WebGL

Bring advanced graphics
to the web.

Jorge González Delgado
Lucas Hernández Abreu

Contact Information

Jorge González Delgado

<alu0101330105@ull.edu.es>

Lucas Hernández Abreu

<alu0101317496@ull.edu.es>

Index

3. Why should I use it
OK, seems good, but why should I use it?

2. How does it work
What is WebGl used for and how does it do it

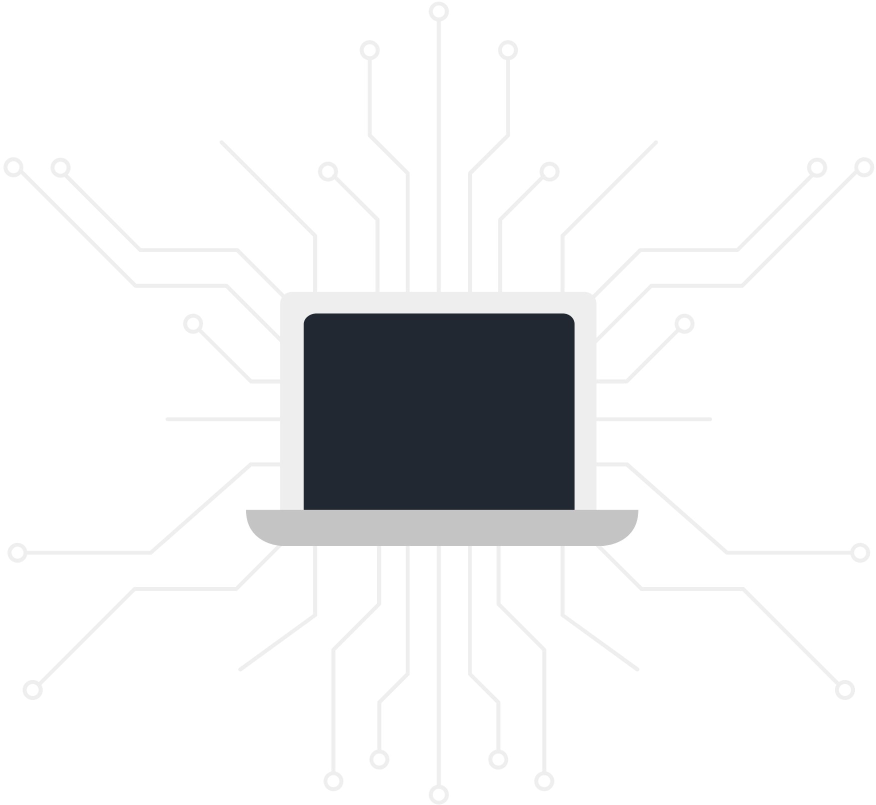
1. What is WebGl
Get to know what is WebGl and where it comes from

4. How to use WebGl
Now i want to start doing some too!

5. Example Construction
Let’s build a webGl application step for step.

6. ThreeJs
Lets not use raw WebGL for this again.

Language specifications



Shader	Sombreador
Fragment Shader	Sombreador de fragmentos
Vertex Shader	Sombreador de vértices
API	

