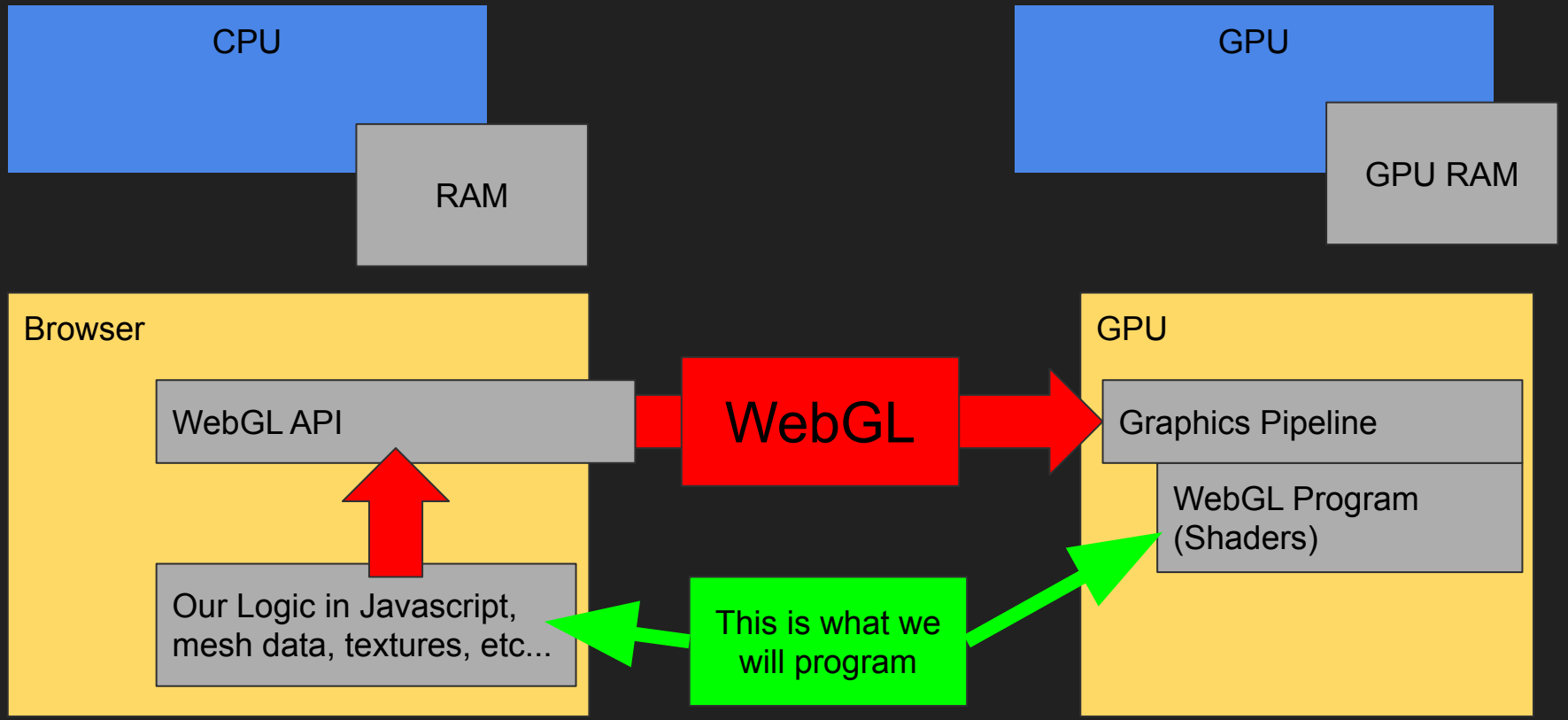
Shader Programming Fundamentals

And Buffer Management

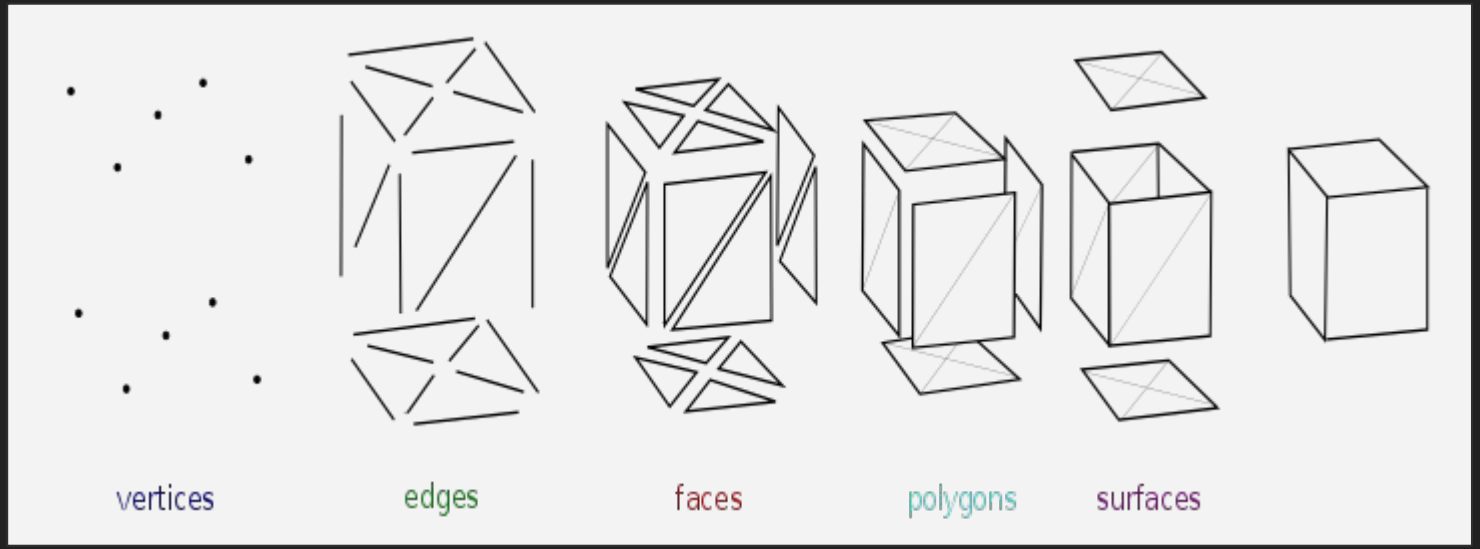
Recap: Program Layout



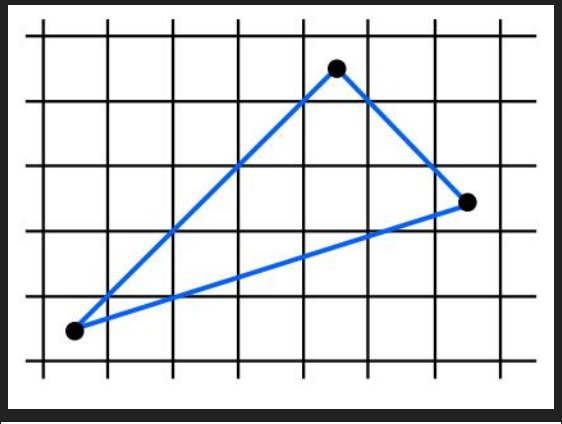
Recap: Task List

1. Create Canvas and get WebGL context.
2. Create array of position data.
3. Create a WebGL-shader-program for the GPU that can accept our position data when we want to push it to the GPU to draw it.
4. Compile shader-program and push it to the GPU.
5. Push our position data to the GPU.
6. Tell WebGL which shader-program to use for the data pushed.
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

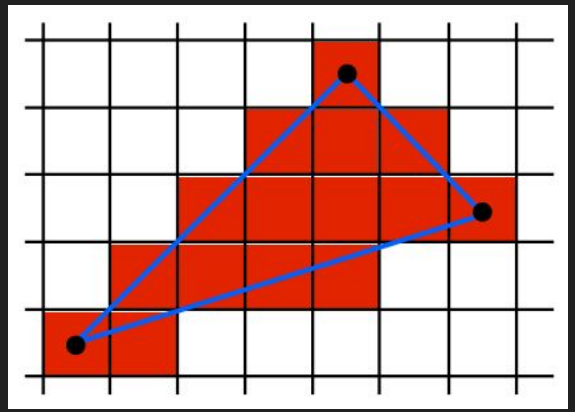
Recap: Drawing rasterized graphics Terminology



Recap: Drawing rasterized graphics



Triangle data send to the GPU.



