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!

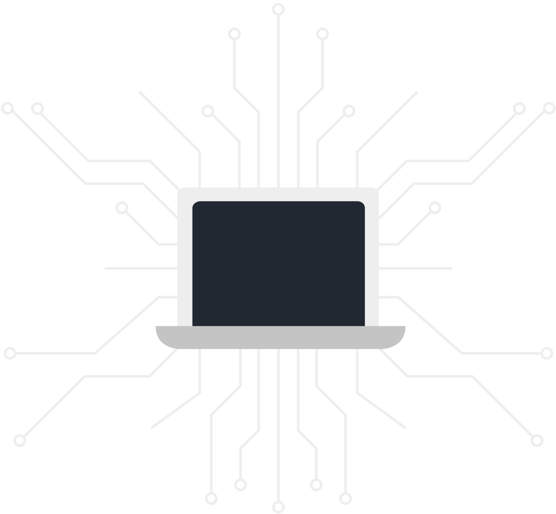
5. Example Construction
Let’s build a webGl application step for step.

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

Context: OpenGL

What is it?

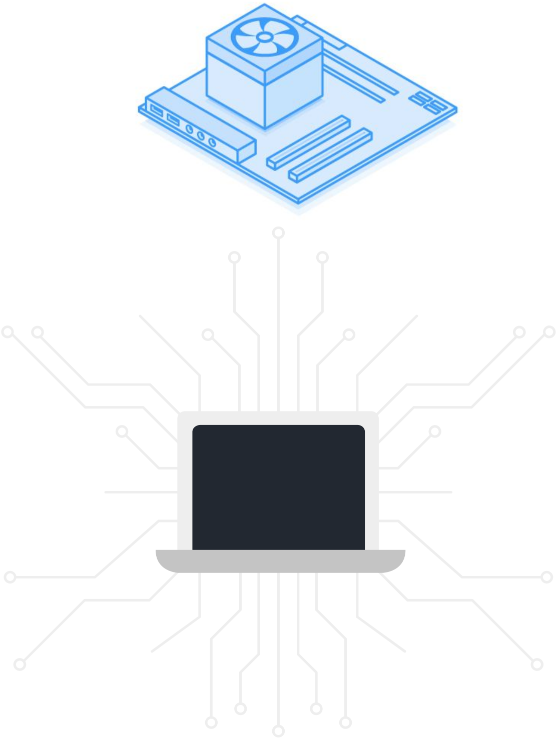
The Open Graphics Library (openGL) is a standard that defines a multi-language and multi-platform API to write programs that produce 2D and 3D graphics



How it works? (brushstrokes)

It works with the use of a high level shader language, the openGL Shading Language (GLSL). Has a syntax based in the C programming language.

Context: Hardware Acceleration



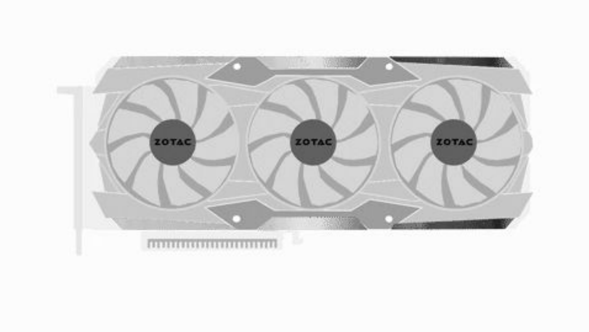
What is it?

Is the use of computer hardware designed to perform specific functions more efficiently when compared to software running on a general-purpose CPU.

Why do we need it?

The making of 2D and 3D graphics require a very high amount of calculations in under a second. The CPU is not specialized in doing this tasks. This is why we let them to the GPU.

Context: GPU



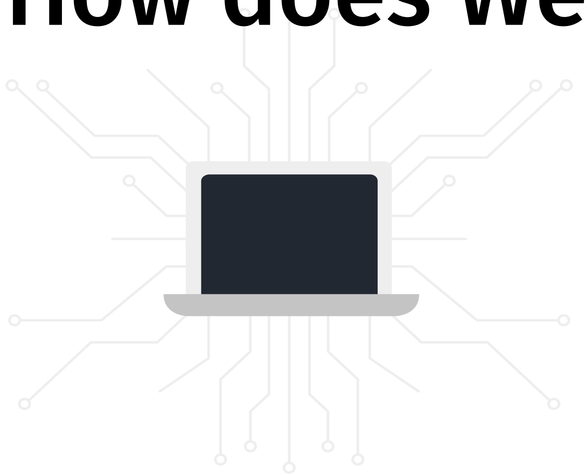
What is it?

