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

```
<!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

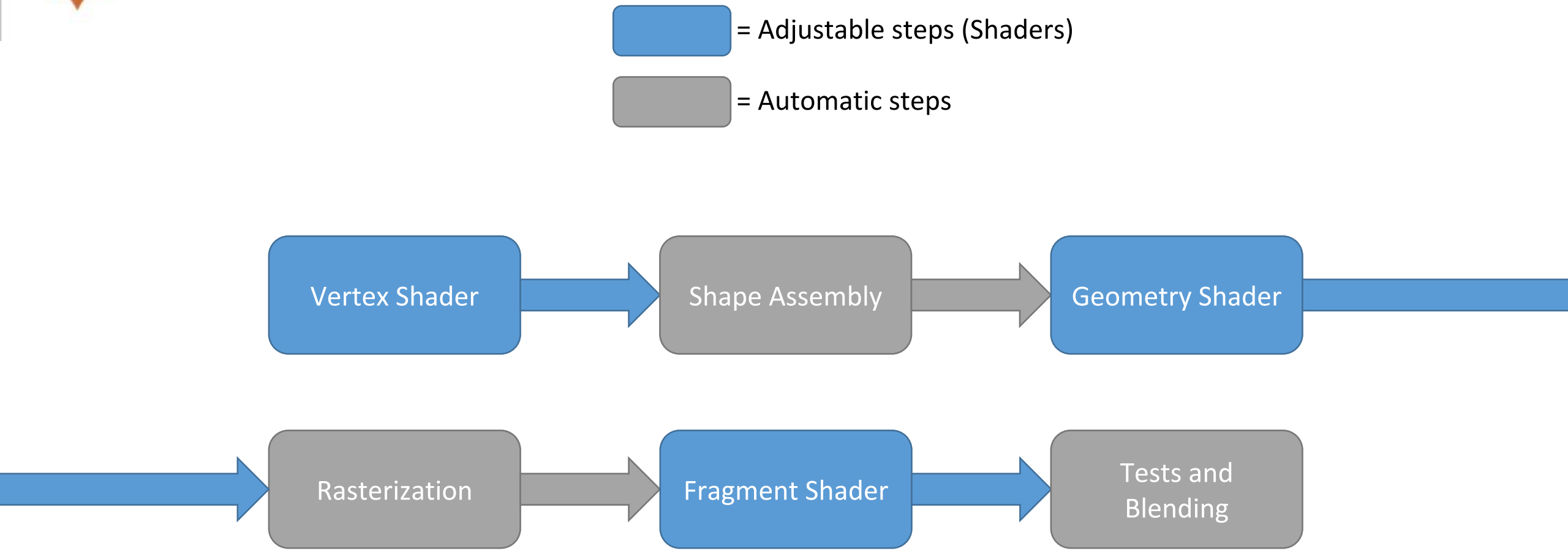
```
main();

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
2. Element array buffer
3. Vertex shader
4. Fragment shader
5. Shader program

Steps for drawing a triangle

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

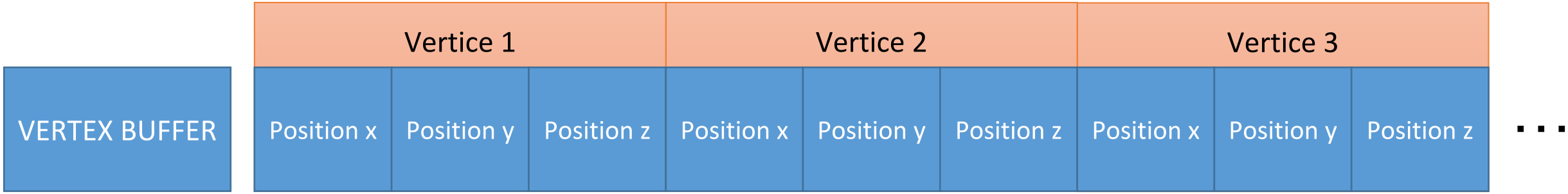
1. Array Buffer Objects
1. Shaders and Shader Program
1. Linking Buffers and Shaders
1. Drawing

Step 1: Array Buffer Objects

Array Buffer Objects

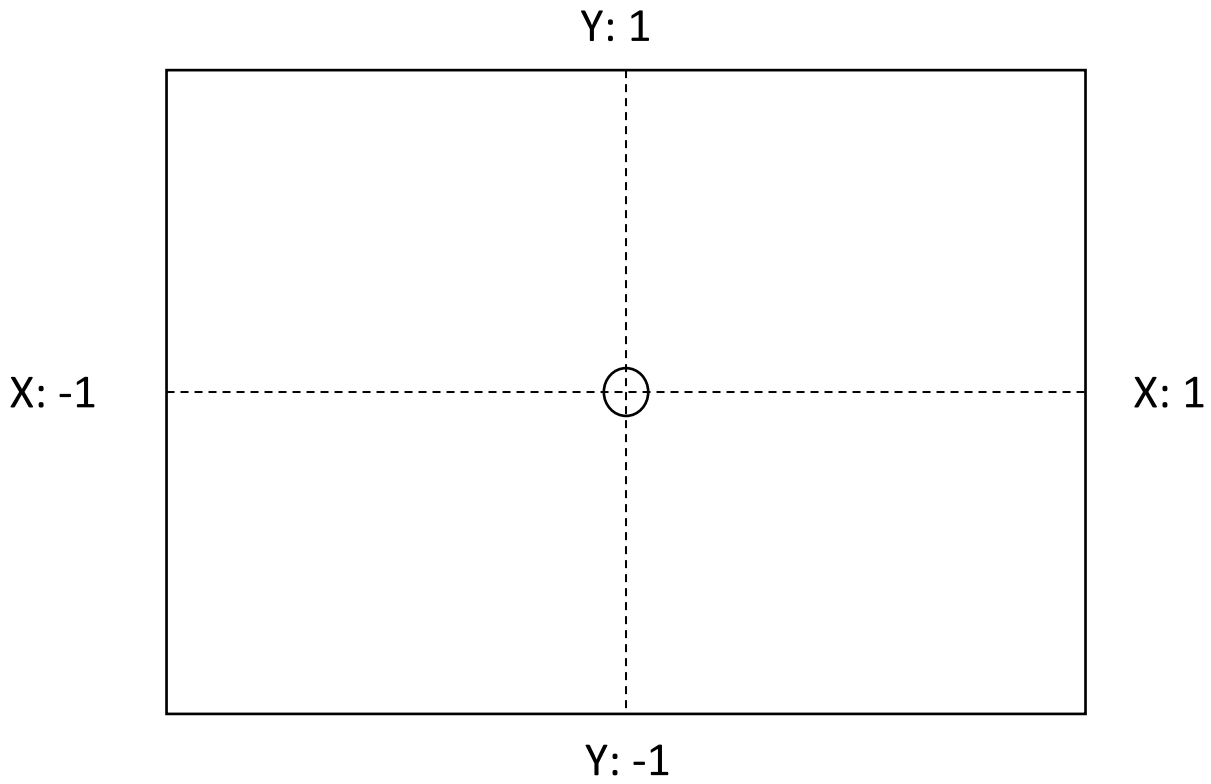
A Vertex Buffer contains information about each vertice, which shaders will use for calculations and translations.

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



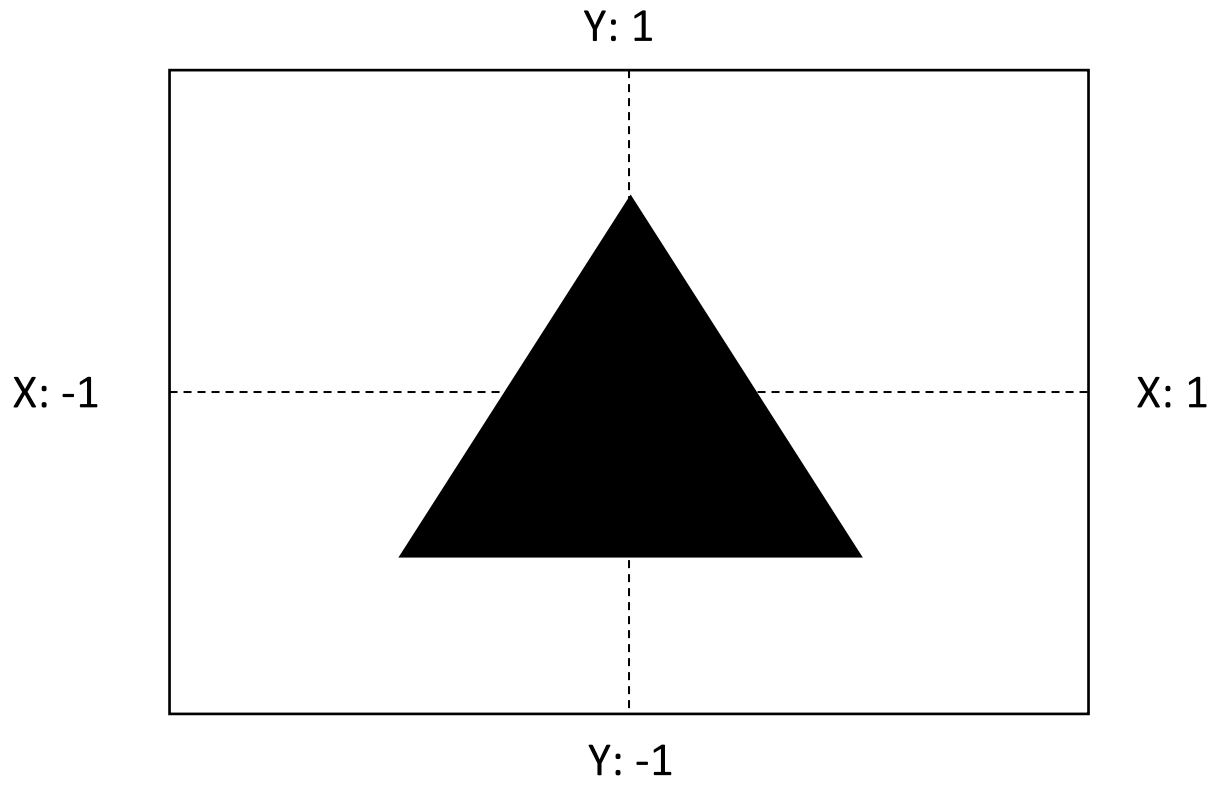
Array Buffer Objects

How does WebGL interpret the view?



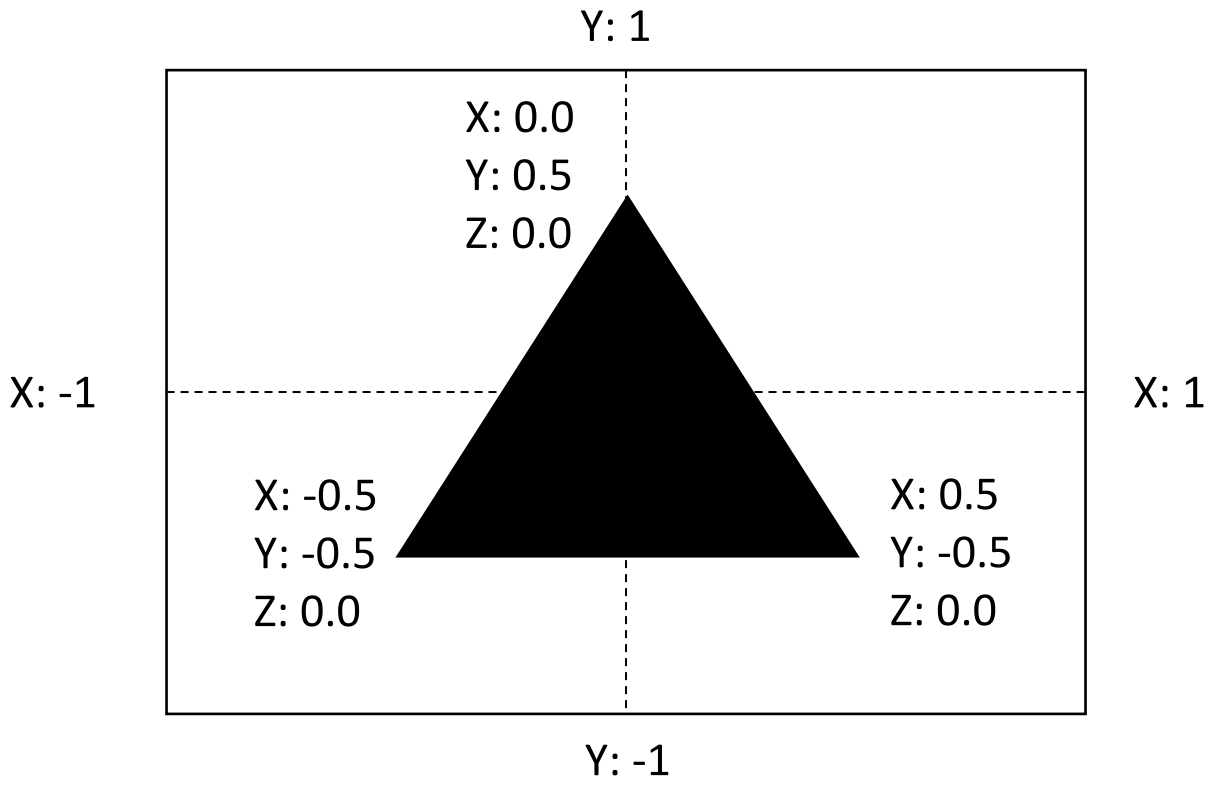
Array Buffer Objects

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

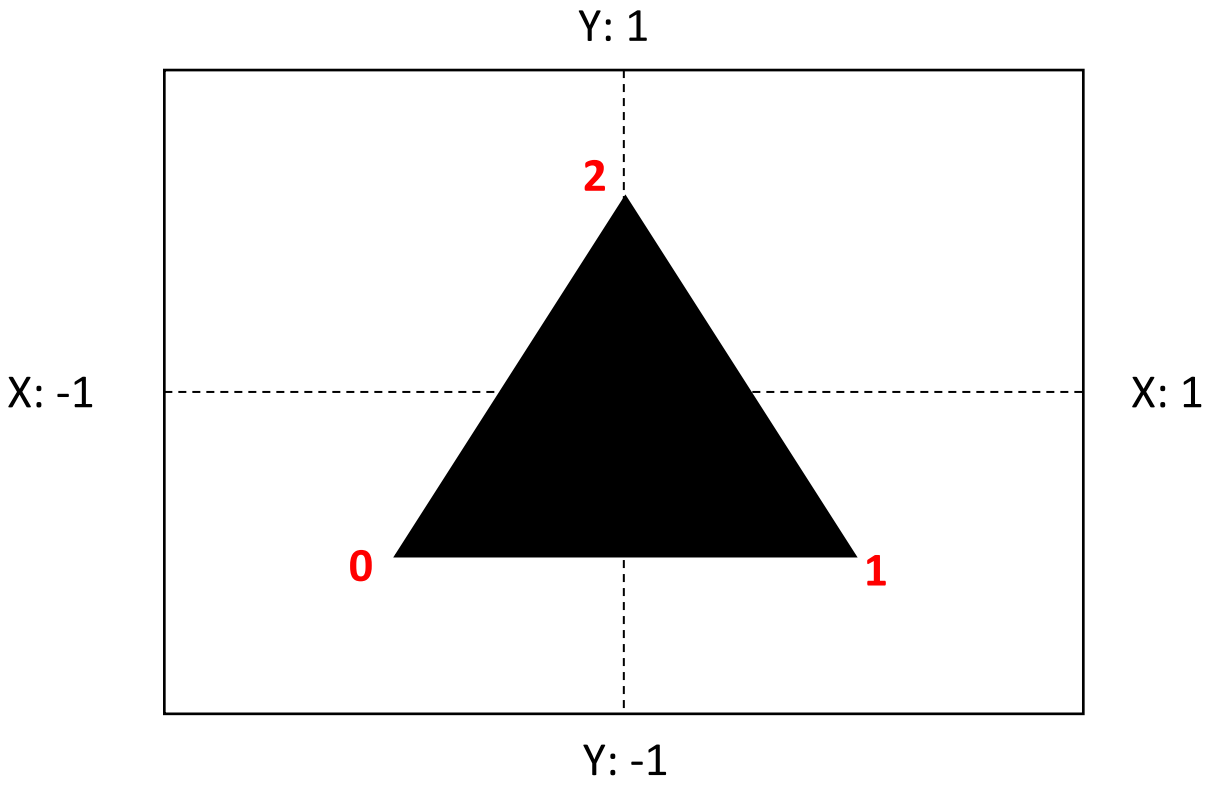
Now let's prepare our buffer so WebGL can eventually send it to the shaders.

```
// 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.

