WebGL™

Part I

Presentation by: Quincy Jacobs

The Tools

Getting all the tools necessary to get started with WebGL is surprisingly easy.

1. Check your browser's compatibility: webglreport.com
2. Open your favorite text editor or IDE for JavaScript.

Now you're good to go.

The HTML

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>WebGL Example 1</title>
  </head>

  <body>
    <canvas id="glcanvas" width="640" height="480"></canvas>
  </body>

  <script src="index.js"></script>
</html>
```

Our actual WebGL code goes into index.js

The CSS

The First Step

Drawing a black background using WebGL

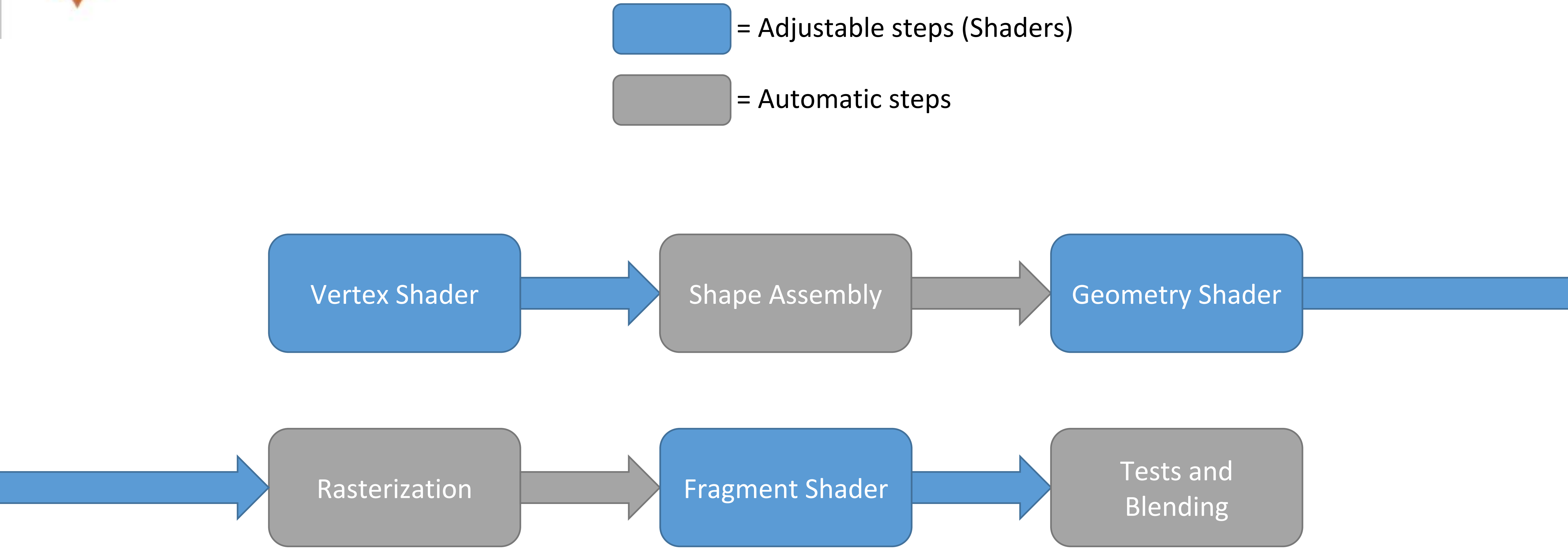
```
main();

function main(){
    const canvas = document.querySelector('#glcanvas');
    const gl = canvas.getContext('webgl2');

    // make sure we have a gl context
    if (!gl) {
        alert('Unable to initialize WebGL. Your browser or machine may not support it.');
        return;
    }

    // set clear color to black, fully opaque
    gl.clearColor(0.0, 0.0, 0.0, 1.0);

    // clear the color buffer with specified clear color
    gl.clear(gl.COLOR_BUFFER_BIT);
}
```

The First Step

You can download/clone the code from my Github:

https://github.com/QuincyJacobs/WebGLTutorial

Graphics pipeline



Steps for drawing a triangle

We will need all of the following WebGL objects to draw a triangle:

1. Vertex array buffer
1. Element array buffer
1. Vertex shader
1. Fragment shader
1. Shader program

Steps for drawing a triangle

These necessary elements are created and used in the following steps:

1. Array Buffer Objects

1. Shaders and Shader Program

1. Linking Buffers and Shaders

1. Drawing

Step 1: Array Buffer Objects

Array Buffer Objects

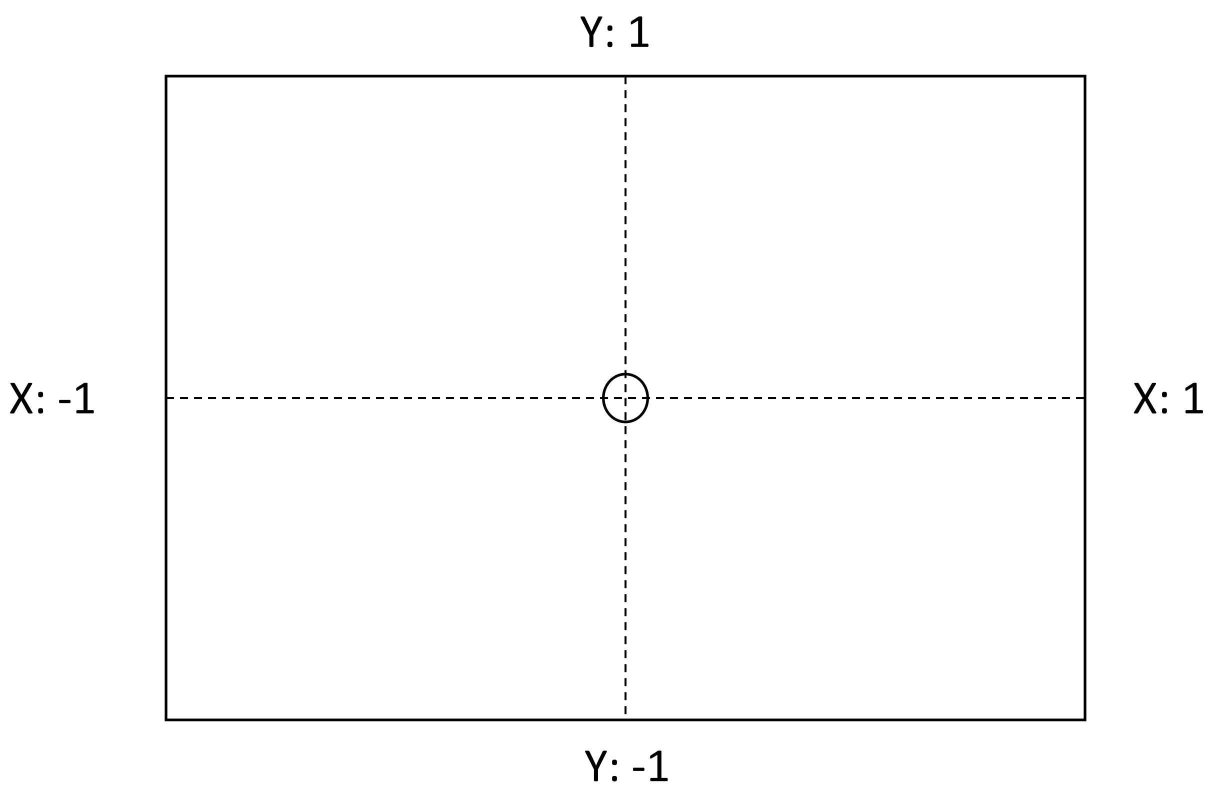
A Vertex Buffer contains information about each vertice, which shaders will use for calculations and translations.

Our vertex buffer will only contain 3d positions for each vertice:

	Vertice 1			Vertice 2			Vertice 3			
VERTEX BUFFER	Position x	Position y	Position z	Position x	Position y	Position z	Position x	Position y	Position z	. . .

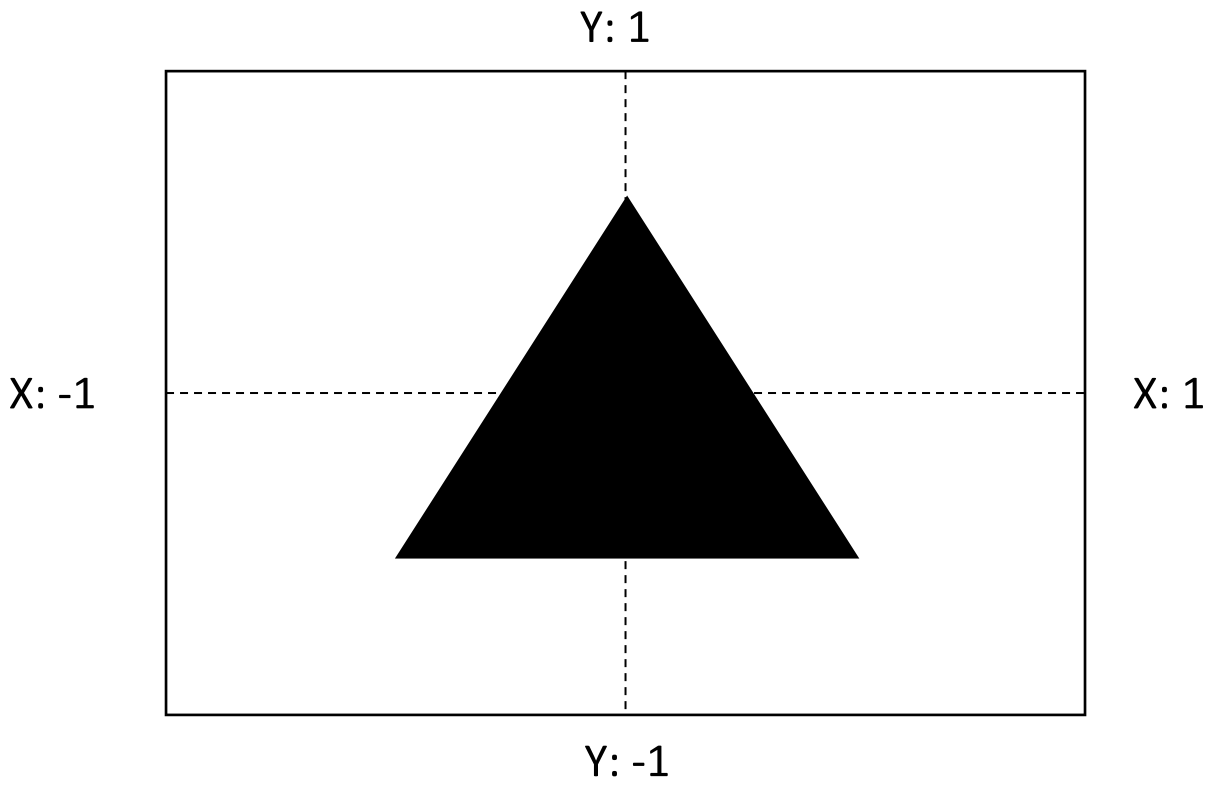
Array Buffer Objects

How does WebGL interpret the view?



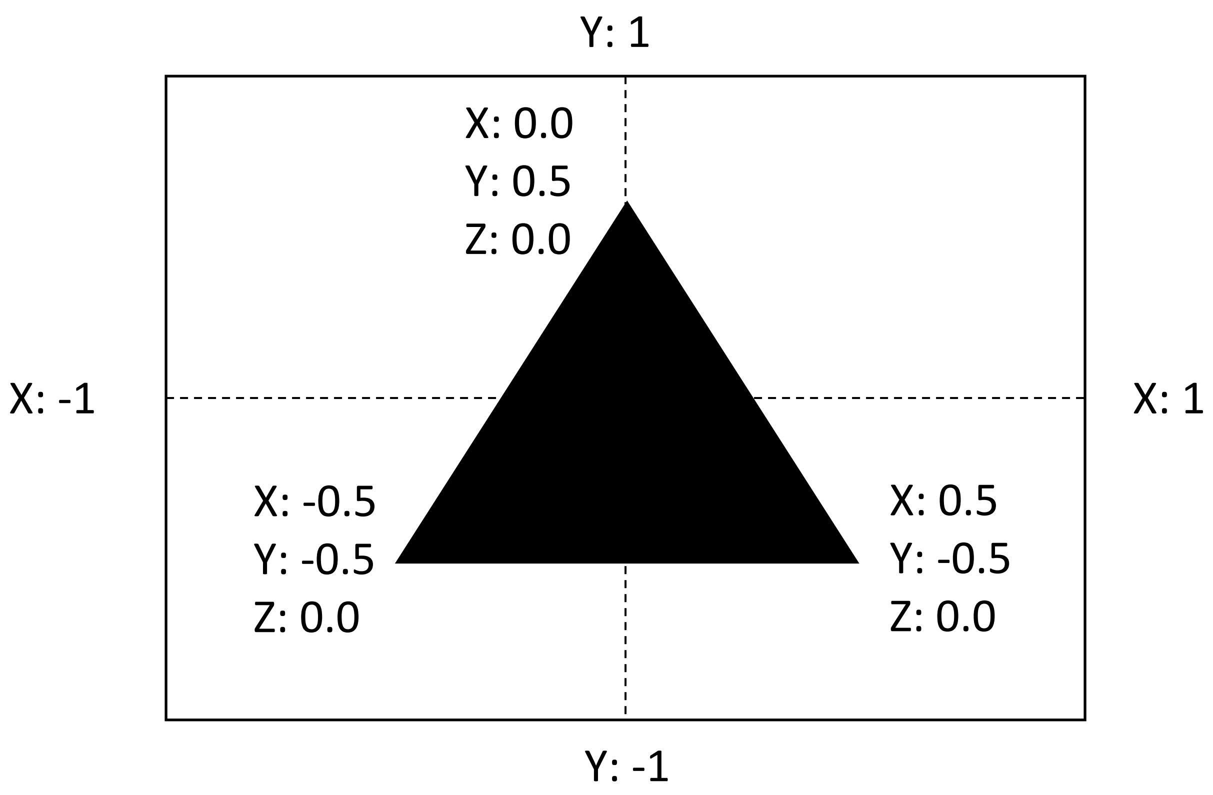
Array Buffer Objects

What will our triangle look like?



Array Buffer Objects

What data do we need?



Array Buffer Objects

The result:

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

Let’s format this mess…

Array Buffer Objects

The result:

```
var positions = [
    // x      y      z
    -0.5, -0.5,   0.0,    // lower left corner
     0.5, -0.5,   0.0,    // lower right corner
     0.0,  0.5,   0.0     // upper corner
];
```

Better!

Array Buffer Objects

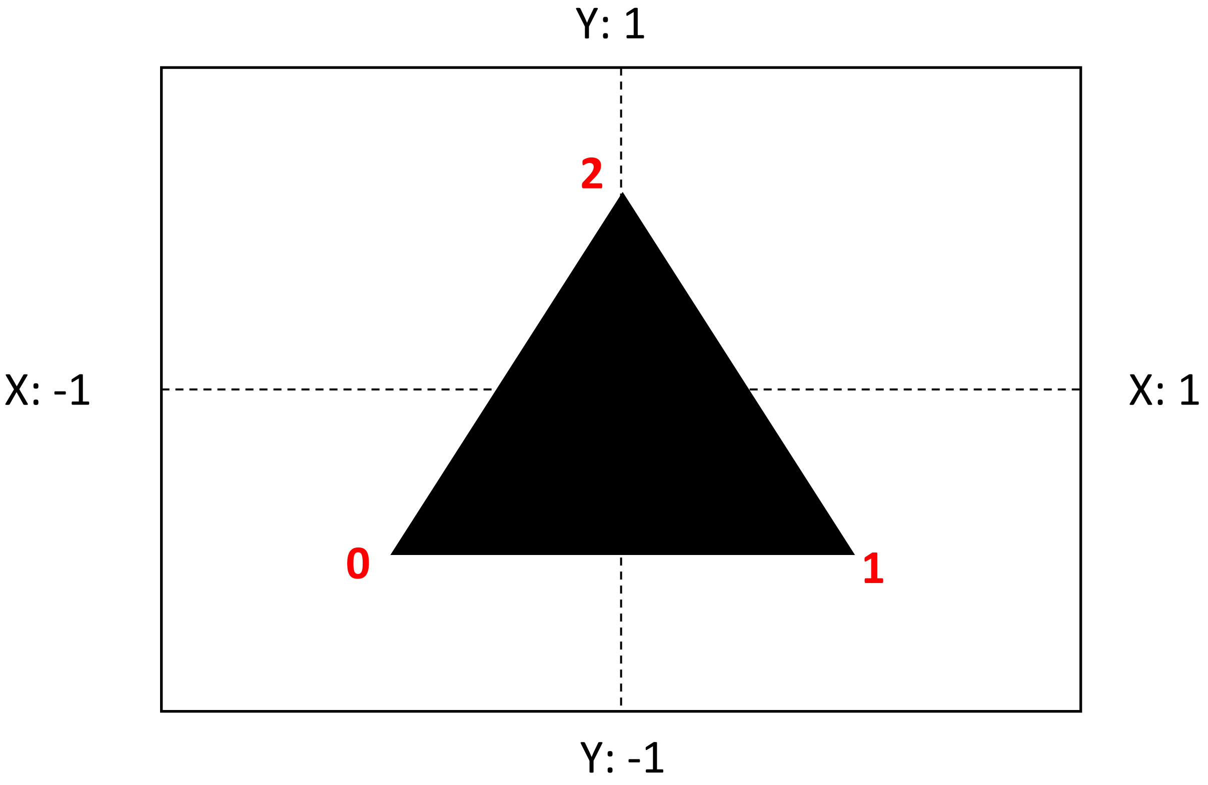
Now let's prepare our buffer so WebGL can eventually send it to the shaders.

```
// create a buffer, bind it to webgl as our active buffer and put our vertices in the buffer
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

// unbind the buffer to prevent unwanted changes later
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

Array Buffer Objects

Next we will need a buffer that holds the indices that form a triangle.

```
     var positions = [
         // x      y      z
0 ->    -0.5,  -0.5,   0.0,    // lower left corner
1 ->     0.5,  -0.5,   0.0,    // lower right corner
2 ->     0.0,   0.5,   0.0     // upper corner
     ];
```



Array Buffer Objects

The Indices will be saved as an Element Array Buffer. The way to do this is similar to the (Vertex) Array Buffer

```
var indices = [0, 1, 2];

// now we will do the same for the indices
var index_buffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Array Buffer Objects Results

```
// create an array holding all our vertex positions.    Note: This will become a triangle
var positions = [
    // x      y      z
    -0.5, -0.5,   0.0,    // lower left corner
     0.5, -0.5,   0.0,    // lower right corner
     0.0,  0.5,   0.0     // upper corner
];

// indices show the 3 points that create a triangle. This will not be of much help with just
// 1 triangle, but once we start drawing more it will start making more sense.
var indices = [0, 1, 2];

// create a buffer object, bind it to webgl as our active buffer and put our vertices in the buffer
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);

// unbind the buffer to prevent unwanted changes later
gl.bindBuffer(gl.ARRAY_BUFFER, null);

// now we will do the same for the indices
var index_buffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

Step 2: Shaders and Shader Program

Shaders and Shader Program

As we recall from the graphics pipeline slide, we can supply code for 3 Shaders:

Vertex Shader

Geometry Shader

Fragment Shader

We will only create a Vertex Shader and a Fragment Shader. The Geometry Shader is not necessary.