What is WebGl

General

Made By

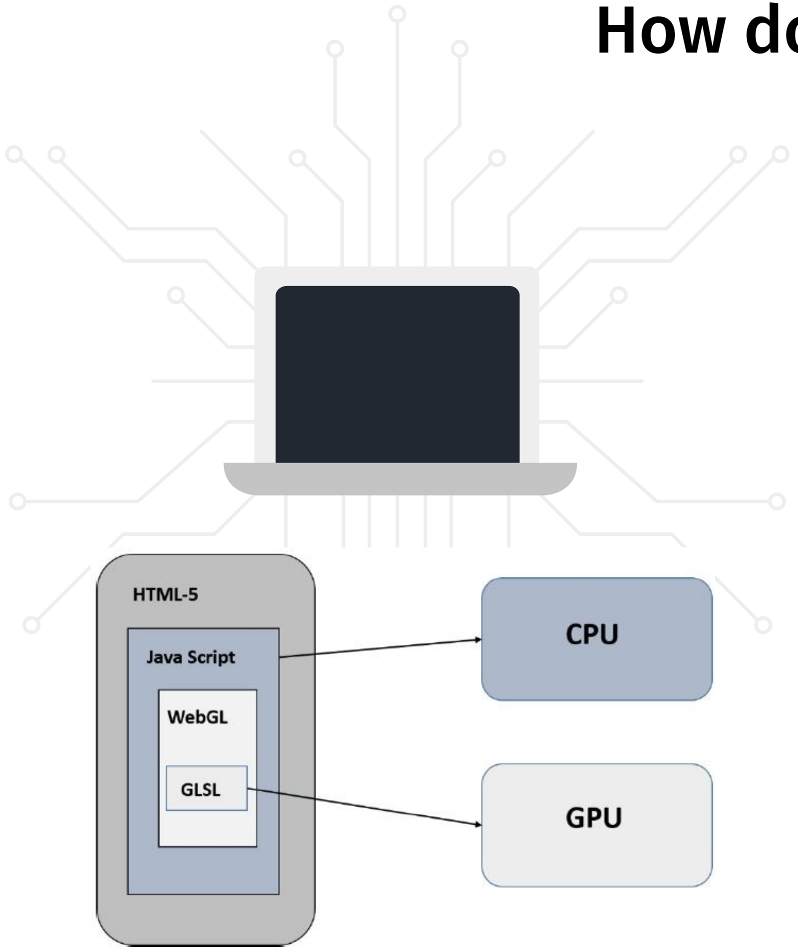
OpenGl

How does it show

How does it work

- Low level 3D graphics API based on OpenGl.
- WebGl was developed by Khronos, a group focused on developing free open standards for 3D graphics, augmented and virtual reality and many more.
- OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
- WebGl makes use of the HTML5 canvas element.
- WebGl uses hardware acceleration to render GLSL shaders as a 3D environment.

How does WebGl work?



Canvas

As said previously, WebGl is represented in a web using the HTML5 canvas element, we just need to use another context.

Script

The JavaScript program will only be used to create a link between the Shader and the canvas. We do not create 3D graphics with JavaScript. Or do we?

Shader

A shader is a piece of code written in GLSL a C/C++ like language that runs entirely in your GPU. We actually need 2 shaders to render any shape: A vertex shader and a fragment shader.

What is a shader

Shader

Code written in GLSL that runs in your GPU

GLSL

Literally OpenGL Shading Language

We need 2

As said earlier, we need 2 shaders, vertex and fragment

Vertex

Simply speaking, this creates the vertex in the space

Fragment

And this paints each pixel with their corresponding color when rasterized

Vertex Shader Data

```
attribute vec2 position
uniform vec2 zoomCenter
uniform sampler2D texture
```

Attribute

Data pulled from buffers.

Uniforms

Values that stay the same for all vertices of a single draw call

Textures

Data from pixels/texels

Vertex Shader

```
precision highp float;
attribute vec2 a_Position;
void main() {
  gl_Position = vec4(a_Position.x, a_Position.y, 0.0, 1.0);
}
```

Objective

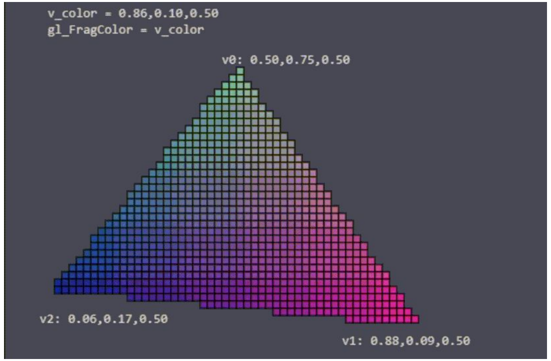
Generate Clip Space Coordinates for each vertex we wanna represent (spoiler: a lot)

Utility

We must create the vertex of our polygons for webGL to rasterize them. We can not see the images if there are no vertex on them

Fragment Shader

```
void main() {
   gl_Position = doMathToMakeClipspaceCoordinates
}
```



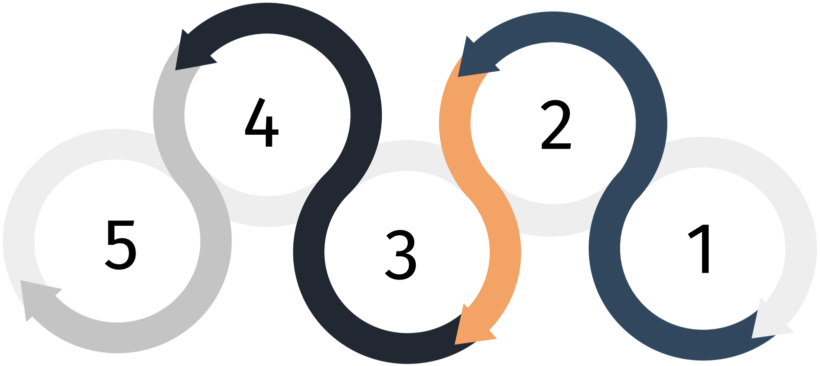
Objective

Shader stage that will process a fragment generated by the Rasterization into a set of colors and a single depth value.

