# Introduction to WebGL

## Overview

**WebGL is often thought of as a 3D API**

- In reality WebGL is just a rasterization engine.
- It draws points, lines, and triangles based on code you supply.

## Overview

**WebGL runs on the GPU on your computer.**

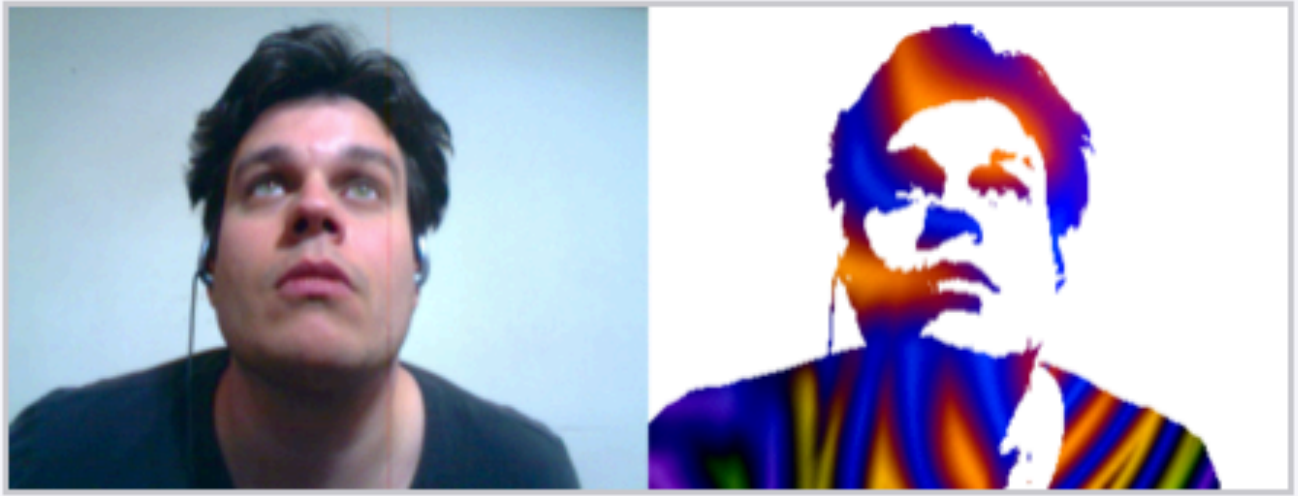
- As such you need to provide the code that runs on that GPU.
- You provide that code in the form of pairs of functions (called shaders).
- The shaders are named **vertex shader** and **fragment shader** and they are written in a very strictly typed C/C++ like language called GLSL. (GL Shader Language).
- Paired together they are called a program.

# What is a shader?

In computer graphics, a shader is a type of computer program that was originally used for shading (the production of appropriate levels of light, darkness, and color within an image) but which now performs a variety of specialized functions in various fields of computer graphics special effects or does video post-processing unrelated to shading, or even functions unrelated to graphics at all.

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

# Shaders



Shaders can also be used for special effects. An example of a digital photograph from a webcam unshaded on the left, and the same image with a special effects shader applied on the right which replaces all light areas of the image with white and the dark areas with a brightly colored texture.

## Shaders

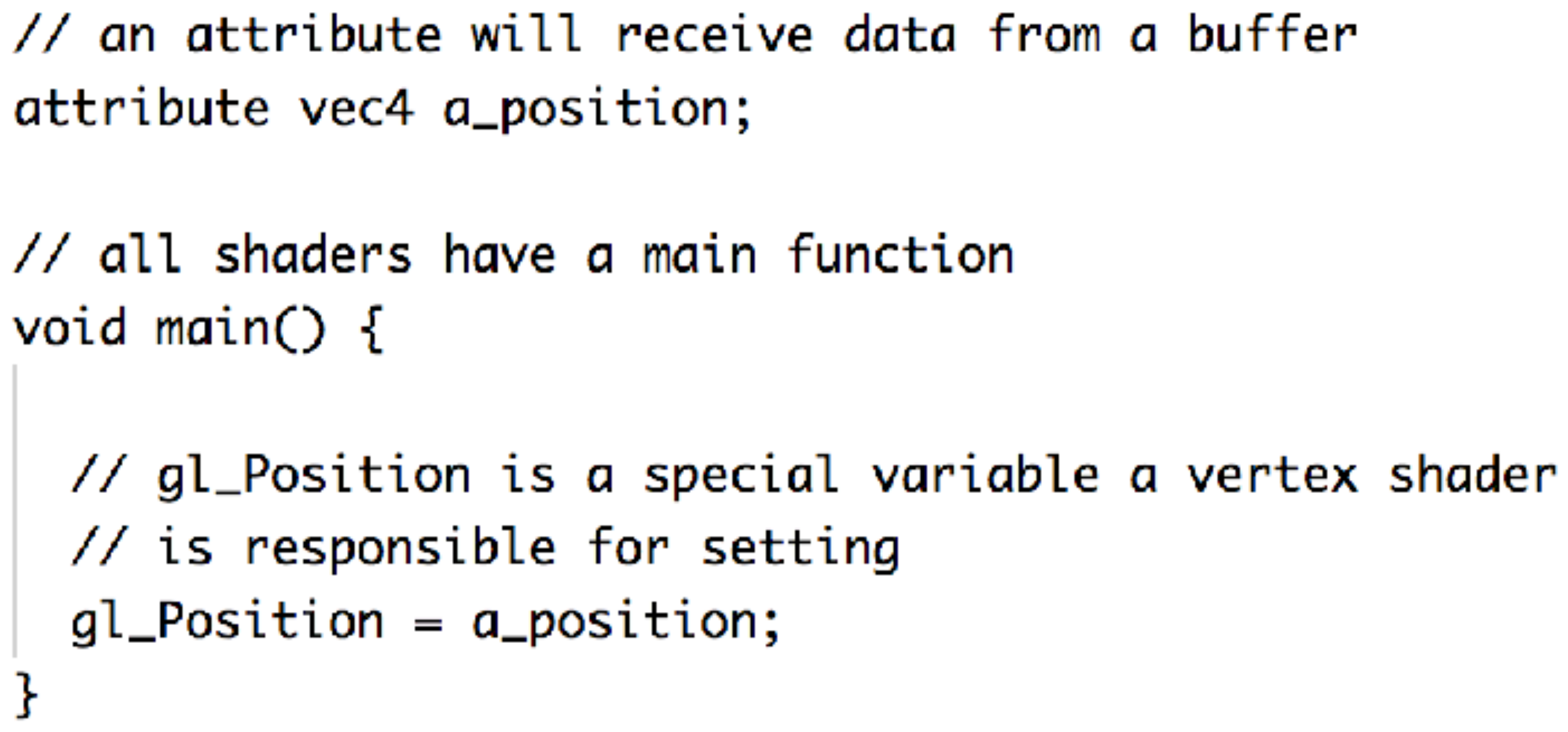
### Vertex Shader

- A vertex shader's job is to compute vertex positions.
  - A vertex (plural vertices) in computer graphics is a data structure that describes certain attributes, like the position of a point in 2D or 3D space, at multiple points on a surface.
- Based on the positions output by the function, WebGL can then rasterize various kinds of primitives including points, lines, or triangles.
- When rasterizing these primitives it calls a second user supplied function called a fragment shader.

### Fragment Sahder

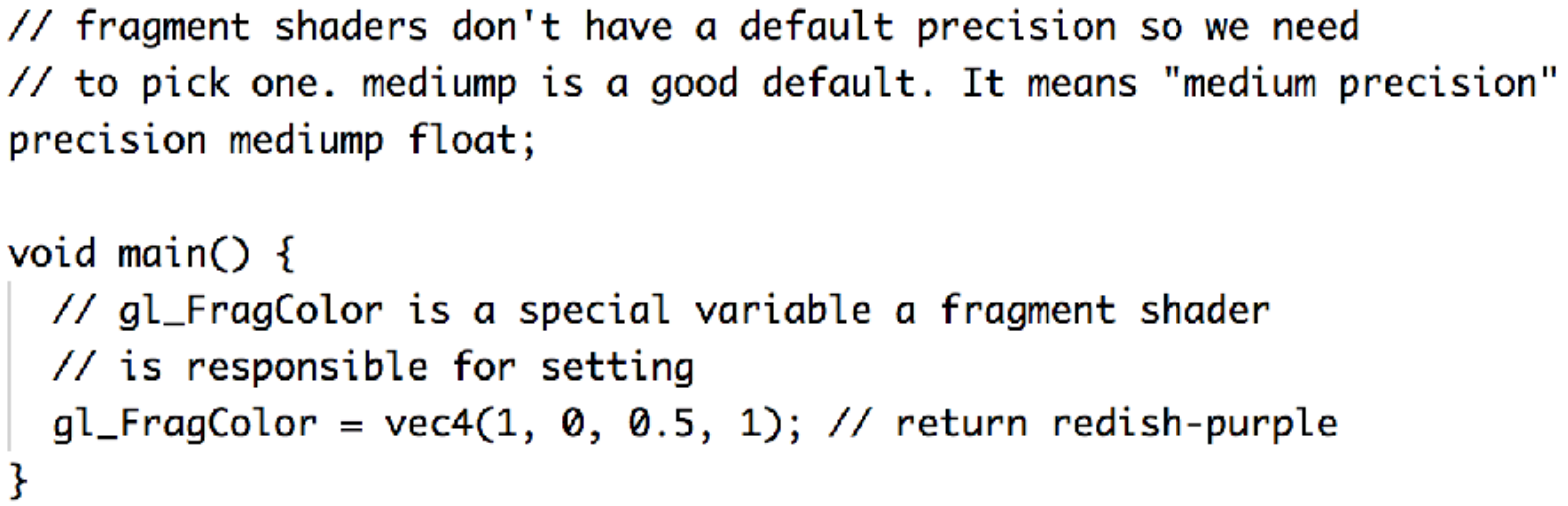
- A fragment shader's job is to compute a color for each pixel of the primitive currently being drawn.