```
0 ->  var positions = [
          // x      y      z
          -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:

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);
```

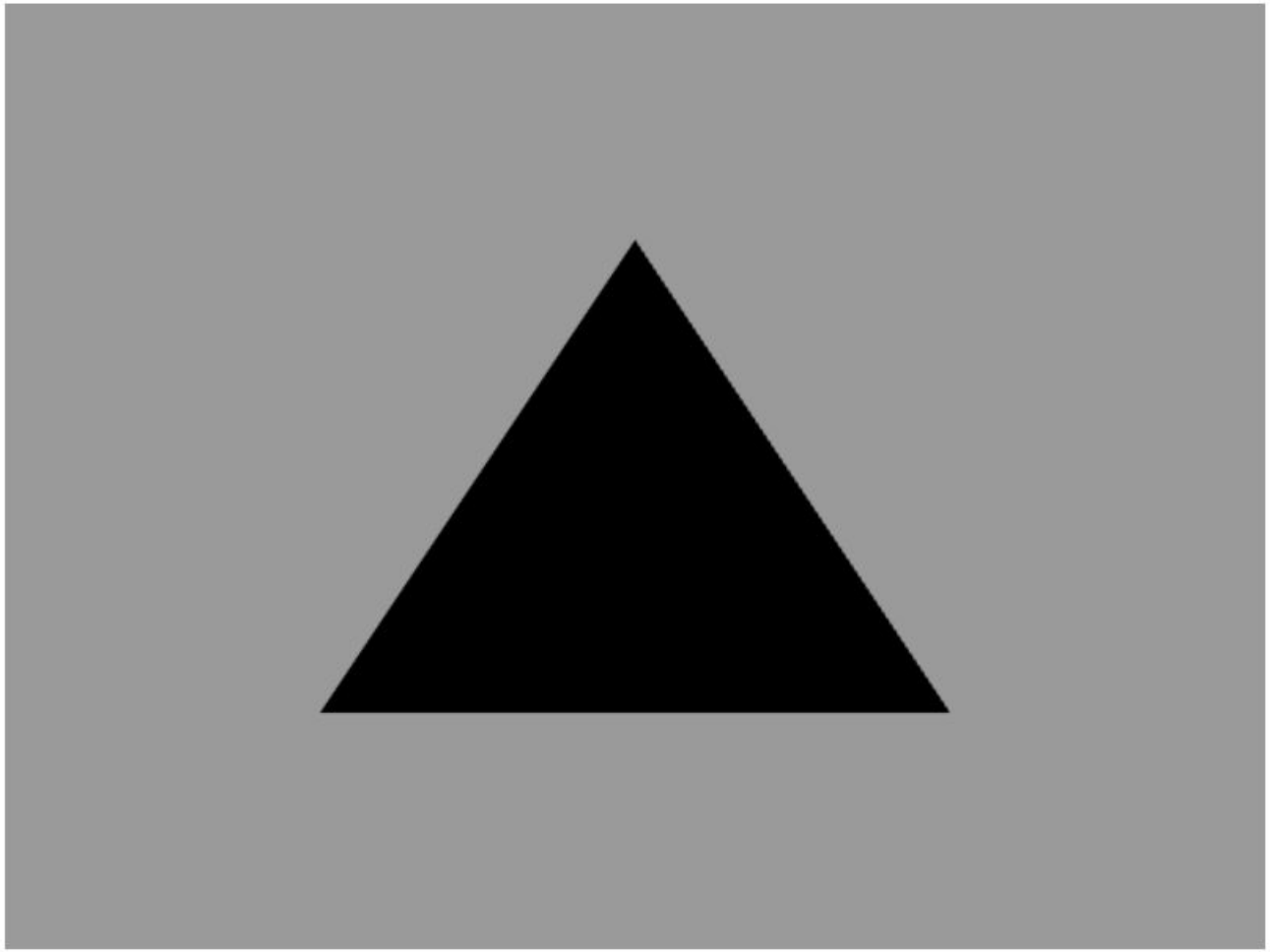
Step 4: Drawing

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.

Part II

Presentation by: Quincy Jacobs

Session preview

This session will focus on these points:

1. Colors array and (color) Vertex Buffer Object

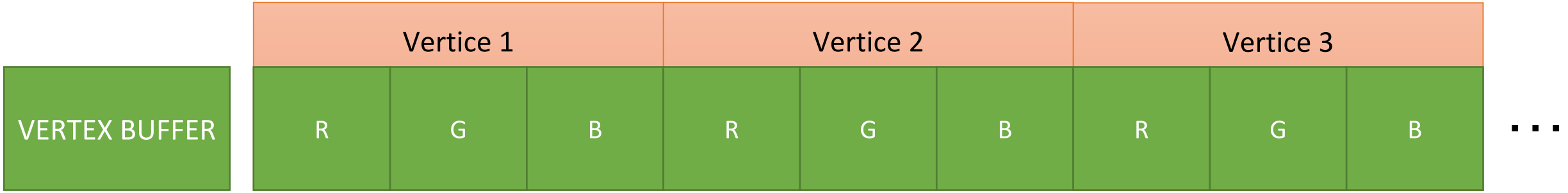
1. Combining the Color and Position Vertex Buffers

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.



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);
    }
`;
```

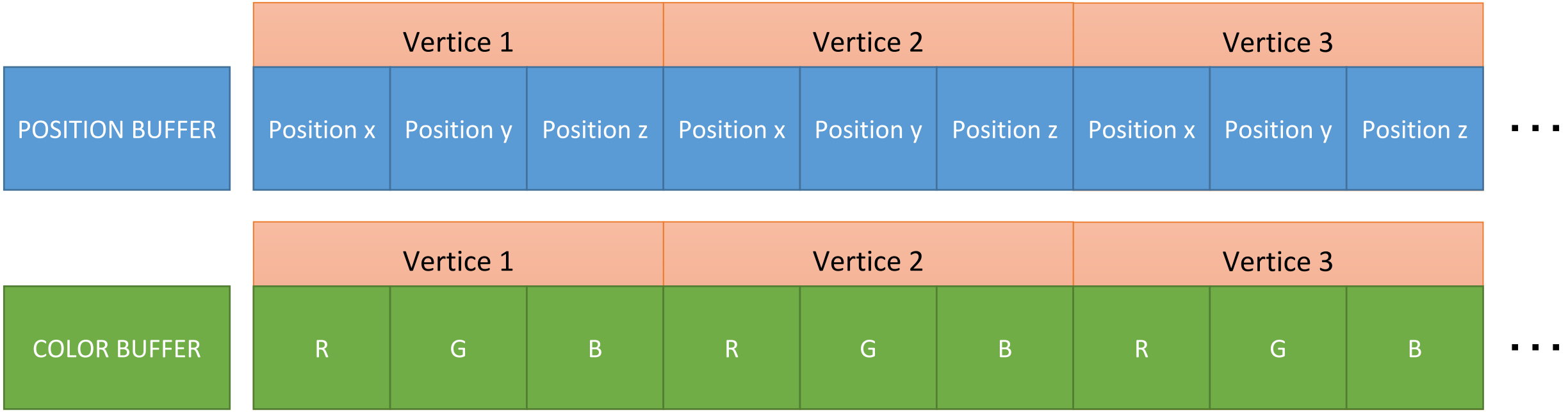
Next we will tie everything together in the code

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



How should we combine these 2 arrays?

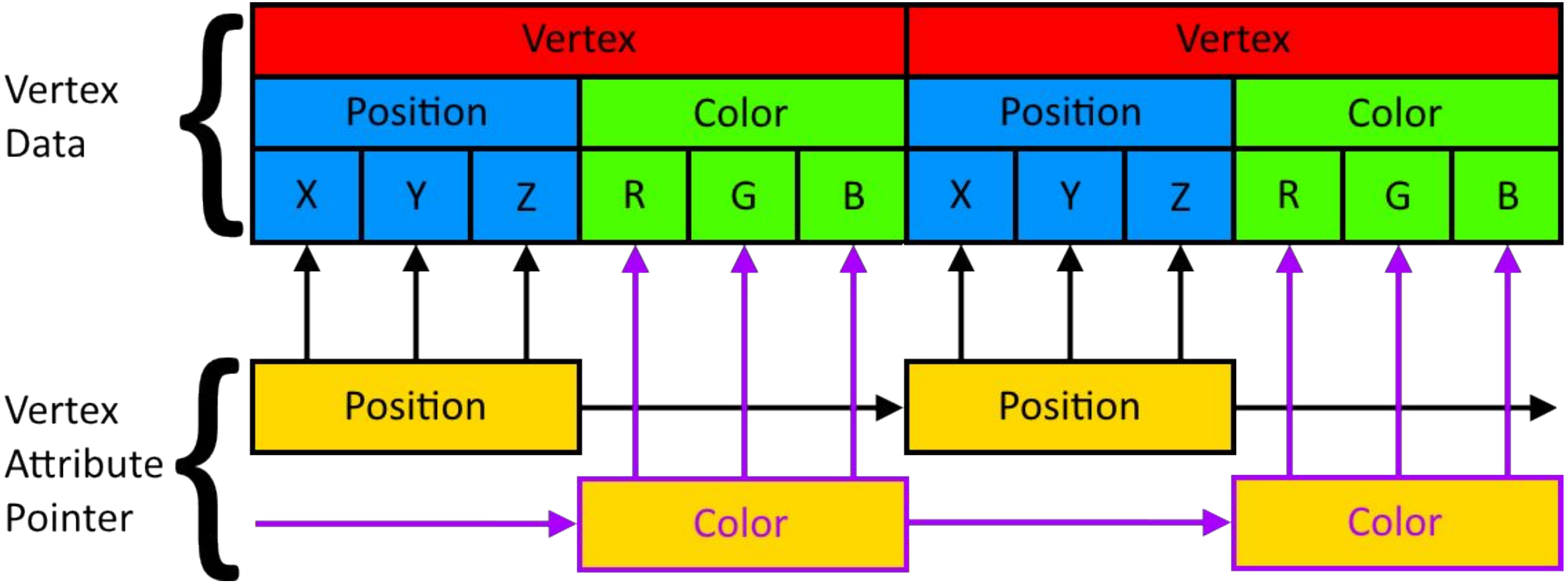
Combining the Color and Position Vertex Buffers