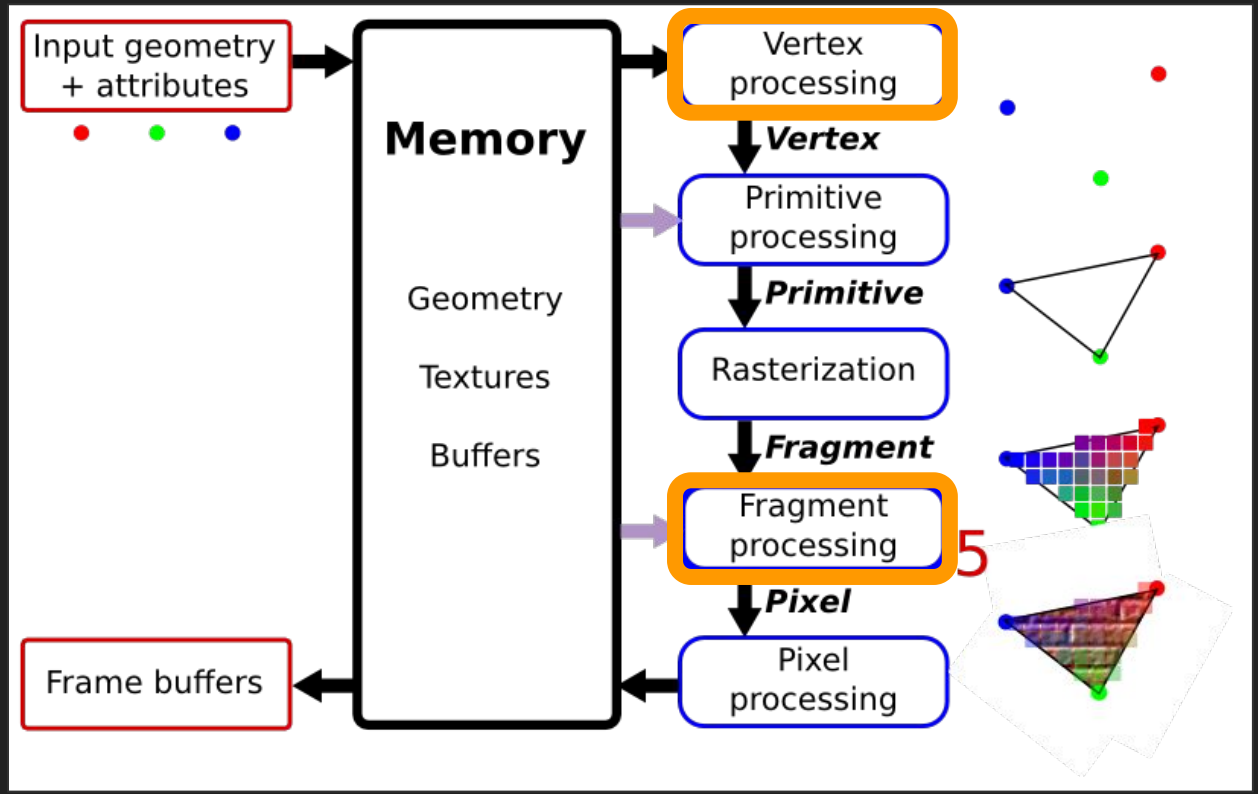
Pixels the GPU will color on the output using normal rasterization.

Recap: GPU Pipeline

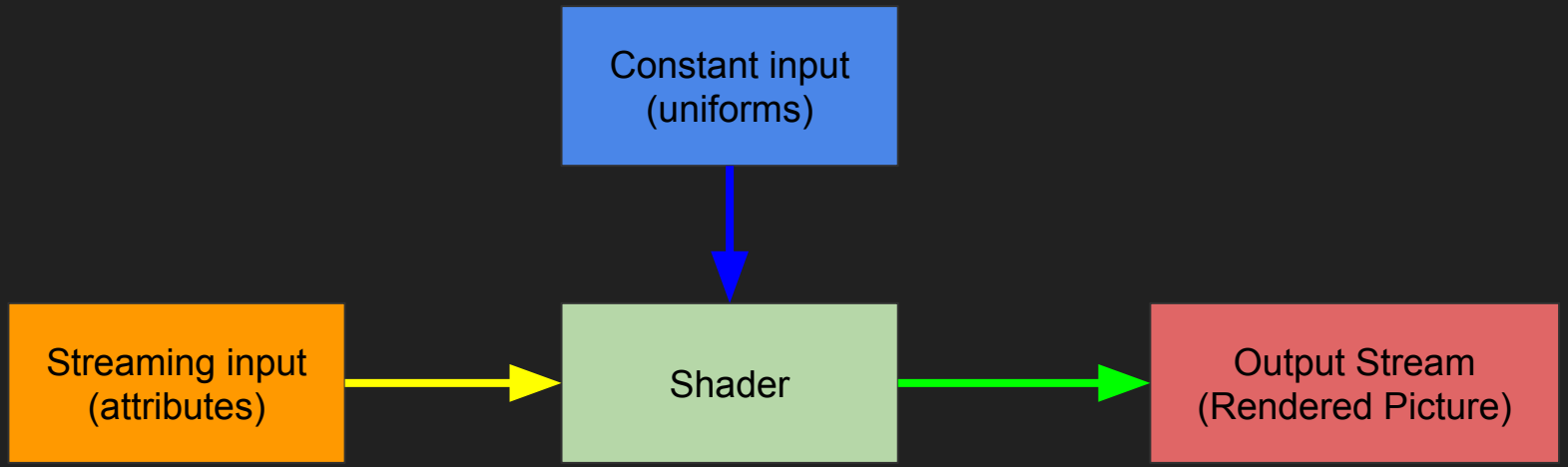
A quick intro to GPU programming:

The GPU Pipeline:

