# Web GL

A JavaScript API for rendering high-performance interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins

Done by Francisco Parrinha

# What is WebGL?

Web GL is a JavaScript API for fast 2D and 3D rendering within web browsers

It is natively supported by **HTML5**, using the **<canvas>** element

```
<canvas className={`Engine-render-window ${this.state.css}`} id={thi
```

*Canvas DOM element, containing the WebGL's context instance, giving the ability to use the API*

# History

**OpenGL ES** was designed for embedded devices, such as mobile phones and video game consoles

**WebGL** was designed as a browser standard for 3D graphics without the use of plug-ins

# History

WebGL 1.0 was officially released in March 2011 using HTML5 <canvas> element

WebGL 2.0 was published In January 2017 based on OpenGL ES 3.0

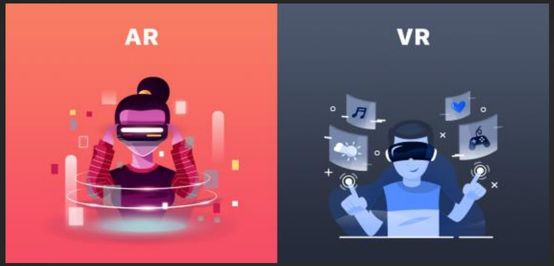
# Support

All browsers shown also support WebGL on their mobile versions

# Where is it applicable?

Video Games

Data Visualization




Virtual Reality
Augmented Reality

# Advantages

## Cross-Platform and Compatibility

Most operating systems support natively browsers that run WebGL

## Performance

WebGL uses the GPU's parallelization abilities to draw sophisticated graphics

## Community Support

WebGL has a strong development community, with libraries like **Three.js** and **Babylon.js**, allowing for an easier use of the API

# Shaders

A shader is a small GPU program written in a high-level shading language whose main purpose is to draw pixels on the screen

Examples of shading languages

**GLSL** from OpenGL

**HLSL** from DirectX

**Metal Shading** from Apple's Metal API

## WebGL uses **GLSL**

```


void main() {

    // Get colors
    v_color = u_color;
    v_textureCoord = a_textureCoord;

    // Get transformations
    mat3 translation = translation(t: u_position);
    mat3 rotation = rotation(angle: u_rotation);
    mat3 scale = scale(s: u_size);

    vec2 position = (translation * scale * rotatio
    gl_Position = vec4(v0: position, v1: 0.0, v2: 1.
}
```

*GLSL code example*

# Important Technical Concepts

**Shader Types**

Vertex used to manipulate and transform vertex data

Fragment used to draw the correct color for the final image

**Variable types**

Uniform used to pass data from the CPU to the GPU. Constant during a draw call

Attribute used to store data per vertex

Varying used to pass data from the Vertex Shader to the Fragment Shader

# Vertex Shader

Transforms and processes vertex data

Typically involves projections, translations, rotations and scaling on the vertices

Supports the following variables:

**Uniform**

**Attribute**

**Varying**

```
/** ATTRIBUTES */
in vec2 a_position;
in vec2 a_textureCoord;

/** VARYINGS */
out vec3 v_color;
out vec2 v_textureCoord;

/** UNIFORMS */
uniform vec3 u_color;
uniform vec2 u_size;
uniform vec2 u_position;
uniform float u_rotation;
uniform float u_zoom;


/** Returns the translation matrix to move the
mat3 translation(vec2 t) {
    return mat3(
      v0: 1, v1: 0, v2: 0,
      v3: 0, v4: 1, v5: 0,
      v6: t.x, v7: t.y, v8: 1);
}

/** Returns the rotation matrix to rotate accor
mat3 rotation(float angle) {
    float radians = radians(degrees: angle);
    float c = cos(angle: radians);
    float s = sin(angle: radians);
    return mat3(
      v0: c,v1: -s, v2: 0,
      v3: s, v4: c, v5: 0,
      v6: 0, v7: 0, v8: 1);
}

/** Returns the scaling matrix to scale the ent
mat3 scale(vec2 s) {
    return mat3(
      v0: s.x, v1: 0, v2: 0,
      v3: 0, v4: s.y, v5: 0,
      v6: 0, v7: 0, v8: 1);
}
```