Shaders and Shader Program

Shaders are written in their own language called GLSL (OpenGL Shader Language)

Our Shader code will be defined as a constant string.

Vertex Shader

```
// the vertex shader source code
const vertexShaderSource = `#version 300 es

    in vec3 coordinates;

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

Fragment Shader

```
// the fragment shader source code
const fragmentShaderSource = `#version 300 es

    precision mediump float;

    out vec4 fragmentColor;

    void main() {
        fragmentColor = vec4(0.0, 0.0, 0.0, 1.0);
    }
`;
```

Shaders and Shader Program

In the next step we create Shader Objects, attach the source code, and compile the source code.

Vertex Shader

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

Fragment Shader

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

Shaders and Shader Program

These lines will check if the shader compiled and give an error if it was not compiled.

```
// debugging the shader
if (!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)) {
    alert('An error occurred compiling the shaders: ' + gl.getShaderInfoLog(fragmentShader));
    gl.deleteShader(fragmentShader);
    return null;
}
```

Shaders and Shader Program

Now we connect the shaders in a program and tell WebGL to use the program.

```
// finally create a shader program that links our shaders to each other.
var shaderProgram = gl.createProgram();

// attach our shaders to the program
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);

// link the shader programs to each other
gl.linkProgram(shaderProgram);

// tell WebGL to use our shader program
gl.useProgram(shaderProgram);
```

Step 3: Linking Buffers and Shaders

Linking Buffers and Shaders

This is all the code necessary to link the buffers and shaders.

```
// bind our buffer objects to WebGL
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);

// get the point in our vertex shader where we can insert our positions
var coordinatePosition = gl.getAttribLocation(shaderProgram, "coordinates");

// tell the vertex shader how to interpret the vertex data.
gl.vertexAttribPointer(coordinatePosition, 3, gl.FLOAT, false, 0, 0);

// tell WebGL to enable the vertex attribute we just specified.
gl.enableVertexAttribArray(coordinatePosition);
```

Linking Buffers and Shaders

These 2 lines tell WebGL where to insert our Buffers into the shaders and how to interpret the Buffers.

```
// get the point in our vertex shader where we can insert our positions
var coordinatePosition = gl.getAttribLocation(shaderProgram, "coordinates");

// tell the vertex shader how to interpret the vertex data.
gl.vertexAttribPointer(coordinatePosition, 3, gl.FLOAT, false, 0, 0);
```

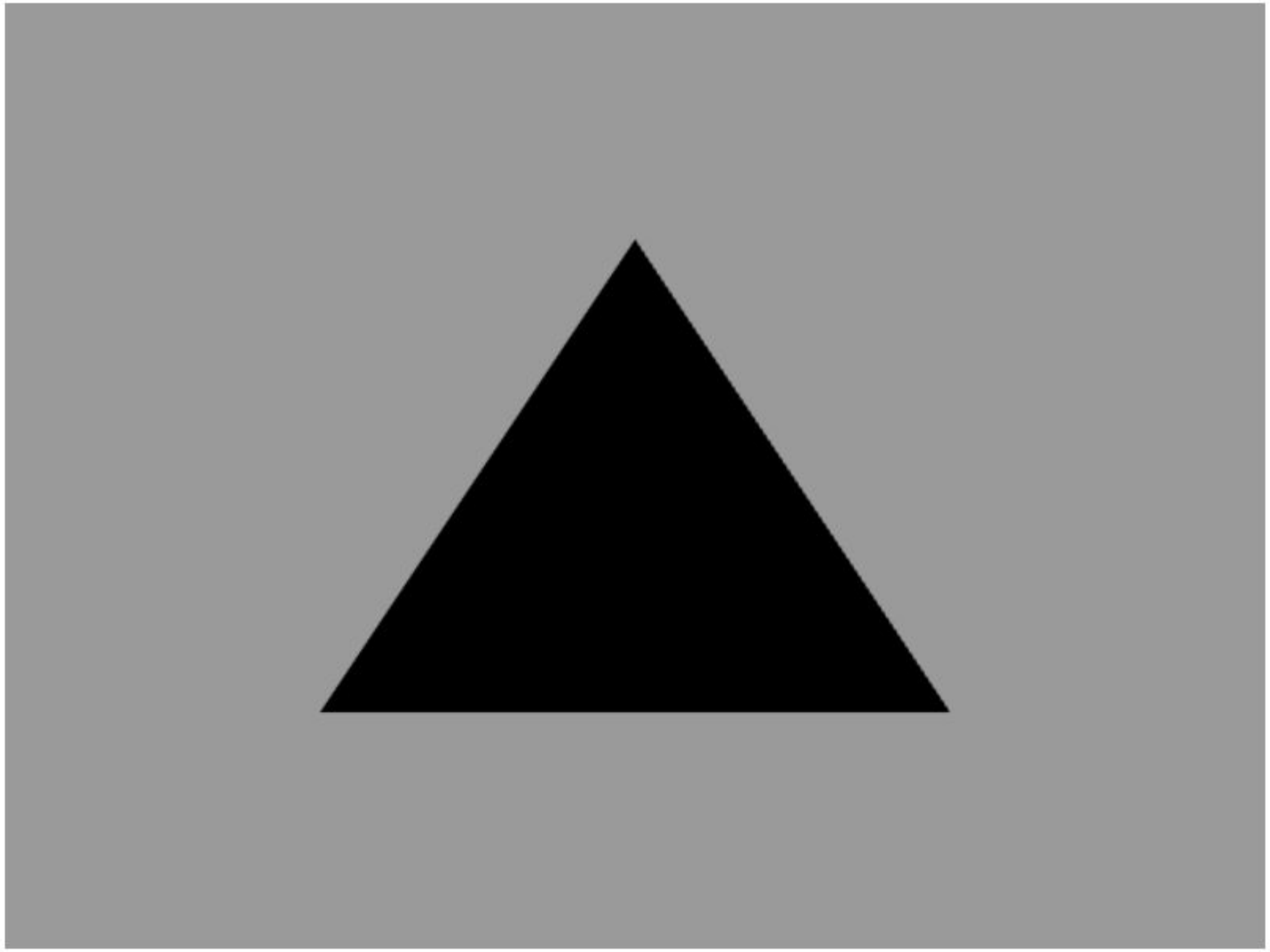
Step 4: Drawing

Drawing

This line tells WebGL to start drawing triangles.

```
// draw the triangle
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);
```

The result



Afsluiting

Thanks for listening.

WebGL™

Part II

Presentation by: Quincy Jacobs

Session preview

This session will focus on these points:

1. Colors array and (color) Vertex Buffer Object

1. Combining the Color and Position Vertex Buffers

1. Transformations and the math behind them
   Part 1: 2D vectors

Step 1: Colors array and (color) Vertex Buffer Object

Colors array and (color) Vertex Buffer Object

Similar to the position vertex array the color vertex array will have 3 elements: r, g, b.

	Vertice 1			Vertice 2			Vertice 3			
VERTEX BUFFER	R	G	B	R	G	B	R	G	B	…

How does WebGL interpret color data?

Colors array and (color) Vertex Buffer Object

WebGL takes in positive floats as RGBA color values between 0 and 1

R	0.0 … 1.0
G	0.0 … 1.0
B	0.0 … 1.0

What will the raw data look like in code?

Colors array and (color) Vertex Buffer Object

The array closely resembles the position array, and even the creation of the object is the same.

```
var colors = [
//    r     g     b
    1.0, 0.0, 0.0,   // lower left corner
    0.0, 1.0, 0.0,   // lower right corner
    0.0, 0.0, 0.0,   // upper left corner
    0.0, 0.0, 1.0,   // upper right corner
];

var color_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, color_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Where do we insert our colors in the shader?

Colors array and (color) Vertex Buffer Object

We have to specify a new input variable in the Vertex Shader, and give it to the Fragment Shader

```
// the vertex shader source code
const vertexShaderSource = `#version 300 es

    in vec3 v_position;
    in vec3 v_color;

    out lowp vec3 f_color;

    void main() {
        f_color = v_color;
        gl_Position = vec4(v_position, 1.0);
    }
`;
```

```
// the fragment shader source code
const fragmentShaderSource = `#version 300 es

    precision mediump float;

    in lowp vec3 f_color;

    out vec4 fragmentColor;

    void main() {
        fragmentColor = vec4(f_color, 1.0);
    }
`;
```

Next we will tie everything together in the code

Colors array and (color) Vertex Buffer Object

```
var colors = [
//    r     g     b
    1.0, 0.0, 0.0,  // lower left corner
    0.0, 1.0, 0.0,  // lower right corner
    0.0, 0.0, 0.0,  // upper left corner
    0.0, 0.0, 1.0   // upper right corner
];

var color_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, color_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);

var color_location = gl.getAttribLocation(shaderProgram, "v_color");
gl.vertexAttribPointer(color_location, 3, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, null);

// bind our buffer objects to WebGL
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bindBuffer(gl.ARRAY_BUFFER, color_buffer);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);

// tell WebGL to enable the vertex attribute we just specified.
gl.enableVertexAttribArray(position_location);
gl.enableVertexAttribArray(color_location);
```

Step 2: Combining the Color and Position Vertex Buffers

Combining the Color and Position Vertex Buffers

	Vertice 1			Vertice 2			Vertice 3			
POSITION BUFFER	Position x	Position y	Position z	Position x	Position y	Position z	Position x	Position y	Position z	. . .

	Vertice 1			Vertice 2			Vertice 3			
COLOR BUFFER	R	G	B	R	G	B	R	G	B	. . .