Programmable steps



Shader input

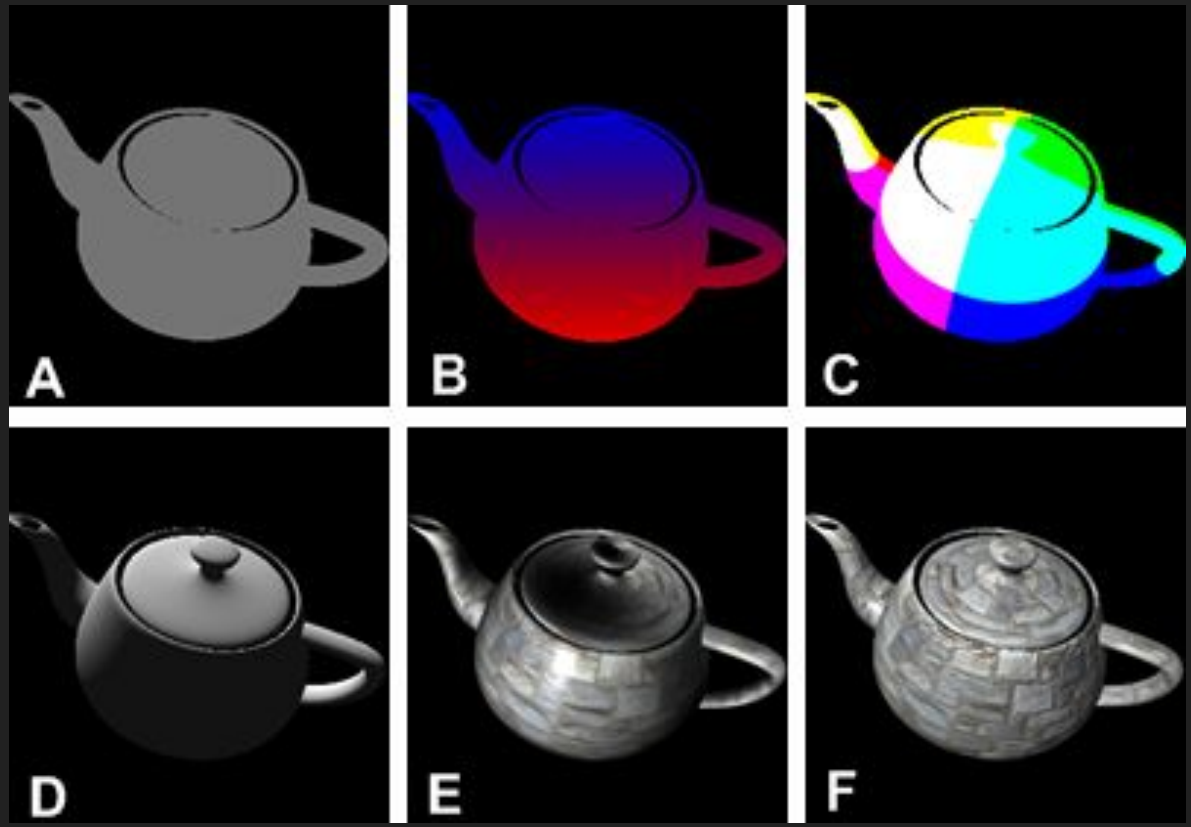


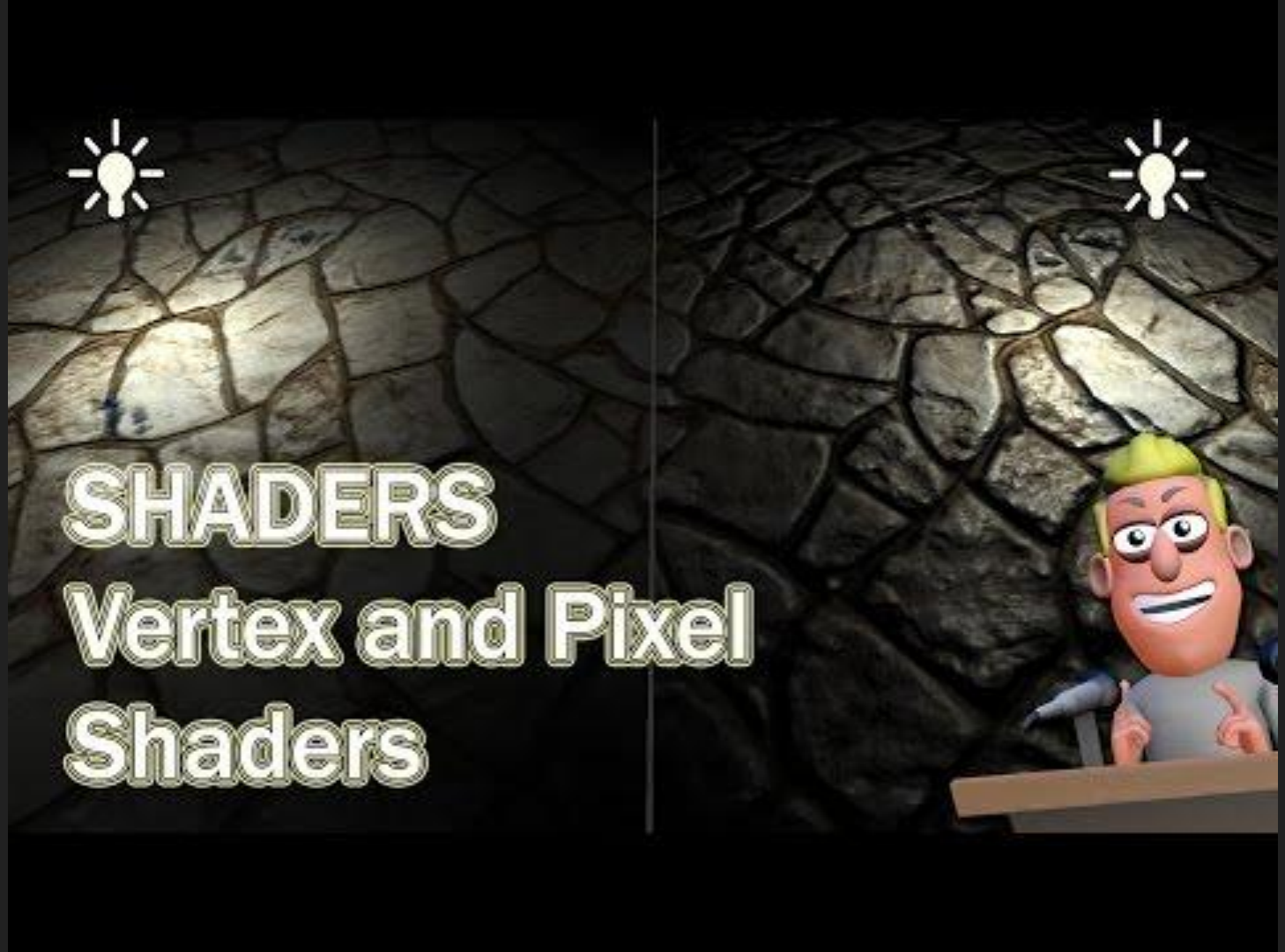
What is a Shader?

Small Program

Runs on the GPU

Can manipulate Vertices and the way Polygons are drawn on the screen.




SHADERS
Vertex and Pixel
Shaders

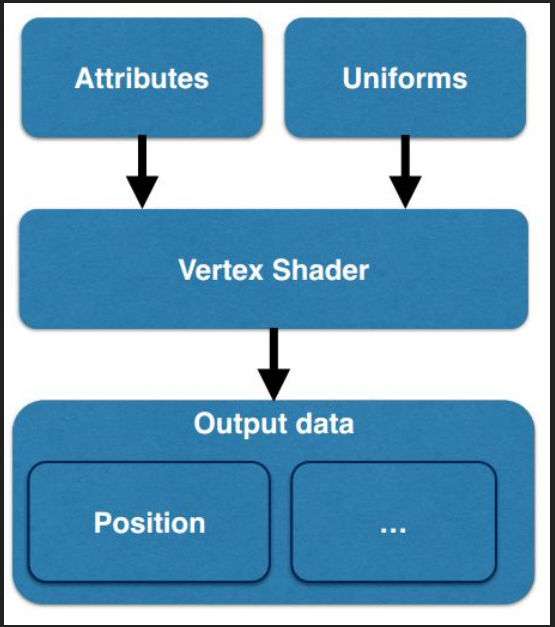
Vertex Shader

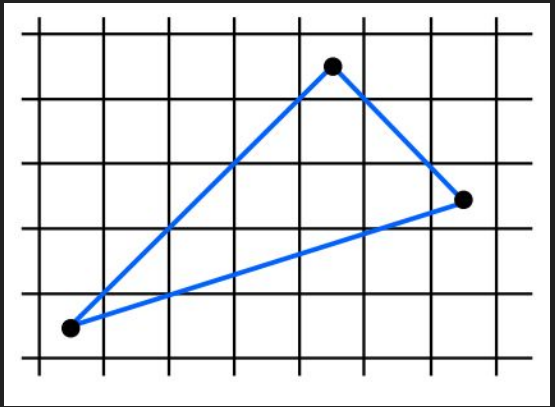
Input:

- Vertex attributes (data per vertex)
- Uniforms (data per draw call)

Output :

- Vertex position (mandatory)
- Other vertex attributes





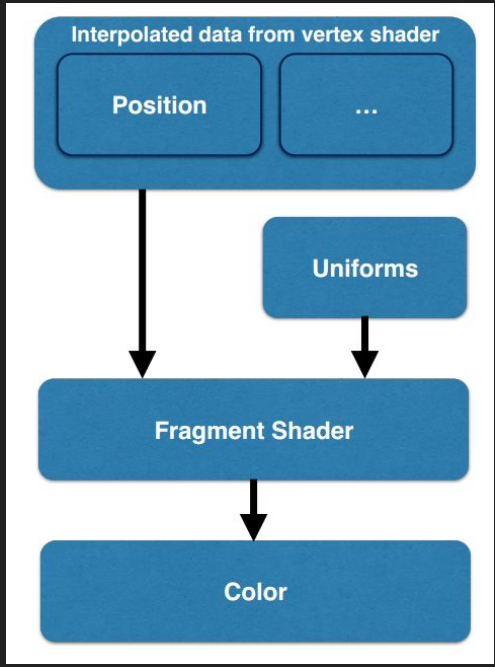
Fragment Shader

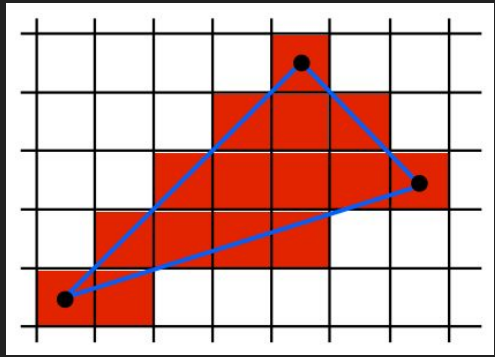
Input:

- Interpolated vertex attributes
- (data per vertex)
- Uniforms (data per draw call -
- material properties)

Output :

- Color to Framebuffer





Shader inputs: Uniforms and Attributes

Attributes:

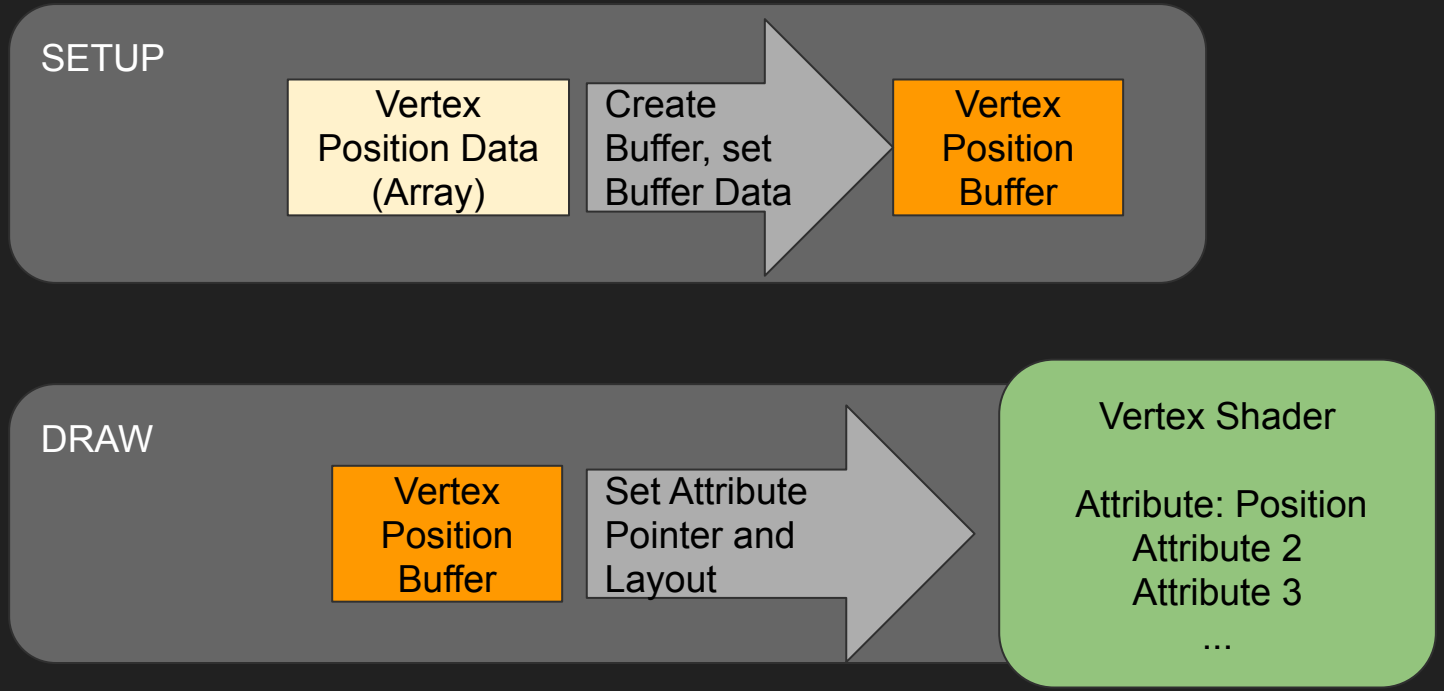
- Data that changes for every Vertex of the model
  - Position
  - Texture Coordinates
  - Direction of Surface Normal (What way is the Vertex “facing, for e.g. lighting)

Uniforms:

- Data that stays constant for the entire model
  - Tint
  - Textures
  - The models position and rotation in Space