Shader

The fragment shader is the OpenGL pipeline stage after a primitive is rasterized

Why should I use WebGl



OpenGl

Since OpenGl is quite popular there is a lot of documentation on internet

Support

WebGl is currently supported by a lot of browsers, including Internet Explorer after version 11

Tasks

It can perform tasks that are just not possible by other technologies, or more accurately would be extremely complex and difficult

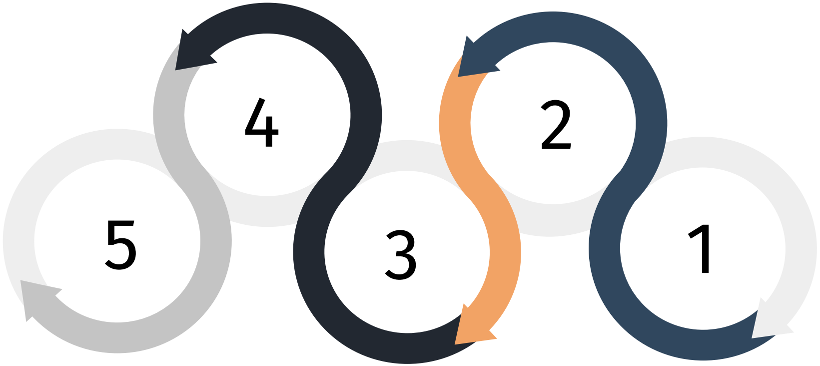
Performance

Performance. WebGL is blindingly fast and fully utilizes hardware acceleration, making it suitable for games or complex visualizations.

Shaders

Shaders are so polivalent the can produce from, a simple sepia filter to real-time complex raymarching

Why shouldn't I use WebGl



Precision problems

WebGL uses 32bit numbers.

There are 3 precision settings:
lowp, mediump and highp.

Since we use the low size numbers to represent the images for device compatibility, we cannot be precise in our drawings because of the lack of bits.

WebGL limitations

WebGL ONLY represents lines, dots and triangles. Anything else you will have to create it with this three elements.