How should we combine these 2 arrays?

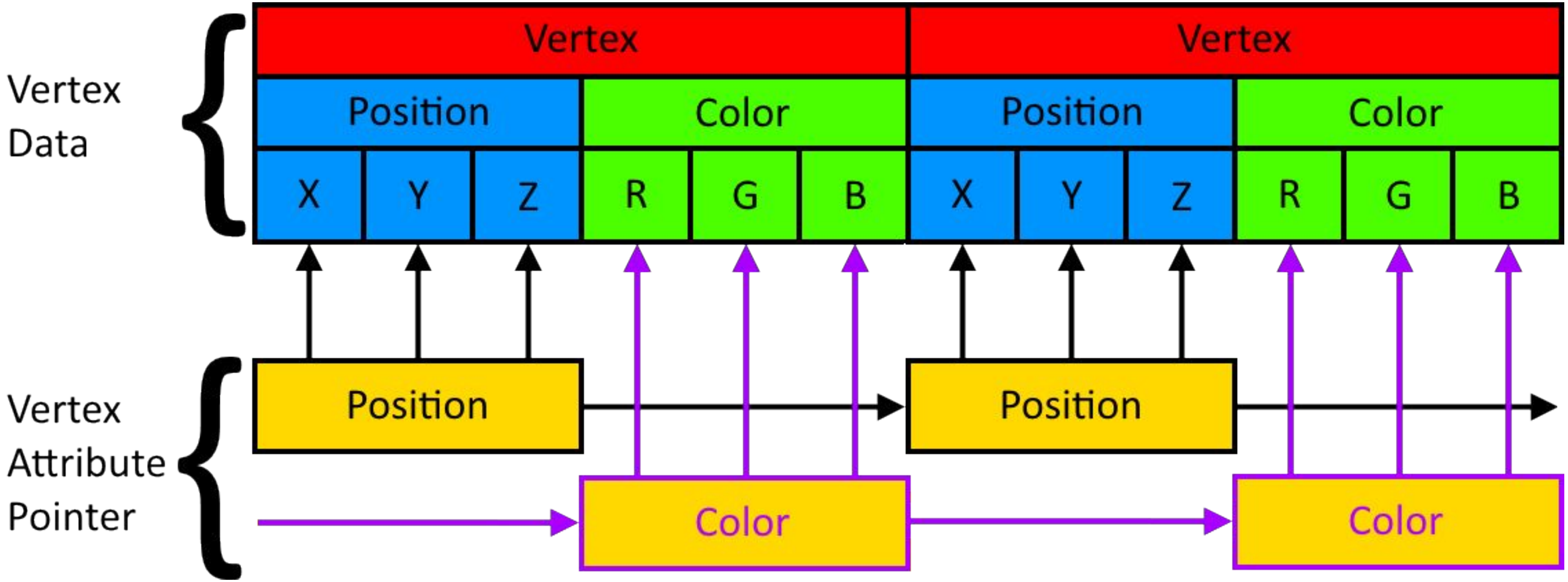
Combining the Color and Position Vertex Buffers

We simply place the position and color of each vertex right behind each other

How would WebGL filter these out and give them to the right in variable in the vertex shader?

Combining the Color and Position Vertex Buffers

We use 2 vertex attribute pointers, one for positions and one for colors. Now let's see the code.



Combining the Color and Position Vertex Buffers

The code simply pastes the rgb values after the xyz values.

```
// create an array holding all our vertex data.
var vertices = [
   //---positions---\   /---colors---\
   // x     y     z       r     g     b
    -0.5, -0.5,  0.0,    1.0, 0.0, 0.0,   // lower left corner
     0.5, -0.5,  0.0,    0.0, 1.0, 0.0,   // lower right corner
    -0.5,  0.5,  0.0,    0.0, 0.0, 0.0,   // upper left corner
     0.5,  0.5,  0.0,    0.0, 0.0, 1.0,   // upper right corner
];
```

How will we feed the Vertex fragment with this data?

Combining the Color and Position Vertex Buffers

We use 2 pointers as specified in the image before.

```
// create a buffer, bind it to webgl as our active buffer and put our vertices in the buffer
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// position attribute pointer
var position_location = gl.getAttribLocation(shaderProgram, "v_position");
gl.vertexAttribPointer(position_location, 3, gl.FLOAT, false, 6*4, 0);

// color attribute pointer
var color_location = gl.getAttribLocation(shaderProgram, "v_color");
gl.vertexAttribPointer(color_location, 3, gl.FLOAT, false, 6*4, 3*4);
```

There is just one more loose end left.

Combining the Color and Position Vertex Buffers

We remove our color buffer and don't bind it anymore.

```
// bind our buffer objects to WebGL
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, index_buffer);

// tell WebGL to enable the vertex attribute we just specified.
gl.enableVertexAttribArray(position_location);
gl.enableVertexAttribArray(color_location);
```

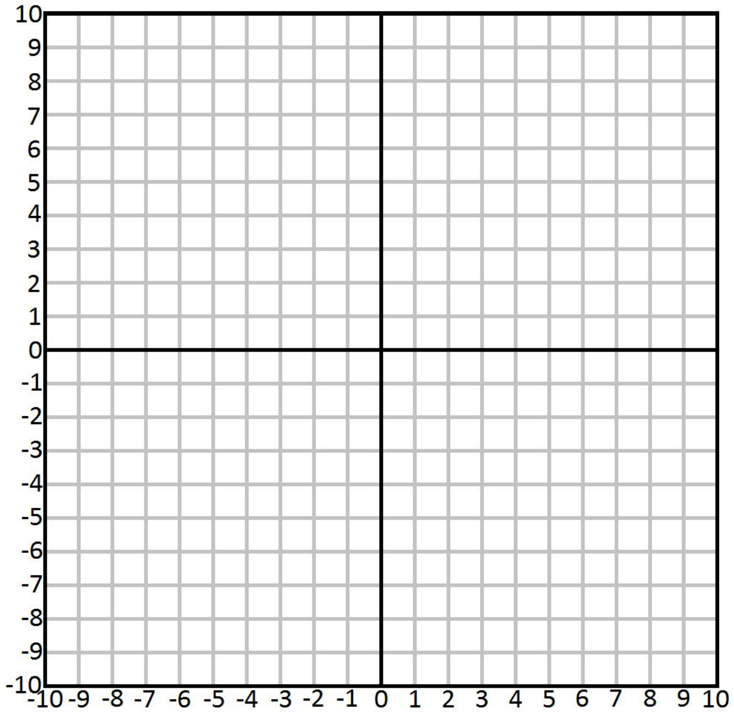
The location is still necessary as that is the point in the shader we will feed data to.

Step 3: Transformations and the math behind them
Part 1: 2D vectors

Disclaimer: I am not a mathematician

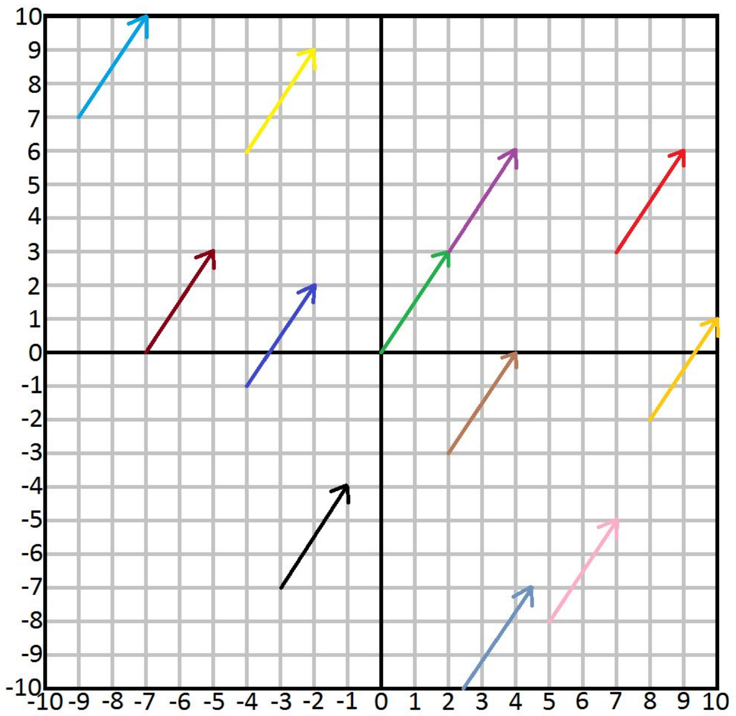
Math part 1: 2D vectors

Let's start off with 2D examples and slowly work into 3D and transformations.
This is a 2D Cartesian system.



Math part 1: 2D vectors

This is a 2D Cartesian system with vectors.



Which arrow indicates vector (2, 3)?

Math part 1: 2D vectors

All of them are the same vector.

A vector in mathematics indicates only 2 things:

1. **Direction**
2. **Magnitude**

A vector is usually notated with a bold symbol or a symbol with an arrow on top.

$$\mathbf{v} = \vec{V} = \begin{pmatrix} x \\ y \end{pmatrix}$$

The 2 numbers in a vector are usually displayed on top of each other for convenience

Math part 1: 2D vectors

Vectors can also be manipulated in many different ways.
Let's start with scalar operations:

Vector: $\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$

Scalar: $s = 5$

Multiplication: $\vec{v} * s = \begin{pmatrix} 2 * 5 \\ 5 * 5 \end{pmatrix} = \begin{pmatrix} 10 \\ 25 \end{pmatrix}$

Division: $\vec{v} \,/\, s = \begin{pmatrix} 2 / 5 \\ 5 / 5 \end{pmatrix} = \begin{pmatrix} 0.4 \\ 1 \end{pmatrix}$

The scalar will simply be applied to every member of the vector, even with 3D vectors.