## Vertex Shader



```
// an attribute will receive data from a buffer
attribute vec4 a_position;

// all shaders have a main function
void main() {

  // gl_Position is a special variable a vertex shader
  // is responsible for setting
  gl_Position = a_position;
}
```

# Fragment Shader



```
// fragment shaders don't have a default precision so we need
// to pick one. mediump is a good default. It means "medium precision"
precision mediump float;

void main() {
  // gl_FragColor is a special variable a fragment shader
  // is responsible for setting
  gl_FragColor = vec4(1, 0, 0.5, 1); // return redish-purple
}
```

## Shaders and Data

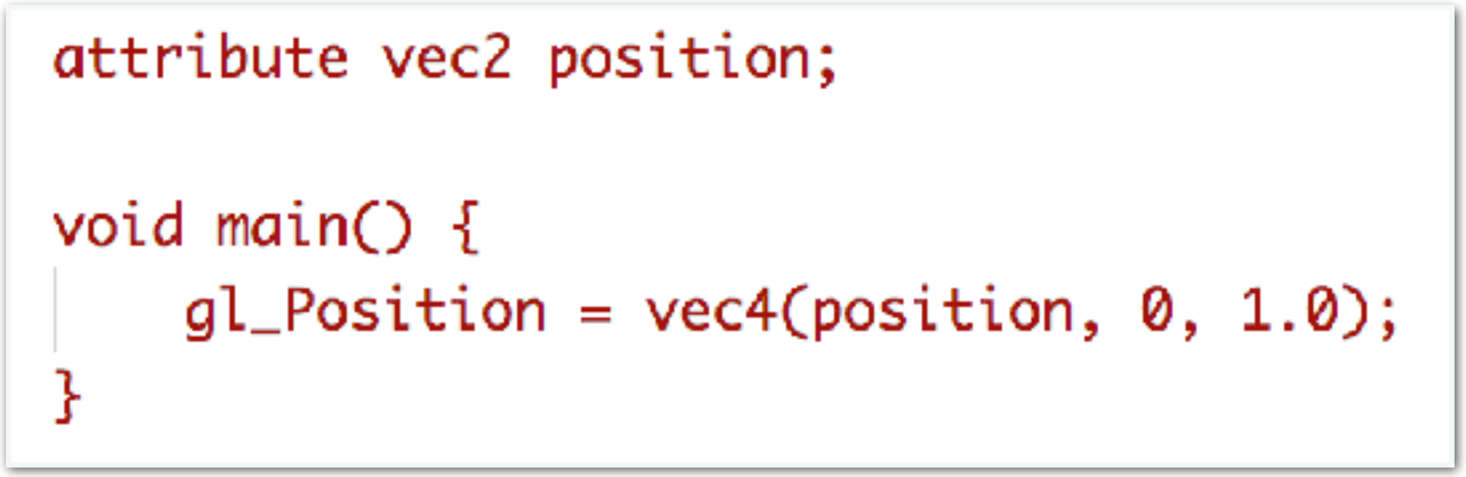
**Any data you want shaders to have access to must be provided to the GPU.**

- There are 4 ways a shader can receive data.
  1. Attributes (and Buffers)
  2. Uniforms
  3. Textures
  4. Varyings

## Attributes

**Attributes are used to specify how to pull data out of your buffers and provide them to your vertex shader.**

- For example you might put positions in a buffer as three 32bit floats per position.
- You would tell a particular attribute which buffer to pull the positions out of, what type of data it should pull out (3 component 32 bit floating point numbers), what offset in the buffer the positions start, and how many bytes to get from one position to the next.



```
attribute vec2 position;

void main() {
    gl_Position = vec4(position, 0, 1.0);
}
```

## Buffers

**Buffers are arrays of binary data you upload to the GPU.**

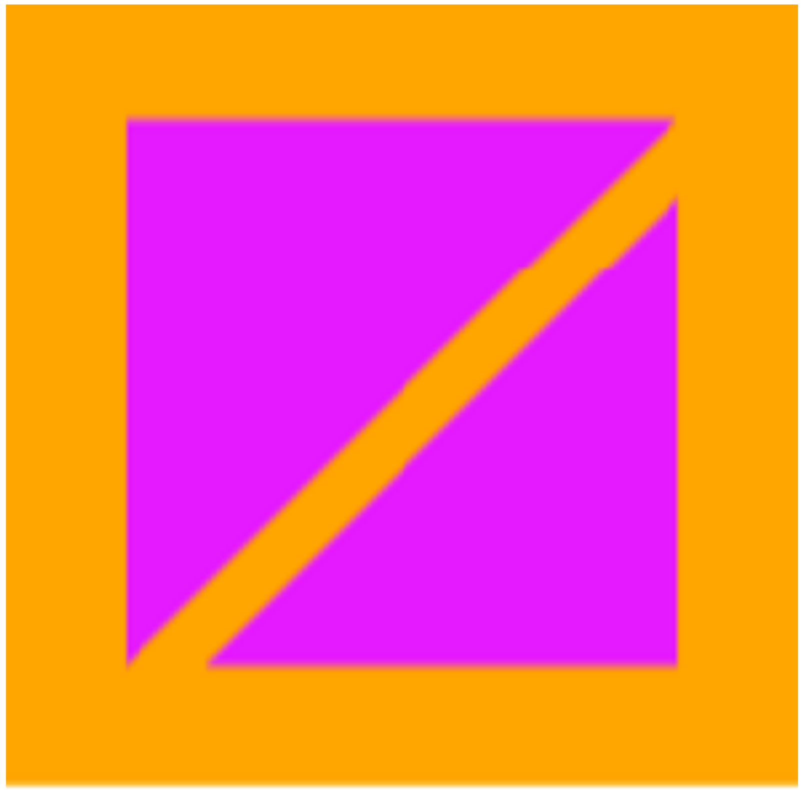
- Usually buffers contain data like positions, normals, texture coordinates, vertex colors, etc although you're free to put anything you want in them.
- Buffers are named memory areas (and not random access)

**A vertex shader is executed a specified number of times.**

- Each time it's executed the next value from each specified buffer is pulled out assigned to an attribute.