WebGl examples

ShaderToy

In Shadertoy you can see what the community is able to do in WebGl

Example: Rainforest

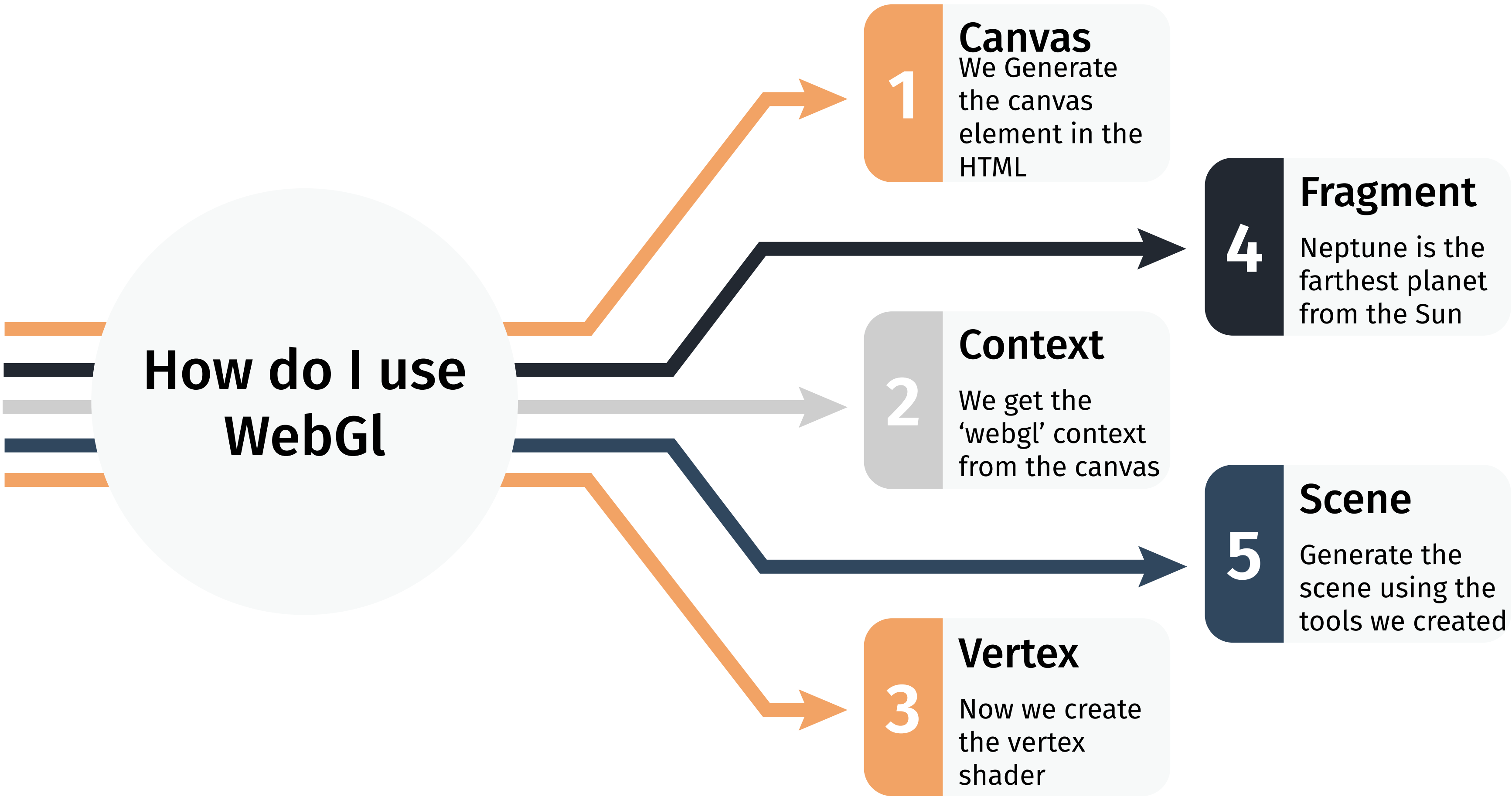
Awesome shader made by Iñigo Quiles
https://www.shadertoy.com/view/4ttSWf

Of course, games too.

MontBlanc Legend Race
https://therace.montblanclegend.com/

Example: Fóvea detector

Visual illusion made by nimitz
https://www.shadertoy.com/view/4dsXzM


How do I use WebGl
1
Canvas
We Generate the canvas element in the HTML
2
Context
We get the ‘webgl’ context from the canvas
3
Vertex
Now we create the vertex shader
4
Fragment
Neptune is the farthest planet from the Sun
5
Scene
Generate the scene using the tools we created

Creating WebGl application
Part 1: Create Canvas

```html
<!doctype html>
<html lang="en">
  <head>
    <title>WebGL Demo</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="./webgl.css" type="text/css">
  </head>

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

Canvas Element

We start by creating the Canvas element inside of our HTML file and fill everything else as we want.

Remember

Remember that you can also create your canvas inside of your script.

Creating WebGl application
Part 2: Script and context

```
function main() {
  const canvas = document.querySelector("#glCanvas");
  // Initialize the GL context
  const gl = canvas.getContext("webgl");

  // Only continue if WebGL is available and working
  if (gl === null) {
    alert("Unable to initialize WebGL. Your browser or 
machine may not support it.");
    return;
  }
}
```

Stating the Script

Now it’s time to start our script and import the canvas we just created from the HTML

Get Context

Instead of getting the 2d context as we always have done, we are gonna get the ‘webgl’ context.
No need to install anything, we can use webGl now.

Creating WebGl application
Part 3: Creating the Vertex Shader

```
  const vsSource = `
    attribute vec4 aVertexPosition;

    uniform mat4 uModelViewMatrix;
    uniform mat4 uProjectionMatrix;

    void main() {
      gl_Position = uProjectionMatrix * uModelViewMatrix *
aVertexPosition;
    }
  `;
```

GLSL

Any Shader must be written in GLSL, and passed as an string to the program itself.

Attribute

Our vertex gets as an attribute a 4D vector as the vertex position.

Perspective

As you see, we also create 2 uniform matrix. That’s a 4x4 float point matrix that contains the transformation parameters to simulate the change the depth produces in the size and shape.

gl_Position

gl_Position is the attribute of each vertex that stores its coordinates, so the goal here is to set this attribute to what we want using mathematical equations.

Creating WebGl application
Part 4: Creating the Fragment Shader