Addition and Subtraction with just a scalar are weird:

Math part 1: 2D vectors

Vectors can also be manipulated in many different ways.
Let's start with scalar operations:

Vector: $\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$

Scalar: $s = 5$

Addition: $\vec{v} + s = \begin{pmatrix} 2 + 5 \\ 5 + 5 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix}$

Subtraction: $\vec{v} - s = \begin{pmatrix} 2 - 5 \\ 5 - 5 \end{pmatrix} = \begin{pmatrix} -3 \\ 0 \end{pmatrix}$

Multiplication: $\vec{v} * s = \begin{pmatrix} 2 * 5 \\ 5 * 5 \end{pmatrix} = \begin{pmatrix} 10 \\ 25 \end{pmatrix}$

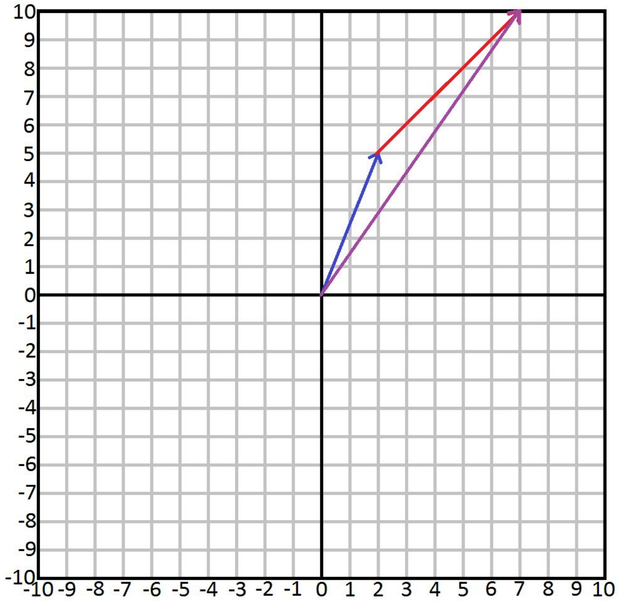
Division: $\vec{v} \,/\, s = \begin{pmatrix} 2 / 5 \\ 5 / 5 \end{pmatrix} = \begin{pmatrix} 0.4 \\ 1 \end{pmatrix}$

The scalar will simply be applied to every member of the vector, even with 3D vectors.

Why are adding and subtracting scalars a weird thing to do to vectors?

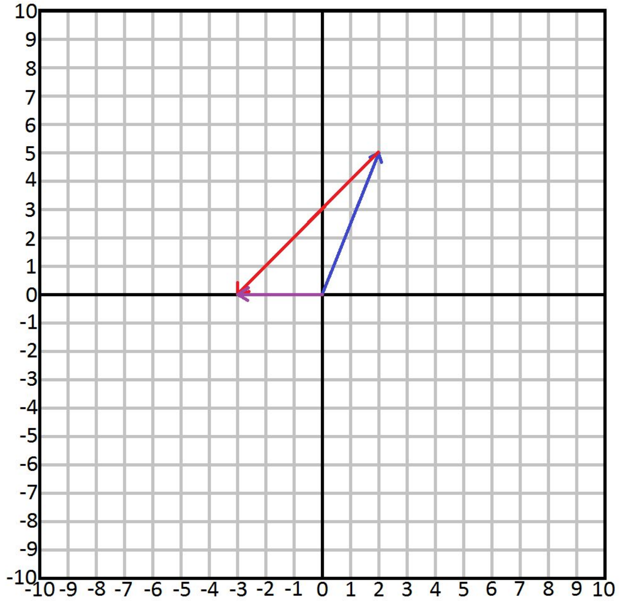
Math part 1: 2D vectors

Addition and subtraction can only move a vector like a bishop in chess.
There is no correlation because both direction and magnitude change in weird ways.



$$\text{Vector: } \mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

$$\text{Scalar: } s = 5$$



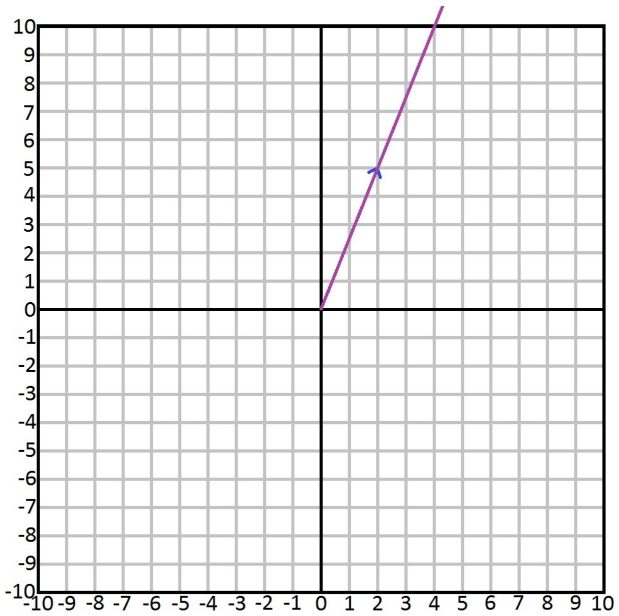
$$\textbf{Addition: } \vec{v} + s = \begin{pmatrix} 2 + 5 \\ 5 + 5 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix}$$

$$\textbf{Subtraction: } \vec{v} - s = \begin{pmatrix} 2 - 5 \\ 5 - 5 \end{pmatrix} = \begin{pmatrix} -3 \\ 0 \end{pmatrix}$$

Multiplying and Dividing by a scalar is much more useful.

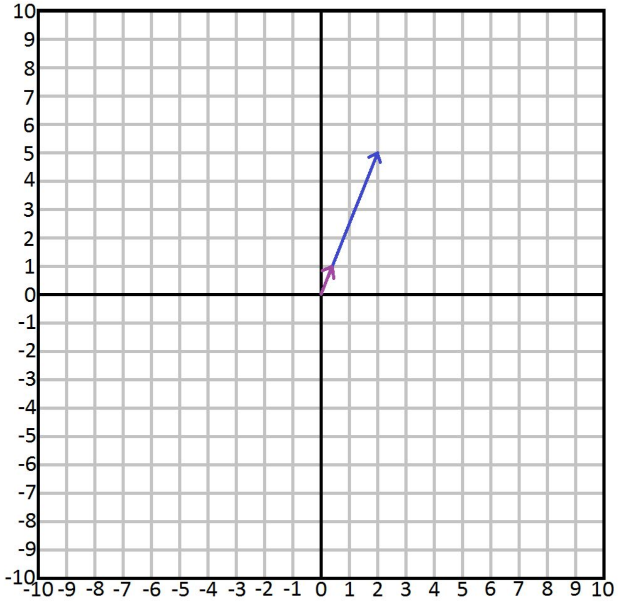
Math part 1: 2D vectors

Multiplication and Division lengthen or shorten the distance of the vector.
The direction will remain, making them very useful.



Vector: $\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$

Scalar: $s = 5$



Multiplication: $\vec{v} * s = \begin{pmatrix} 2 * 5 \\ 5 * 5 \end{pmatrix} = \begin{pmatrix} 10 \\ 25 \end{pmatrix}$

Division: $\vec{v} \,/\, s = \begin{pmatrix} 2 / 5 \\ 5 / 5 \end{pmatrix} = \begin{pmatrix} 0.4 \\ 1 \end{pmatrix}$

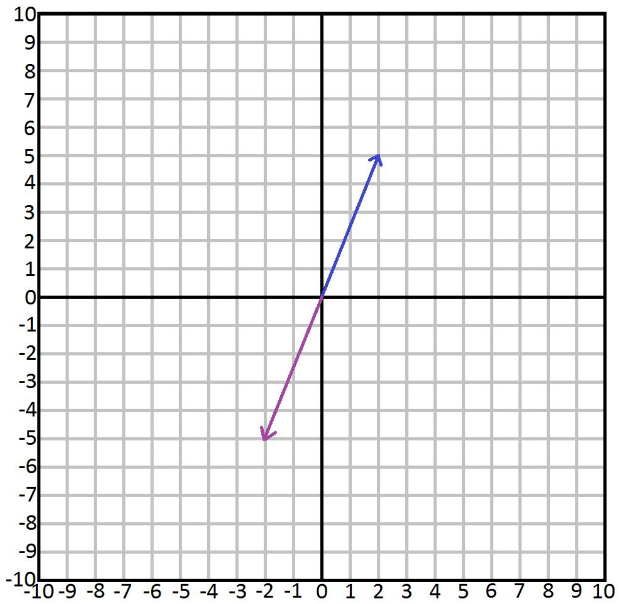
Another useful operation is flipping the vector. Any idea how you can do this?

Math part 1: 2D vectors

Flipping a vector is done by simple multiplying the vector with the scalar -1.

Vector: $\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$

Scalar: s = -1



Multiplication: $\vec{v} * s = \begin{pmatrix} 2 * -1 \\ 5 * -1 \end{pmatrix} = \begin{pmatrix} -2 \\ -5 \end{pmatrix}$

Let's continue to vector on vector operations.