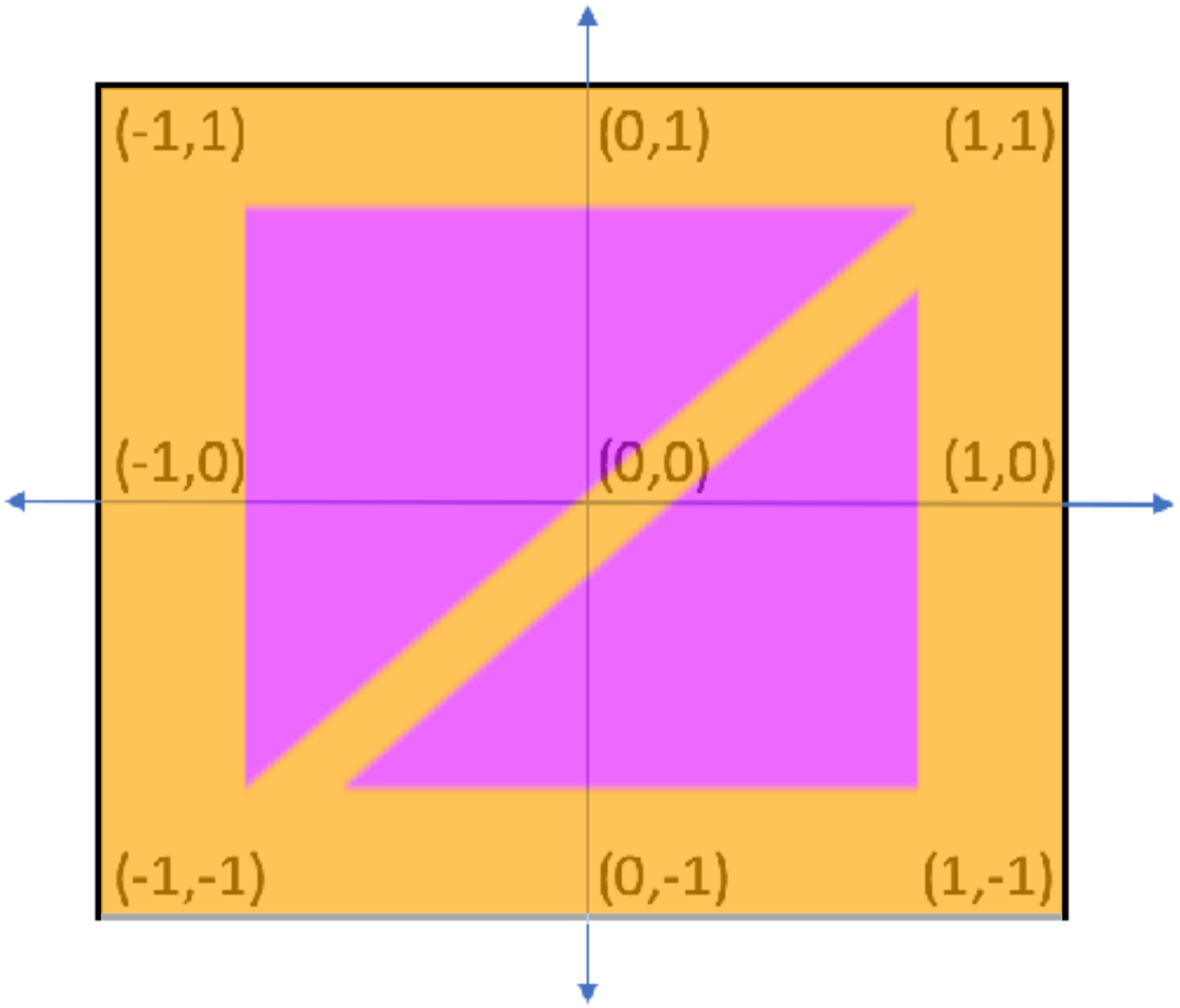
## Buffers

```
const vertices = new Float32Array([
    -0.7, -0.7,
    -0.7, 0.7,
    0.7, 0.7,
    -0.5, -0.7,
    0.7, 0.5,
    0.7, -0.7
]);
```



```
const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
const positionLocation = gl.getAttribLocation(program, 'position');
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionLocation);
```

```
(x1, y1)  = (-0.7, -0.7)
(x2, y2)  = (-0.7, 0.7)
(x3, y3)  = (0.7, 0.7)
(x4, y4)  = (-0.5, -0.7)
(x5, y5)  = (0.7, 0.5)
(x6, y6)  = (0.7, -0.7)
```



## Uniforms, textures and varyings

### Uniforms

- Uniforms are effectively global variables you set before you execute your shader program.

### Textures

- Textures are arrays of data you can randomly access in your shader program.
- The most common thing to put in a texture is image data but textures are just data and can just as easily contain something other than colors.

### Varyings

- Varyings are a way for a vertex shader to pass data to a fragment shader.
- Depending on what is being rendered, points, lines, or triangles, the values set on a varying by a vertex shader will be interpolated while executing the fragment shader.

## GLSL

### OpenGL Shader Language

- Closely modelled on C/C++

# Overview of GLSL

## Version specification

- The first line is usually the version specification
- #version 130

| OpenGL version | GLSL Version |
|---|---|
| 2.0 | 110 |
| 2.1 | 120 |
| 3.0 | 130 |
| 3.1 | 140 |
| 3.2 | 150 |
| 3.3 | 330 |

| OpenGL version | GLSL Version |
|---|---|
| 4.0 | 400 |
| 4.1 | 410 |
| 4.2 | 420 |
| 4.3 | 430 |
| 4.4 | 440 |
| 4.5 | 450 |

## Overview of GLSL

### Important Data types

- float (32 bit floating point number)
  - more common than integers in shader language
- int (32 bit integer)
- bool
- sampler__ (sampler2D etc)

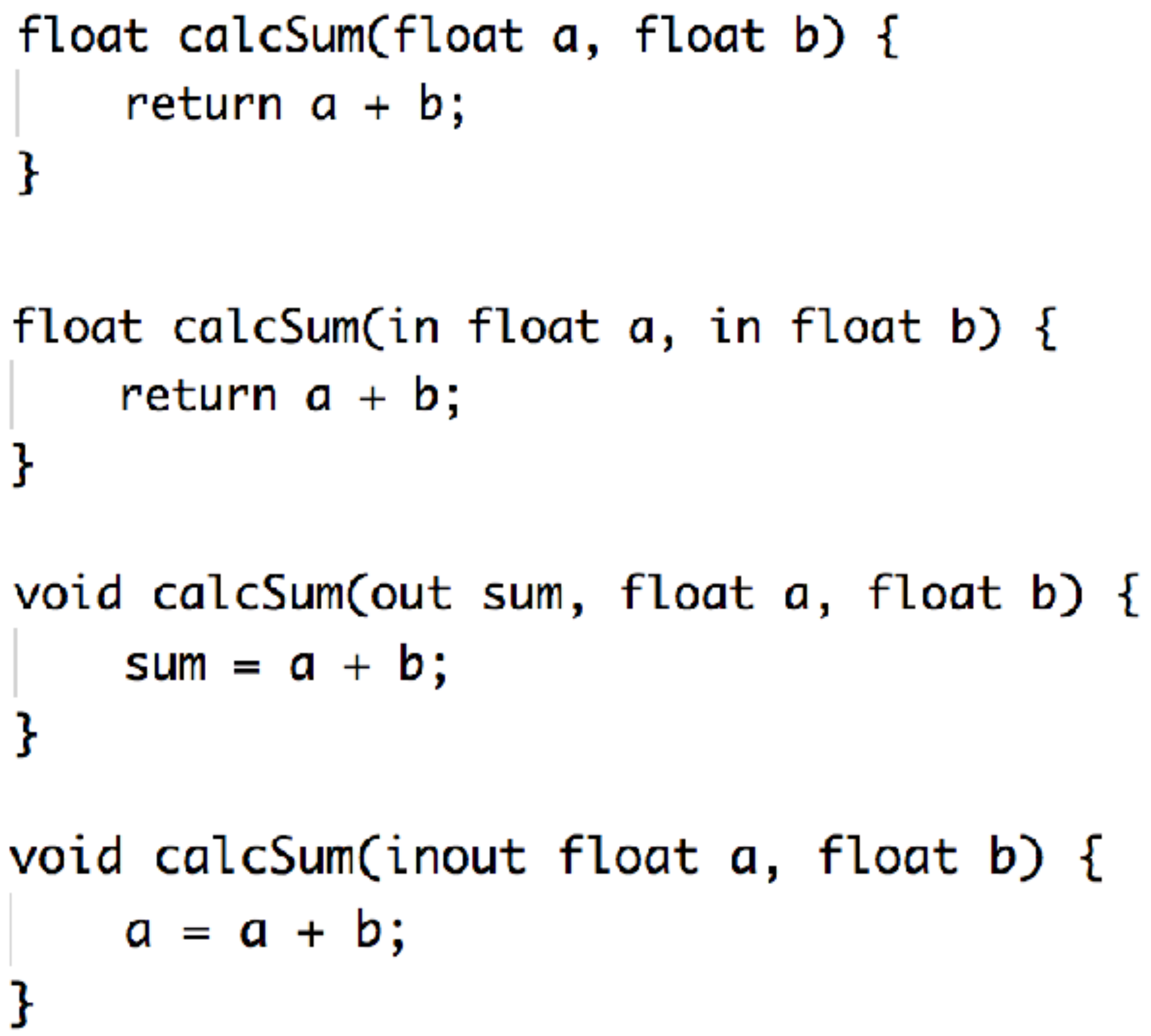
### Both float and int have vector variants

- vec2, ivec2
- vec3, ivec3
- vec4, ivec4

### The float also has matrix variants

- mat2, mat3, mat4

## Function declaration



```
float calcSum(float a, float b) {
    return a + b;
}


float calcSum(in float a, in float b) {
    return a + b;
}

void calcSum(out sum, float a, float b) {
    sum = a + b;
}

void calcSum(inout float a, float b) {
    a = a + b;
}
```