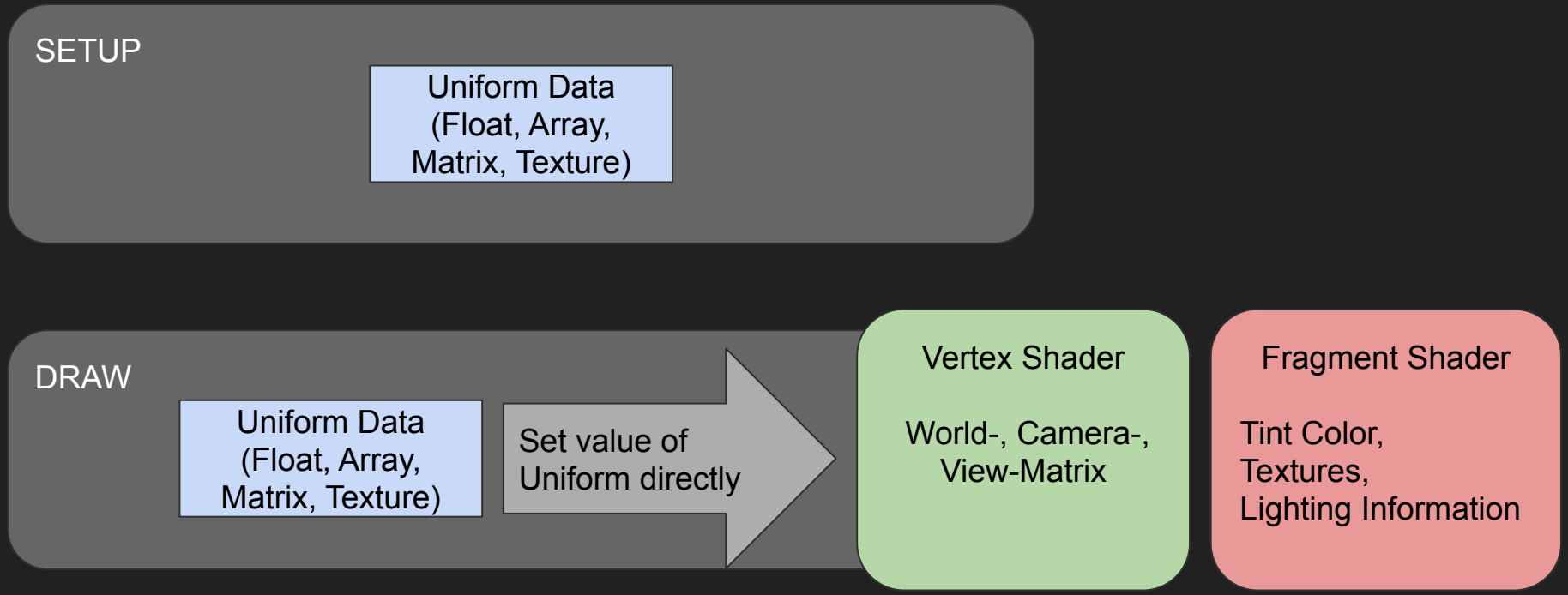
Attributes:

- Data contained in Buffer.
- Every Vertex gets one set of data.
- Example: Vertex position



Uniforms:

- Only one set of data for all vertices
- Do not need a Buffer
- Example: Object tint (color)



GLSL Syntax: Attributes and Uniforms

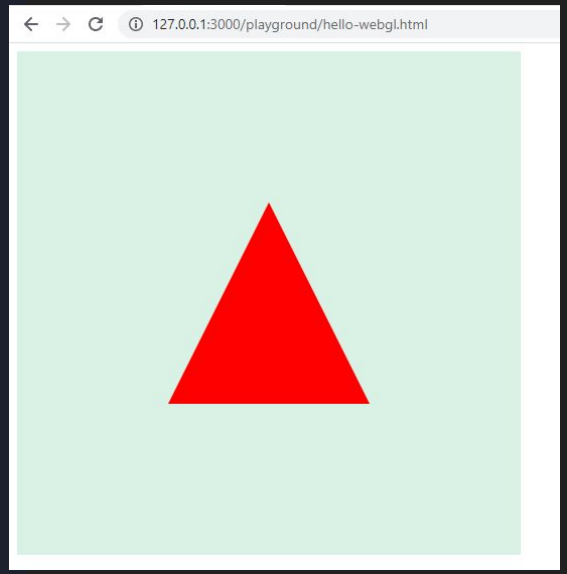
```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;
    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings
    uniform vec3 u_color;
    void main()
    {
        gl_FragColor = vec4(u_color,1); // rbga
    }
</script>
```

Position: Attribute, changes per vertex.

Color: Uniform, the same for all vertices of the draw call.

GLSL Syntax: Attributes and Uniforms

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;
    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings
    uniform vec3 u_color;
    void main()
    {
        gl_FragColor = vec4(u_color,1); // rbga
    }
</script>
```



GLSL Syntax: Varying (version < es 3.0)

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;
    attribute vec3 a_color;
    varying vec3 color;
    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
        color = a_color;
    }
</script>
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings
    varying vec3 color;
    void main()
    {
        gl_FragColor = vec4(color,1); // rbga
    }
</script>
```

Position and color: Attributes, change per vertex.

Color: input from vertex shader (varying)

GLSL Syntax: Es 3.0+
more modern version of GLSL

Version Identifier on first line!

Old syntax (< es 3.0)

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;
    attribute vec3 a_color;
    varying vec3 color;
    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
        color = a_color;
    }
</script>
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings
    varying vec3 color;
    void main()
    {
        gl_FragColor = vec4(color,1); // rbga
    }
</script>
```

New syntax (es 3.0+)

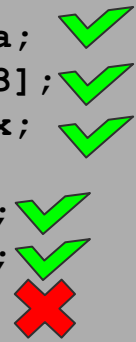
```
<script type="vertex-shader" id="vertexShader">#version 300 es
    in vec2 a_position;
    in vec3 a_color;
    out vec3 color;
    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
        color = a_color;
    }
</script>
<script type="fragment-shader" id="fragmentShader">#version 300 es
    precision highp float; //float precision settings
    in vec3 color;
    out vec4 finalColor;
    void main()
    {
        finalColor = vec4(color,1); // rbga
    }
</script>
```

GLSL Syntax

Types

- Scalar types: float, int, uint, bool
- Vectors are also built-in types:
  - vec2, vec3, vec4
  - ivec*, uvec*, bvec*
- Access components three ways:
  - .x, .y, .z, .w position or direction
  - .r, .g, .b, .a color
  - .s, .t, .p, .q texture coordinate

```
vec4 myColor = vec4(1,1,0,1);

float alpha = myColor.a;
float alpha = myColor[3];
float alpha = myColor.x;

vec3 rgb = myColor.rgb;
vec3 bgr = myColor.bgr;
vec3 rgb = myColor.sgz
```



Mini Exercise:
what's the final value for v1, v2 and f?

```
1.  vec4 v1 = vec4(1,2,3,4);
2.  vec2 v2 = v1.yx;
3.  float f = v1.a;
4.  v1.zw = v2 + v2;
```

Mini Exercise:
what's the final value for v1, v2, v3 and f?

```
1.  vec4 v1 = vec4(1,2,3,4);
2.  vec2 v2 = v1.yx;
3.  float f = v1.a;
4.  v1.zw = v2 + v2;
```

```
v1 = [1, 2, 4, 2]

v2 = [2, 1]

f = 4
```

Transferring data to the GPU: Uniforms

Get location and set data using correct method format

```
let colorLocation = this.gl.getUniformLocation(shaderProgram, "u_color");
this.gl.uniform3fv(colorLocation, this.color);
```

Method formats

`WebGLRenderingContext.uniform[1234][fi][v]()`

Syntax

```
void gl.uniform1f(location, v0);
void gl.uniform1fv(location, value);
void gl.uniform1i(location, v0);
void gl.uniform1iv(location, value);

void gl.uniform2f(location, v0, v1);
void gl.uniform2fv(location, value);
void gl.uniform2i(location, v0, v1);
```

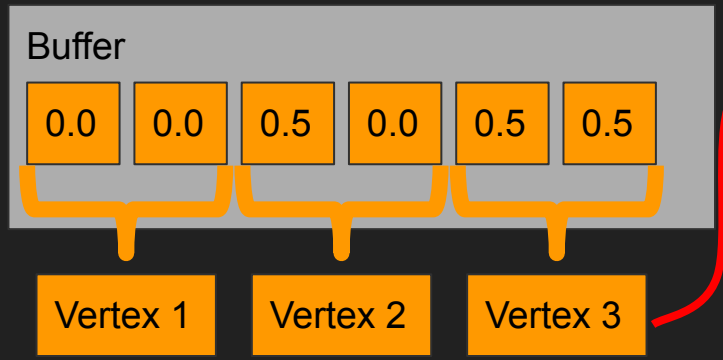
For more info, check documentation
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform

Transferring data to the GPU: Attributes

```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

```
let positionBuffer = gl.createBuffer();
// set id to the current active array buffer (only one can be active)
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// upload buffer data
gl.bufferData(gl.ARRAY_BUFFER, positionArray, gl.STATIC_DRAW);
```

Buffer data transfer: Attribute stream




```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;

    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
```

Buffer values need to go in chunks of two into `attribute vec2 a_position` of our vertex shader.