Math part 1: 2D vectors

Take these 2 vectors:

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \qquad \mathbf{w} = \vec{w} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Addition: $\vec{v} + \vec{w} = \begin{pmatrix} 2 + 3 \\ 5 + 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$

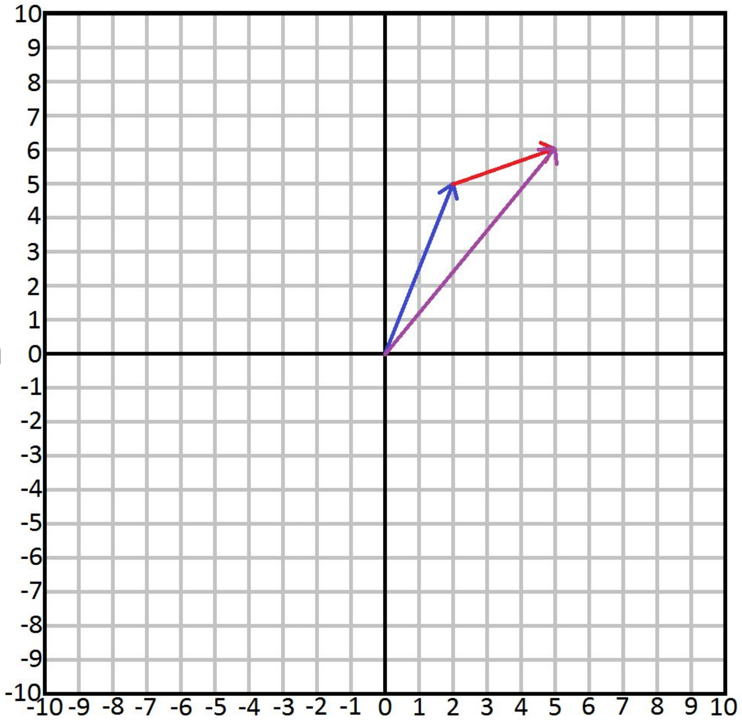
Subtraction: $\vec{v} - \vec{w} = \begin{pmatrix} 2 - 3 \\ 5 - 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$

Seeing this visually will make it much clearer.

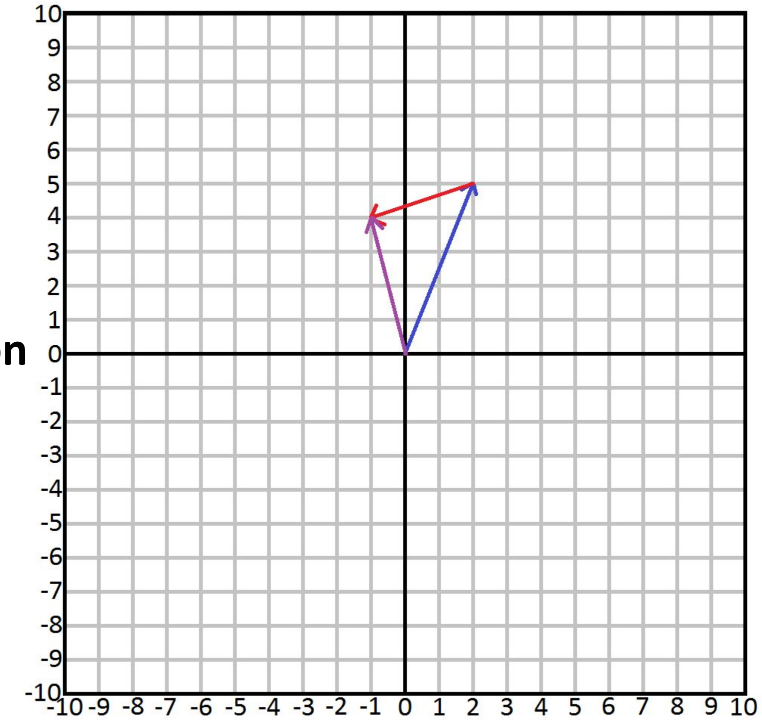
Math part 1: 2D vectors

This is the visual representation of the Addition and Subtraction of our **v** and **w** vectors

Addition



Subtraction

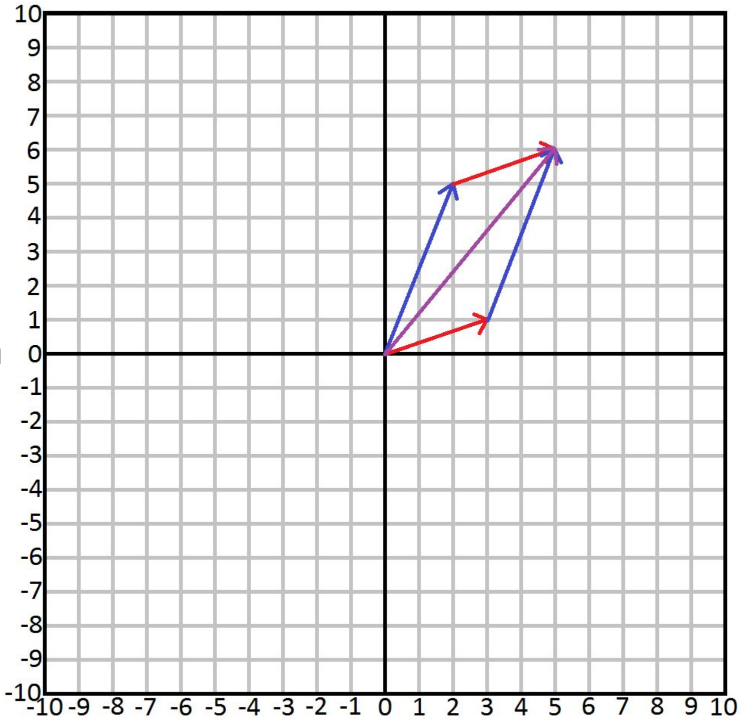


Will these vectors have the same outcome regardless of order?

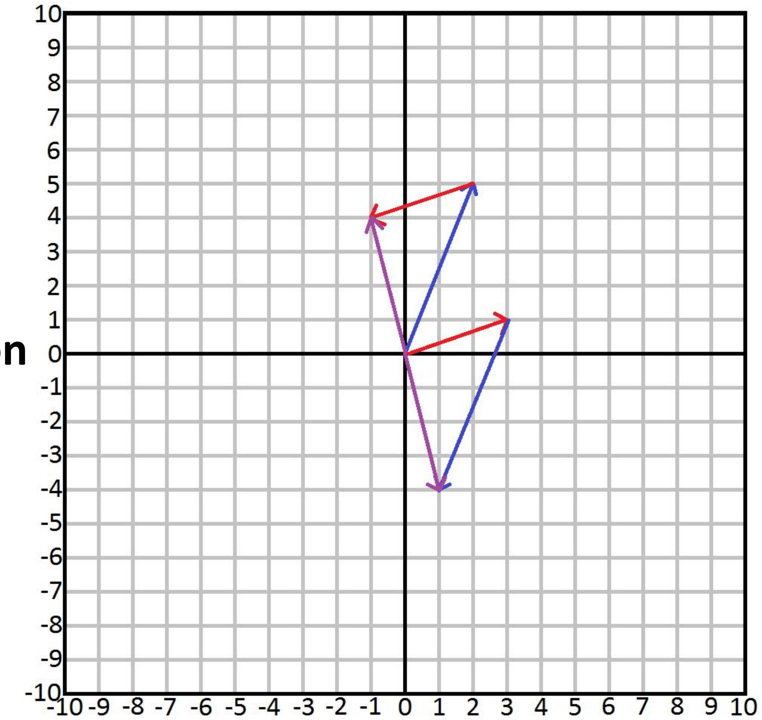
Math part 1: 2D vectors

Addition does, subtraction does not.

Addition



Subtraction



The math will continue at the end of part III

Math part 1: 2D vectors

To be continued next time…

Afsluiting

Thanks for listening.

WebGL™

Part III

Presentation by: Quincy Jacobs

Session preview

This session will focus on these points:

1. Adding textures

1. Transformations and the math behind them
   Part 2: 2D vectors

Step 1: Adding textures

Adding textures

In part 3 we will first add textures to our program before jumping into the math again.

We will render this image over a square.

Adding textures

First we have to create a texture and bind it, very similar to our buffers.

```
// create a texture and bind it to the gl context
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);

// image placeholder until it has been loaded from a file
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE,
        new Uint8Array([127, 127, 127, 255]));
```

Next we will load an image from an external source.

Adding textures

We load the image from an external path.

```
var image = new Image();
path = 'https://image.source/image';
requestCORSIfNotSameOrigin(image, path);
image.src = path;
```

Loading an image means dealing with Cross-Origin Resource Sharing (CORS)

```
// function to request CORS for Cross-Origin Images
function requestCORSIfNotSameOrigin(img, url)
{
  if ((new URL(url)).origin !== window.location.origin)
  {
    img.crossOrigin = "";
  }
}
```

Next we will set the texture parameters once the image has been loaded

Adding textures

We want to set the texture settings after the image is loaded.

```
// when the image is loaded, set the texture properties
image.addEventListener('load', function()
{
    // set 4 different texture settings (https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/texParameter)
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

    // instead of setting texParameteri 4 times we can use the line below,
    // but we will also lose the ability to alter texture behavior.
    //gl.generateMipmap(gl.TEXTURE_2D);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

});
```

Let’s briefly visit each of these texture parameters

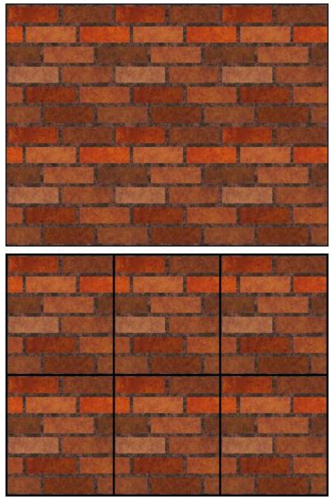
Adding textures

First we set the TEXTURE_WRAP_S parameter.