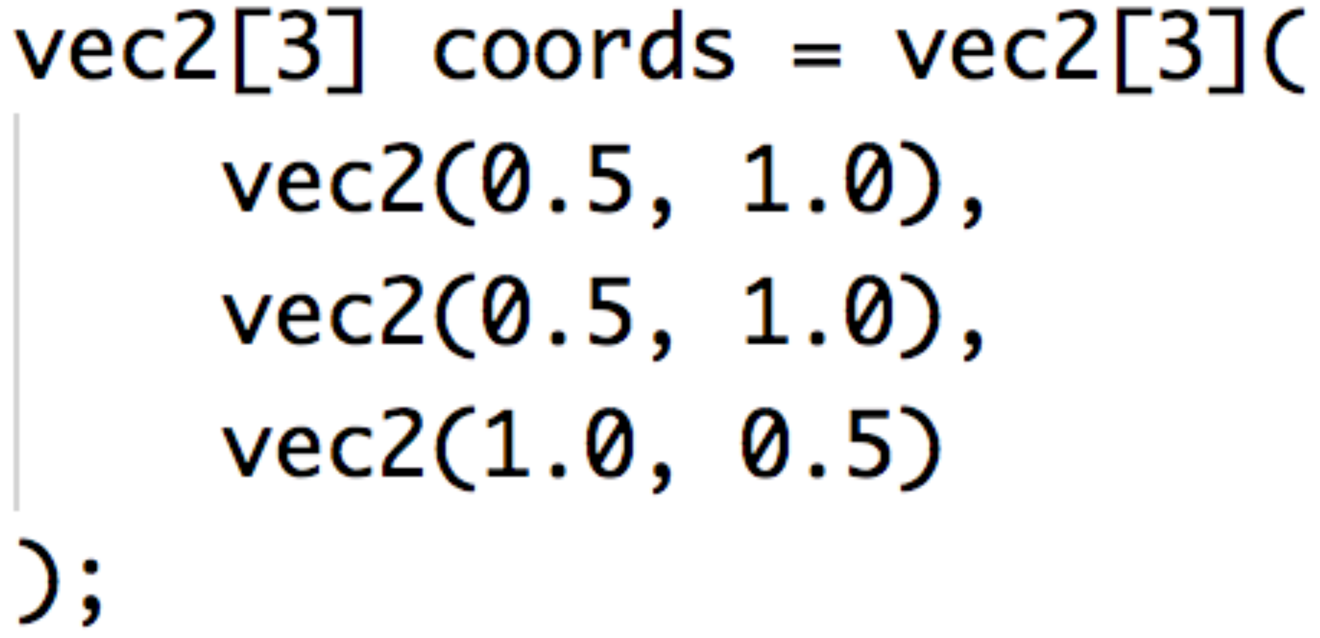
## Variable declarations

```
float x;
float y = 1.23;
float p = 1.2, q = 3.4;

vec3 color = vec3(0.0, 0.1, 0.2);

mat3 idMat = mat3(
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0
);
```

## Variable declarations

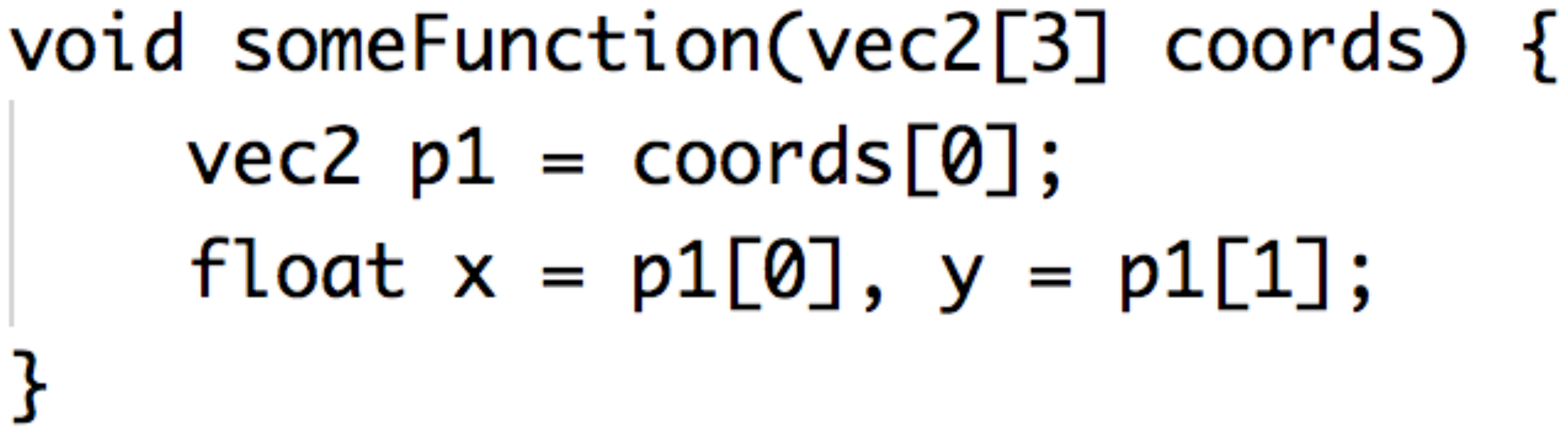
```
vec3 color = vec3(1.0);

vec3 color = vec3(0.5, 1.0, 0.2);
vec4 colorAlpha = vec4(color, 1.0);
```

## Array declarations



```
vec2[3] coords = vec2[3](
        vec2(0.5, 1.0),
        vec2(0.5, 1.0),
        vec2(1.0, 0.5)
);
```

## Array accessors



```
void someFunction(vec2[3] coords) {
    vec2 p1 = coords[0];
    float x = p1[0], y = p1[1];
}
```

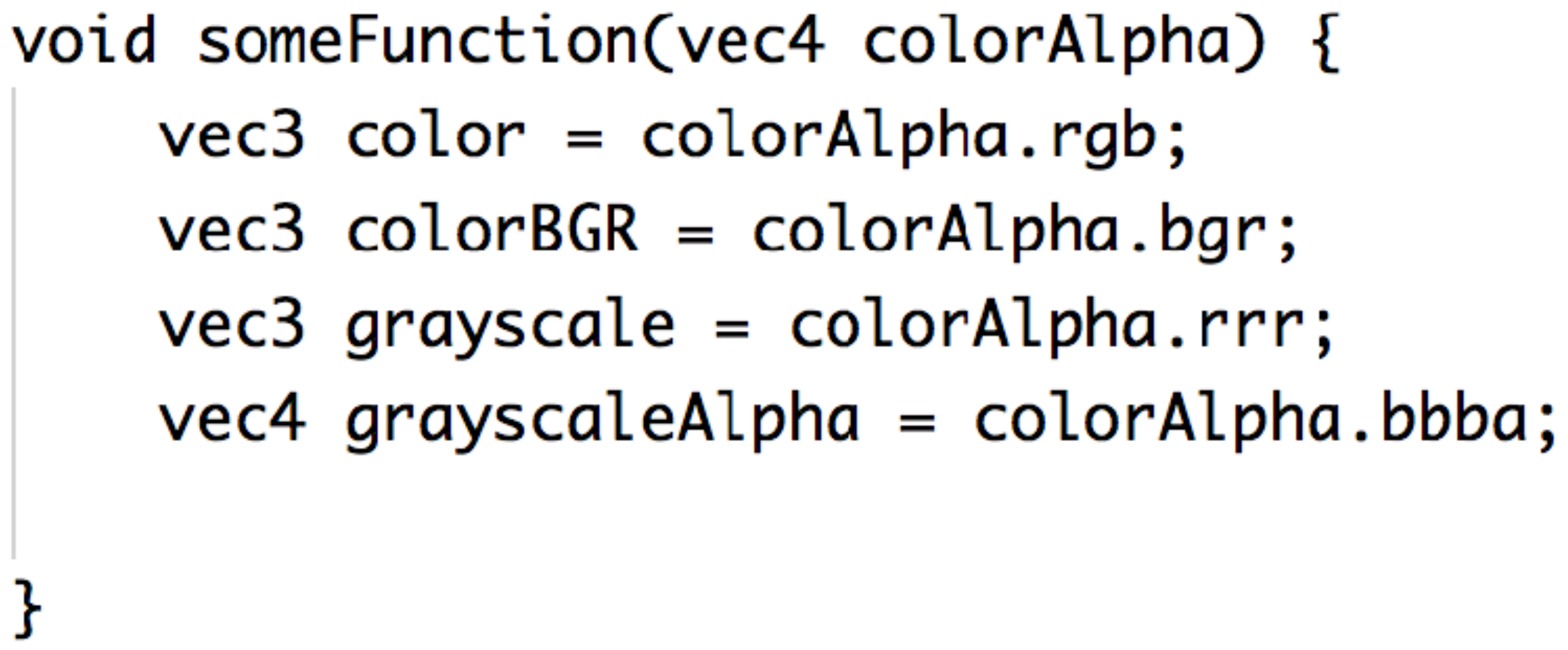
## Vector Swizzling

```
void someFunction(vec2 position) {
    float x = position.x;
    float y = position.y;

    position.x = 1.0;
    position.y = 0.5;
}
```

