Universidad de La Laguna

WebGL

Programación de Aplicaciones Interactivas

Cristo Daniel Navarro Rodríguez
Luciana Varela Díaz

INDEX

INTRODUCTION

- Web Graphics Library
- Javascript API
- Runs on your GPU (Graphics Processing Unit)

INTRODUCTION

- Primitive figures:
  - Points
  - Lines
  - Triangles

Canvas VS WebGL

- Platforms
- Learning rate
- Capabilities
- Applications
- Performance

Canvas VS WebGL

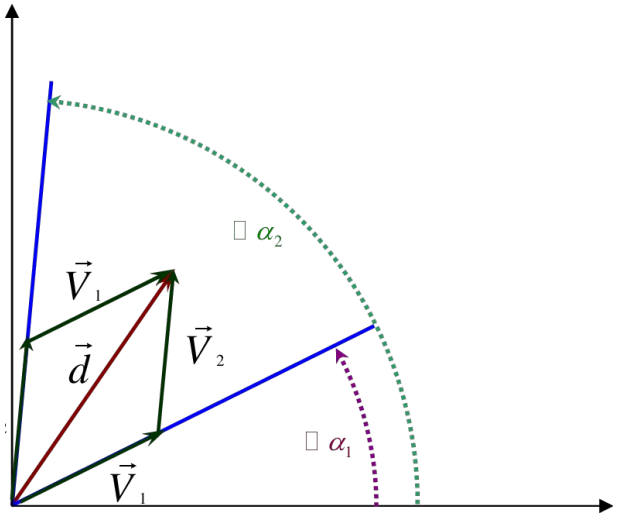
Platforms:

- Canvas: browsers
- WebGL: browsers + mobile

Canvas VS WebGL

Learning rate:

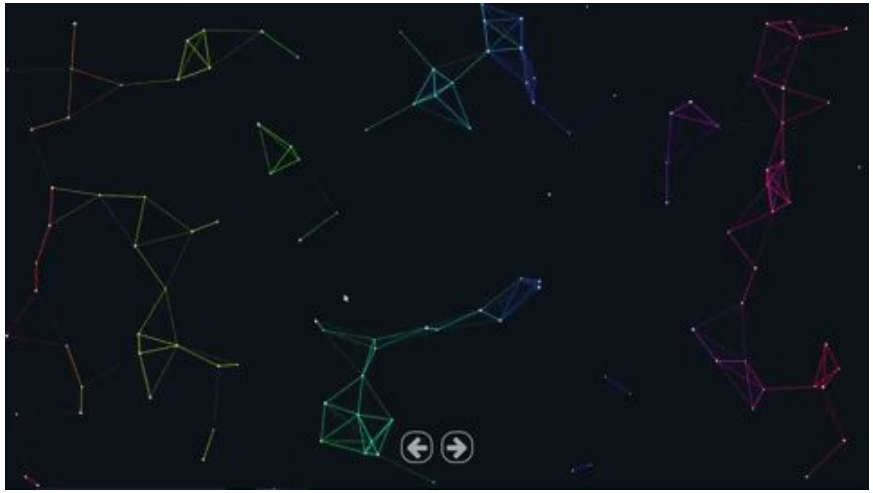
- Canvas: way easier
- WebGL: complicated

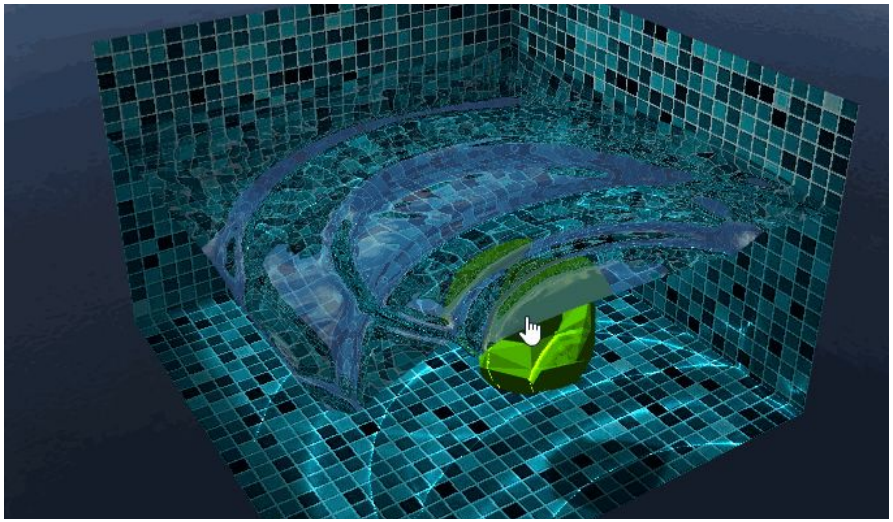


Canvas VS WebGL

Applications:

- Canvas: 2D
- WebGL: 3D

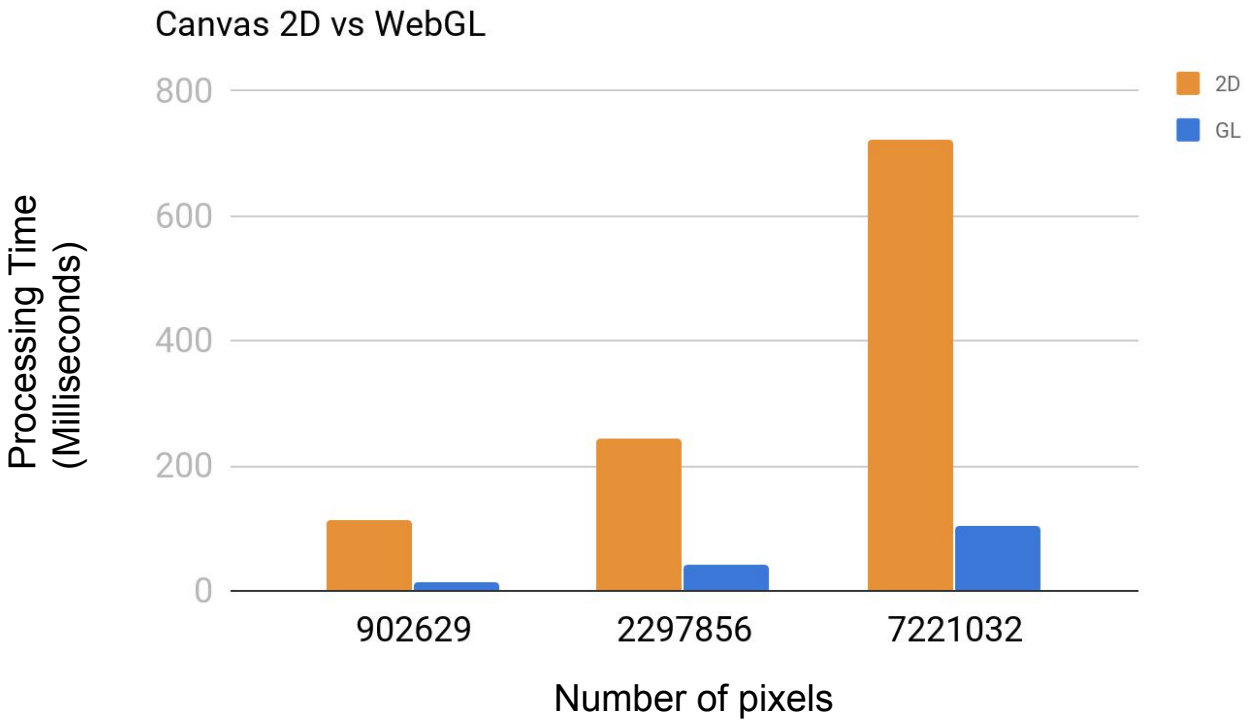




Canvas VS WebGL

Performance

- Rendering on GPU.



How does it work?

- Prepare the canvas and the rendering context.
- Define geometry and store it.
- Create and compile shader programs.
- Associate the shaders with buffer objects.
- Draw the objects.

How does it work?

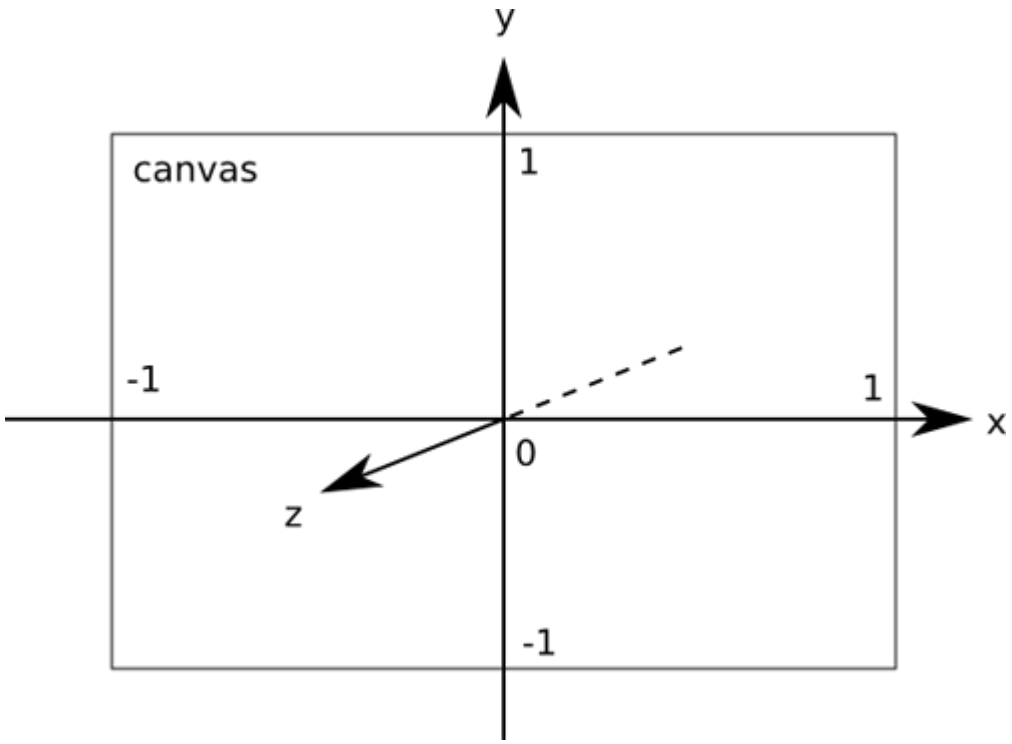
Prepare the canvas and the rendering context.

```
const CANVAS =
document.getElementById('canvas');
const CONTEXT = CANVAS.getContext('webgl');
```