*Vertex Shader example*

# Fragment Shader

On the fragment shader, the developer can choose how to color the pixels chosen by the rasterizer, using the primitives assembled by the **Vertex Shader**

Can also be useful for lightning and shadowing purposes and other special effects

Supports the following variables:

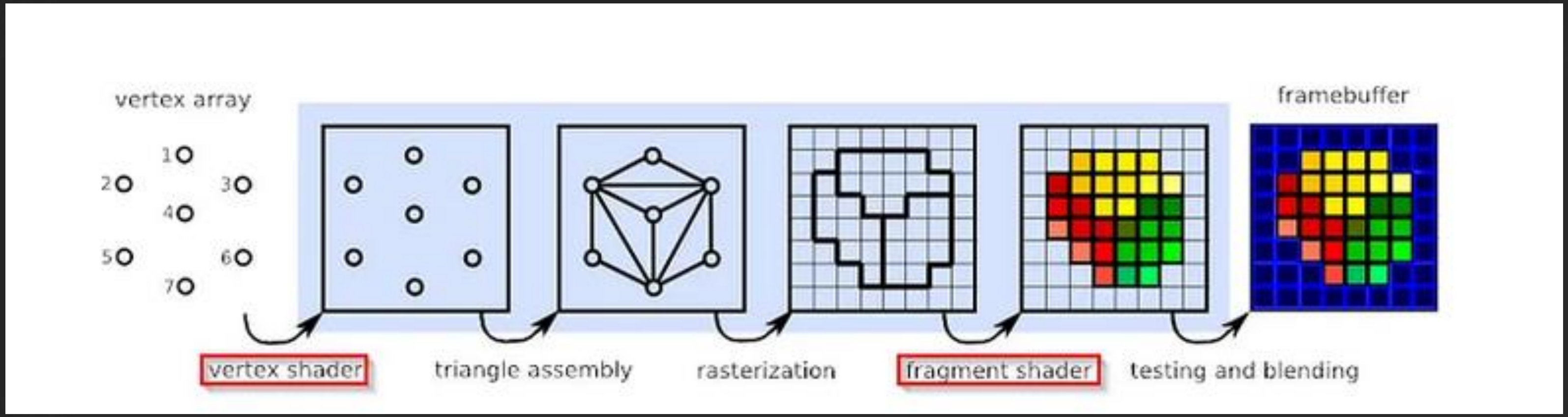
**Uniform**

**Varying**

```
void main() {

    // Get correct offsets in Sprite Space
    vec2 animPosition = getSpritePosition(spriteSize: u_sprit
    vec2 inSpriteCoordinates = toSpaceCoordinates(spaceSize:

    // Get correct UV postions for sprite animation
    vec2 uvAnimationOffset = toSpaceCoordinates(spaceSize: UV
    vec2 uvRescaler = toSpaceCoordinates(spaceSize: UV_SPACE,
    vec2 UVs = v_textureCoord * uvRescaler + uvAnimationOffs

    // Calculate final color with texture
    vec4 tex =  texture(sampler: u_textureSampler, P: UVs);
    vec3 finalColor = tex.rgb * v_color;
    fragColor = vec4(v0: finalColor, v1: tex.a);
}
```