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);
```

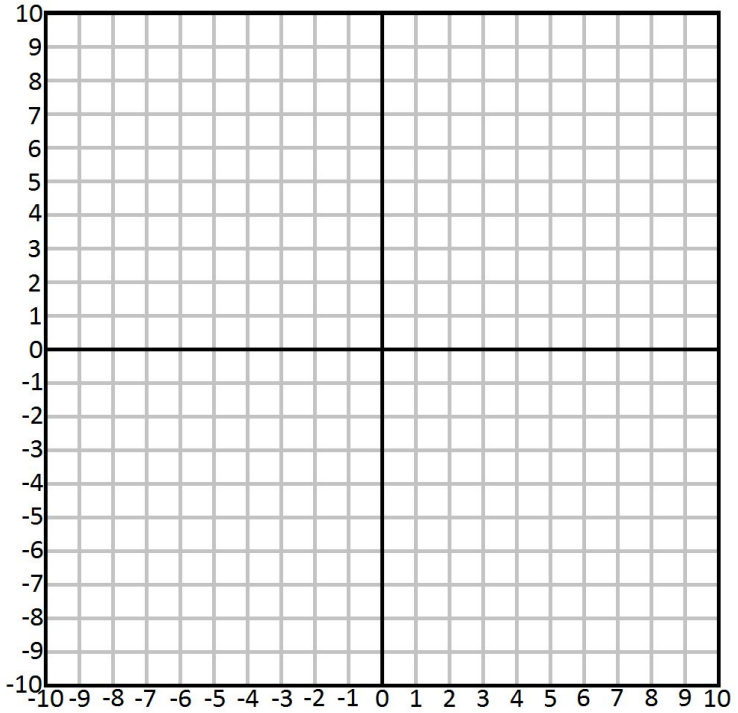
The location is still necessary as that is the point in the shader we will feed data to.

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

Disclaimer: I am not a mathematician

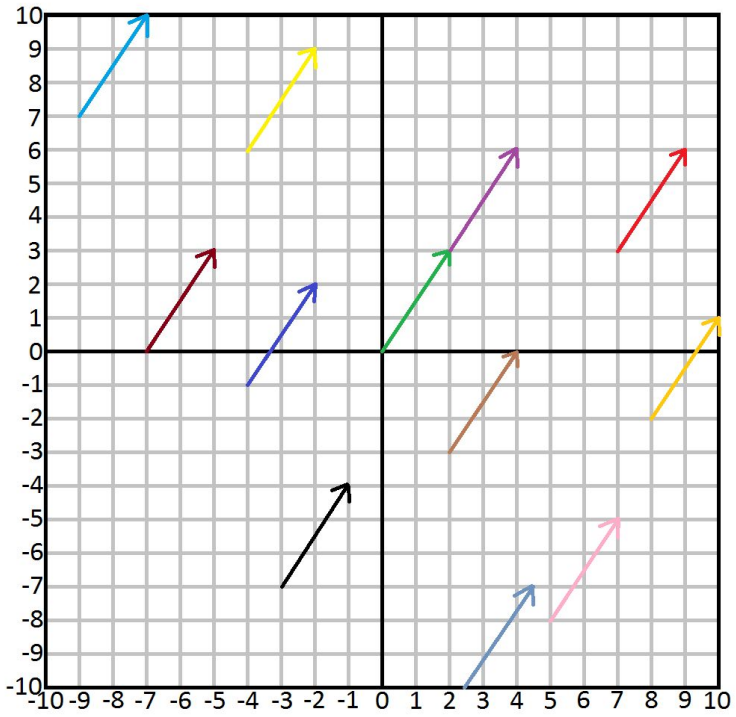
Math part 1: 2D vectors

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

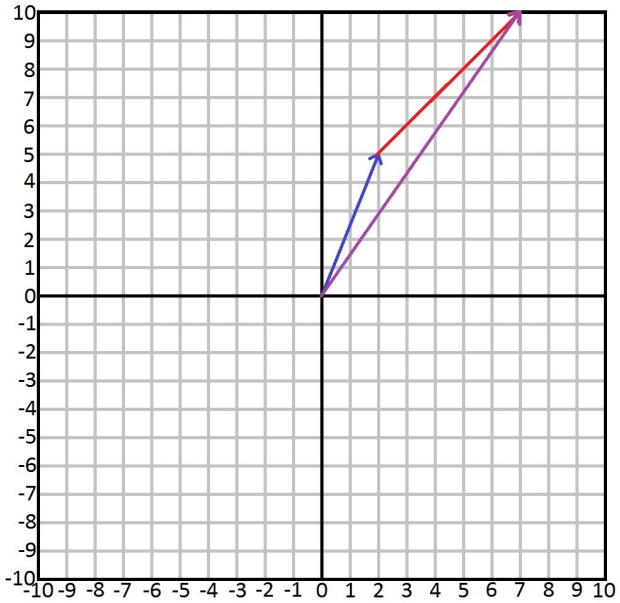
Division: $\vec{v} / s = \begin{pmatrix} 2/5 \\ 5/5 \end{pmatrix} = \begin{pmatrix} 0.4 \\ 1 \end{pmatrix}$

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?

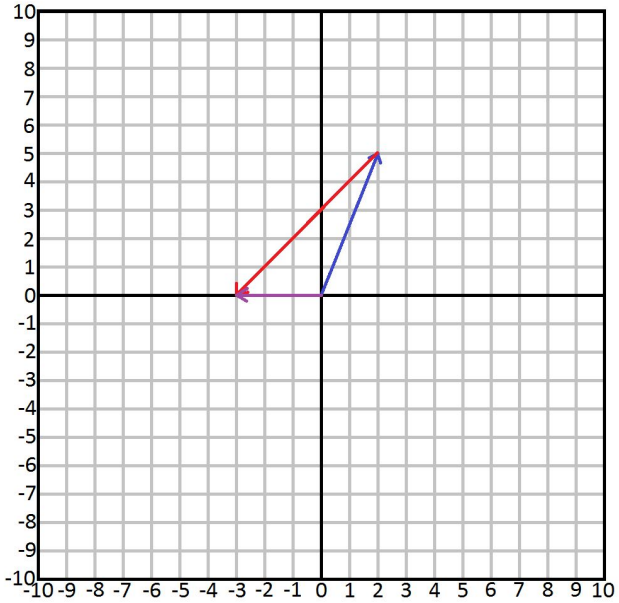
Math part 1: 2D vectors

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.



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

Scalar: $s = 5$



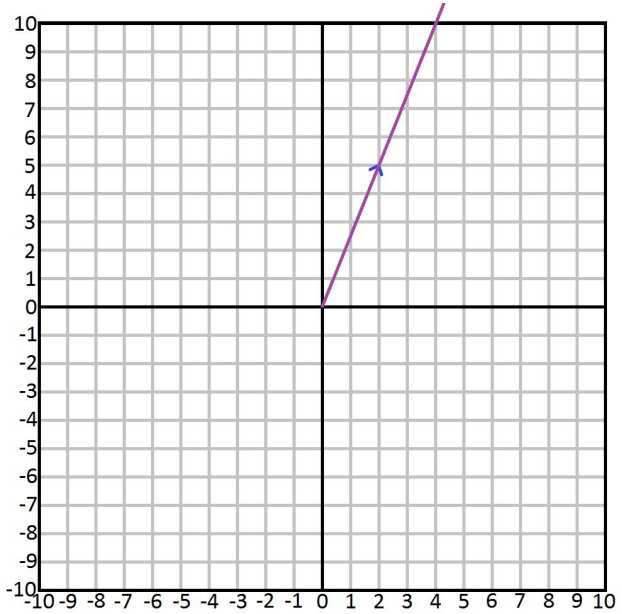
Addition: $\vec{v} + s = \begin{pmatrix} 2+5 \\ 5+5 \end{pmatrix} = \begin{pmatrix} 7 \\ 10 \end{pmatrix}$

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

Multiplying and Dividing by a scalar is much more useful.

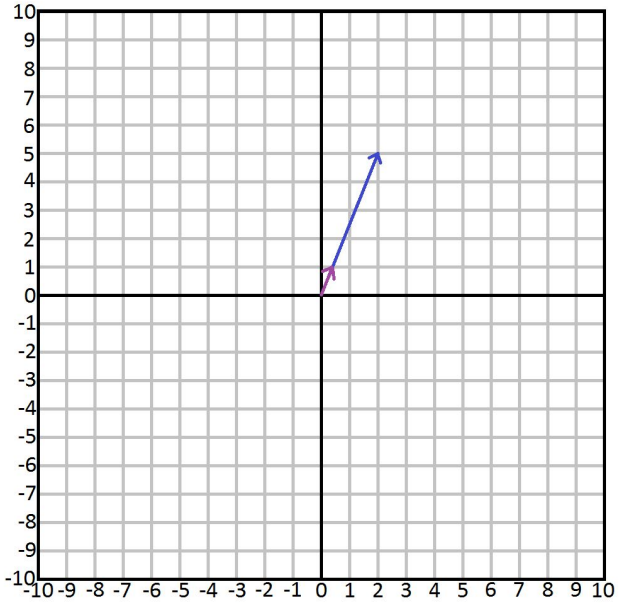
Math part 1: 2D vectors

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

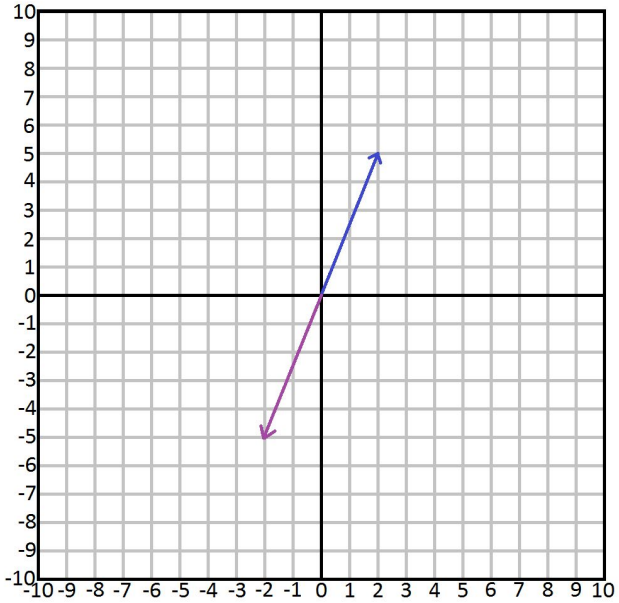
Another useful operation is flipping the vector. Any idea how you can do this?

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

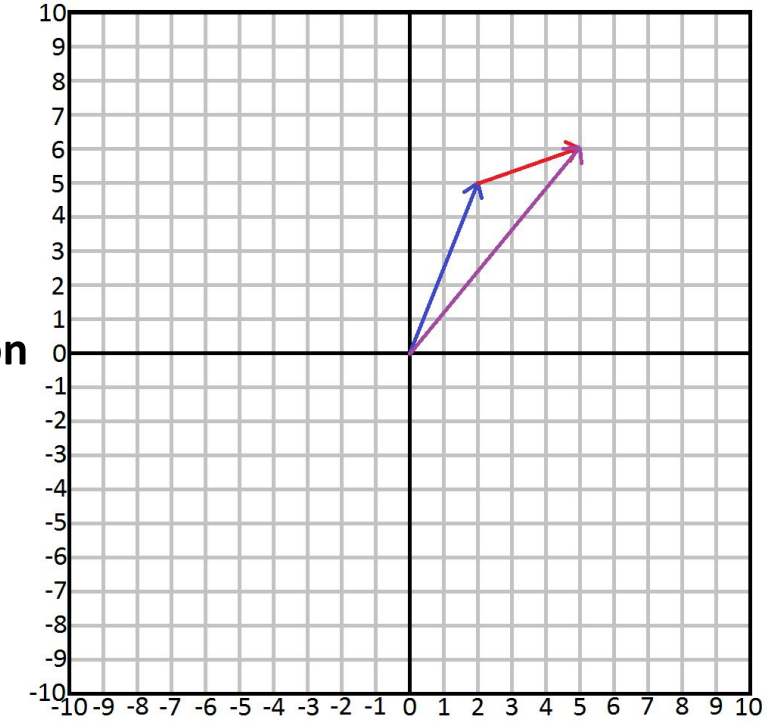
Subtraction: $\vec{v} - \vec{w} = \begin{pmatrix} 2-3 \\ 5-1 \end{pmatrix} = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$

Seeing this visually will make it much clearer.

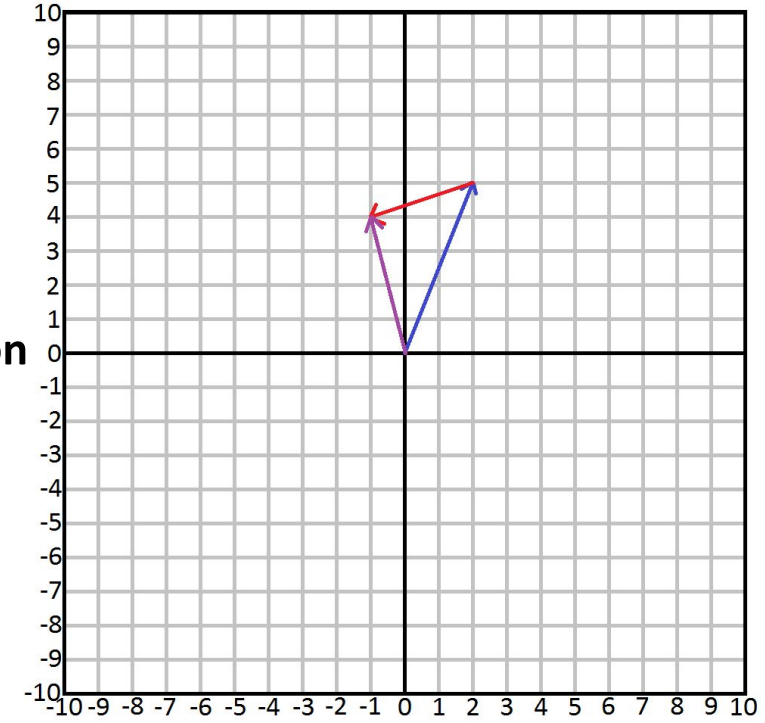
Math part 1: 2D vectors

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

Addition



Subtraction

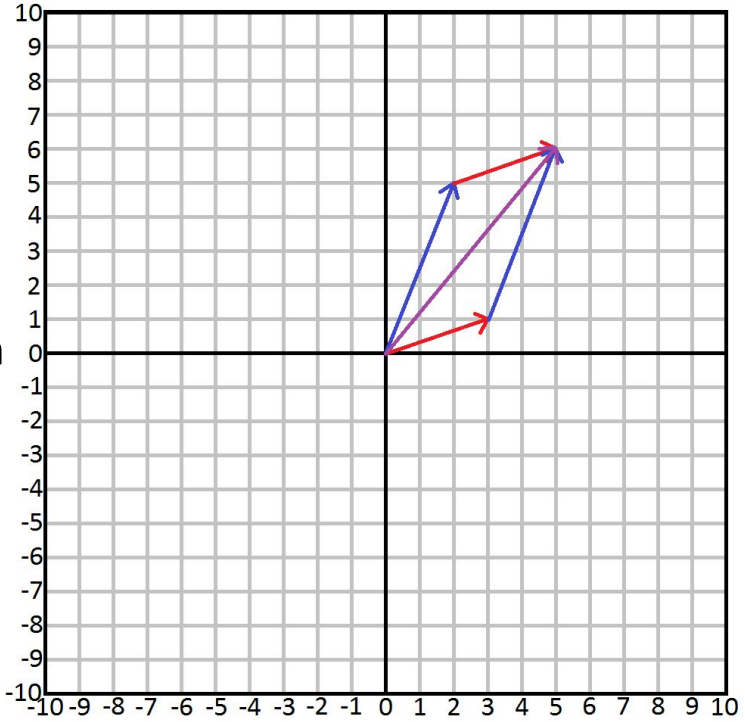


Will these vectors have the same outcome regardless of order?

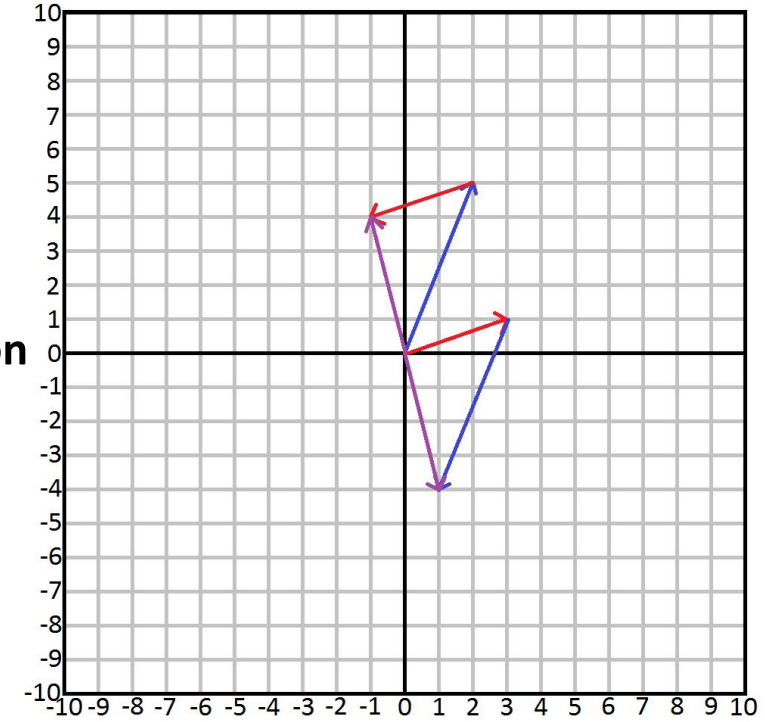
Math part 1: 2D vectors

Addition does, subtraction does not.

Addition



Subtraction



The math will continue at the end of part III

To be continued next time…

Thanks for listening.

Part III

Textures, Vectors, and 4 matrices to move your triangles.

Presentation by: Quincy Jacobs

Session preview

This session will focus on these points:

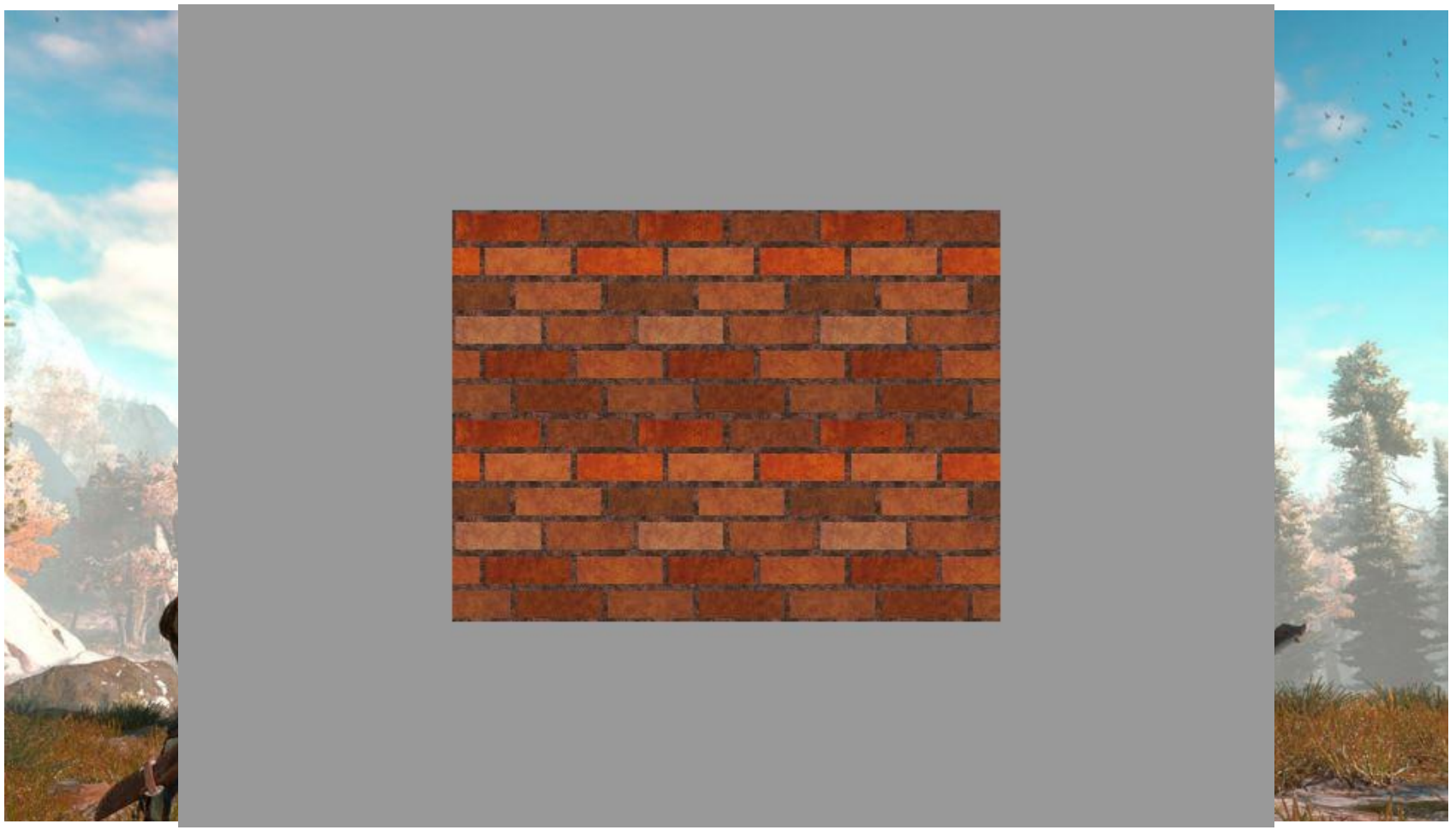
1. **Adding a texture**
2. **Math**
    - 2D/3D vectors
    - Matrices
3. **Applying math in WebGL**

Step 1: Adding a texture

Step 2: Math

Step 3: Applying math in WebGL

Adding a texture



Adding a texture

To-do list

- ☐ Texture object
- ☐ Loading image
- ☐ Texture parameters
- ☐ Texture in shader
- ☐ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

Creating a texture object.

1. Let WebGL create the object
2. Bind the object to WebGL
3. Create a placeholder image