```
const fsSource = `
  void main() {
    gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
  }
`;
```

Color

As the Fragment Shader's purpose is to determine the color to show in each pixel of the canvas, to simplify things at first we opt to go for a full black shape.

gl_FragColor

As we saw before with the vertex, this is the attribute to store the color each pixel in the canvas it's gonna receive when the image is rasterized.

Creating WebGl application
Part 5.0: Initializing the Shaders

```
function initShaderProgram(gl, vsSource, fsSource) {
  const vertexShader = loadShader(gl, gl.VERTEX_SHADER,
vsSource);
  const fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER,
fsSource);

  // Create the shader program

  const shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);

  // If creating the shader program failed, alert

  if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
    alert('Unable to initialize the shader program: ' +
gl.getProgramInfoLog(shaderProgram));
    return null;
  }

  return shaderProgram;
}
```

Load

The strings we created with the shader program in it, are just that, a string so we need to give them format in order to work properly. For that purpose we use loadShader and tell them what kind of shader it is.

Initialize

In order to use our shaders we need to tell webGl to use them. To do so we create what's called a program. To this we attach each shader and link it back to the context.

Creating WebGl application
Part 5.1: Loading Shaders

```
function loadShader(gl, type, source) {
  const shader = gl.createShader(type);

  // Send the source to the shader object

  gl.shaderSource(shader, source);

  // Compile the shader program

  gl.compileShader(shader);

  // See if it compiled successfully

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

  return shader;
}
```

```
function main() {
    //...

  // Initialize the shaders
  const shaderProgram = initShaderProgram(gl, vsSource, fsSource);
}
```

Shader

Now we create a new Shader object to work with, we determine the type. In it we source the GLSL code string we get as argument in the function.

Compile

As with any code of any language, our shader needs to be compiled for it to work in our GPU.

Initialized Shaders

The initialized Shaders will be stored in a constant in the main program.

Creating WebGl application

Part 6: Easy access to attributes

```
  const programInfo = {
    program: shaderProgram,
    attribLocations: {
      vertexPosition: gl.getAttribLocation(shaderProgram,
'aVertexPosition'),
    },
    uniformLocations: {
      projectionMatrix: gl.getUniformLocation(shaderProgram,
'uProjectionMatrix'),
      modelViewMatrix: gl.getUniformLocation(shaderProgram,
'uModelViewMatrix'),
    },
  };
```

Attributes

We actually have 2 uniforms and one attribute, the way it works is that attributes receive the next value from a buffer each iteration.

Uniforms

Uniforms are more like JS global variables, they stay with the same value in all iterations.

Locations

We store inside an object the memory address webGl put our attributes and uniforms so we can access them easily if needed.

Creating WebGl application

Part 7: Creating the shape buffer

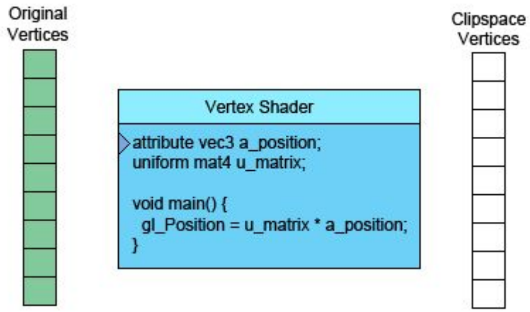
```
function initBuffers(gl) {

  // Create a buffer for the square's positions.

  const positionBuffer = gl.createBuffer();

  // Select the positionBuffer as the one to apply buffer
  // operations to from here out.

  gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

  // Now create an array of positions for the square.

  const positions = [
     1.0,  1.0,
    -1.0,  1.0,
     1.0, -1.0,
    -1.0, -1.0,
  ];

  // Now pass the list of positions into WebGL to build the
  // shape. We do this by creating a Float32Array from the
  // JavaScript array, then use it to fill the current buffer.