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
```

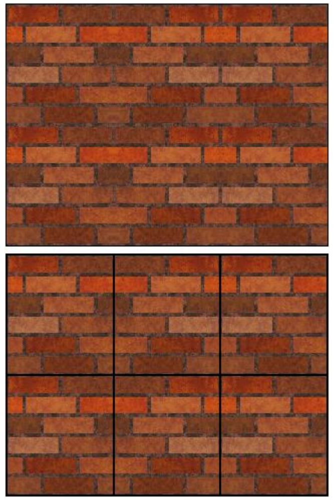
There are 3 options for this parameter:

gl.REPEAT



gl.CLAMP_TO_EDGE

gl.MIRRORED_REPEAT



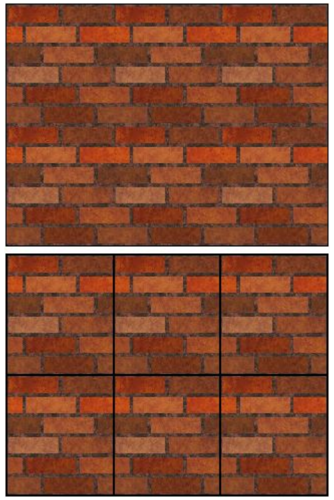
Adding textures

Next we set the TEXTURE_WRAP_T parameter.

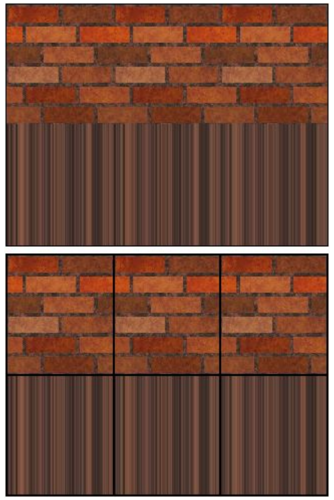
```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

The parameters are the same as TEXTURE_WRAP_S, but travel on the y-axis:

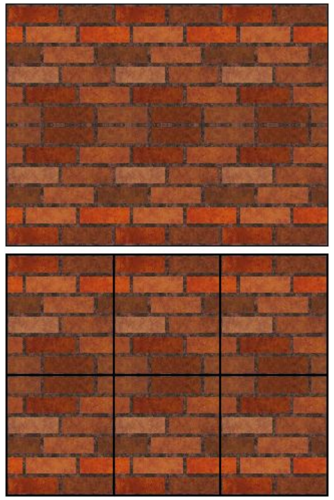
gl.REPEAT



gl.CLAMP_TO_EDGE



gl.MIRRORED_REPEAT



Adding textures

Lastly we set the TEXTURE_MIN_FILTER and TEXTURE_MAG_FILTER parameters.

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

These options have more delicate differences, so here are the options without image:

GL_NEAREST - (no filtering, no mipmaps)

GL_LINEAR - (filtering, no mipmaps)

GL_NEAREST_MIPMAP_NEAREST - (no filtering, sharp switching between mipmaps)

GL_NEAREST_MIPMAP_LINEAR - (no filtering, smooth transition between mipmaps)

GL_LINEAR_MIPMAP_NEAREST - (filtering, sharp switching between mipmaps)

GL_LINEAR_MIPMAP_LINEAR - (filtering, smooth transition between mipmaps)

Adding textures

Now we have everything we need to load the image.

We need a few more things to actually show the image on our object:

1. Texture logic in Shaders
2. Texture Vertex Buffer Object
3. Loading in a texture

Let’s start with the shaders

Adding textures

We will abuse the vertex shader as a conduit, just like we did with colors.

```
// the vertex shader source code
const vertexShaderSource = `#version 300 es

    in vec3 v_position;
    in vec3 v_color;
    in vec2 v_texture;

    out lowp vec3 f_color;
    out lowp vec2 f_texture;

    void main() {
        f_color = v_color;
        f_texture = v_texture;
        gl_Position = vec4(v_position, 1.0);
    }
`;
```

Do note that in contrast to color, texture is a vec2.

Adding textures

In the fragment shader we will use the texture instead of the color.

```
// the fragment shader source code
const fragmentShaderSource = `#version 300 es

    precision mediump float;

    in lowp vec3 f_color;
    in lowp vec2 f_texture;

    out vec4 fragmentColor;

    uniform sampler2D u_texture;

    void main() {
        fragmentColor = texture(u_texture, f_texture);
        //fragmentColor = vec4(f_color, 1.0);
    }
`;
```

Let’s briefly go through what happens here.

Adding textures

Receiving the texture coordinate values from the vertex shader should be nothing new.

```
in lowp vec2 f_texture;
```

Now we declare a uniform where we input the actual image of the texture.

```
uniform sampler2D u_texture;
```

Finally we let webgl do the work using the texture and the texture coordinates.

```
fragmentColor = texture(u_texture, f_texture);
```

This finishes up the shaders, we will now move on to the Vertex Buffer Object.

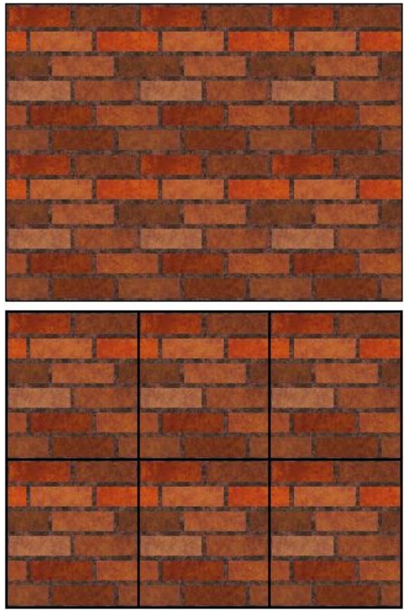
Adding textures

First we have to add vertice data for our texture coordinates.

Let’s grab the images from a few slides back to understand how these coordinates work.

The actual texture

How it will be implemented



Adding textures

The first step is to look at how the fragment shader will interpret the data.

This is how a single texture file is seen in coordinates:

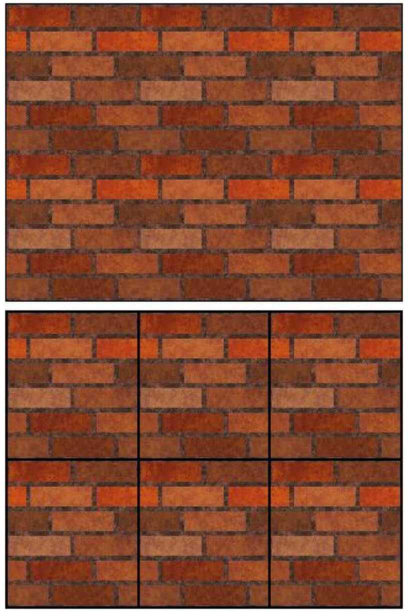
0.0 : 0.0 1.0 : 0.0

0.0 : 1.0 1.0 : 1.0

Adding textures

In the example I have implemented my wall a few times over only 4 points.

Can you guess what the “texture coordinate” values should be?



0.0 : 0.0 3.0 : 0.0

0.0 : 2.0 3.0 : 2.0

Adding textures

Now let's implement those points into our vertice data.

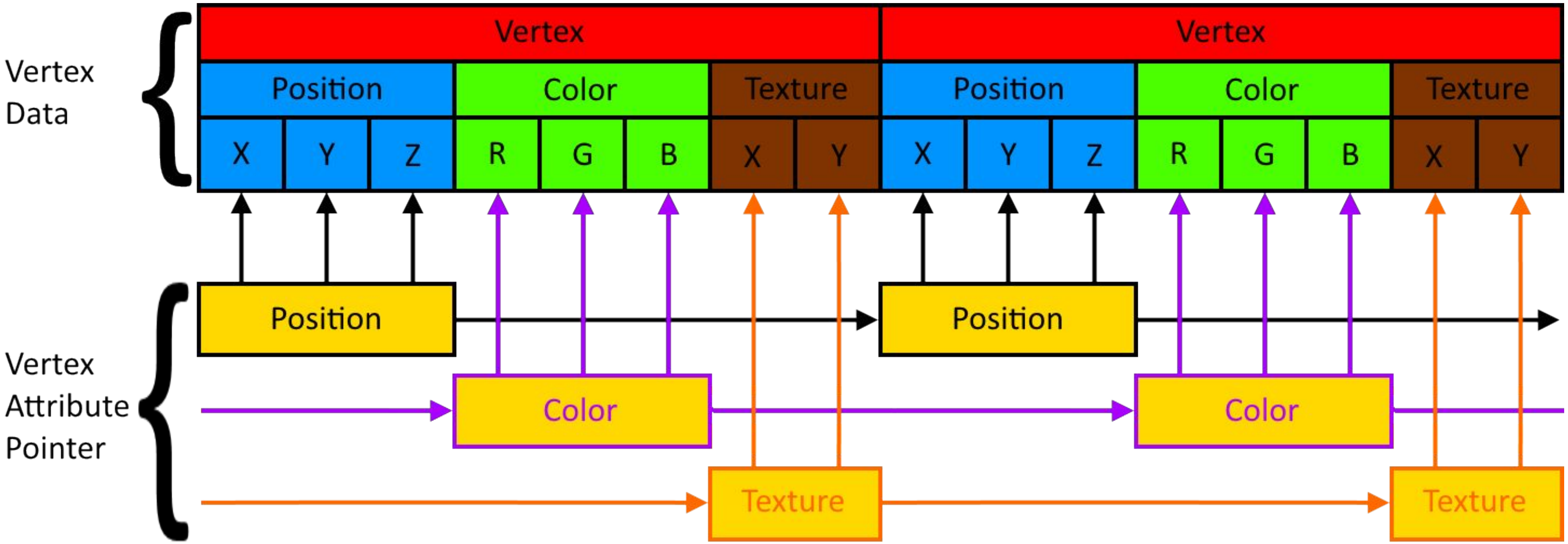
```
// create an array holding all our vertex data.
var vertices = [
    //---positions---\   /---colors---\  /--texture--\
    // x       y       z       r     g     b
     -0.5, -0.5,  0.0,   1.0, 0.0, 0.0,    0.0, 2.0,      // lower left corner
      0.5, -0.5,  0.0,   0.0, 1.0, 0.0,    3.0, 2.0,      // lower right corner
     -0.5,  0.5,  0.0,   0.0, 0.0, 0.0,    0.0, 0.0,      // upper left corner
      0.5,  0.5,  0.0,   0.0, 0.0, 1.0,    3.0, 0.0       // upper right corner
];
```

Notice how easy they are to implement in our vertex data.