```
// 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]));
```

Adding a texture

To-do list

- ☒ Texture object
- ☐ Loading image
- ☐ Texture parameters
- ☐ Texture in shader
- ☐ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

Loading an image.

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

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 = "";
  }
}
```

Adding a texture

To-do list

- ☒ Texture object
- ☒ Loading image
- ☐ Texture parameters
- ☐ Texture in shader
- ☐ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

Set texture parameters after image load

```
// 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);

});
```

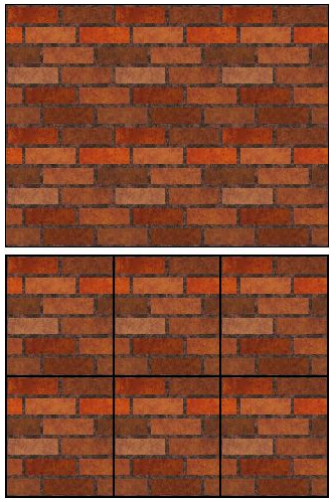
Adding a texture

TEXTURE_WRAP_S

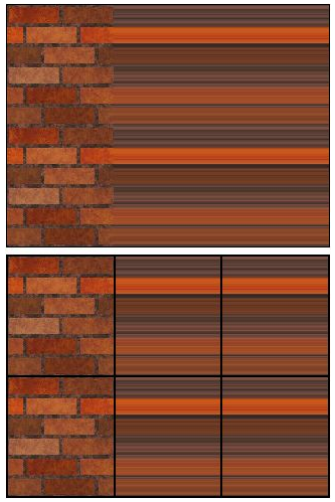
```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
```

3 options for TEXTURE_WRAP_S:

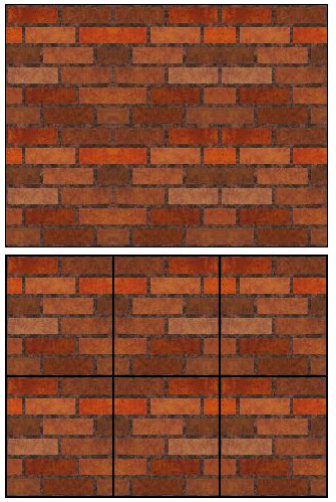
gl.REPEAT



gl.CLAMP_TO_EDGE



gl.MIRRORED_REPEAT



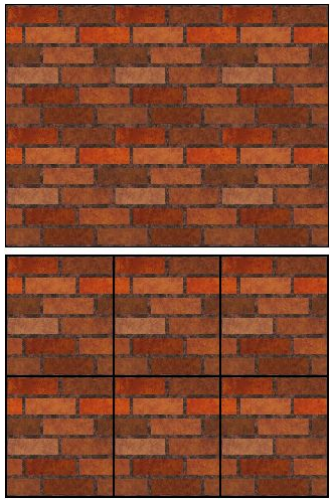
Adding a texture

TEXTURE_WRAP_T

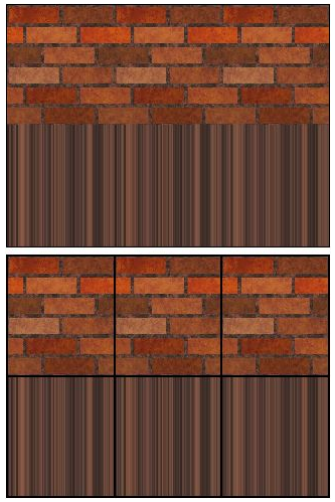
```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

Same as TEXTURE_WRAP_S, but on the y-axis:

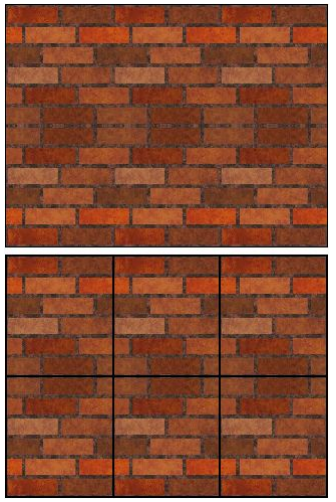
gl.REPEAT



gl.CLAMP_TO_EDGE



gl.MIRRORED_REPEAT



Adding a texture

TEXTURE_MIN_FILTER and **TEXTURE_MAG_FILTER**

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

Control the stretching of the image with 6 options:

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 a texture

To-do list

- ☒ Texture object
- ☒ Loading image
- ☒ Texture parameters
- ☐ Texture in shader
- ☐ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

Vertex shader

Abuse the vertex shader as a conduit

```
// 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);
    }
`;
```

Adding a texture

Fragment shader

```
// 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);
    }
`;
```

Adding a texture

To-do list

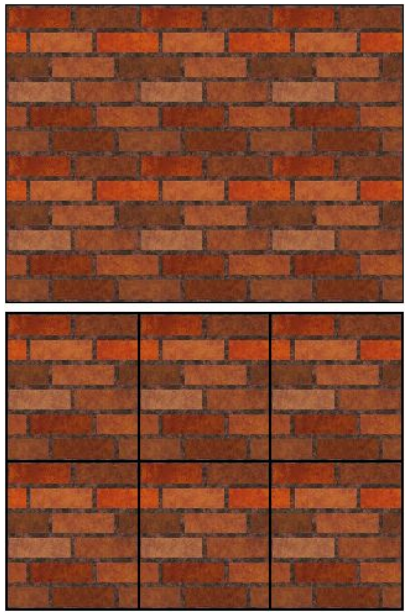
- ☒ Texture object
- ☒ Loading image
- ☒ Texture parameters
- ☒ Texture in shader
- ☐ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

Inputting texture coordinates

The actual texture

Desired result



Adding a texture

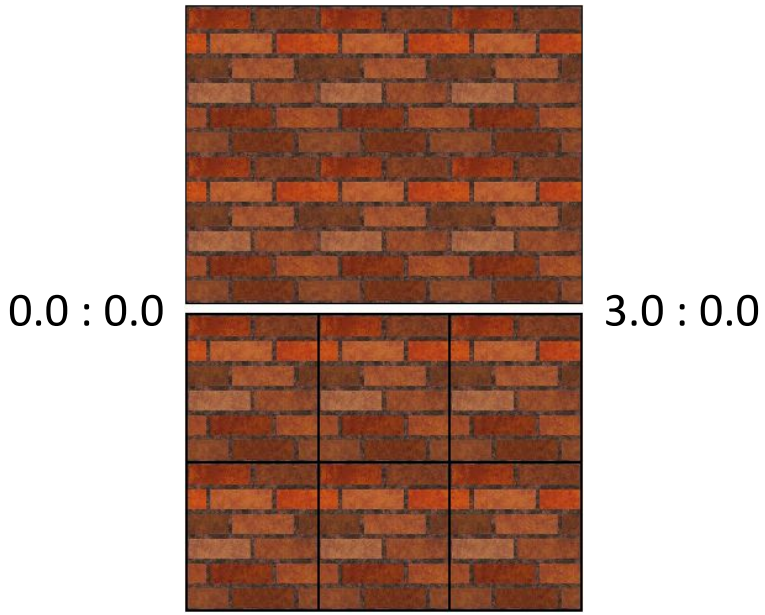
The coordinates for the image itself

0.0 : 0.0 1.0 : 0.0

0.0 : 1.0 1.0 : 1.0

Adding a texture

The coordinates for the desired result:



Adding a texture

Implementing the coordinates

```
// 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
];
```

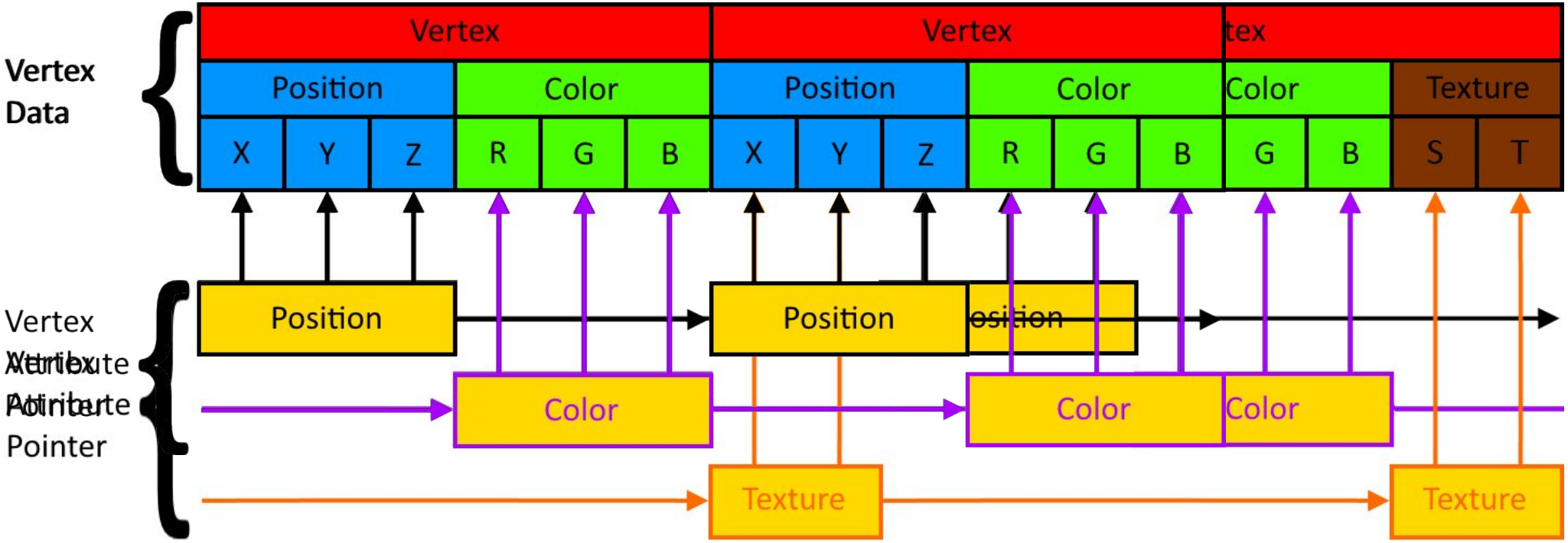
Adding a texture

To-do list

- ☒ Texture object
- ☒ Loading image
- ☒ Texture parameters
- ☒ Texture in shader
- ☒ Texture “vertices”
- ☐ Texture attribute pointer

Adding a texture

The **texture attribute pointer** adds in just like the **color attribute pointer**.



Adding a texture

Add the texture **vertex attribute pointer**