How does it work?

Define geometry and store it.

- Clipspace



How does it work?

Define geometry and store it.

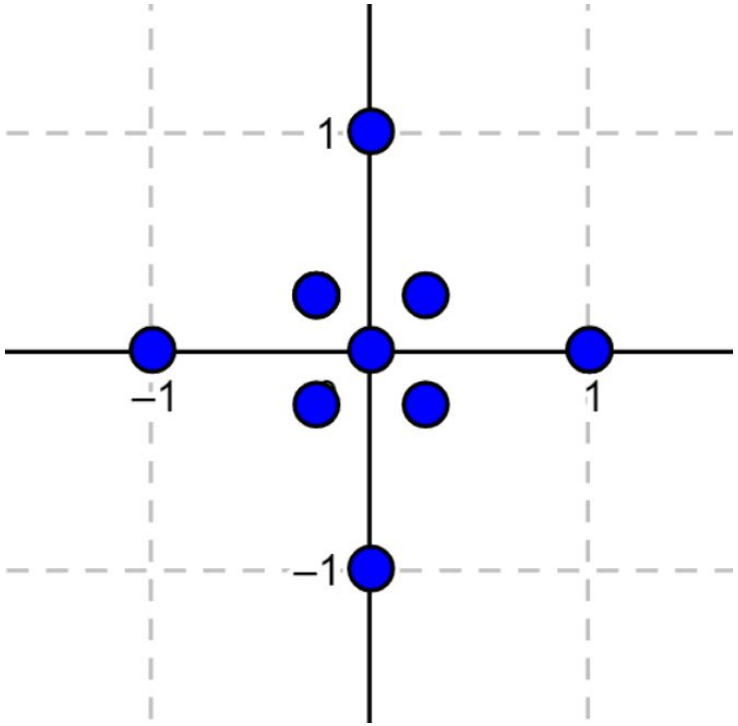
- Positions

```
let vertex = [-1.0, 0.0,   -0.25, 0.25,    0.0, 0.0,
               0.0, 0.0,   -0.25, 0.25,    0.0, 1.0,
               0.0, 1.0,    0.25, 0.25,    0.0, 0.0,
               0.0, 0.0,    0.25, 0.25,    1.0, 0.0,
               0.0, 0.0,    1.0, 0.0,      0.25, -0.25,
               0.0, 0.0,    0.25, -0.25,   0.0, -1.0,
               0.0, 0.0,    0.0, -1.0,    -0.25, -0.25,
               0.0, 0.0,   -0.25, -0.25,  -1.0, 0.0];
```

How does it work?

Define geometry and store it.

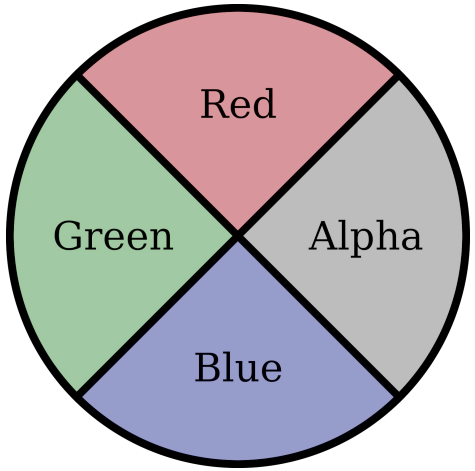
- Positions



How does it work?

Define geometry and store it.