Adding textures

The next step is to create a vertex attribute pointer for textures.

Textures will be added to the mix just like colors did.



Adding textures

With the last slide in mind, how will we implement the texture Vertex Attribute Pointer?

This was the state before textures:

```
// create a buffer, bind it to webgl as our active buffer and put our vertices in the buffer
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// position attribute pointer
var position_location = gl.getAttribLocation(shaderProgram, "v_position");
gl.vertexAttribPointer(position_location, 3, gl.FLOAT, false, 6*4, 0);

// color attribute pointer
var color_location = gl.getAttribLocation(shaderProgram, "v_color");
gl.vertexAttribPointer(color_location, 3, gl.FLOAT, false, 6*4, 3*4);
```

Note: We called the shader input “v_texture”

Adding textures

The state of Vertex Attribute Pointers with textures added will be:

```
// create a buffer, bind it to webgl as our active buffer and put our vertices in the buffer
var vertex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// position attribute pointer
var position_location = gl.getAttribLocation(shaderProgram, "v_position");
gl.vertexAttribPointer(position_location, 3, gl.FLOAT, false, 8*4, 0);

// color attribute pointer
var color_location = gl.getAttribLocation(shaderProgram, "v_color");
gl.vertexAttribPointer(color_location, 3, gl.FLOAT, false, 8*4, 3*4);

// texture attribute pointer
var texture_location = gl.getAttribLocation(shaderProgram, "v_texture");
gl.vertexAttribPointer(texture_location, 2, gl.FLOAT, false, 8*4, 6*4);
```

Notice how we have 8*4 now because we have 8 elements in our vertex array (and a float is 4 byte in javascript)

Adding textures

Don’t forget to enable the Vertex Attribute Array

```
// tell WebGL to enable the vertex attribute we just specified.
gl.enableVertexAttribArray(position_location);
gl.enableVertexAttribArray(color_location);
gl.enableVertexAttribArray(texture_location);
```

Now there are just a few things left to use our loaded texture.

Adding textures

First we activate a texture unit to store our texture in.

Then we bind our loaded in texture to the activated texture unit.

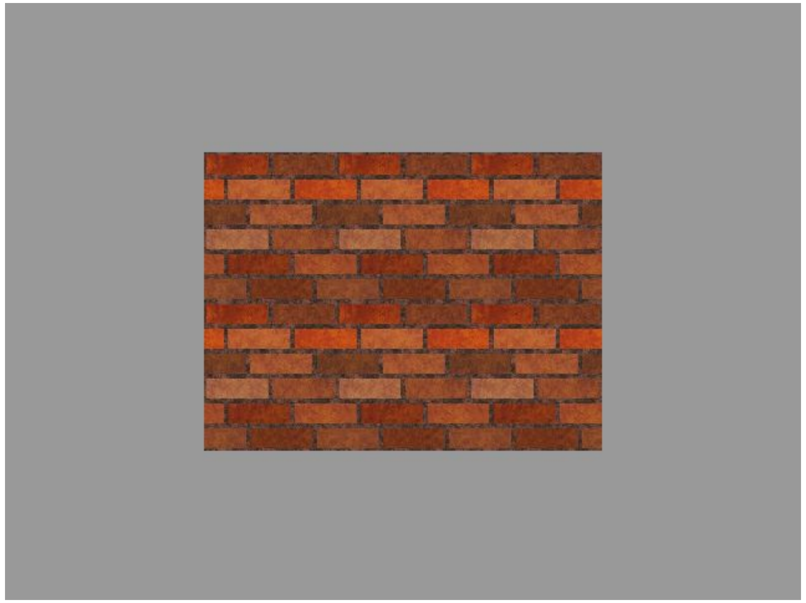
```
// tell WebGL we want to affect texture unit 0
gl.activeTexture(gl.TEXTURE0);

// bind the texture to the texture unit
gl.bindTexture(gl.TEXTURE_2D, texture);
```

The sampler2D in the fragment shader takes a bound texture unit, and defaults to gl.TEXTURE0 in this case.

```
uniform sampler2D u_texture;
```

Adding textures

Here is the result of our hard work:



Now let’s jump back into the math.

Step 2: Transformations and the math behind them
Part 2: 2D/3D vectors

Disclaimer: I am still not a mathematician

Math part 2: 2D/3D vectors

Last time we discussed adding and subtracting vectors.

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \qquad \mathbf{w} = \vec{w} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Addition: $\vec{v} + \vec{w} = \begin{pmatrix} 2 + 3 \\ 5 + 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \end{pmatrix}$

Subtraction: $\vec{v} - \vec{w} = \begin{pmatrix} 2 - 3 \\ 5 - 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$

Now let's do these calculations with 3D vectors.

Math part 2: 2D/3D vectors

Switching over to 3D vectors.

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \\ 4 \end{pmatrix} \qquad \mathbf{w} = \vec{w} = \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix}$$

Addition: $\vec{v} + \vec{w} = \begin{pmatrix} 2+3 \\ 5+1 \\ 4+2 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 6 \end{pmatrix}$

Subtraction: $\vec{v} - \vec{w} = \begin{pmatrix} 2-3 \\ 5-1 \\ 4-2 \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \\ 2 \end{pmatrix}$

We will get to the "multiplying" of vectors soon, but before that we have 2 other subjects.

Math part 2: 2D/3D vectors

The **length** (or **magnitude**) of a vector is how far the vector travels in any direction.

The length of a vector is denoted as a character between 2 (or 4) vertical lines: $|v|$

The length of a vector is always an absolute value, meaning it can not be negative.

The length of a vector is calculated using this formula:

$$|v| = \sqrt{(v_x^2) + (v_y^2)}$$

Math part 2: 2D/3D vectors

The length (or magnitude) of a vector is calculated using this formula:

$$|v| = \sqrt{(v_x)^2 + (v_y)^2}$$

This formula is called Pythagoras' theorem:

$$c^2 = a^2 + b^2$$

For our vector **V** the length should be:

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

$$|v| = \sqrt{(3)^2 + (4)^2} = 5$$

Math part 2: 2D/3D vectors

The length (or magnitude) of a 3D vector is calculated using this formula:

$$|v| = \sqrt{(v_x)^2 + (v_y)^2 + (v_y)^2}$$

Pythagoras' theorem still holds in 3D:

$$d^2 = a^2 + b^2 + c^2$$

For our vector **V** the length should be:

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

$$|v| = \sqrt{(3)^2 + (4)^2 + (5)^2} \approx 7.07$$

Math part 2: 2D/3D vectors

A **unit vector** is a vector where the length of the vector equals 1.

A unit vector is denoted as a character with a roof over it's head: $\hat{v}$

Unit vectors are really useful for calculations that use the length of a vector, as it will be 1.

A unit vector is calculated using this formula:

$$\hat{v} = \vec{v} \,/\, |v|$$

Calculating the unit vector is called **normalizing** a vector.

Math part 2: 2D/3D vectors

A unit vector is calculated by dividing each component of the original vector by the vector's length. So:

Vector $\mathbf{v} = \vec{v} = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$

Length / Magnitude $|v| = \sqrt{(3)^2 + (4)^2} = 5$

Unit vector $\hat{v} = \vec{v} \,/\, |v| = \begin{pmatrix} 3 / 5 \\ 4 / 5 \end{pmatrix} = \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix}$

And the proof that the length (or magnitude) of this vector equals 1:

$$|v| = \sqrt{(0.6)^2 + (0.8)^2} = \sqrt{(0.36 + 0.64)} = \sqrt{1} = 1$$

Math part 2: 2D/3D vectors

A unit vector will behave the same in 3D, just adding a 3rd number to the equation.

Vector

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix}$$

Length / Magnitude

$$|v| = \sqrt{(3)^2 + (4)^2 + (5)^2} \approx 7.07$$

Unit vector

$$\hat{v} = \vec{v} \,/\, |v| = \begin{pmatrix} 3 / 7.07 \\ 4 / 7.07 \\ 5 / 7.07 \end{pmatrix} = \begin{pmatrix} 0.424 \\ 0.566 \\ 0.707 \end{pmatrix}$$

And the proof that the length (or magnitude) of this vector equals 1:

$$|v| = \sqrt{(0.424)^2 + (0.566)^2 + (0.707)^2} = \sqrt{(0.18 + 0.32 + 0.50)} = \sqrt{1} = 1$$

Math part 2: 2D/3D vectors

"Multiplying" a vector can be done using 2 methods.

Dot product

$$\vec{a} \cdot \vec{b} = |a| \cdot |b| \cdot \cos\theta$$

Cross product

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y \cdot b_z - a_z \cdot b_y \\ a_z \cdot b_x - a_x \cdot b_z \\ a_x \cdot b_y - a_y \cdot b_x \end{pmatrix} = \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix}$$

We will shortly go through both, and then finally get to the transformation/scaling and rotation matrices.

Math part 2: 2D/3D vectors

TO DO:

1. Dot Product
2. Cross product
3. 3D vectors
4. Revisit all for 3D
5. Matrixes
6. Identity matrix
7. Translation matrix
8. Scale matrix
9. Rotation matrix