```
// 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);

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

Adding a texture

Enable the **vertex attribute arrays**

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

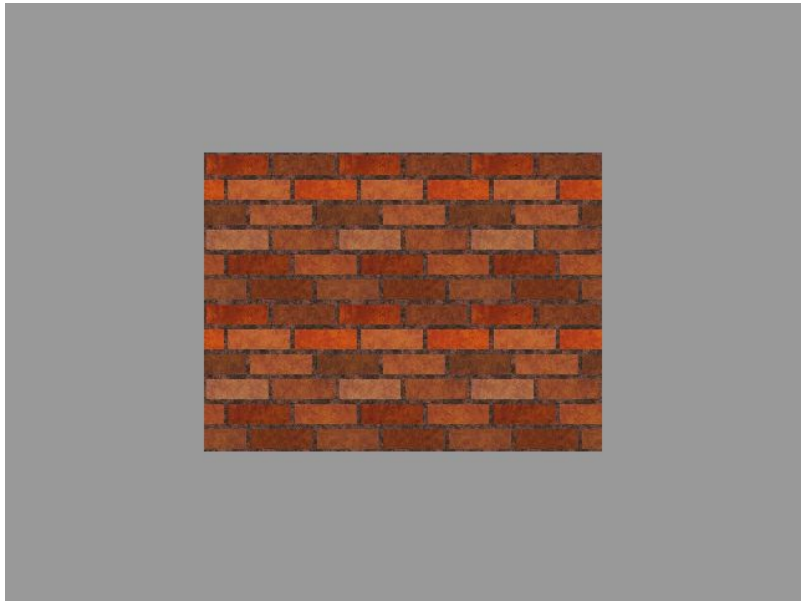
Adding a texture

To-do list

- ☒ Texture object
- ☒ Loading image
- ☒ Texture parameters
- ☒ Texture in shader
- ☒ Texture “vertices”
- ☒ Texture attribute pointer

Adding a texture

The final result:



Step 1: Adding a texture

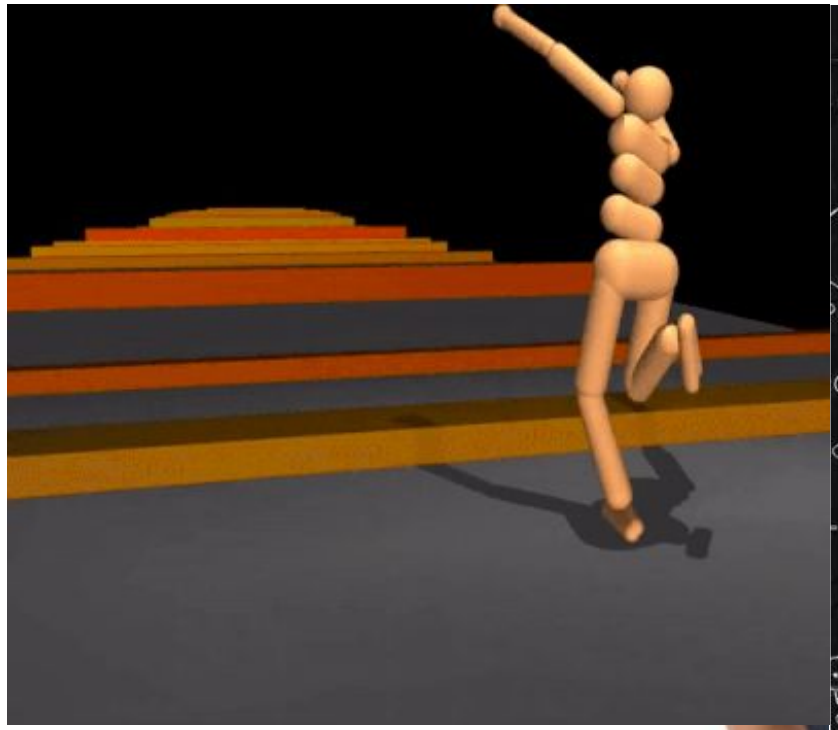
Step 2: Math

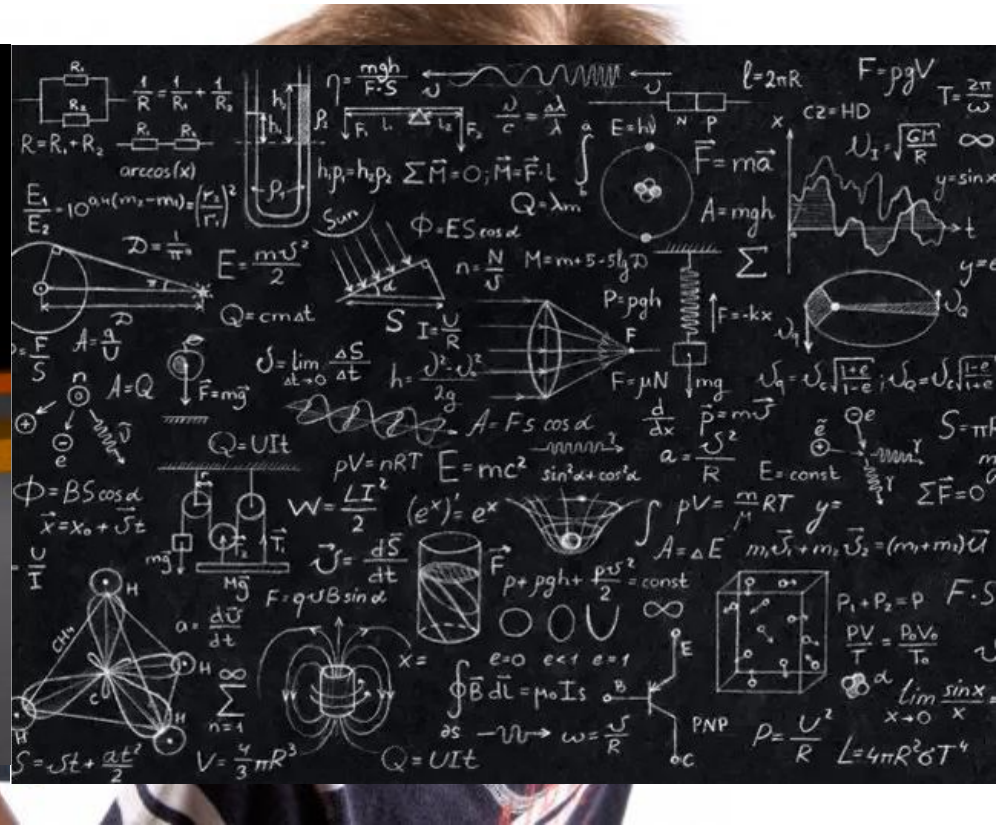
Step 3: Applying math in WebGL

Disclaimer: I am not a mathematician

Math

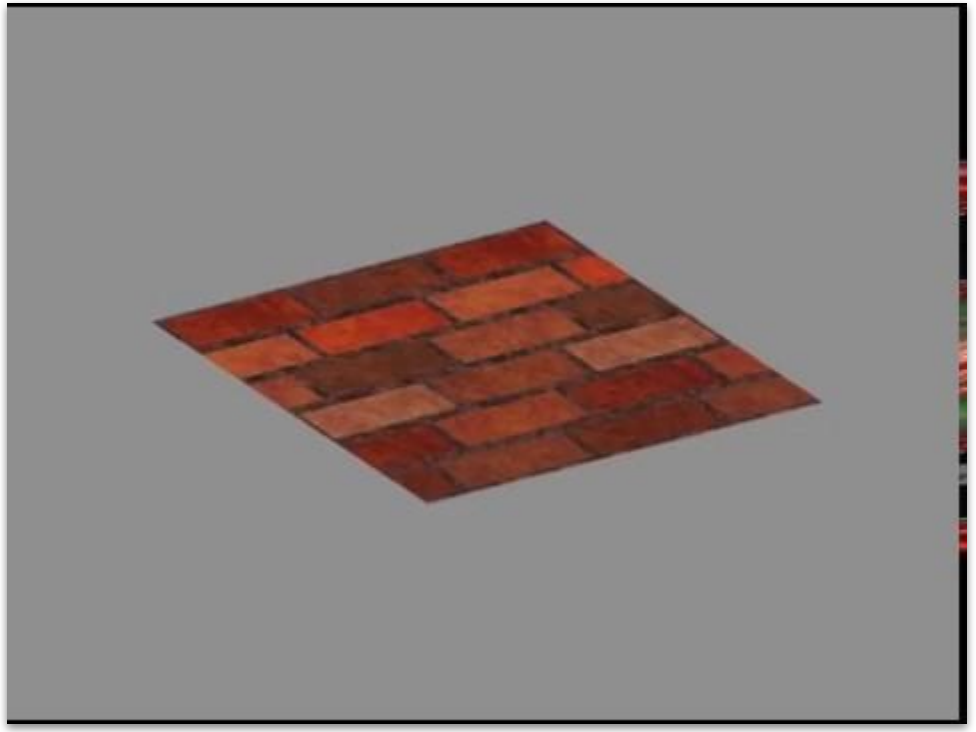
What we discussed last term.





Math

What you will actually learn.



Step 2: Math

2D/3D vectors

Matrices

Disclaimer: I am still not a mathematician

Math: 2D/3D vectors

To-do list

- ☐ 3D vector
- ☐ Vector length
- ☐ Unit vector
- ☐ Dot product
- ☐ Cross product

Math: 2D/3D vectors

Last time: addition and subtraction with 2D vectors.

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

$$\mathbf{w} = \vec{w} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Addition:

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

Subtraction:

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

Math: 2D/3D vectors

Addition and subtraction with 3D vectors.

$$\mathbf{v} = \vec{v} = \begin{pmatrix} 2 \\ 5 \\ 4 \end{pmatrix}$$

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

Addition:

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

Subtraction:

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

Math: 2D/3D vectors

To-do list

- ☒ 3D vector
- ☐ Vector length
- ☐ Unit vector
- ☐ Dot product
- ☐ Cross product

Math: 2D/3D vectors

Length (or **magnitude**) of a vector:

- indicates how far a vector travels
- independent of direction
- always positive
- usually written as $|v|$

Math: 2D/3D vectors

The **length** (or **magnitude**) of a 2D vector:

- formula: $|v| = \sqrt{(v_x)^2 + (v_y)^2}$
- rearrangement of: $c^2 = a^2 + b^2$
- proof: $|v| = \sqrt{(3)^2 + (4)^2} = 5$

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

Math: 2D/3D vectors

The **length** (or **magnitude**) of a 3D vector:

- formula: $|v| = \sqrt{(v_x)^2 + (v_y)^2 + (v_z)^2}$
- rearrangement of: $d^2 = a^2 + b^2 + c^2$
- proof: $|v| = \sqrt{(3)^2 + (4)^2 + (5)^2} \approx 7.07$

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

Math: 2D/3D vectors

To-do list

- ☒ 3D vector
- ☒ Vector length
- ☐ Unit vector
- ☐ Dot product
- ☐ Cross product

Math: 2D/3D vectors

The **unit vector**:

- length of 1
- useful in calculations
- **normalizing**
- usually written as $\hat{v}$

Math: 2D/3D vectors

Vector

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

Length

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

The (2D) **unit vector**:

- formula: $\hat{v} = \vec{v} \,/\, |v|$
- proof: $\hat{v} = \begin{pmatrix} 3/5 \\ 4/5 \end{pmatrix} = \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix}$

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

Math: 2D/3D vectors

Vector

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

Length

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

The (3D) **unit vector**:

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

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

Math: 2D/3D vectors

To-do list

- ☒ 3D vector
- ☒ Vector length
- ☒ Unit vector
- ☐ Dot product
- ☐ Cross product

Math: 2D/3D vectors

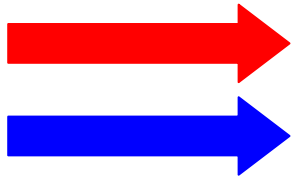
The **dot product**:

- “multiplying” vectors
- results in a scalar
- angle
- usually written as $a \cdot b$

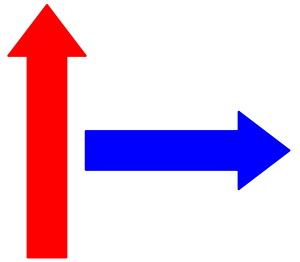
Math: 2D/3D vectors

The **dot product**:

- positive:



- 0:



- negative

Math: 2D/3D vectors

The **dot product**:

- Formula 1:

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

- Formula 2:

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y (+ a_z \cdot b_z)$$

Math: 2D/3D vectors

Vector

$$\mathbf{a} = \vec{a} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} \qquad \mathbf{b} = \vec{b} = \begin{pmatrix} 6 \\ 2 \\ 1 \end{pmatrix}$$

Length

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

$$|b| = \sqrt{(6)^2 + (2)^2 + (1)^2} \approx 6.4$$

Unit vector

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

$$\hat{b} = \vec{b} \,/\, |b| = \begin{pmatrix} 6 / 6.4 \\ 2 / 6.4 \\ 1 / 6.4 \end{pmatrix} \approx \begin{pmatrix} 0.938 \\ 0.313 \\ 0.156 \end{pmatrix}$$

Math: 2D/3D vectors

Let’s try to calculate the dot product.

Method 1

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z$$

1. $\vec{a} \cdot \vec{b} = 3 \cdot 6 + 4 \cdot 2 + 5 \cdot 1$
2. $\vec{a} \cdot \vec{b} = 18 + 8 + 5$
3. $\vec{a} \cdot \vec{b} = 31$

Math: 2D/3D vectors

Let's try to calculate the dot product.