## Vector Swizzling

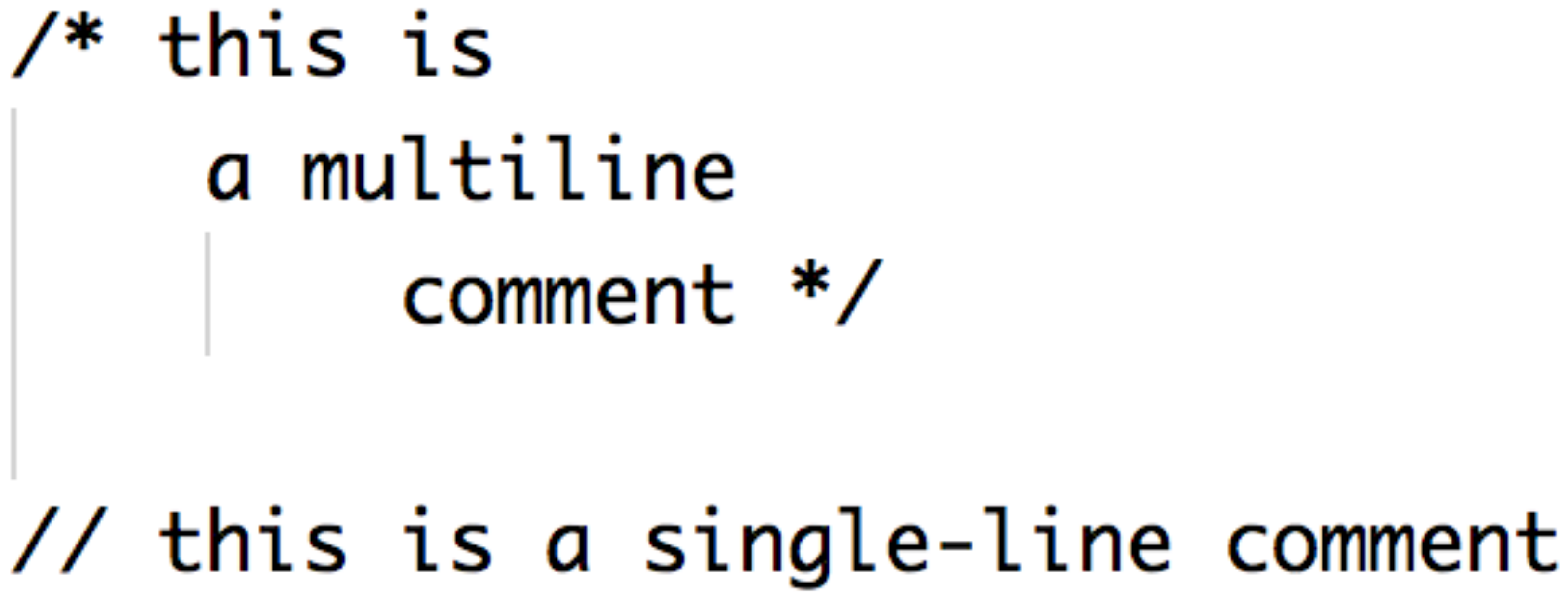
| Array Style Accessor | Coordinate Dimension | Color Channel | Texture Dimension |
|---|---|---|---|
| [0] | .x | r | s |
| [1] | .y | g | t |
| [2] | .z | b | p |
| [3] | .w | a | q |

## Vector Swizzling



```
void someFunction(vec4 colorAlpha) {
    vec3 color = colorAlpha.rgb;
    vec3 colorBGR = colorAlpha.bgr;
    vec3 grayscale = colorAlpha.rrr;
    vec4 grayscaleAlpha = colorAlpha.bbba;

}
```

## Code commenting

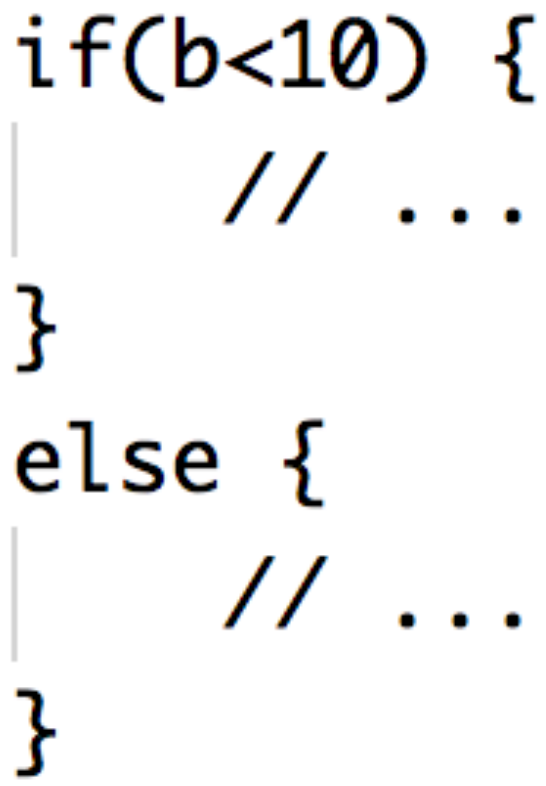


```
/* this is
    a multiline
        comment */

// this is a single-line comment
```

## Calling functions

```
float y = someFunction(x);

float z = someOtherFunction(1.0, 0.5);
```

## Control structures



```
if(b<10) {
    // ...
}
else {
    // ...
}
```

## Control structures

```
for(int i=0; i<10; i++) {
    // ...
}

int i = 0;
while(i<10) {
    // ...
    i++;
}
```

## Control statements

**Change the flow of control**

- return;
- return x;
- break;
- continue;
- discard;

**‘goto’ is not allowed**

## Operators

```
+   -   *   /

=   +=   -=   *=   /=   ++   --

%   &   |   ^   <<   >>   ~

%=   &=   |=   ^=   <<=   >>=

!   &&   ||   ==   !=   <   <=   >   >=

?  :
```

## Operators

```
9 / 2 == 4
9.0 / 2.0 == 4.5
```

## Operators

```
vec3(x1, y1, z1) + vec3(x2, y2, z2)

== vec3(x1+x2, y1+y2, z1+z2)
```

## Operators

```
float(k) + vec3(x, y, z)

 == vec3(x+k, y+k, z+k)
```

```
float(k) * vec3(x, y, z)

 == vec3(x*k, y*k, z*k)
```

## Operators

```
scalar * vector

vector * vector

matrix * vector

matrix * matrix
```

## Operators

```
vec3(x1, y1, z1) * vec3(x2, y2, z2)

== vec3(x1*x2, y1*y2, z1*z2)
```

Not an established operation in linear algebra; useful for color bending

## Operators

```
dot(vec3(x1, y1, z1) , vec3(x2, y2, z2))

== x1*x2 + y1*y2 + z1*z2


cross(vec3(x1, y1, z1) , vec3(x2, y2, z2))

== vec3(y1*z2-z1*y2, z1*x2-x1*z2, x1*y2-y1*x2)
```

## Builtin functions

```
abs(x) sign(x) floor(x) ceil(x)

fract(x) ---> fraction part of a float x

min(a,b) max(a,b) mod(x,modulo)

sqrt(x) pow(x,exponent) exp(x) log(x)

sin(x) cos(x) tan(x) asin(x) acos(x) atan(x)
```