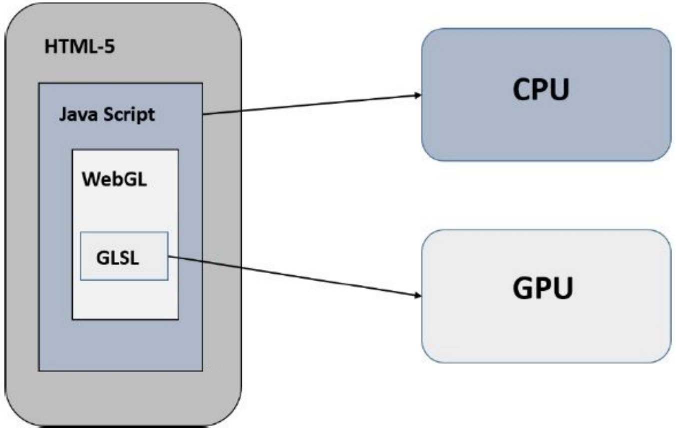
Is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer.

What is WebGL?

- **General**: Low level 3D graphics API based on OpenGl.
- **Made By**: Khronos, developers of 3D graphics.
- **OpenGl**: OpenGL: API for rendering 2D and 3D vector graphics.
- **Showed by?**: The use of the HTML5 canvas element.
- **Work by?**: Using hardware acceleration to render the 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.

What is a shader?

Shader

Code written in GLSL that runs in your GPU

We need 2

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

GLSL

Literally OpenGL Shading Language

Fragment

And this paints each pixel with their corresponding color when rasterized

Vertex

Simply speaking, this creates the vertex in the space

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

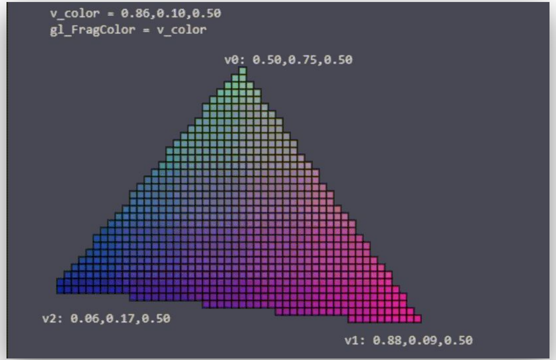
Generate Clip Space Coordinates for each vertex we wanna represent (spoiler: a lot)

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

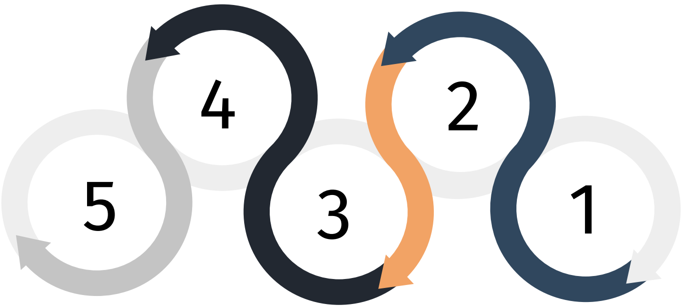
```
void main() {
  gl_FragColor = someMathTOGetColor
}
```



Objective

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

Why should I use WebGL?



Performance

WebGL is blindingly fast and fully utilizes hardware acceleration.

Support

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

OpenGl

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

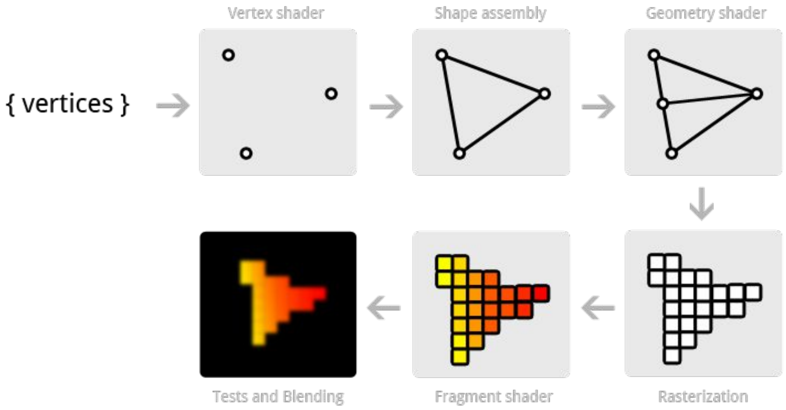
Tasks

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

Shaders

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

Why shouldn't I use WebGL?



WebGL limitations

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

Precision problems

WebGL uses 32 bit numbers. There are 3 precision settings: lowp, mediump and highp.

We cannot be precise in our drawings because of the lack of bits when looking for device compatibility.