Method 2

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

1. $31 = \sqrt{50} \cdot \sqrt{41} \cdot \cos\theta$
2. $\cos\theta = 31 / (\sqrt{50} \cdot \sqrt{41})$
3. $\cos\theta \approx 0.6847\ldots.$
4. $\theta = \cos^{-1}(0.6847\ldots.) = 46.79$
5. $\sqrt{50} \cdot \sqrt{41} \cdot \cos(46.79) = 31$

Math: 2D/3D vectors

To-do list

- ☒ 3D vector
- ☒ Vector length
- ☒ Unit vector
- ☒ Dot product
- ☐ Cross product

Math: 2D/3D vectors

The **cross product**:

- “multiplying” vectors
- results in a vector
- right angle
- usually written as $a \times b$

Math: 2D/3D vectors

The **cross product**:

- Formula 1:

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

- Formula 2:

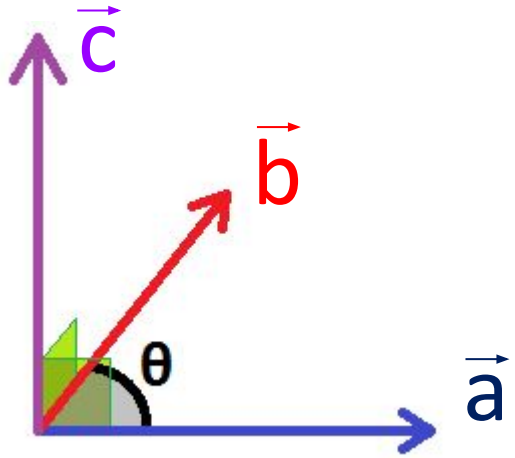
$$\vec{a} \times \vec{b} = |a| \cdot |b| \cdot \sin\theta \cdot \mathbf{n}$$

Math: 2D/3D vectors

The **cross product**:

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

$$\vec{a} \times \vec{b} = \vec{c}$$



Math: 2D/3D vectors

The **cross product**:

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

$$\vec{a} \times \vec{b} = \begin{pmatrix} 3 \\ 4 \\ 5 \end{pmatrix} \times \begin{pmatrix} 6 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \cdot 1 - 5 \cdot 2 \\ 5 \cdot 6 - 3 \cdot 1 \\ 3 \cdot 2 - 4 \cdot 6 \end{pmatrix} = \begin{pmatrix} -6 \\ 27 \\ -18 \end{pmatrix}$$

Math: 2D/3D vectors

To-do list

- ☒ 3D vector
- ☒ Vector length
- ☒ Unit vector
- ☒ Dot product
- ☒ Cross product

Step 2: Math

2D/3D vectors

Matrices

Disclaimer: I am still not a mathematician

Math: Matrices

What you think you will learn

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Math: Matrices

To-do list

- ☐ Matrix specifications
- ☐ Matrix addition
- ☐ Matrix multiplications
- ☐ Identity matrix
- ☐ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

A **matrix**:

- is rectangular
- **n** x **n** dimensions
- represents a linear function

Math: Matrices

A **matrix**:

- usually written as

(3 x 3)

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

(2 x 2)

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

(3 x 2)

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☐ Matrix addition
- ☐ Matrix multiplications
- ☐ Identity matrix
- ☐ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

Matrix + **Scalar**

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} + 5 = \begin{bmatrix} 1+5 & 4+5 & 7+5 \\ 2+5 & 5+5 & 8+5 \\ 3+5 & 6+5 & 9+5 \end{bmatrix} = \begin{bmatrix} 6 & 9 & 12 \\ 7 & 10 & 13 \\ 8 & 11 & 14 \end{bmatrix}$$

Scalars will be applied to every element in the matrix.

Math: Matrices

Matrix + **Matrix**

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} + \begin{bmatrix} 8 & 5 & 2 \\ 7 & 4 & 1 \\ 6 & 3 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 1+8 & 4+5 & 7+2 \\ 2+7 & 5+4 & 8+1 \\ 3+6 & 6+3 & 9+0 \end{bmatrix} = \begin{bmatrix} 9 & 9 & 9 \\ 9 & 9 & 9 \\ 9 & 9 & 9 \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☐ Matrix multiplications
- ☐ Identity matrix
- ☐ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

Matrix · Matrix

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \cdot \begin{bmatrix} 8 & 5 & 2 \\ 7 & 4 & 1 \\ 6 & 3 & 0 \end{bmatrix}$$

You can **multiply** a matrix as long as the amount of **columns** on the **left** equal the amount of **rows** on the **right**.

Math: Matrices

Matrix · Matrix

$$\begin{bmatrix} (1\cdot 8+4\cdot 7+7\cdot 6) & (1\cdot 5+4\cdot 4+7\cdot 3) & (1\cdot 2+4\cdot 1+7\cdot 0) \\ (2\cdot 8+5\cdot 7+8\cdot 6) & (2\cdot 5+5\cdot 4+8\cdot 3) & (2\cdot 2+5\cdot 1+8\cdot 0) \\ (3\cdot 8+6\cdot 7+9\cdot 6) & (3\cdot 5+6\cdot 4+9\cdot 3) & (3\cdot 2+6\cdot 1+9\cdot 0) \end{bmatrix}$$

$$\begin{bmatrix} 78 & 42 & 6 \\ 99 & 54 & 9 \\ 120 & 66 & 12 \end{bmatrix}$$

Math: Matrices

Matrix · Matrix

$$\begin{bmatrix} (1\cdot 8)+(4\cdot 7)+(7\cdot 6) \\ (2\cdot 8)+(5\cdot 7)+(8\cdot 6) \\ (3\cdot 8)+(6\cdot 7)+(9\cdot 6) \end{bmatrix} \cdot \begin{bmatrix} 8 & 5 & 2 \\ 7 & 4 & 1 \\ 6 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 78 & 42 & 6 \\ 99 & 54 & 9 \\ 120 & 66 & 12 \end{bmatrix}$$

Math: Matrices

Matrix · Matrix

Can you multiply these matrices?

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \cdot \begin{bmatrix} 8 & 5 \\ 7 & 4 \\ 6 & 3 \end{bmatrix}$$

Math: Matrices

Matrix · Matrix

Can you multiply these matrices?

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \cdot \begin{bmatrix} 8 & 5 \\ 7 & 4 \\ 6 & 3 \end{bmatrix}$$

Math: Matrices

Matrix · Vector

Can you multiply this matrix by this vector?

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \cdot \begin{bmatrix} 8 \\ 7 \\ 6 \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☒ Matrix multiplications
- ☐ Identity matrix
- ☐ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

An **identity matrix**:

- consists of 0’s
- with diagonal 1’s
- starting point
- same result

Math: Matrices

The **identity matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Math: Matrices

An **identity matrix**:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☒ Matrix multiplications
- ☒ Identity matrix
- ☐ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

A **scale matrix**:

- diagonal scale factor
- multiplies members of vector
- modifies length
- does not modify direction

Math: Matrices

A **scale matrix**:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Math: Matrices

A **scale matrix**:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Math: Matrices

A **scale matrix**:

Scaling each member by a factor 2.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 \\ 2 \cdot 4 \\ 2 \cdot 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \\ 10 \\ 1 \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☒ Matrix multiplications
- ☒ Identity matrix
- ☒ Scale matrix
- ☐ Translation matrix
- ☐ Rotation matrix

Math: Matrices

A **translation matrix**:

- column addition factor
- adds to members of vector
- modifies length
- modifies direction

Math: Matrices

A **translation matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & T_1 \\ 0 & 1 & 0 & T_2 \\ 0 & 0 & 1 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Math: Matrices

A **translation matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & T_1 \\ 0 & 1 & 0 & T_2 \\ 0 & 0 & 1 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_1 \\ y + T_2 \\ z + T_3 \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Math: Matrices

A **translation matrix**:

Translating with the vector [1, 2, 3].

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 3+1 \\ 4+2 \\ 5+3 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 8 \\ 1 \end{bmatrix}$$

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☒ Matrix multiplications
- ☒ Identity matrix
- ☒ Scale matrix
- ☒ Translation matrix
- ☐ Rotation matrix

Math: Matrices

A **rotation matrix**:

- rotates around an axis
- modifies other 2 axis
- does not modify length
- modifies direction

Math: Matrices

Rotation matrices:

x-axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

y-axis

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

z-axis

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Math part 4: Matrix · Vector

x-axis rotation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \cdot \cos(\theta) + z \cdot \sin(\theta) \\ y \cdot -\sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

(θ is the rotation)

Math part 4: Matrix · Vector

x-axis rotation matrix:

Rotating by 1 **radian**. (almost 57.3 degrees)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(1) & \sin(1) & 0 \\ 0 & -\sin(1) & \cos(1) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \cdot 0.54 + 5 \cdot 0.84 \\ 4 \cdot -0.84 + 5 \cdot 0.54 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 6.37 \\ -0.66 \\ 1 \end{bmatrix}$$

(Rotation in radians)

Math part 4: Matrix · Vector

<u>y-axis rotation matrix</u>:

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\theta) + z \cdot -\sin(\theta) \\ y \\ x \cdot \sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

(θ is the rotation)

Math part 4: Matrix · Vector

y-axis rotation matrix:

Rotating by 1 **radian**. (almost 57.3 degrees)

$$\begin{bmatrix} \cos(1) & 0 & -\sin(1) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(1) & 0 & \cos(1) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 0.54 + 5 \cdot -0.84 \\ 4 \\ 3 \cdot 0.84 + 5 \cdot 0.54 \\ 1 \end{bmatrix} = \begin{bmatrix} -2.59 \\ 4 \\ 5.23 \\ 1 \end{bmatrix}$$

(Rotation in radians)

Math part 4: Matrix · Vector

z-axis rotation matrix:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\theta) + y \cdot \sin(\theta) \\ x \cdot -\sin(\theta) + y \cdot \cos(\theta) \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

(θ is the rotation)

Math part 4: Matrix · Vector

z-axis rotation matrix:

Rotating by 1 **radian**. (almost 57.3 degrees)

$$\begin{bmatrix} \cos(1) & \sin(1) & 0 & 0 \\ -\sin(1) & \cos(1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \cdot 0.54 + 4 \cdot 0.84 \\ 3 \cdot -0.84 + 4 \cdot 0.54 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 4.99 \\ -0.36 \\ 5 \\ 1 \end{bmatrix}$$

(Rotation in radians)

Math: Matrices

To-do list

- ☒ Matrix specifications
- ☒ Matrix addition
- ☒ Matrix multiplications
- ☒ Identity matrix
- ☒ Scale matrix
- ☒ Translation matrix
- ☒ Rotation matrix

Step 1: Adding a texture

Step 2: Math

Step 3: Applying math in WebGL

Disclaimer: I am not a mathematician

Applying the math

What's left we'll learn this lesson.

How much we've learned so far.

Applying the math

To-do list

- ☐ Create math library
- ☐ Edit Vertex Shader
- ☐ Identity matrix code
- ☐ Matrix multiplications code
- ☐ Scale matrix in code
- ☐ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

We'll build our own small math library.

```
// Object holding functions for 3d math
var math3d = new function()
{

}
```

Applying the math

Converter for radians/degrees.

```
this.degreesToRadians = function(degrees)
{
    return (degrees * (Math.PI / 180.0));
}

this.radiansToDegrees = function(radians)
{
    return (radians * (180.0 / Math.PI));
}
```

Applying the math

To-do list

- [x] Create math library
- [ ] Edit Vertex Shader
- [ ] Identity matrix code
- [ ] Matrix multiplications code
- [ ] Scale matrix in code
- [ ] Translation matrix in code
- [ ] Rotation matrices in code

Applying the math

Change the Vertex Shader.

```
// 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;

    uniform mat4 transform;

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

We only supply the transformation

Applying the math

Get the transform uniform location in the vertex shader.

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
```

Supply our transform matrix.

```
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☐ Identity matrix code
- ☐ Matrix multiplications code
- ☐ Scale matrix in code
- ☐ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

The **identity matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying the math

Identity matrix function:

```
// Object holding functions for 3d math
var math3d = new function()
{
    // Return the identity matrix (a matrix with diagonal 1's so each row and column will always have a 1 once)
    this.identityMatrix = function()
    {
        return new Float32Array(
            [1, 0, 0, 0,
             0, 1, 0, 0,
             0, 0, 1, 0,
             0, 0, 0, 1]);
    }
}
```

This calls returns an identity matrix:

```
var matrix = math3d.identityMatrix();
```

Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☒ Identity matrix code
- ☐ Matrix multiplications code
- ☐ Scale matrix in code
- ☐ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

Matrix · Matrix

$$\begin{bmatrix} (1\cdot 8+4\cdot 7+7\cdot 6) & (1\cdot 5+4\cdot 4+7\cdot 3) & (1\cdot 2+4\cdot 1+7\cdot 0) \\ (2\cdot 8+5\cdot 7+8\cdot 6) & (2\cdot 5+5\cdot 4+8\cdot 3) & (2\cdot 2+5\cdot 1+8\cdot 0) \\ (3\cdot 8+6\cdot 7+9\cdot 6) & (3\cdot 5+6\cdot 4+9\cdot 3) & (3\cdot 2+6\cdot 1+9\cdot 0) \end{bmatrix}$$

$$\begin{bmatrix} 78 & 42 & 6 \\ 99 & 54 & 9 \\ 120 & 66 & 12 \end{bmatrix}$$

Applying the math

The **matrix multiplication** function:

```
this.multiplyMatrix = function(m1, m2)
{
    return new Float32Array(
        [m1[0]*m2[0] + m1[1]*m2[4] + m1[2]*m2[8] + m1[3]*m2[12],
         m1[0]*m2[1] + m1[1]*m2[5] + m1[2]*m2[9] + m1[3]*m2[13],
         m1[0]*m2[2] + m1[1]*m2[6] + m1[2]*m2[10] + m1[3]*m2[14],
         m1[0]*m2[3] + m1[1]*m2[7] + m1[2]*m2[11] + m1[3]*m2[15],
         m1[4]*m2[0] + m1[5]*m2[4] + m1[6]*m2[8] + m1[7]*m2[12],
         m1[4]*m2[1] + m1[5]*m2[5] + m1[6]*m2[9] + m1[7]*m2[13],
         m1[4]*m2[2] + m1[5]*m2[6] + m1[6]*m2[10] + m1[7]*m2[14],
         m1[4]*m2[3] + m1[5]*m2[7] + m1[6]*m2[11] + m1[7]*m2[15],
         m1[8]*m2[0] + m1[9]*m2[4] + m1[10]*m2[8] + m1[11]*m2[12],
         m1[8]*m2[1] + m1[9]*m2[5] + m1[10]*m2[9] + m1[11]*m2[13],
         m1[8]*m2[2] + m1[9]*m2[6] + m1[10]*m2[10] + m1[11]*m2[14],
         m1[8]*m2[3] + m1[9]*m2[7] + m1[10]*m2[11] + m1[11]*m2[15],
         m1[12]*m2[0] + m1[13]*m2[4] + m1[14]*m2[8] + m1[15]*m2[12],
         m1[12]*m2[1] + m1[13]*m2[5] + m1[14]*m2[9] + m1[15]*m2[13],
         m1[12]*m2[2] + m1[13]*m2[6] + m1[14]*m2[10] + m1[15]*m2[14],
         m1[12]*m2[3] + m1[13]*m2[7] + m1[14]*m2[11] + m1[15]*m2[15]]);
}
```

(Calculates 1 array item per line, 16 in total)

Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☒ Identity matrix code
- ☒ Matrix multiplications code
- ☐ Scale matrix in code
- ☐ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

A **scale matrix**:

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Applying the math

The code for the **scale** function.

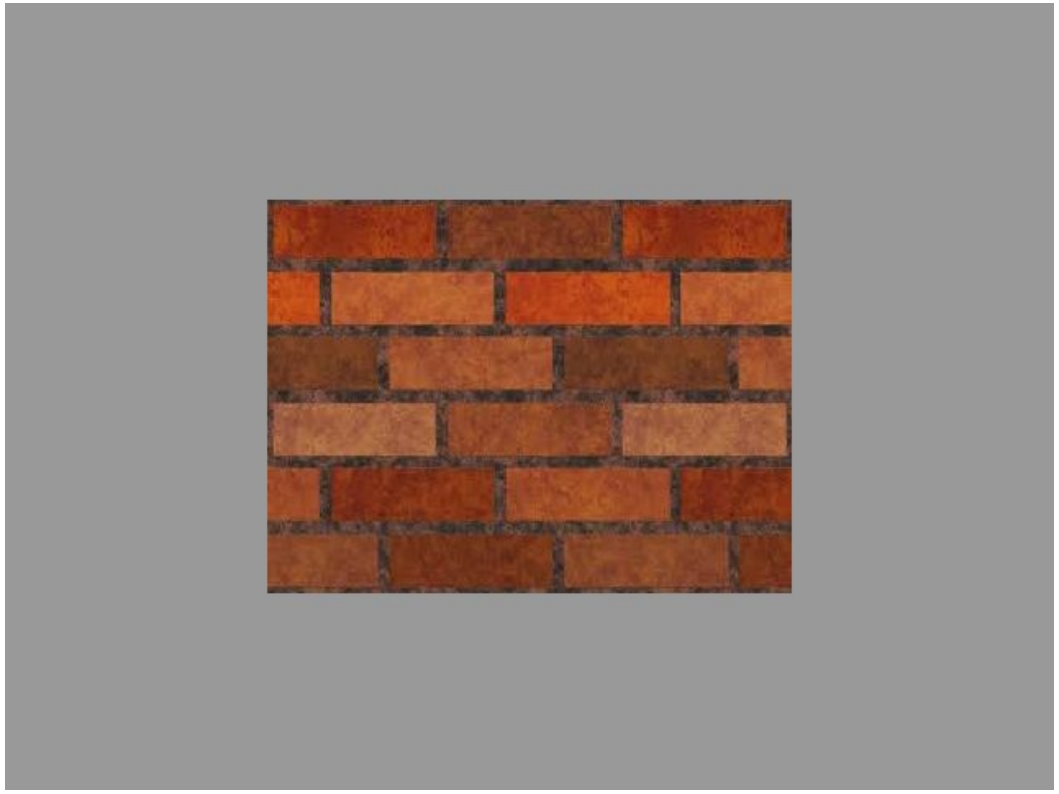
```
this.scaleMatrix = function(s1, s2, s3)
{
    return new Float32Array(
        [s1, 0, 0, 0,
         0, s2, 0, 0,
         0, 0, s3, 0,
         0, 0, 0, 1]);
}
```

Usage

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
var matrix = math3d.identityMatrix();
matrix = math3d.multiplyMatrix(matrix, math3d.scaleMatrix(1.5, 1.5, 1.5));
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Applying the math

Scaling by 1.5:



Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☒ Identity matrix code
- ☒ Matrix multiplications code
- ☒ Scale matrix in code
- ☐ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

A **translation matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & T_1 \\ 0 & 1 & 0 & T_2 \\ 0 & 0 & 1 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_1 \\ y + T_2 \\ z + T_3 \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Applying the math

The code for the **translate** function.

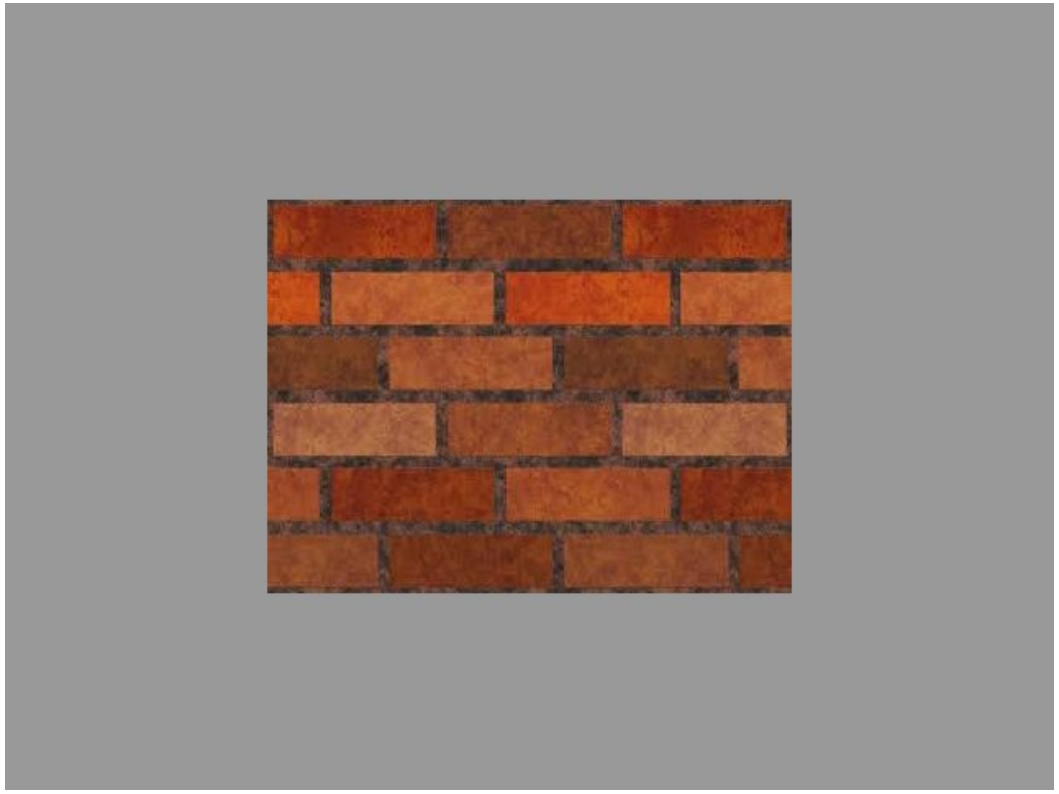
```
this.translationMatrix = function(t1, t2, t3)
{
    return new Float32Array(
        [1, 0, 0, t1,
         0, 1, 0, t2,
         0, 0, 1, t3,
         0, 0, 0, 1]);
}
```

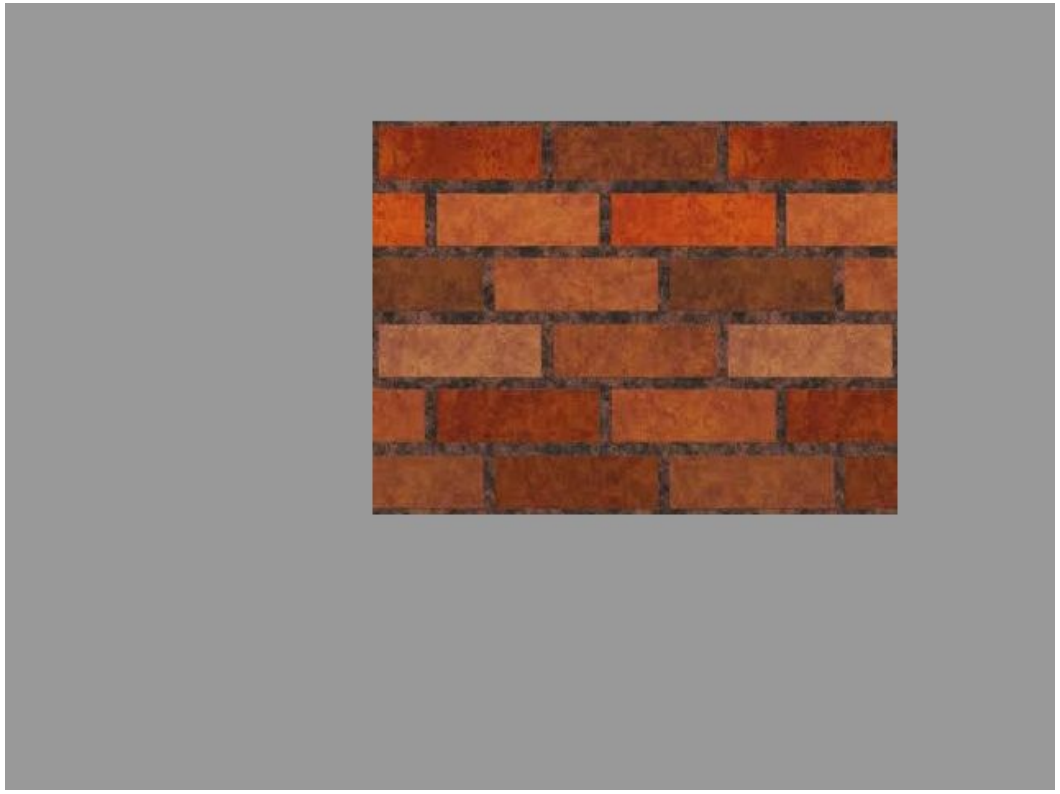
Usage

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
var matrix = math3d.identityMatrix();
matrix = math3d.multiplyMatrix(matrix, math3d.translationMatrix(0.2, 0.2, 0.2));
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Applying the math

Translating by [0.2, 0.2, 0.2]:





Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☒ Identity matrix code
- ☒ Matrix multiplications code
- ☒ Scale matrix in code
- ☒ Translation matrix in code
- ☐ Rotation matrices in code

Applying the math

An **x-axis rotation matrix**:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \cdot \cos(\theta) + z \cdot \sin(\theta) \\ y \cdot -\sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Applying the math

The code for the x rotation function.

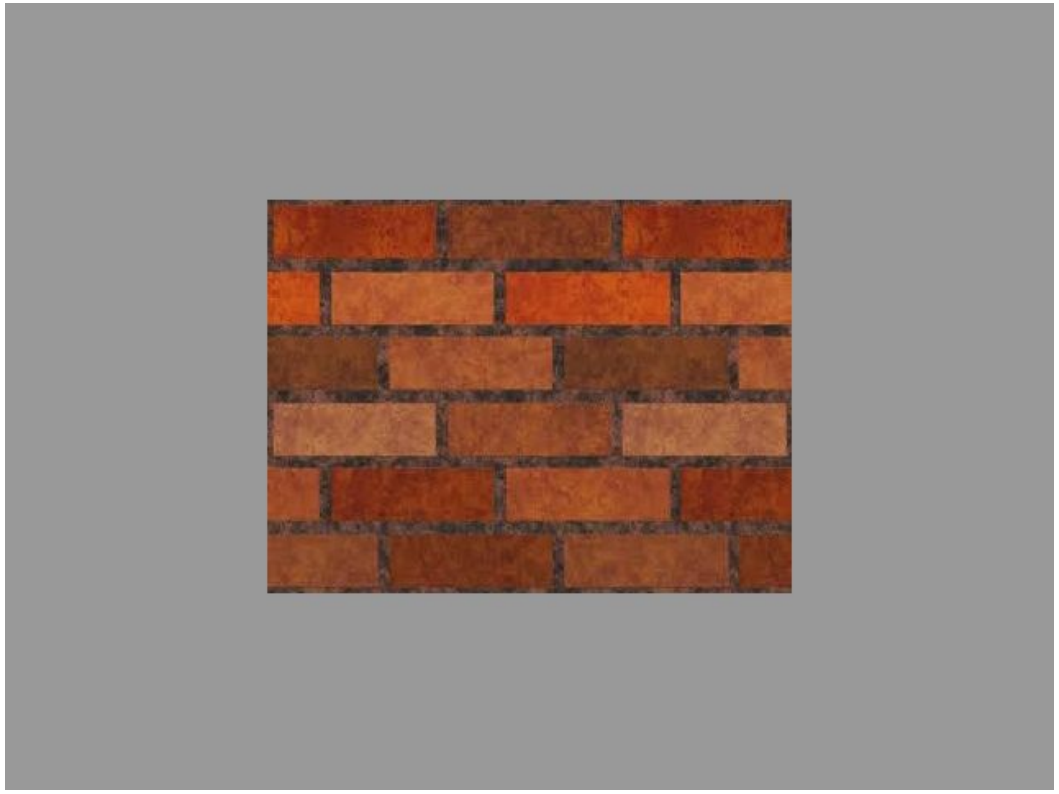
```
// Clockwise rotation on the x axis
this.xRotationMatrix = function(rotationInDegrees)
{
    var rotationInRadians = this.degreesToRadians(rotationInDegrees);
    var c = Math.cos(rotationInRadians);
    var s = Math.sin(rotationInRadians);