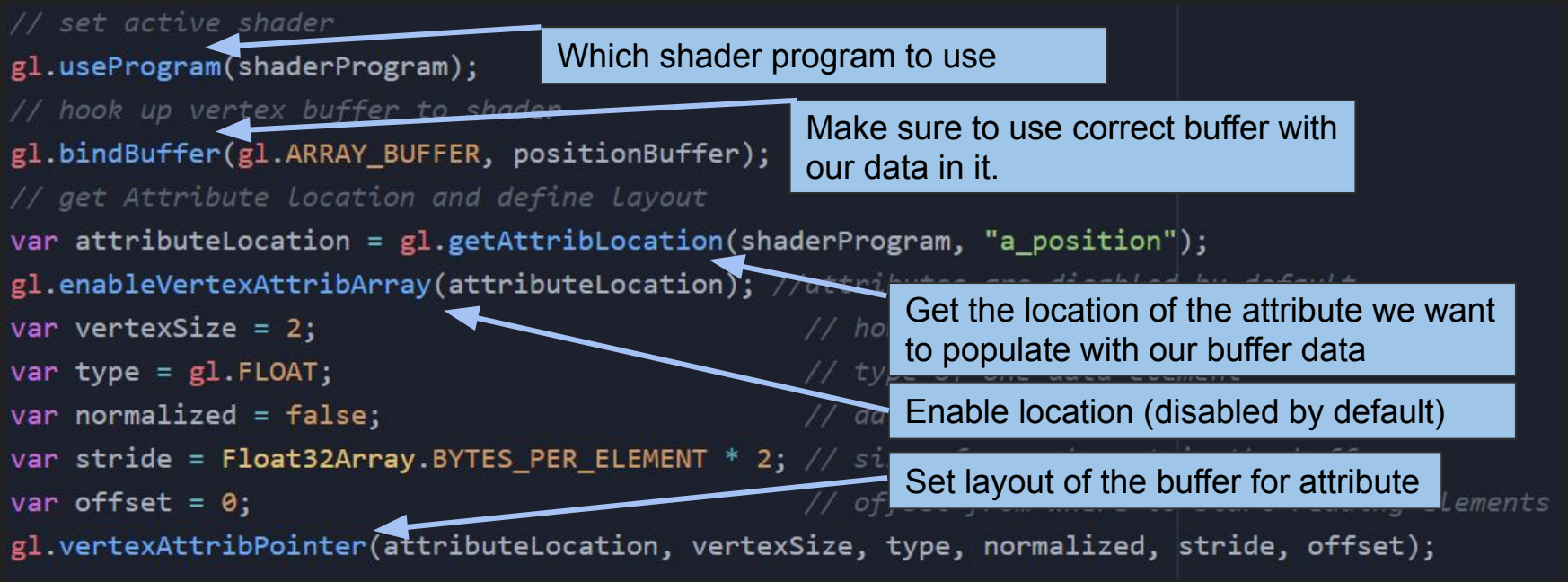
Transferring data to the GPU: Attribute Pointer

The GPU needs to know the layout of the buffer

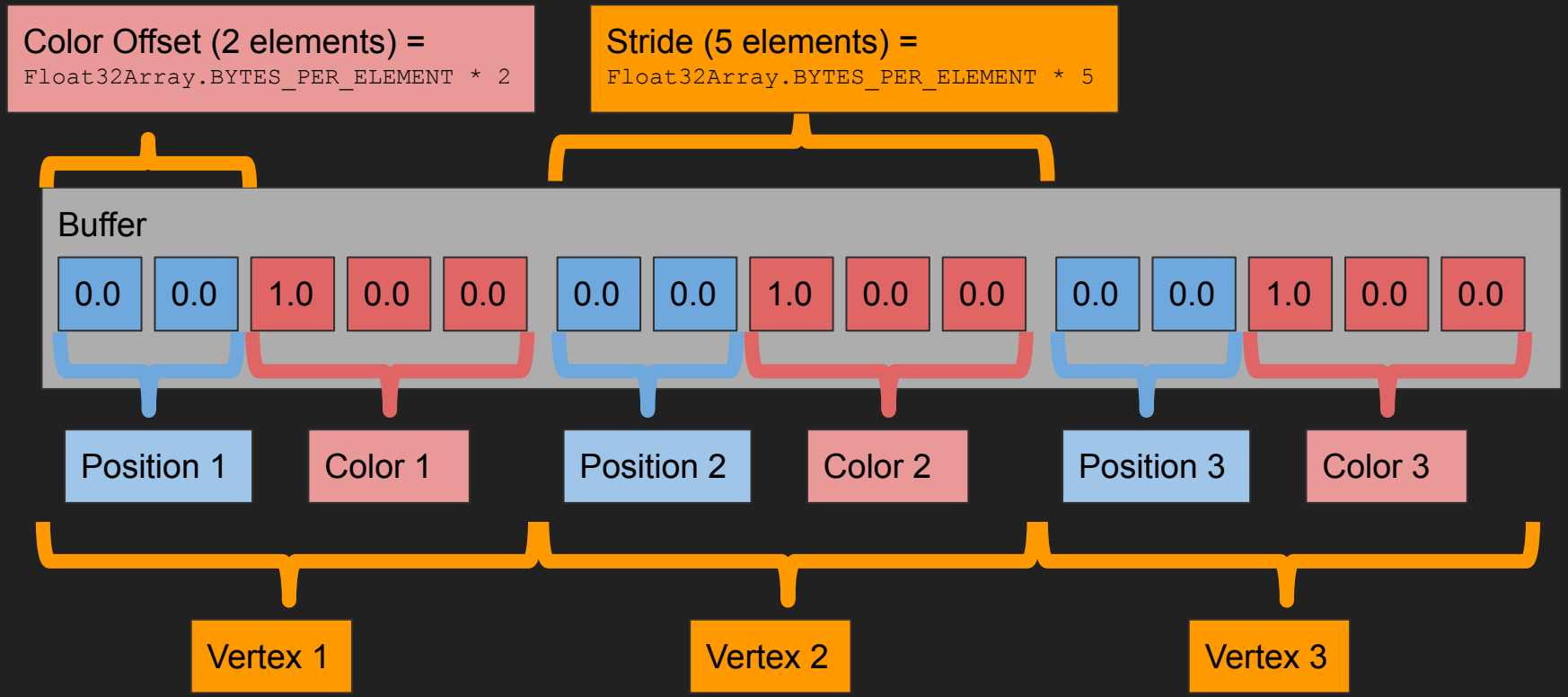
```
// set active shader
gl.useProgram(shaderProgram);
// hook up vertex buffer to shader
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// get Attribute location and define layout
var attributeLocation = gl.getAttribLocation(shaderProgram, "a_position");
gl.enableVertexAttribArray(attributeLocation); //attributes are disabled by default
var vertexSize = 2;                            // how many elements per attribute
var type = gl.FLOAT;                           // type of one data element
var normalized = false;                        // data needs to be normalized?
var stride = Float32Array.BYTES_PER_ELEMENT * 2; // size of one element in the buffer
var offset = 0;                                // offset from where to start reading elements
gl.vertexAttribPointer(attributeLocation, vertexSize, type, normalized, stride, offset);
```

Transferring data to the GPU: Attributes

Attributes always require buffers!