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

  return {
    position: positionBuffer,
  };
}
```

Buffer

Buffers are kind of an array that gets sequential read only.

Vertex in Buffer

We store the vertex we want to represent inside the bufferData of our webGl. We explicitly convert the array into a Float32Array to avoid the soft typing of JS to bite us back.



Creating WebGl application

Part 7.0: Rendering the Scene, Preparations

```
function drawScene(gl, programInfo, buffers) {
  gl.clearColor(1.0, 1.0, 1.0, 1.0);  // Clear to White, fully
opaque
  gl.clearDepth(1.0);                 // Clear everything
  gl.enable(gl.DEPTH_TEST);           // Enable depth testing
  gl.depthFunc(gl.LEQUAL);            // Near things obscure
far things

  // Clear the canvas before we start drawing on it.

  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  //...
```

Render the Scene

Since we created all we needed now it's finally time to start making some shapes.

Preparations

We need to configure some basic things like the background and some functions like the depth to make object block light to other objects.

Clear the space

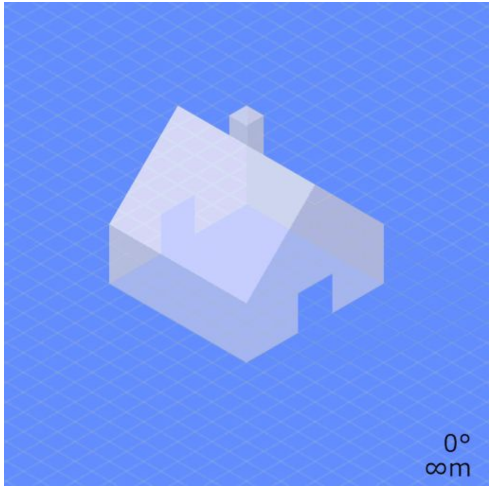
Just as a preventive step, we also clear the whole canvas to start drawing. This is useful in cases where we use the canvas priorly to the webGl render.

Creating WebGl application
Part 7.1: Perspective Matrix

```
function drawScene(gl, programInfo, buffers) {

  //...

  // Create a perspective matrix, a special matrix that is
  // used to simulate the distortion of perspective in a
camera.
  // Our field of view is 45 degrees, with a width/height
  // ratio that matches the display size of the canvas
  // and we only want to see objects between 0.1 units
  // and 100 units away from the camera.

  const fieldOfView = 45 * Math.PI / 180;   // in radians
  const aspect = gl.canvas.clientWidth /
gl.canvas.clientHeight;
  const zNear = 0.1;
  const zFar = 100.0;
  const projectionMatrix = mat4.create();

  // note: glmatrix.js always has the first argument
  // as the destination to receive the result.
  mat4.perspective(projectionMatrix,
                   fieldOfView,
                   aspect,
                   zNear,
                   zFar);

  //...
```

Perspective Matrix

We create a basic perspective matrix to help simulate the distortion a image seen through a lens look like. We do not see the word in orthographic perspective.



Creating WebGl application
Part 7.2: Position

```
function drawScene(gl, programInfo, buffers) {

  //...

  // Set the drawing position to the "identity" point, which is
  // the center of the scene.
  const modelViewMatrix = mat4.create();

  // Now move the drawing position a bit to where we want to
  // start drawing the square.

  mat4.translate(modelViewMatrix,     // destination matrix
                 modelViewMatrix,     // matrix to translate
                 [-0.0, 0.0, -6.0]);  // amount to translate

  //...
```

Positioning

We establish the starting position of our drawing at the centre and then we move it towards the position we want.

Creating WebGl application

Part 7.3: Vertex position extraction

```
function drawScene(gl, programInfo, buffers) {

  //...

  // Tell WebGL how to pull out the positions from the position
  // buffer into the vertexPosition attribute.
  {
    const numComponents = 2;  // pull out 2 values per
iteration
    const type = gl.FLOAT;    // data in the buffer is 32bit
floats
    const normalize = false;  // don't normalize
    const stride = 0;         // how many bytes to get from one
set of values to the next
                              // 0 = use type and numComponents
above
    const offset = 0;         // how many bytes inside the
buffer to start from
    gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
    gl.vertexAttribPointer(
        programInfo.attribLocations.vertexPosition,
        numComponents,
        type,
        normalize,
        stride,
        offset);
    gl.enableVertexAttribArray(
        programInfo.attribLocations.vertexPosition);
  }

  //...
```

Vertex position extraction

Since WebGl is very low level, we have a lot of options when we are going to do things, even when reading the vertex buffer and dumping it into the vertex shader.

Creating WebGl application
Part 7.4: Program selection and Draw

```
function drawScene(gl, programInfo, buffers) {

  //...