## Builtin functions

```
sin(vec3(x, y, z))

== vec3(sin(x), sin(y), sin(z))
```

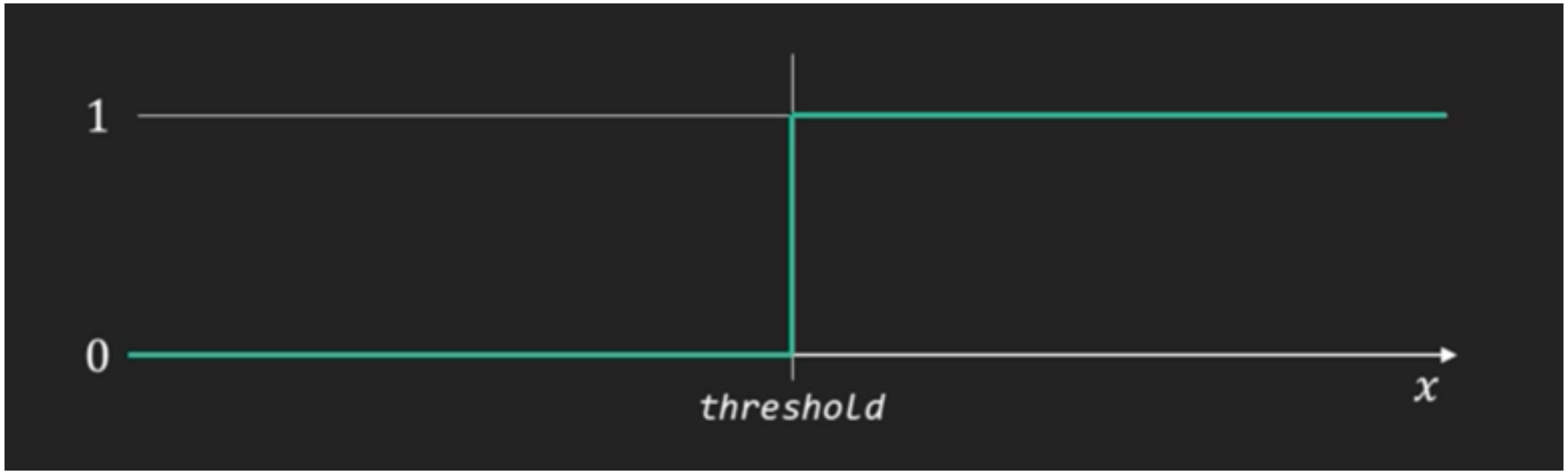
## Builtin functions

```
clamp(x, minimum, maximum)

== min(max(x, minimum), maximum)
```

```
clamp(color, 0.5, 1.0)
```

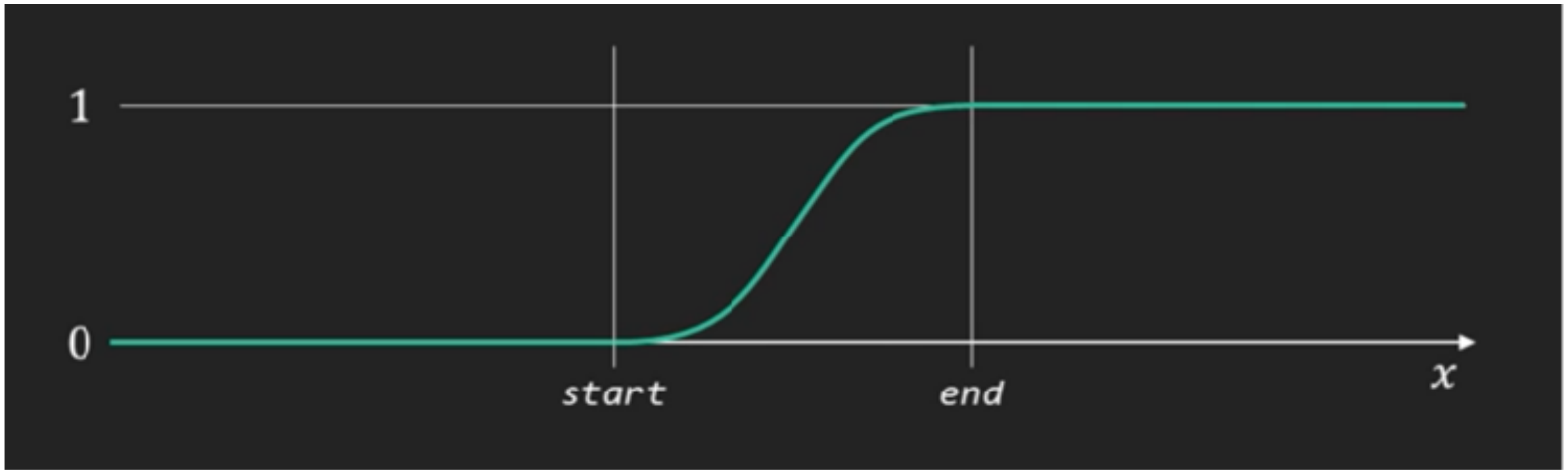
## Builtin functions

```
step(threshold, x)
```



## Builtin functions

```
smoothstep(start, end, x)
```



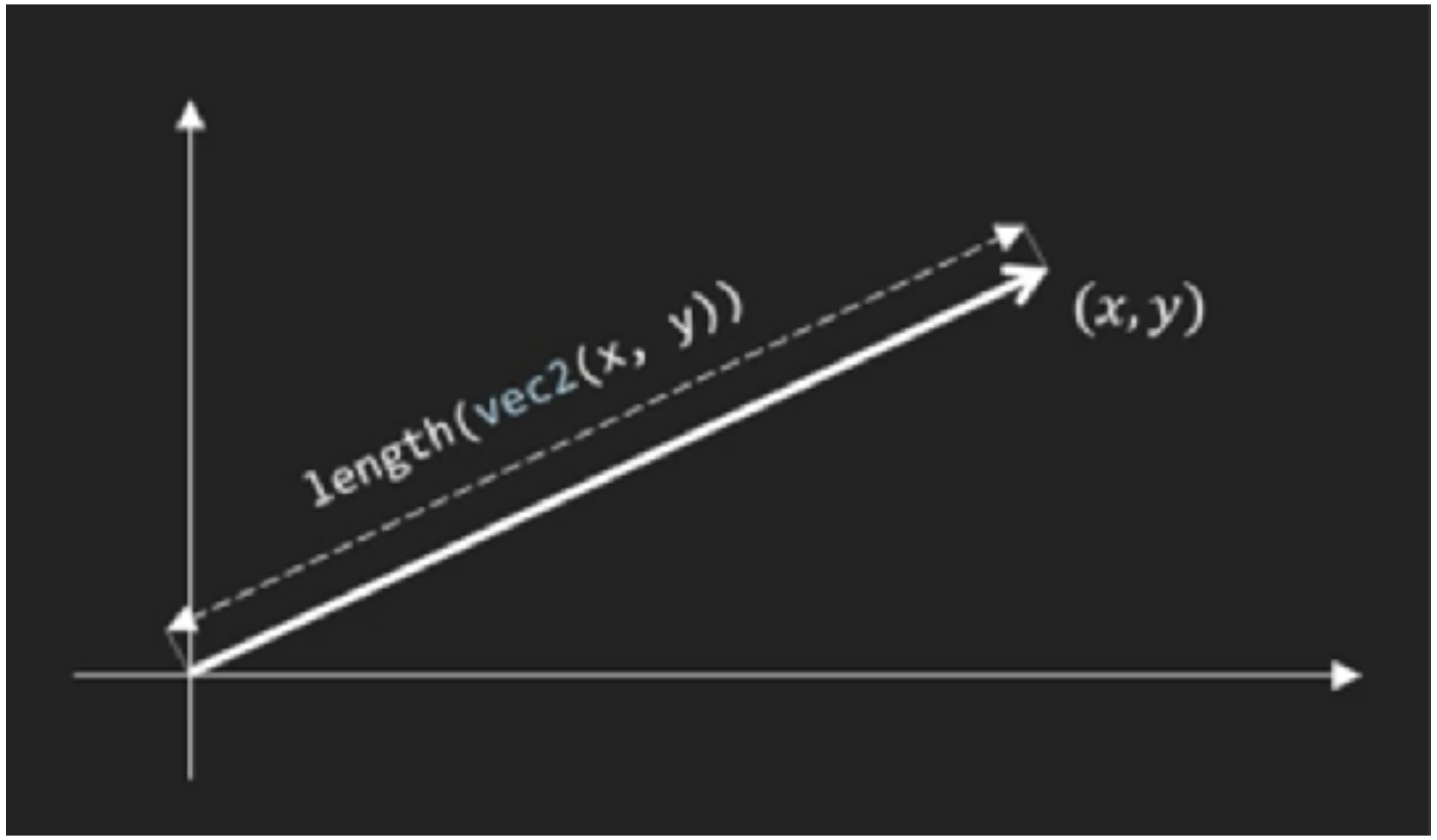
## Builtin functions

```
        mix(a, b, ratio)
  == (1.0 - ratio) * a + ratio * b
```

```
mix(red, yellow, position.x)
```

