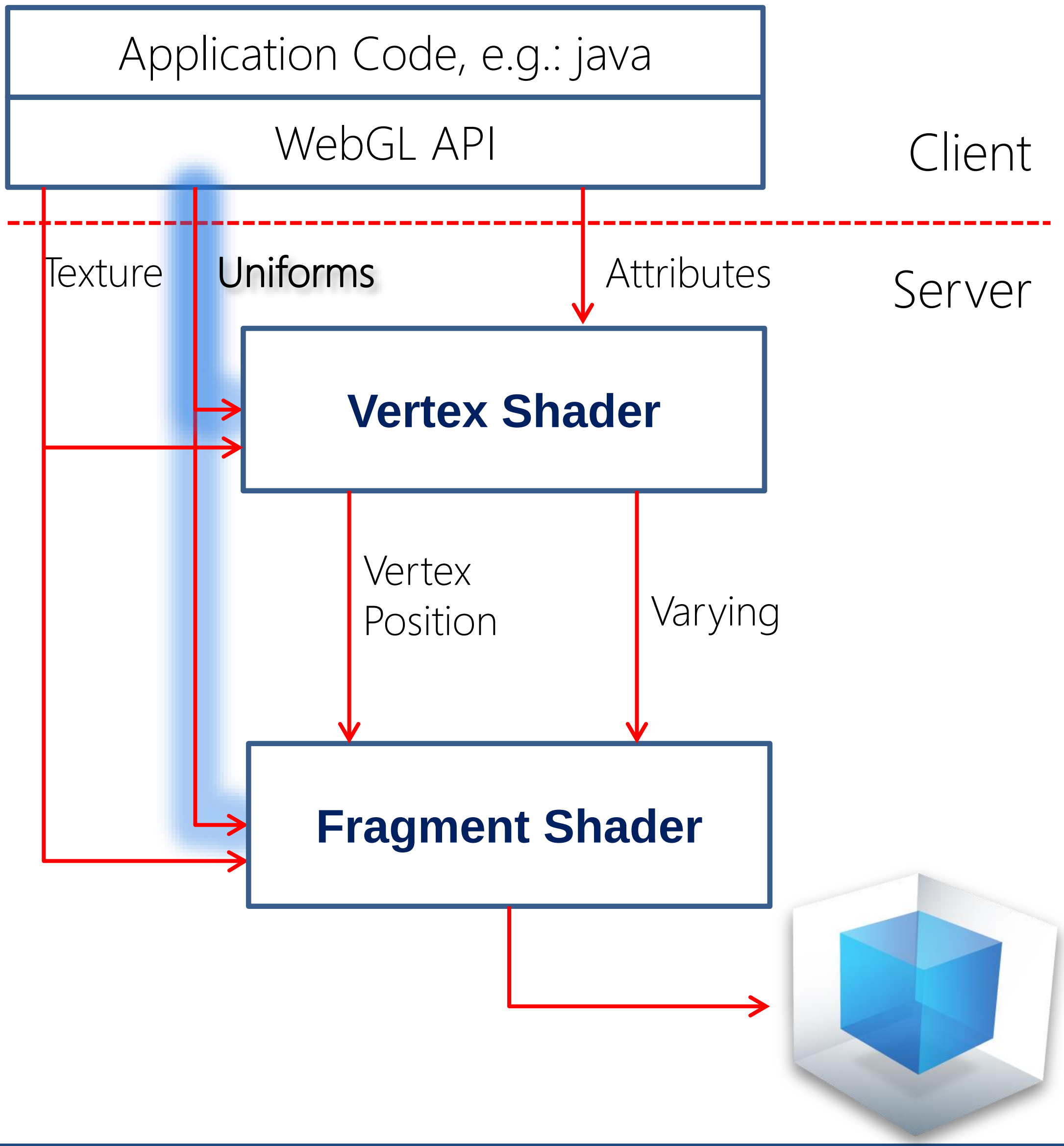
Drawing with Indexing

It is much more convenient to define the attributes for V1, V2, V3, V4, V5, V6, V7

draw 1,2,3,1,3,4,1,4,5,1,5,6,1,6,7,1,7,2



Passing uniform variables



Passing uniform variables

- To feed uniform variables to the rendering pipeline, you need to retrieve the location of the variable from the GLSL program

```
var colorLoc = gl.getUniformLocation(program, "u_color"); //uniform vec4 u_color in GLSL
```

- Then, the function **gl.uniform[1234][fi][v](location, value/values)** is used
  - The number chosen between [1234] corresponds to the number of values you're passing
  - [fi] distinguishes between float and integers to be passed
  - [v] is used to specify you're passing and array with all the values instead of passing them singularly
- This function is NOT used for passing matrices

```
var color = [0.5,0.6,1.0,1];
gl.uniform4fv(colorLoc, color);
```

- https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform
- gl.uniform[1234][fi][v]() must be used *before* the draw call but *after* gl.useProgram()

Drawing with Indexing

```
[...]
//Set up the VBO for the vertices
var positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
gl.enableVertexAttribArray(positionAttributeLocation);
gl.vertexAttribPointer(positionAttributeLocation, 4, gl.FLOAT, false, 0, 0);

//Create the buffer that will hold the indices and send the data
var indices = [0,1,2];
var indexBuffer = gl.createBuffer();
//Here the buffer must be gl.ELEMENT_ARRAY_BUFFER to specify it contains indices
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);

//bind index buffer to be sure that is the current active
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
//drawElements uses the indices to draw the primitives
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0 );
```

Compiling and Linking the code

A shader must be **compiled** and **linked** by calling the proper functions in the **client code**.

1) A shader object must be created for both VS and FS using the function `Object createShader(enum Type)` and its handler stored in a variable.

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

Compiling and Linking the code

2) Shaders must be loaded with a source string using the function:
`void shaderSource(Object shader, string source)`

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

GLSL: Compiling and Linking the code

3) At this point shaders can be compiled using the function

`void compileShader(Object shader)`

NB:
This can generate compilation errors that must be intercepted explicitly or they go silent

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

GLSL: Compiling and Linking the code

4) We then need a **shader program** object that will contain the complete pipeline definition. This is achieved with function:
`Object createProgram()`
that returns a pointer we can later use to operate with it.

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

Compiling and Linking the code

5) The compiled shaders must be attached to this program

`void attachShader (Object program, Object shader)`

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

Compiling and Linking the code

6) Now the program can be linked using
`void linkProgram (Object program)`

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fragmentShaderSource);
gl.compileShader(fragmentShader);

var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
```

OpenGL ES Workflow Summary