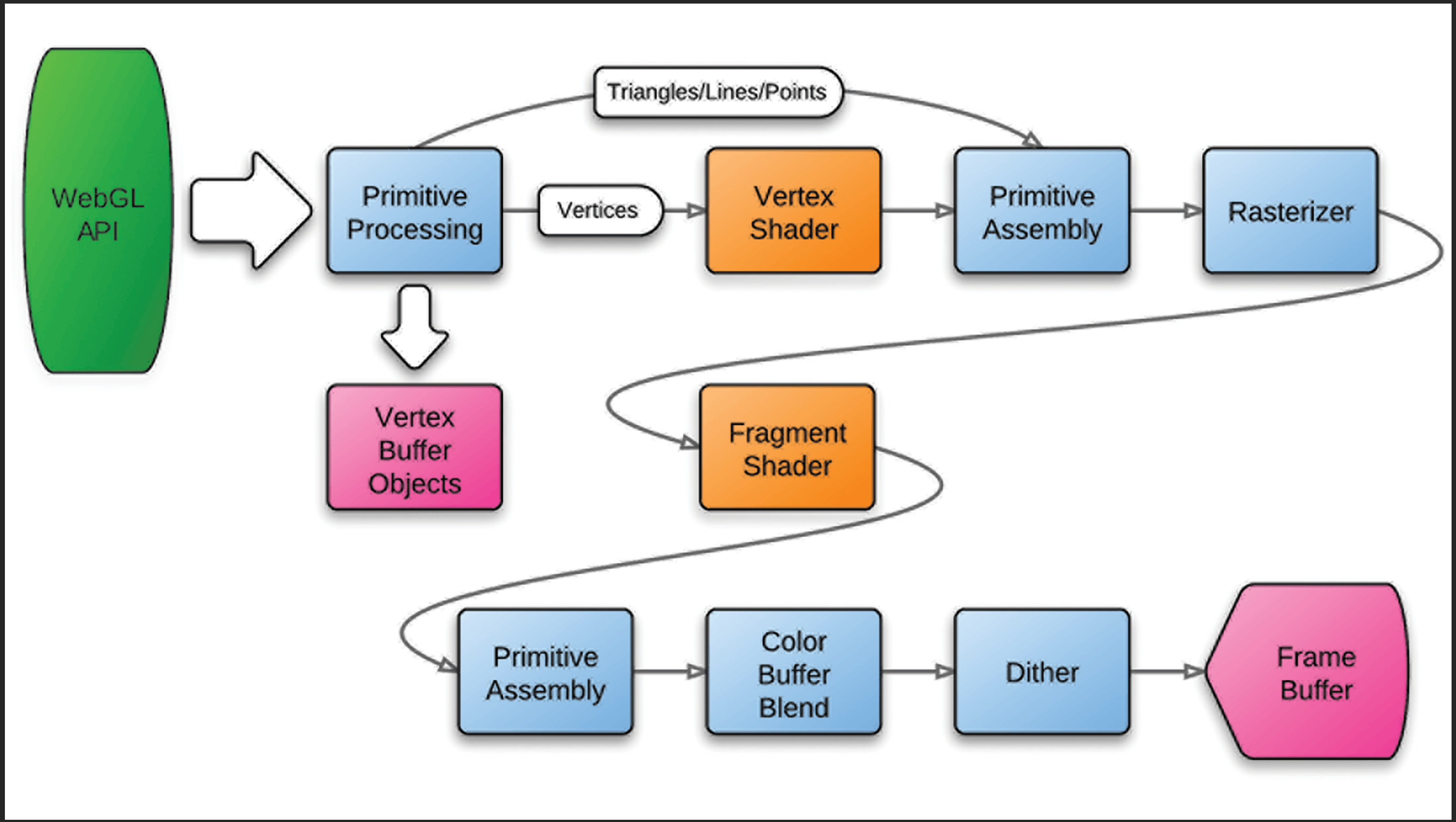
*Fragment Shader example*

# General Architecture




The rendering pipeline

# It is more complicated...



# Setting up the program...

The WebGL context is an object containing the API tools to create the GLSL program

The <canvas> element contains an instance of the object

```
 * @returns WebGL context
 */
findGlContext() {
    this.canvas = document.getElementById(`${this.id}`);

    // Check window's canvas exists
    if (isNull(this.canvas)) {
        alert("No canvas found in DOM level");
        return null;
    }

    // Setup main game window resizer
    window.addEventListener('resize', resizeGamewindow);
    resizeGamewindow();
    return this.canvas.getContext(WEBGL);
}
```