It is written in a low level language, so it's so difficult to do even the simple things

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: Steps

1
Canvas
We Generate the canvas element in the HTML

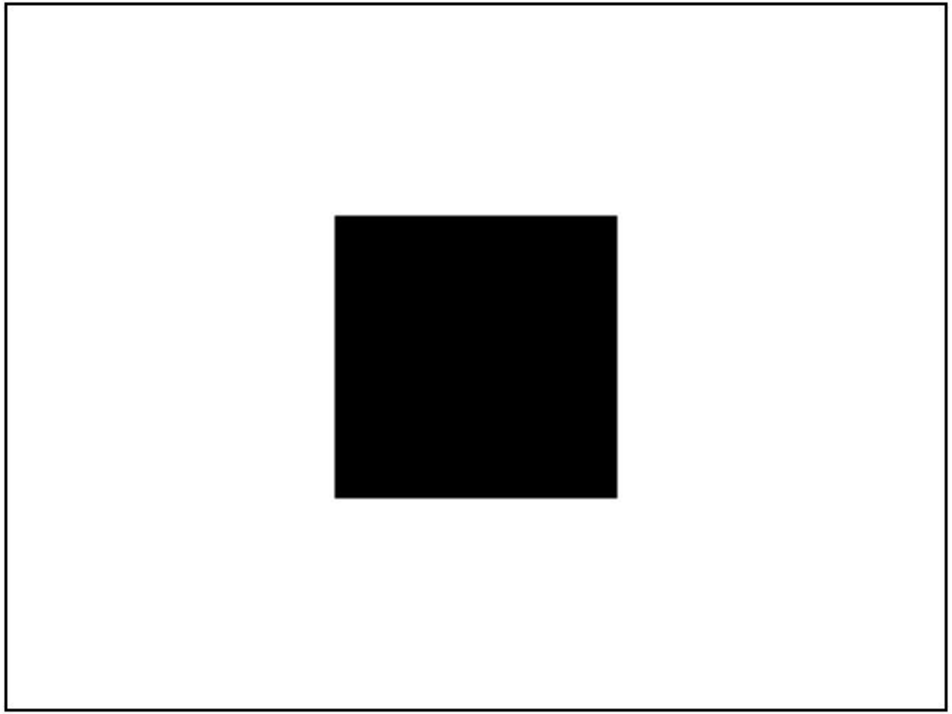
2
Context
We get the ‘webgl’ context from the canvas

3
Vertex
Now we create the vertex shader

4
Fragment
Create the fragment shader

5
Scene
Generate the scene using the tools we created

Result of a simple webGL application



Now, let's show you the process

Go to the VS Code Or Github. All the code is subdivided in different archives in the github repo.

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.

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

No need to install anything, we can use webGl just getting the 'webgl' context.

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.

gl_Position

The goal here is to set this attribute to what we want using mathematical equations.

Perspective

The objective of those two matrix is to perform change in the position to simulate perspective

Part 4: Creating the fragment shader

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

gl_FragColor

This is the attribute to store the color for each pixel in the canvas it's gonna receive when the image is rasterized. In our case it is opaque black.

Part 5: 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

To initialize the program, first we need to format and compile the shaders.

Initialize

We create what’s called a program. To this we attach each shader and link it back to the context.

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

First we create a new shader. 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.

Part 5.2: 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'),
    },
  };
```

Locations

Now we store the locations of all the attributes and variables we are gonna use for easy access

Part 6: Creating the shape buffer

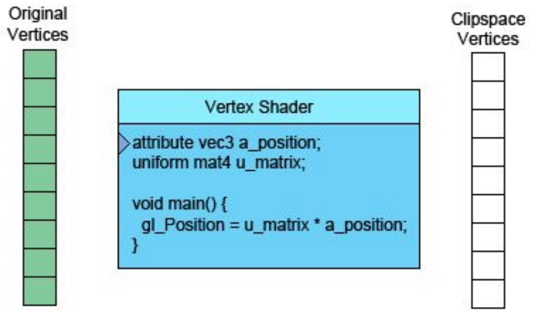
```
function initBuffers(gl) {

  // 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.



Part 7: 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);

  //...
```

Preparations

We establish things like the ‘clear color’ and features like making objects opaque.

Clear the space

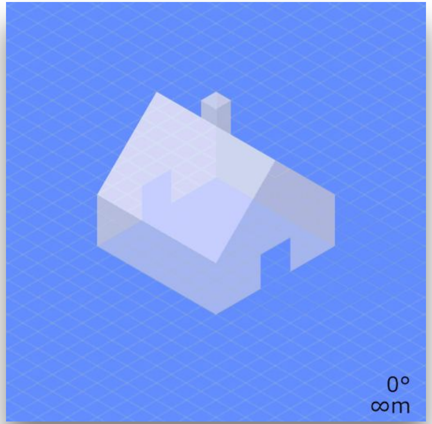
Just as a preventive step, we also clear the whole canvas to start drawing.

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.



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.

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.

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

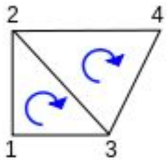
Now we tell WebGl to use the program we created in the initialization process, that includes the compiled shaders.

Set the Uniforms

Now we assign the vertex positions to the matrix created before

drawArrays

Now we call the array drawer function.



TRIANGLE_STRIP

Threejs

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.

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.

Motivation

Do not suffer with raw WebGl for small things.

When shouldn’t I use it

Full control of OpenGl is BETTER that any other option if you know how to use it.

Webgraphy

Code example

https://gpfault.net/posts/mandelbrot-webgl.txt.html

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

Repository

https://github.com/ULL-ESIT-PAI-2021-2022/2021-2022-pai-webgl-jorge-lucas.git