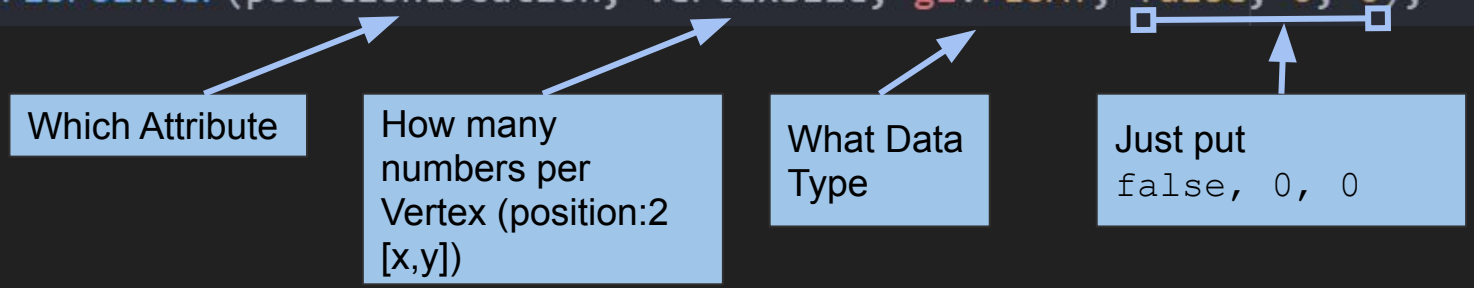
One Buffer can contain data for multiple attributes



The short version

How to connect a Buffer to a Vertex Shader Attribute during `draw()`

```
// hook up position buffer to shader
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// get Attribute location and define Pointer
let positionLocation = gl.getAttribLocation(shaderProgram, "a_position");
gl.enableVertexAttribArray(positionLocation); //attributes are disabled by default
let vertexSize = 2;                           //how many elements per attribute
gl.vertexAttribPointer(positionLocation, vertexSize, gl.FLOAT, false, 0, 0);
```



All attributes work the same,
no matter the type or what they are used for,
position, color etc….

Live example: Color buffer

Optimizing buffer management

During init:

1. Get Data
2. Format to FloatArray32
3. Create Buffer
4. Update Buffer Data

During draw:

1. Get Attribute Location and define Pointer
2. Draw

One Buffer can contain data for multiple attributes

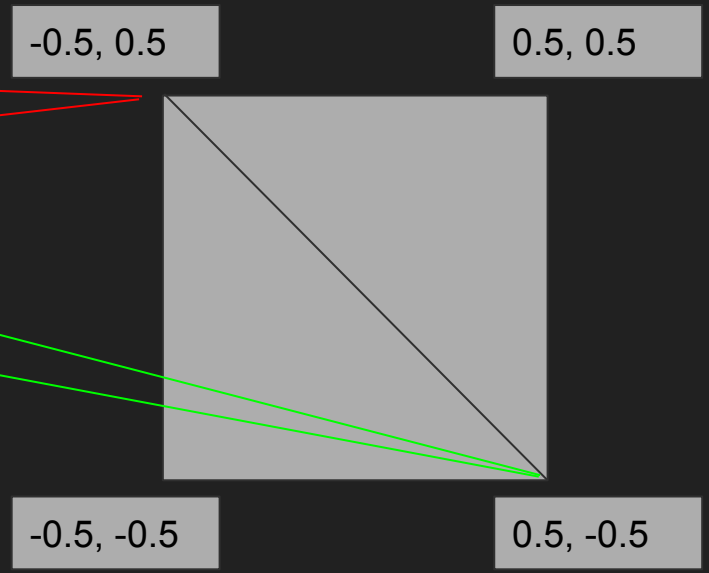
```
attributeBuffer = createArrayBuffer([
   -0.4,-0.4, // position 1
    1.0, 0.0, 0.0, // color 1
    0.4,-0.4, // position 2
    0.0, 1.0, 0.0, // color 2
    0.0, 0.4, // position 3
    0.0, 0.0, 1.0 // color 3
]);
```

```
// create Attribute location and define layout
var attributeLocation = gl.getAttribLocation(shaderProgram, "a_position");
var colorAttributeLocation = gl.getAttribLocation(shaderProgram, "a_color");
gl.enableVertexAttribArray(attributeLocation); //attributes are disabled by default
gl.enableVertexAttribArray(colorAttributeLocation);

var positionSize = 2;
var colorSize = 3;
var type = gl.FLOAT;
var normalized = false;
var stride = Float32Array.BYTES_PER_ELEMENT * (positionSize + colorSize); // size of one element in the buffer
var offset = 0;
gl.vertexAttribPointer(attributeLocation, positionSize, type, normalized, stride, offset);
offset = Float32Array.BYTES_PER_ELEMENT * positionSize;
gl.vertexAttribPointer(colorAttributeLocation, colorSize, type, normalized, stride, offset);

// draw geometry
gl.drawArrays(gl.TRIANGLES, 0, vertices);
```

Optimizing drawing: Element Arrays (Indexing)

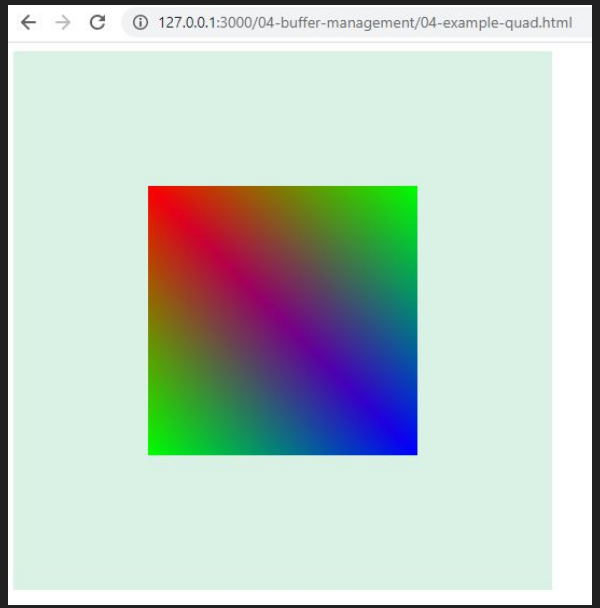
To draw complex shapes we need to repeat vertices that are part of more than one triangle

```
let positionBuffer = createArrayBuffer([
    -0.5, 0.5,
    -0.5,-0.5,
     0.5,-0.5,
    -0.5, 0.5,
     0.5,-0.5,
     0.5, 0.5
]);
```



Every attribute needs to exist for every vertex!

```
let positionBuffer = createArrayBuffer([
    -0.5, 0.5,
    -0.5,-0.5,
     0.5,-0.5,
    -0.5, 0.5,
     0.5,-0.5,
     0.5, 0.5
]);

let colorBuffer = createArrayBuffer([
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    1.0, 0.0, 0.0,
    0.0, 0.0, 1.0,
    0.0, 1.0, 0.0,
]);
```



Using element array buffers (index buffers)

```
let positionBuffer = createArrayBuffer([
    -0.5, 0.5,
    -0.5,-0.5,
     0.5,-0.5,
     0.5, 0.5
]);
let colorBuffer = createArrayBuffer([
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0,
    0.0, 1.0, 0.0
]);
let indexData = [0, 1, 2,    //first triangle
                 2, 3, 0];  //second triangle
```

New indexed data with only 4 positions and color for our 4 vertices and corresponding index data listing triangles

```
// hook up index buffer to vao
let indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indexData), gl.STATIC_DRAW);
```

Index Buffer

Drawing with Element Arrays (indexed drawing)

```
// draw geometry
gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
```