# Setting up the program...

After having the context, it is necessary to create the required shaders

```
    const glShader = gl.createShader(shaderType);

    gl.shaderSource(glShader, shader);
    gl.compileShader(glShader);
    if (!gl.getShaderParameter(glShader, gl.COMPILE_STATUS)) {
        const compileInfo = gl.getShaderInfoLog(glShader);
        console.error(compileInfo);
        return null;
    }

    return glShader;
}
```

With all shaders created, it's time to create the final program that links both **Vertex** and **Fragment** shaders together

```
 * @param {WebGL2RenderingContext} gl WebGL rendering context
 * @param {Shader} vertexShader vertex shader to attach
 * @param {Shader} fragmentShader fragment shader to attach
 * @returns shader program
 */
function createShaderProgram (gl, vertexShader, fragmentShader) {
    const shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);
    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        const compileInfo = gl.getProgramInfoLog(shaderProgram);
        console.error(compileInfo);
        return null;
    }

    return shaderProgram;
}
```

# Program is set

Having the basic initiation steps, the GLSL program is ready to accept new attributes to be able to render vertices on the screen

It is **necessary** to call **useProgram()** to the correct program before trying to pass new attributes

```
loadVertexData() {
    this.gl.bindBuffer(this.gl.ARRAY_BUFFER, this.vertexBuffer);
    this.gl.bufferData(this.gl.ARRAY_BUFFER, this.vertexBufferData, this.gl.STATIC_DRAW);

    //* Load attributes */
    this.loadAttribute(SpriteShaderAttributes.POSITION, 2);
}
```

```
loadAttribute(attribute, size) {
    const attributeLocation = this.gl.getAttribLocation(this.shaderProgram, attribute);
    if(attributeLocation < 0) {
        alert(`Error getting attribute ${attributeLocation}`);
        return;
    }


    this.gl.vertexAttribPointer(attributeLocation, size, this.gl.FLOAT, false, size * Float32Array.BYTES_PER_ELEMENT, 0);
    this.gl.enableVertexAttribArray(attributeLocation);
}
```

# Finally drawing

After having the shader program setup and all vertex attributes booted up, the developer can finally structure his own draw call

```
/**
 * Tells WebGL context to use the material's shader program
 *
 * WARNING: don't forget to load the uniform locations first
 *
 * PRE: all coordinates must be within clip space coordinates
 * @param {vec2} position position uniform value
 * @param {vec2} size size uniform value
 * @param {Number} rotation rotation uniform value
 * @param {vec3} color color uniform value
 * @param {Sprite} sprite texture to upload to the shader //TODO Make texture binding happen on material boot, not on the draw call
 */
drawCall(position, size, rotation, color, sprite) {
    this.gl.useProgram(this.shaderProgram);

    // Change texture data according to texture given
    this.bindTextureFile(sprite.getTexture());
    sprite.tick();

    // Upload uniforms
    this.gl.uniform2fv(this.uniformLocations.position, position.vec);
    this.gl.uniform2fv(this.uniformLocations.size, size.vec);
    this.gl.uniform1f(this.uniformLocations.rotation, rotation);
    this.gl.uniform3fv(this.uniformLocations.color, color.vec);
    this.gl.uniform2fv(this.uniformLocations.spriteSize, sprite.getSize());
    this.gl.uniform1f(this.uniformLocations.spriteCounter, sprite.getCounter());

    // Final draw
    this.gl.drawArrays(this.gl.TRIANGLES, 0, ShaderVertexCount.QUAD);
}
```

# Important Resources

**WebGL Fundamentals** Website

https://webglfundamentals.org/

**Khronos Group** Website

https://www.khronos.org/webgl/

# End

Presentation done by Francisco Parrinha