    return new Float32Array(
        [1, 0, 0, 0,
         0, c, s, 0,
         0, -s, c, 0,
         0, 0, 0, 1]);
}
```

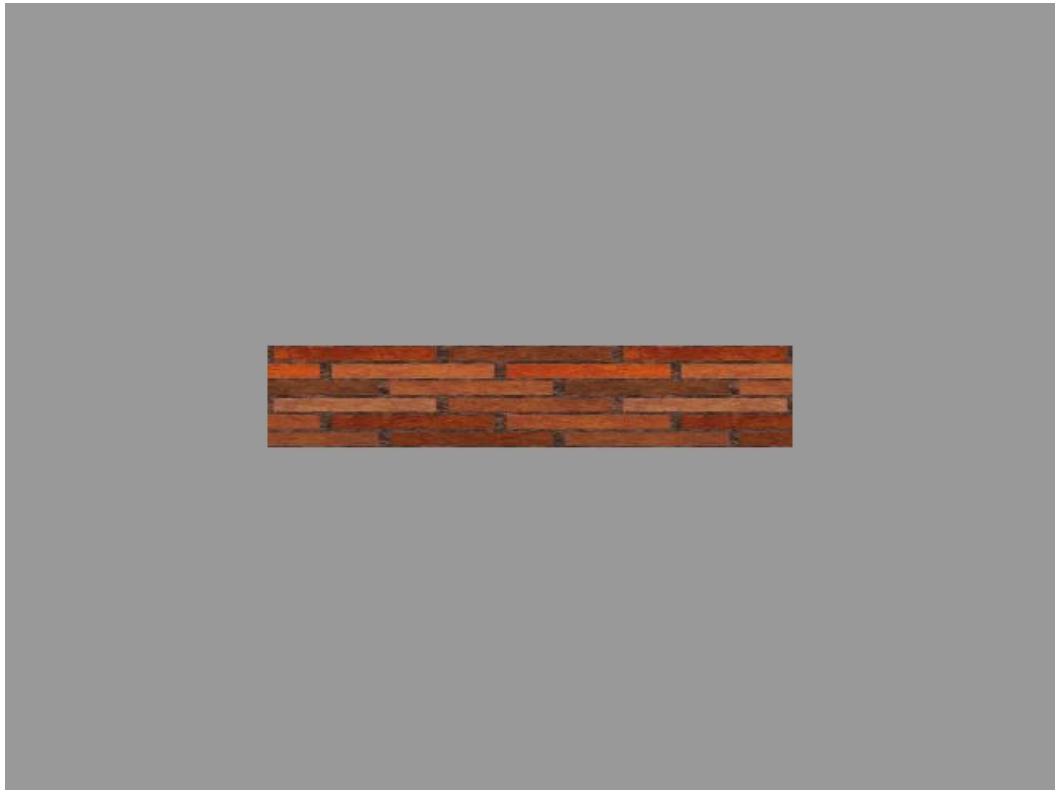
Usage

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
var matrix = math3d.identityMatrix();
matrix = math3d.multiplyMatrix(matrix, math3d.xRotationMatrix(55));
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Applying the math

Rotating around the x axis by 75 degrees.





Applying the math

A **<u>y-axis rotation matrix</u>**:

$$\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\theta) + z \cdot -\sin(\theta) \\ y \\ x \cdot \sin(\theta) + z \cdot \cos(\theta) \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Applying the math

The code for the y rotation function.

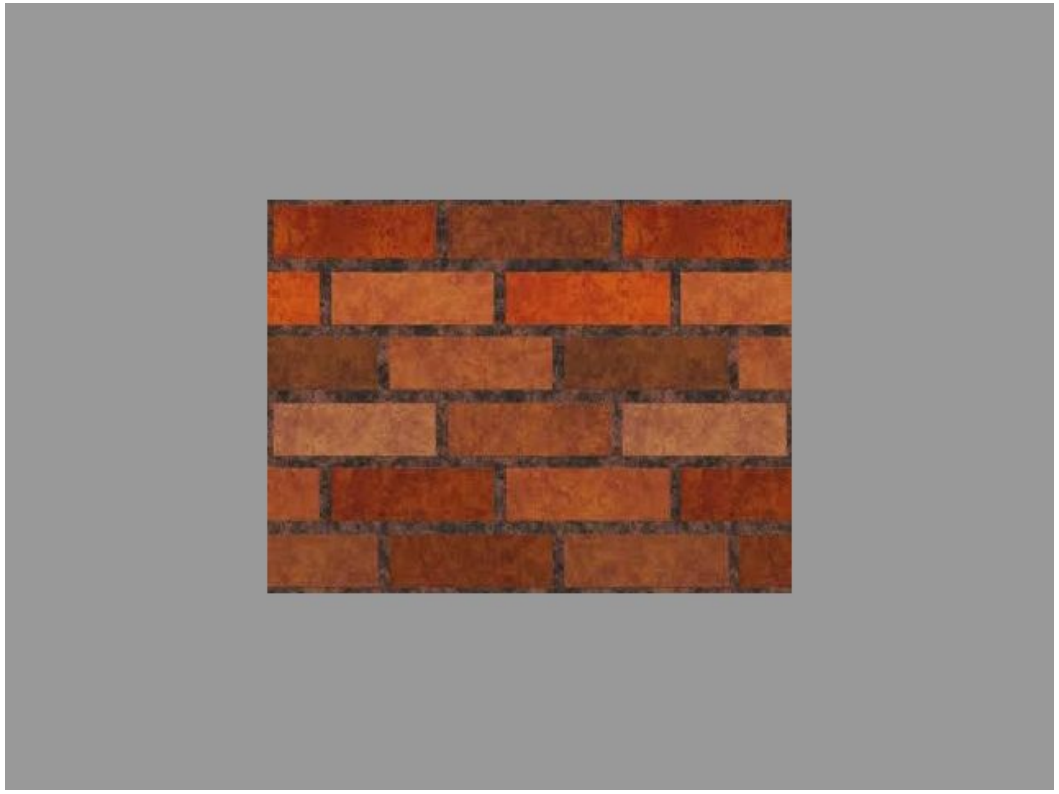
```
// Clockwise rotation on the y axis
this.yRotationMatrix = function(rotationInDegrees)
{
    var rotationInRadians = this.degreesToRadians(rotationInDegrees);
    var c = Math.cos(rotationInRadians);
    var s = Math.sin(rotationInRadians);

    return new Float32Array(
        [c, 0, -s, 0,
         0, 1, 0, 0,
         s, 0, c, 0,
         0, 0, 0, 1]);
}
```

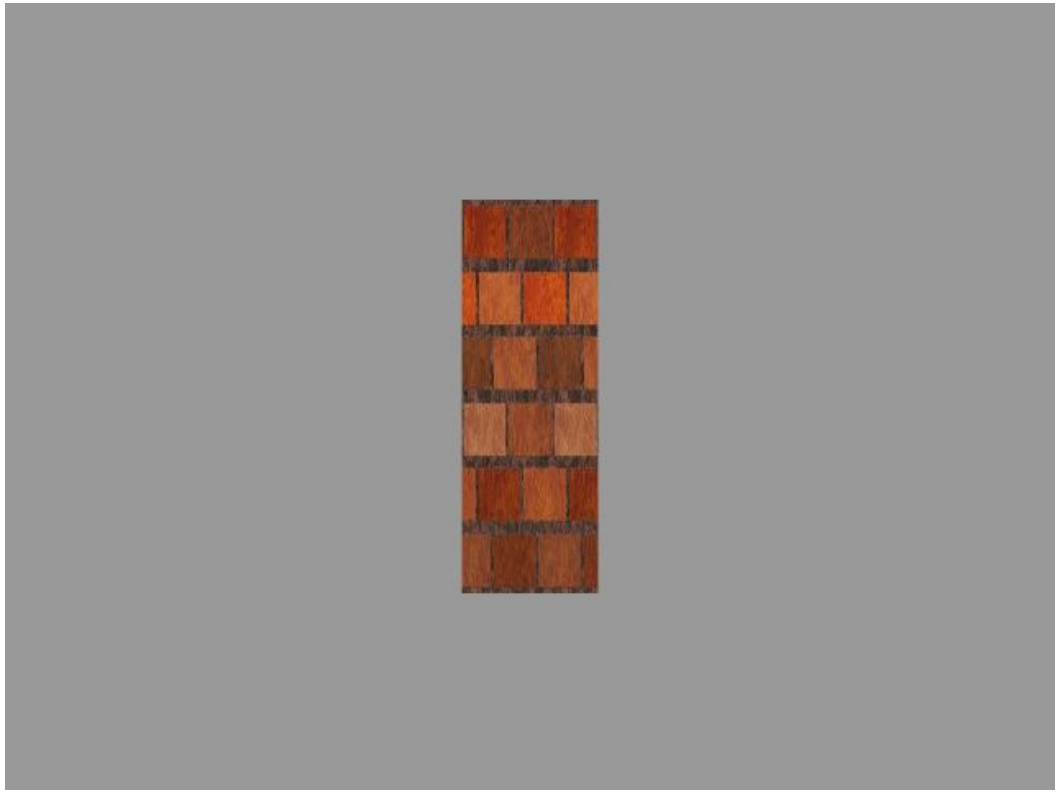
Usage

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
var matrix = math3d.identityMatrix();
matrix = math3d.multiplyMatrix(matrix, math3d.yRotationMatrix(55));
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Math part 4: Matrix · Vector

Rotating around the y axis by 75 degrees.





Applying the math

A **z-axis rotation matrix**:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\theta) + y \cdot \sin(\theta) \\ x \cdot -\sin(\theta) + y \cdot \cos(\theta) \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ 1 \end{bmatrix}$$

Applying the math

The code for the z rotation function.

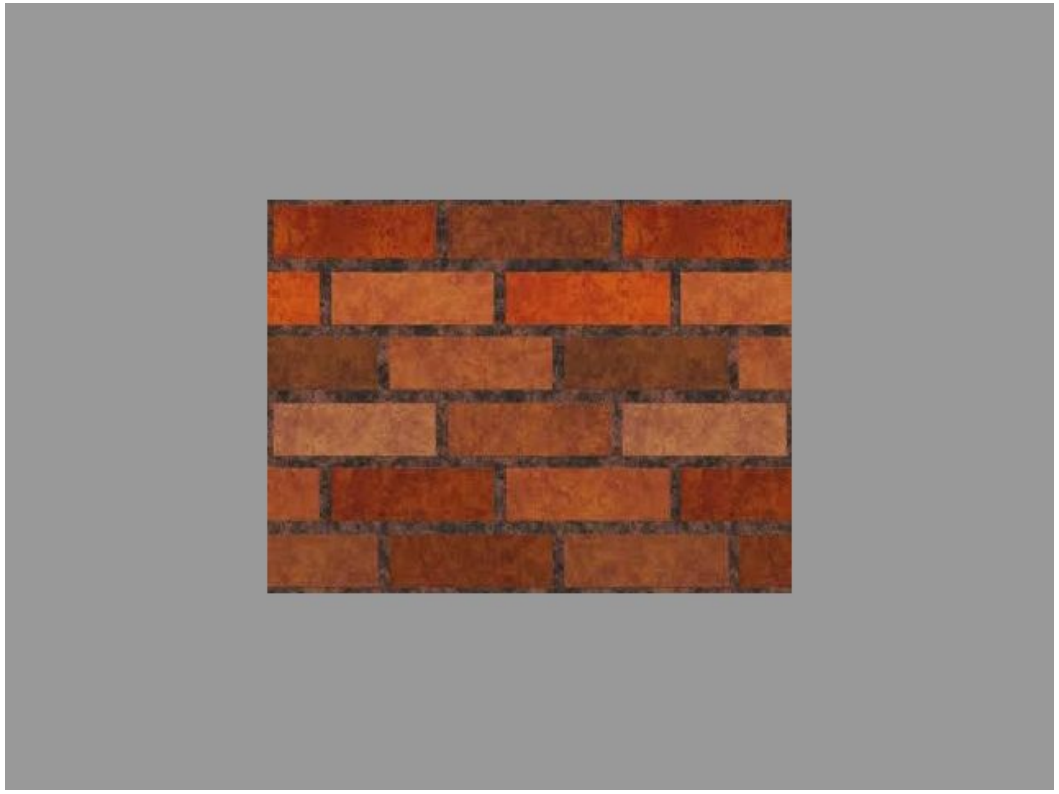
```
// Clockwise rotation on the z axis
this.zRotationMatrix = function(rotationInDegrees)
{
    var rotationInRadians = this.degreesToRadians(rotationInDegrees);
    var c = Math.cos(rotationInRadians);
    var s = Math.sin(rotationInRadians);

    return new Float32Array(
        [c, s, 0, 0,
         -s, c, 0, 0,
         0, 0, 1, 0,
         0, 0, 0, 1]);
}
```

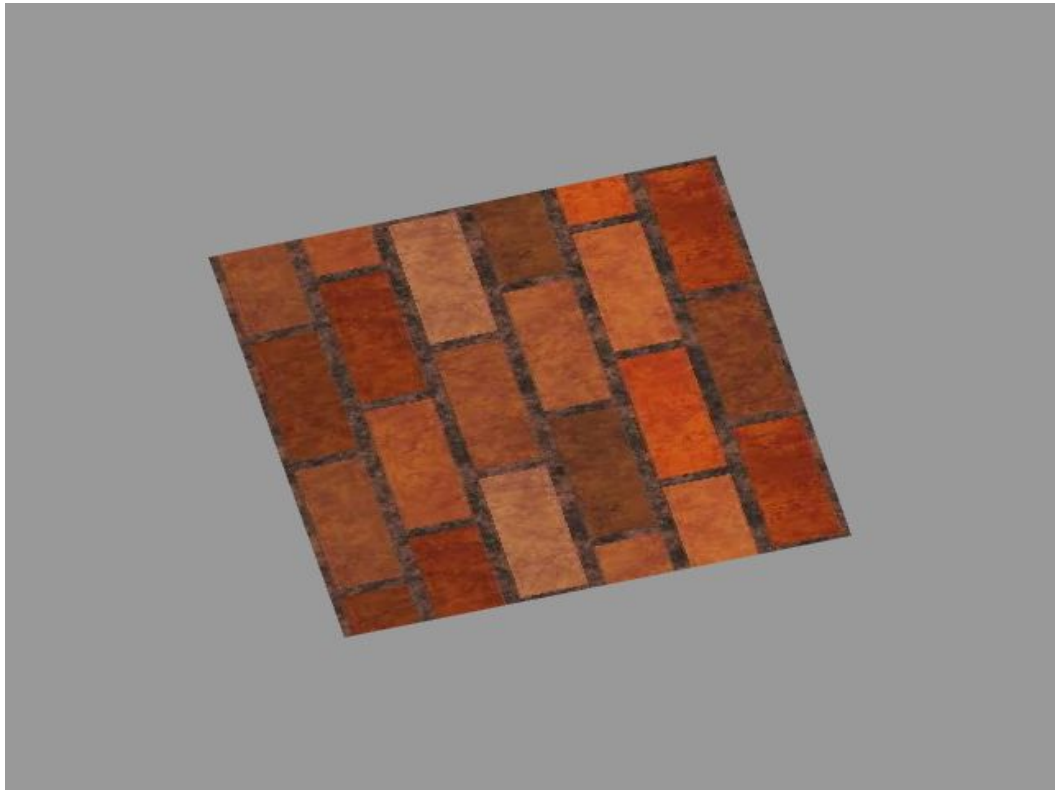
Usage

```
var transform_location = gl.getUniformLocation(shaderProgram, "transform");
var matrix = math3d.identityMatrix();
matrix = math3d.multiplyMatrix(matrix, math3d.zRotationMatrix(55));
gl.uniformMatrix4fv(transform_location, false, matrix);
```

Applying the math

Rotating around the z axis by 75 degrees.





Applying the math

To-do list

- ☒ Create math library
- ☒ Edit Vertex Shader
- ☒ Identity matrix code
- ☒ Matrix multiplications code
- ☒ Scale matrix in code
- ☒ Translation matrix in code
- ☒ Rotation matrices in code

To be continued next time…

Thanks for listening.

Part IV

Presentation by: Quincy Jacobs