- Colors




```
let triangleColors = [[1.0, 0.0, 1.0, 1.0],
                      [1.0, 0.0, 0.0, 1.0],
                      [0.0, 1.0, 0.0, 1.0],
                      [0.0, 0.0, 1.0, 1.0],
                      [0.0, 0.0, 0.0, 1.0],
                      [1.0, 1.0, 0.0, 1.0],
                      [0.0, 1.0, 1.0, 1.0],
                      [1.0, 0.5, 0.5, 1.0]];
```

How does it work?

Define geometry and store it.

- Colors

```
let colors = [];
for (let index = 0; index < triangleColors.length;
  index++) {
  const CURRENT_COLOR = triangleColors[index];
  colors = colors.concat(CURRENT_COLOR, CURRENT_COLOR,
    CURRENT_COLOR);
}
```

How does it work?

Define geometry and store it.

- Buffers
  - Contiguous block of memory
  - Data for attributes

How does it work?

Define geometry and store it.

- Buffers

```
// Create a new buffer object
const VERTEX_BUFFER = context.createBuffer();  // New ID
// Bind an empty array buffer to it
context.bindBuffer(context.ARRAY_BUFFER, VERTEX_BUFFER);  // Active
// Pass the vertices data to the buffer
context.bufferData(context.ARRAY_BUFFER, new Float32Array(vertex),
  context.STATIC_DRAW);
// Unbind the buffer
context.bindBuffer(context.ARRAY_BUFFER, null);  // Inactive
```

How does it work?

Create and compile shader programs.

- Shader source code
  - Vertex shader
  - Fragment shader
- Uses the language GLSL

How does it work?

Create and compile shader programs.

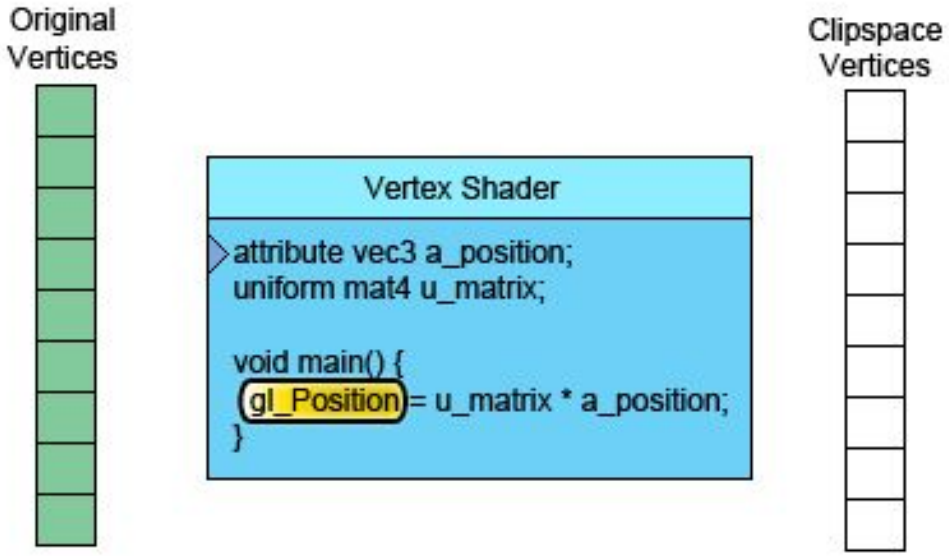
- Vertex shader
  - Generate the clipspace coordinates
  - Called once per vertex
  - Assigns the coordinates to “gl_Position”

Always vec4!!!

How does it work?

Create and compile shader programs.

- Vertex shader



How does it work?

Create and compile shader programs.

- Vertex shader

```
const VERTEX_CODE = `
  attribute vec2 coordinates;
  attribute vec4 aVertexColor;
  varying lowp vec4 vColor;
  void main(void) {
    gl_Position = vec4(coordinates, 0.0, 1.0);
    vColor = aVertexColor;
  }`;
```

How does it work?

Create and compile shader programs.

- Vertex shader

```
//Create a vertex shader object
const VERTEX_SHADER =
context.createShader(context.VERTEX_SHADER);
//Attach vertex shader source code
context.shaderSource(VERTEX_SHADER, VERTEX_CODE);
//Compile the vertex shader
context.compileShader(VERTEX_SHADER);
```

How does it work?

Create and compile shader programs.

- Fragment shader
  - Assigns the color to the current pixel
  - Called once per pixel
  - Assigns the color to “gl_FragColor”

How does it work?

Create and compile shader programs.

- Fragment shader
  - Precision
    - lowp
    - mediump
    - highp

How does it work?

Create and compile shader programs.