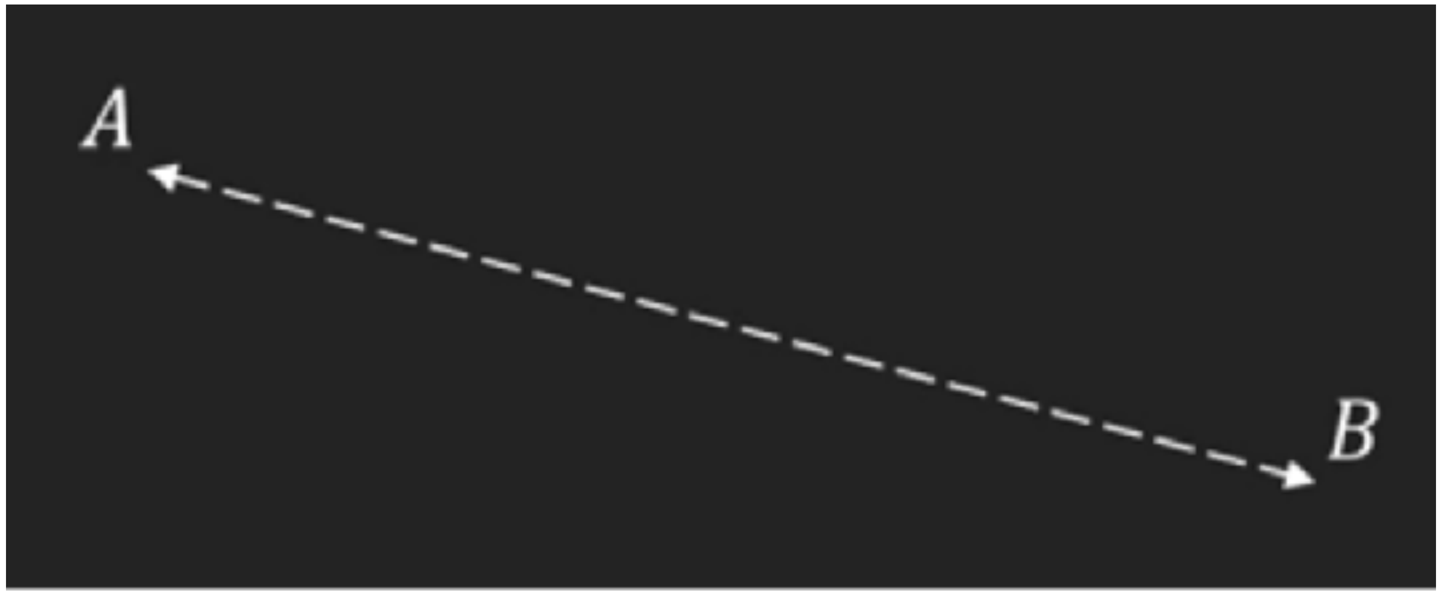
# Builtin functions

```
length(vec2(x, y))

  == sqrt(x*x + y*y)
```



# Builtin functions

```
distance(A, B)

  == length(B - A)
```



# Builtin functions

**vec4**

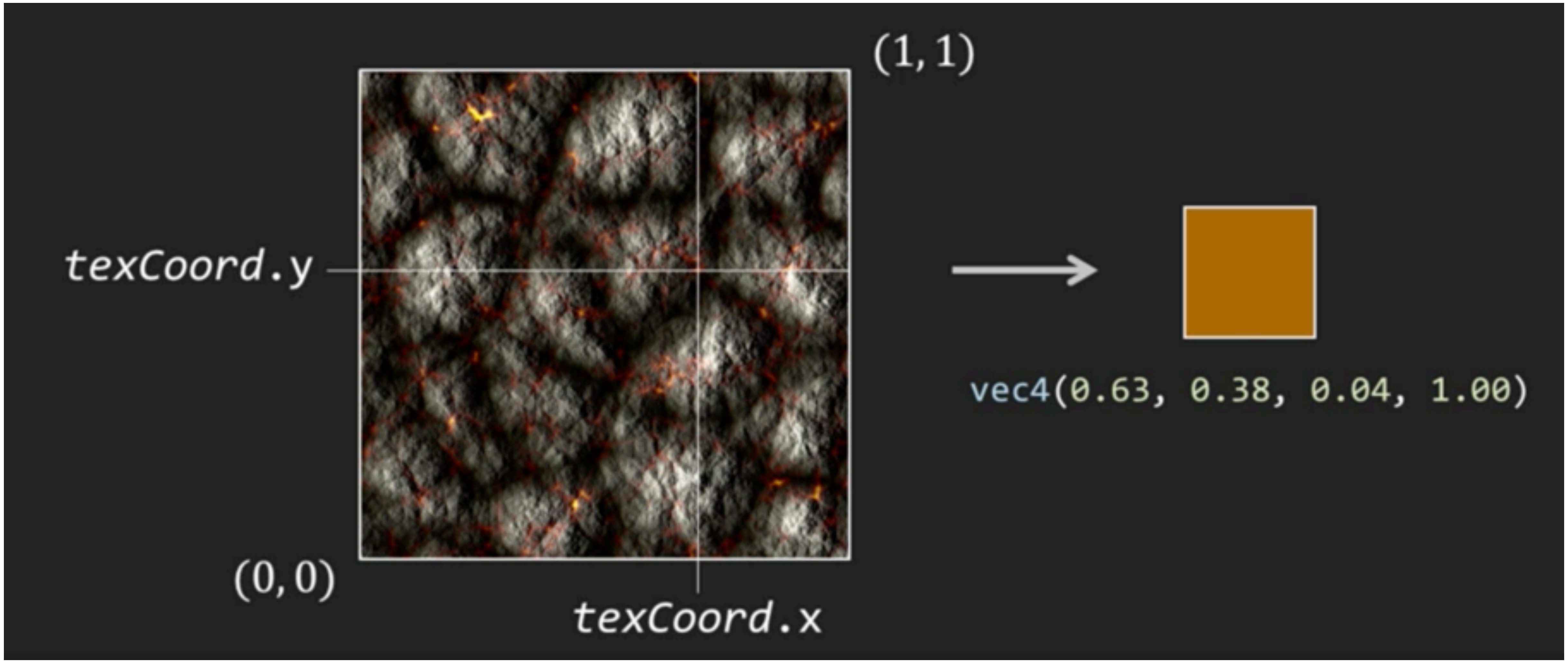
```
texture(aTexture, texCoords)
```

**vec2**

**sampler2D or**
**image or animation**

# Builtin functions

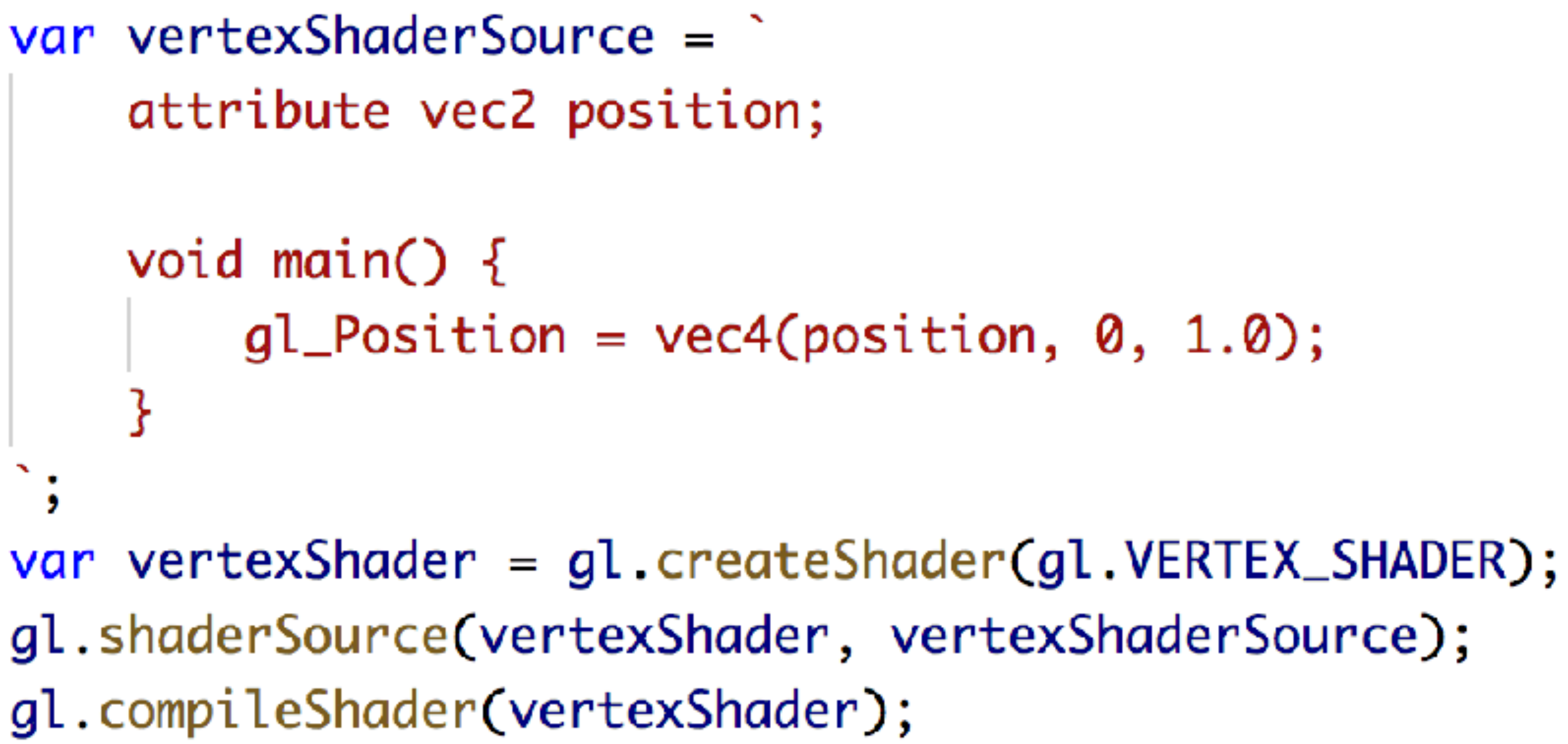
```
texture(aTexture, texCoords)
```



## Putting it together: the canvas

```
var canvas = document.createElement('canvas');
document.body.appendChild(canvas);
var gl = canvas.getContext('webgl');
```