Syntax

```
void gl.drawElements(mode, count, type, offset);
```

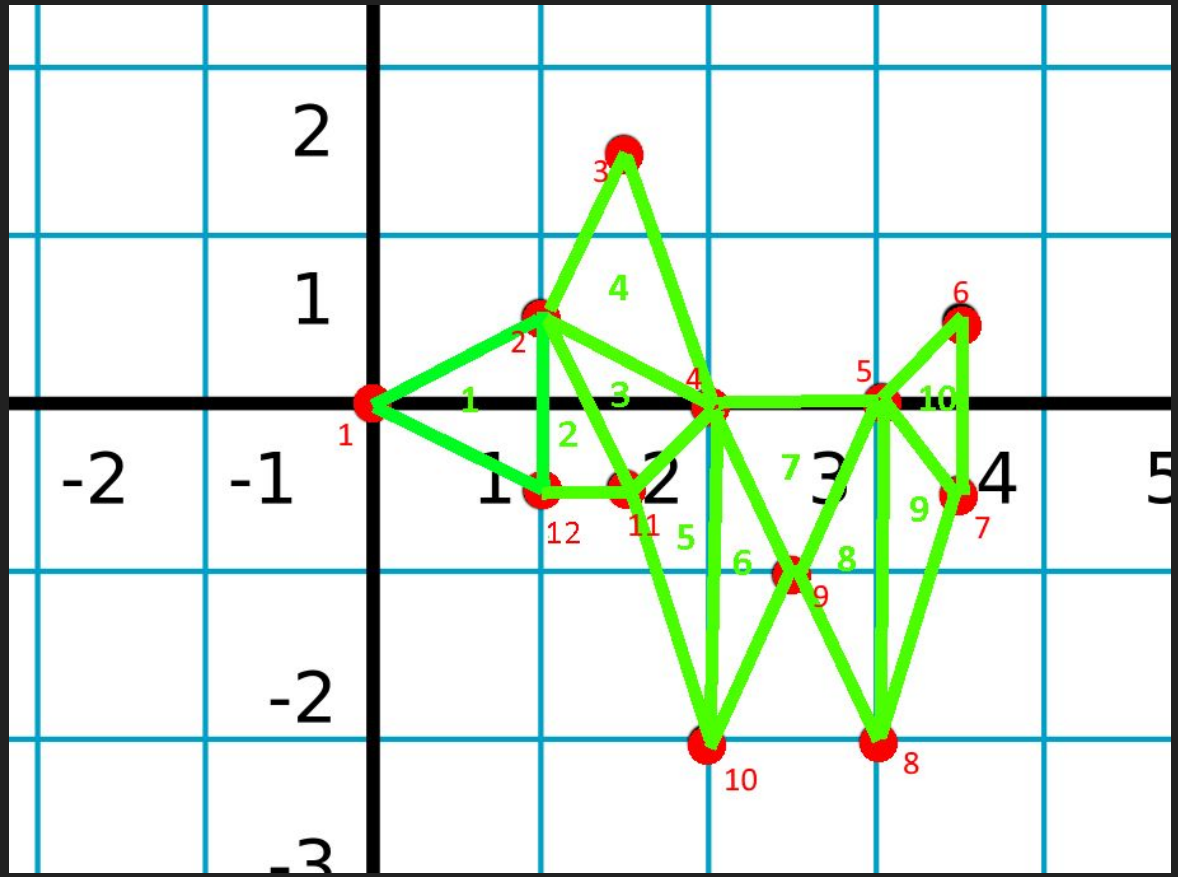
Mini Exercise: Draw this:

Triangles

#			
1	1	2	12
2	12	2	11
3	11	2	4
4	4	2	3
5	10	11	4
6	10	4	9
7	9	4	5
8	8	9	5
9	7	8	5
10	6	7	5

Vertices

#	X	Y
1	0.0,	0.0
2	1.0,	0.5
3	1.5,	1.5
4	2.0,	0.0
5	3.0,	0.0
6	3.5,	0.5
7	3.5,	-0.5
8	3.0,	-2.0
9	2.5,	-1.0
10	2.0,	-2.0
11	1.5,	-0.5
12	1.0,	-0.5

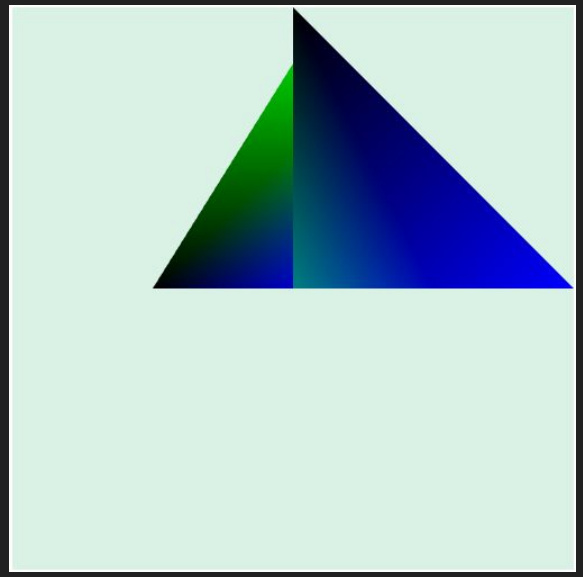

2
1
-2
-1
1
2
3
4
5
-2
-3
1
2
3
4
5
6
7
8
9
10
11
12
1
2
3
4
5
6
7
8
9
10

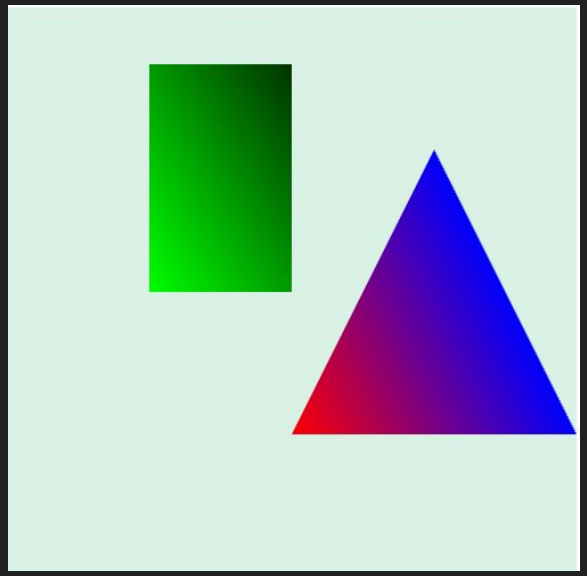
Exercise: Fixme

Fix: class04-exercise-fixme.html

Tipp: include webgl-debug.js

The data arrays for color and position can not be altered!
Their information is correct.
There are multiple errors and bugs.





Correct Solution

Check Buffer Size

How to check if your buffer has the correct Data:

During `draw()` {for each buffer that has to be attached}

After calling
```
gl.bindBuffer(gl.ARRAY_BUFFER, myBuffer);
```

Get the amount of BYTES contained in the currently bound Buffer by calling:
```
Let bf = gl.getBufferParameter(gl.ARRAY_BUFFER, gl.BUFFER_SIZE);
```
Print with
```
console.log("currently bound buffer contains: " + bf + " Bytes");
```

Before calling
```
gl.vertexAttribPointer(...)
```

Compare with the array that you used to fill the bugger (gl.bufferData())
The size (in BYTES) should be [number of floats in array] * 4

Example: Triangle Color Data:
[1.0, 0.0, 0.0,
0.0, 0.0, 1.0,
0.0, 0.0, 1.0] => 9 Data Points

Buffer Size should be 9*4=36 b