- Fragment shader

```
const FRAG_CODE = `
    varying lowp vec4 vColor;
    void main(void) {
      gl_FragColor = vColor;
    }`;
// Create fragment shader object
const FRAG_SHADER = context.createShader(context.FRAGMENT_SHADER);
// Attach fragment shader source code
context.shaderSource(FRAG_SHADER, FRAG_CODE);
// Compile the fragment shader
context.compileShader(FRAG_SHADER);
```

How does it work?

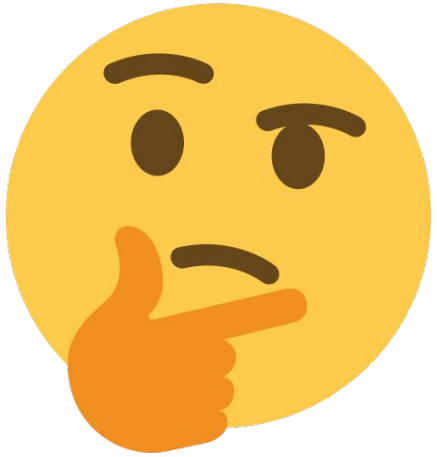
Create and compile shader programs.

```
// Create a shader program object to store combined shader
program
const SHADER_PROGRAM = context.createProgram();
// Attach a vertex shader
context.attachShader(SHADER_PROGRAM, VERTEX_SHADER);
// Attach a fragment shader
context.attachShader(SHADER_PROGRAM, FRAG_SHADER);
// Link both programs
context.linkProgram(SHADER_PROGRAM);
// Use the combined shader program object
context.useProgram(SHADER_PROGRAM);
```

How does it work?

Associate the shaders with buffer objects.

- How do we pull the data from the buffer?



How does it work?

Associate the shaders with buffer objects.

- We tell WebGL where to put the data:

```
context.getAttribLocation(SHADER_PROGRAM,
  'coordinates');


context.getAttribLocation(SHADER_PROGRAM,
  'aVertexColor');
```

How does it work?

Associate the shaders with buffer objects.

- We tell WebGL where to put the data:

```
const VERTEX_CODE = `
    attribute vec2 coordinates;
    attribute vec4 aVertexColor;
    varying lowp vec4 vColor;
    void main(void) {
        gl_Position = vec4(coordinates, 0.0, 1.0);
        vColor = aVertexColor;
    }`;
```

How does it work?

Associate the shaders with buffer objects.

- We tell WebGL how to do it:

```
const NUM_COMPONENTS = 2;     // Pull out per iteration
const TYPE = gl.FLOAT;        // Data type
const NORMALIZE = false;      // Needed to normalize?
const STRIDE = 0;             // Bytes between elements
const OFFSET = 0;             // Bytes to start
```

How does it work?

Associate the shaders with buffer objects.

- We tell WebGL where to put the data:

```
// Assigns the vertex attributes for the shader program
context.vertexAttribPointer(
    COORD,
    NUM_COMPONENTS,
    TYPE,
    NORMALIZE,
    STRIDE,
    OFFSET);

context.enableVertexAttribArray(COORD);
```

How does it work?

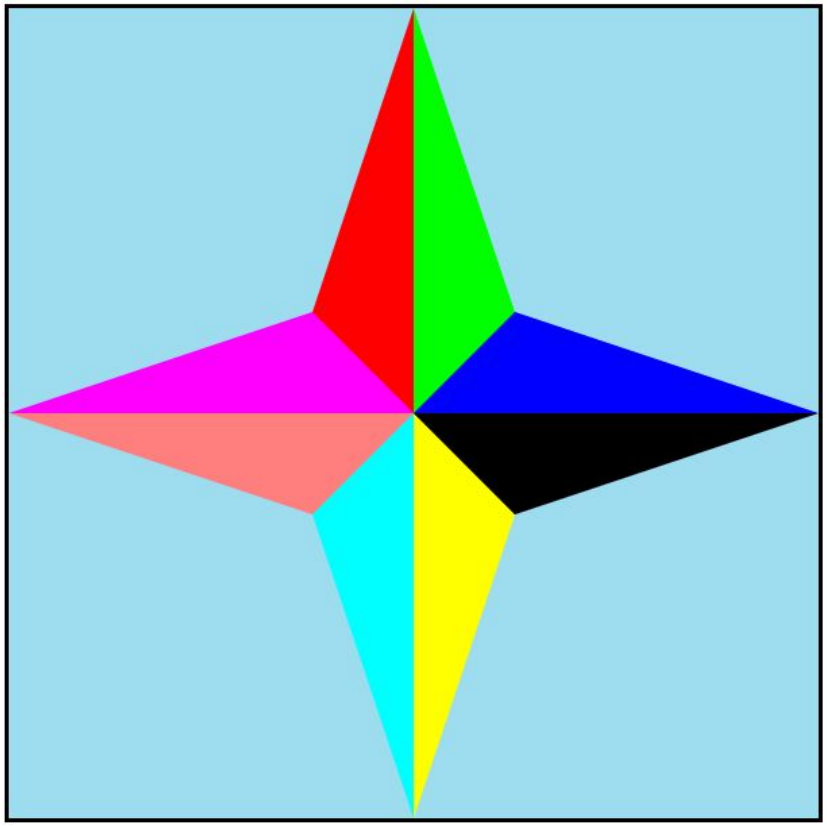
Draw the objects.