  // Tell WebGL to use our program when drawing

  gl.useProgram(programInfo.program);

  // Set the shader uniforms

  gl.uniformMatrix4fv(
      programInfo.uniformLocations.projectionMatrix,
      false,
      projectionMatrix);
  gl.uniformMatrix4fv(
      programInfo.uniformLocations.modelViewMatrix,
      false,
      modelViewMatrix);

  {
    const offset = 0;
    const vertexCount = 4;
    gl.drawArrays(gl.TRIANGLE_STRIP, offset, vertexCount);
  }
}
```

Program

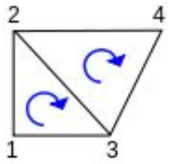
Now we tell WebGl to use the program we created in the initialization process, that includes the compiled shaders.

Set the Uniforms

Lastly we set the uniforms, those matrix we created at the beginning in the vertex shader to accurately represent the vertex in our virtual 3D world.

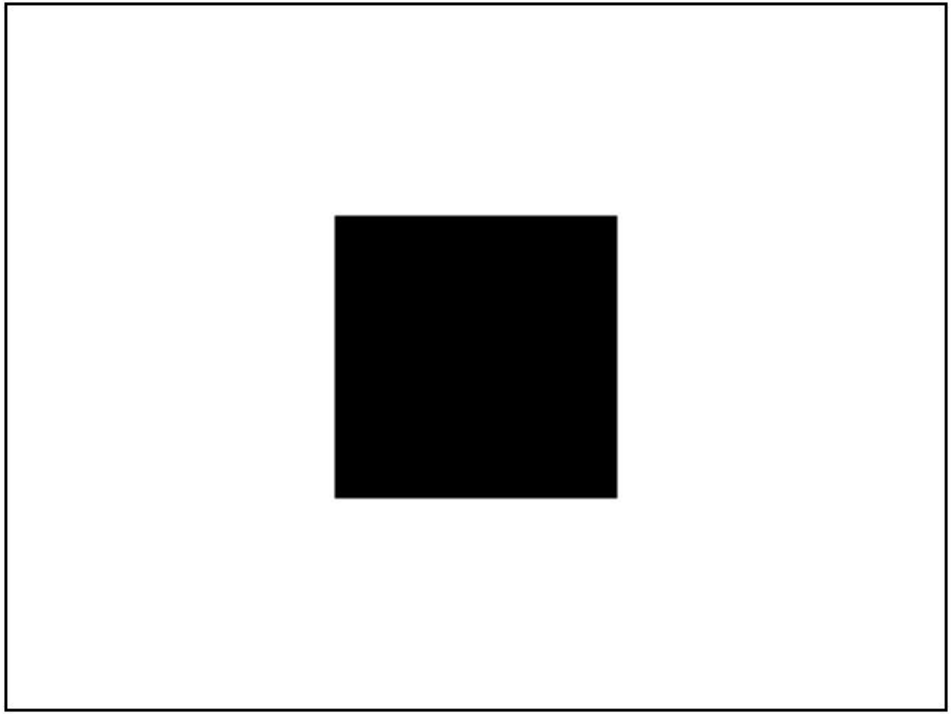
drawArrays

Finally it's time to draw our shape. We do so calling the drawArrays method that requires a mode, a starting offset and a count.



TRIANGLE_STRIP

Creating WebGl application
Part 7.5: Result



We did it!!

Now officially we have drawn a square.
Basically this is what it’s called:
“To kill a fly with a warship”

Three.js

What is Three.js

Three.js is a 3D library that tries to make it as easy as possible to get 3D content on a webpage.

Motivation

Do not suffer with raw WebGl for small things.

Why should I use it

When the complexity of the scene isn't too high there is an advantage in using this library to make it easier.

When shouldn't I use it

When dealing with complex scenes, which require a lot of fine detailing, manual retouches would impair the performance quite a bit compared to do them directly in webGl.

Biography

Code example

https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL

General information

https://www.khronos.org/opengl/
https://webglfundamentals.org/
https://threejs.org/
https://www.toptal.com/javascript/3d-graphics-a-webgl-tutorial

Recommendations

https://www.youtube.com/channel/UCdmAhiG8HQDlz8uyekw4ENw