## Putting it together: vertex shader



```
var vertexShaderSource = `
    attribute vec2 position;

    void main() {
        gl_Position = vec4(position, 0, 1.0);
    }
`;
var vertexShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertexShader, vertexShaderSource);
gl.compileShader(vertexShader);
```

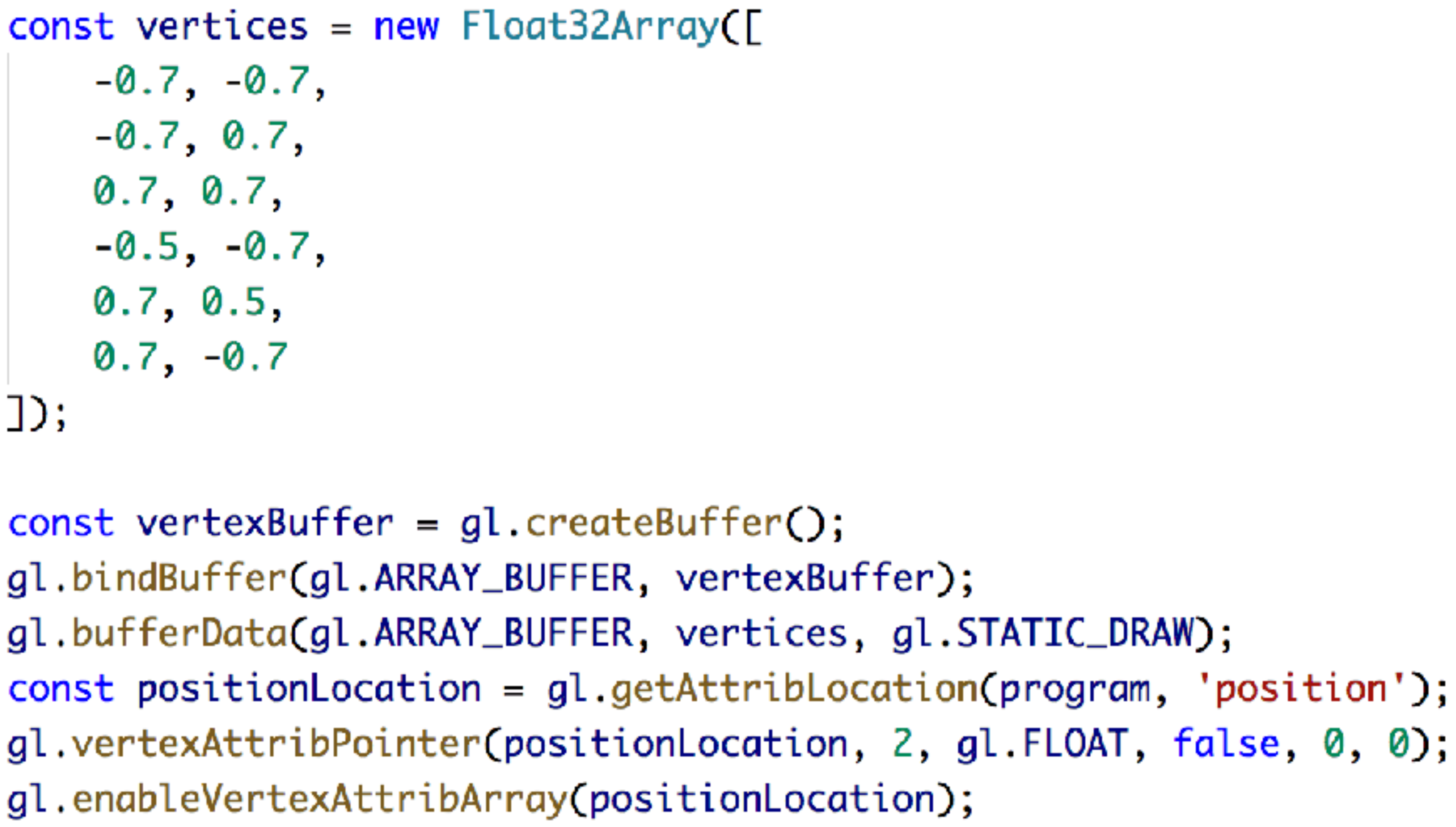
## Putting it together: fragment shader

```
var fragShaderSource = `
    void main() {
        gl_FragColor = vec4(0.9, 0.1, 1.0, 1.0);
    }
`;

var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragShaderSource);
gl.compileShader(fragShader);
```

## Putting it together: program

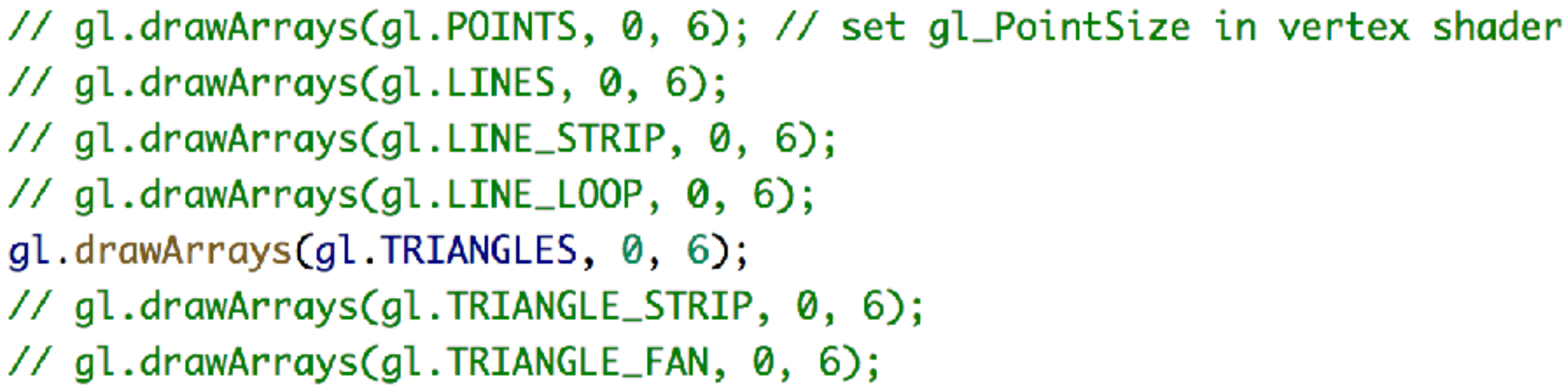
```
var program = gl.createProgram();
gl.attachShader(program, vertexShader);
gl.attachShader(program, fragShader);
gl.linkProgram(program);
gl.useProgram(program);
```

## Putting it together: vertex data



```
const vertices = new Float32Array([
    -0.7, -0.7,
    -0.7, 0.7,
    0.7, 0.7,
    -0.5, -0.7,
    0.7, 0.5,
    0.7, -0.7
]);

const vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
const positionLocation = gl.getAttribLocation(program, 'position');
gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionLocation);
```

## Putting it together: drawing



```
// gl.drawArrays(gl.POINTS, 0, 6); // set gl_PointSize in vertex shader
// gl.drawArrays(gl.LINES, 0, 6);
// gl.drawArrays(gl.LINE_STRIP, 0, 6);
// gl.drawArrays(gl.LINE_LOOP, 0, 6);
gl.drawArrays(gl.TRIANGLES, 0, 6);
// gl.drawArrays(gl.TRIANGLE_STRIP, 0, 6);
// gl.drawArrays(gl.TRIANGLE_FAN, 0, 6);
```

# What is drawn..