- drawArrays
- drawElements

```
const MODE = CONTEXT.TRIANGLE_STRIP // Figure we
are going to draw
const FIRST = 0; // Starting index of the array of
vertex
const COUNT = 24; // Number of vertex to be
rendered
// Draw the figure
context.drawArrays(MODE, FIRST, COUNT);
```

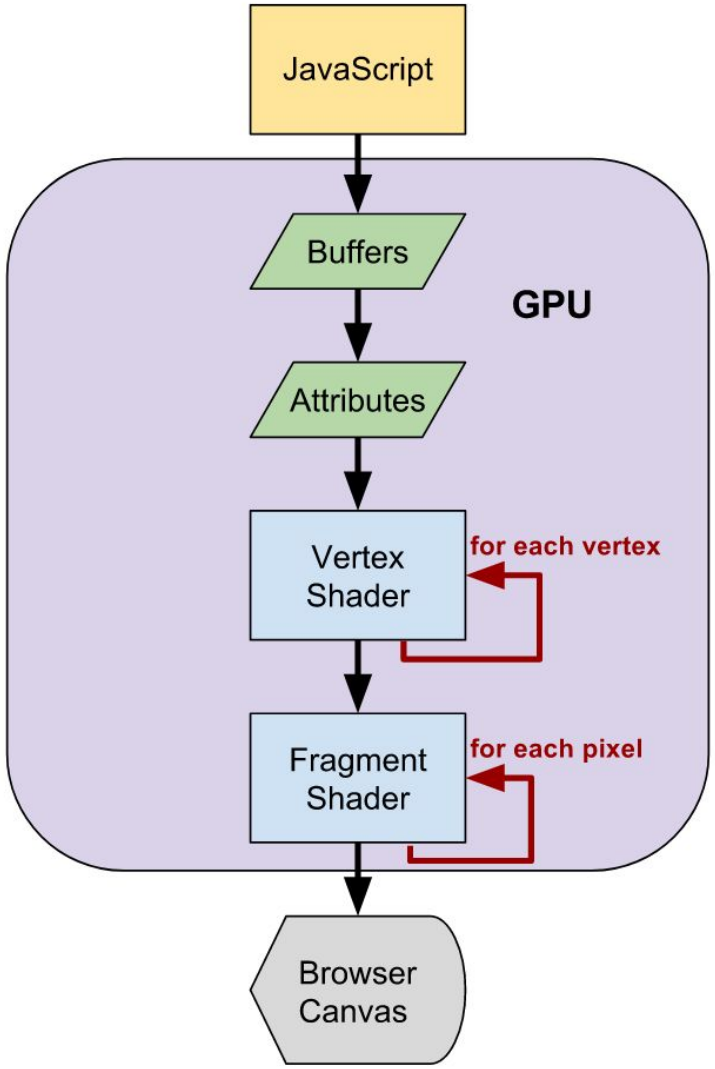
How does it work?

Final result.



How does it work?

Summary



Basic functions and Qualifiers

- Qualifiers:
  - **const**: constants during the execution
  - **attribute**: global variable that is constant in each vertex

Basic functions and Qualifiers

- Qualifiers:
  - **uniform**: global variable that is constant in each primitive
  - **varying**: share information between the vertex shader and the fragment shader

Basic functions and Qualifiers

- Buffer Objects
- Programs and shaders
- Uniforms and attributes
- Writing to the Draw Buffer
- Qualifiers

Animations

In order to perform an animation:

- Track the movement.
- Update our shaders.
- Draw the current movement.
- Update our movement.

Animations

Shaders

- We need to tell Webgl how to rotate the figure.
- Add a new variable to our shader to perform the rotation.

Animations

Shaders

```
const VERTEX_CODE = `
attribute vec2 coordinates;
attribute vec4 aVertexColor;
varying lowp vec4 vColor;
uniform mat4 uModelViewMatrix;  ← New
void main(void) {
  gl_Position = vec4(coordinates, 0.0, 1.0)
* uModelViewMatrix;
  vColor = aVertexColor;
}`;
```

Animations

Shaders

- Associate shaders to our buffer.
- Rotate the temporal matrix and assign the transformation again.

```
coord = CONTEXT.getUniformLocation(SHADER_PROGRAM,
'uModelViewMatrix');
const modelViewMatrix = mat4.create();
mat4.rotate(modelViewMatrix, modelViewMatrix, rotation,
[0, 0, 1]); ← How we are going to rotate our figure
CONTEXT.uniformMatrix4fv(coord, false, modelViewMatrix);
```

Animations

Shaders

- Transformations

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

Rotate X Matrix

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

Rotate Y Matrix

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

Rotate Z Matrix

Animations

Updating rotation

- Variable that tracks rotation.

```
let rotation = 0.0;
```

- Each time we draw the figure, we update the rotation and the shaders.

Animations

Updating rotation

- Each time we draw, we update the rotation.

```
function render(currentTime) {
  ...
  draw();
  rotation += rotationIncrease;
}
```

Animations

Rendering

- Using the current time to know how far to move the figure.
- **requestAnimationFrame** passes the number of milliseconds since the last frame was rendered to it's callback.

Animations

Rendering

```
function render(currentTime) {
  associateShaders(createBuffer(), createShaders());
  currentTime *= 0.001;
  let rotationIncrease = currentTime - previousTime;
  previousTime = currentTime;
  draw();
  rotation += rotationIncrease;
  requestAnimationFrame(render);
}
```

Animations

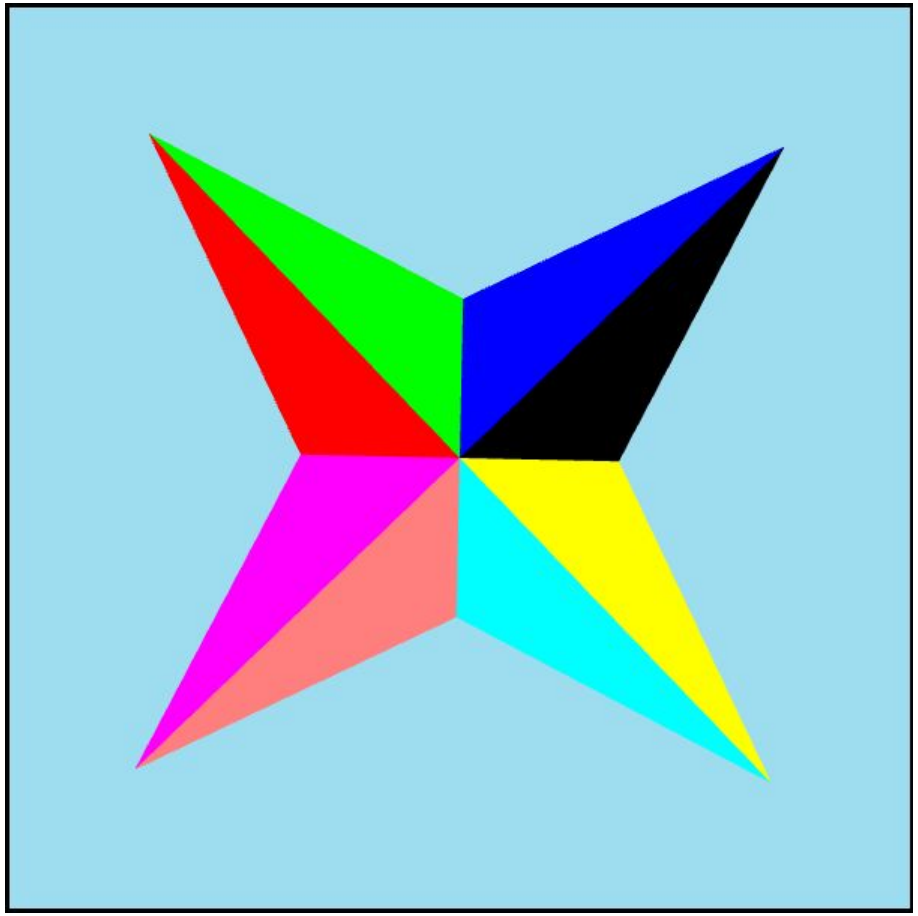
Starting the animation

- We use the **requestAnimationFrame** function to call our rendering function.

```
function main() {
  ...
  requestAnimationFrame(render);
}
```

Animations

Final result



Creating 3D figures

- Add new coordinates.
- Set the colors of the new elements.
- Update drawing parameters:
  - Vertex shader
  - Buffer access
  - Number of vertex to draw

Creating 3D figures

- Adding new coordinates

```
let vertex = [-1.0, 0.0, 0.0,
              -0.25, 0.25, 0.0,
               0.0, 0.0, 0.25,
                    .
                    .
                    .
               -1.0, 0.0, 0.0,
               -0.25, 0.25, 0.0,
                0.0, 0.0, -0.25,
                    .
                    .
                    .
               ];
```


Opposite vertex

Creating 3D figures

- Set the colors of the new elements.

```
let triangleColors = [[1.0, 0.0, 1.0, 1.0],
                      ...
                      [1.0, 0.0, 1.0, 0.8]];
```

Creating 3D figures

- Update vertex shader.

```
let VERTEX_CODE = `
  attribute vec3 coordinates;
  attribute vec4 aVertexColor;
  uniform mat4 uModelViewMatrix;
  varying lowp vec4 vColor;
  void main(void) {
    gl_Position = vec4(coordinates, 1.0) * uModelViewMatrix;
    vColor = aVertexColor;
  }`;
```

Creating 3D figures

- Update buffer access.

```
const NUM_COMPONENTS = 3;     // Pull out per iteration
const TYPE = gl.FLOAT;        // Data type
const NORMALIZE = false;      // Needed to normalize?
const STRIDE = 0;             // Bytes between elements
const OFFSET = 0;             // Bytes to start
```

Creating 3D figures

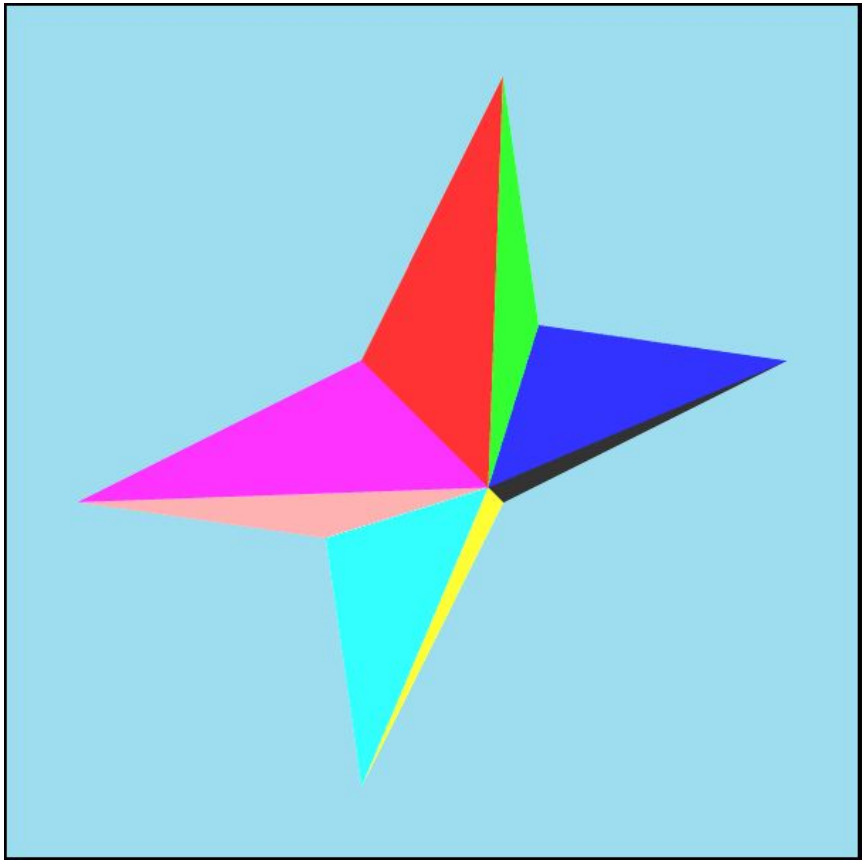
- Update the number of vertex to draw.

```
const MODE = CONTEXT.TRIANGLE_STRIP // Figure we are going to draw
const FIRST = 0; // Starting index of the array of vertex
const COUNT = 48; // Number of vertex to be rendered

// Draw the figure
context.drawArrays(MODE, FIRST, COUNT);
```

Creating 3D figures

Final result



Summary

- Canvas VS WebGL
- Geometry and colors in WebGL
- Shader programs
- Create a simple animation
- Create a simple 3D figure

If this is not enough...

- Apply textures to the figures
- Apply lighting
- User interface
- three.js

Impressive WebGL works

- Jellyfish
- Videogames
  - Quake
  - WebGL Games
- Reflektor - Arcade Fire

Bibliografía

- https://www.toptal.com/javascript/3d-graphics-a-webgl-tutorial
- https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial
- https://webglfundamentals.org/
- https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf
- https://www.tutorialspoint.com/webgl/webgl_sample_application.htm
- http://glmatrix.net/docs/mat4.js.html
- https://www.tutorialspoint.com/webgl/webgl_shaders.htm
- https://en.wikipedia.org/wiki/Triangle_strip

Any questions?

Cristo Daniel Navarro Rodríguez
(alu0101024608@ull.edu.es)
Luciana Varela Díaz
(alu0101106175@ull.edu.es)


Universidad
de La Laguna