

# Auxiliar N°5

## Shaders

---

Auxiliar: Pablo Pizarro R. [@ppizarro](#)

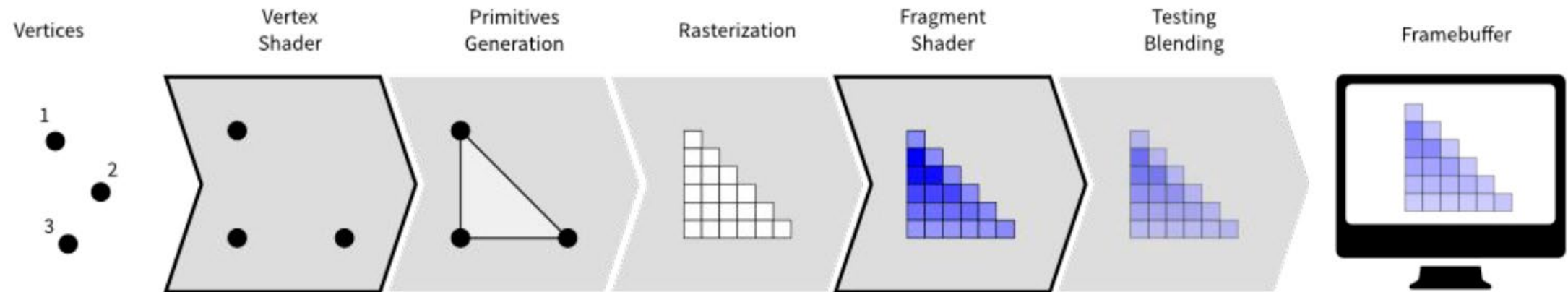
# Introducción

- Shaders
- Ejemplo de shader en Three.js
  - Introducción a Three.js
  - Esquema de código
  - Ejemplo práctico (visualizador fractales)

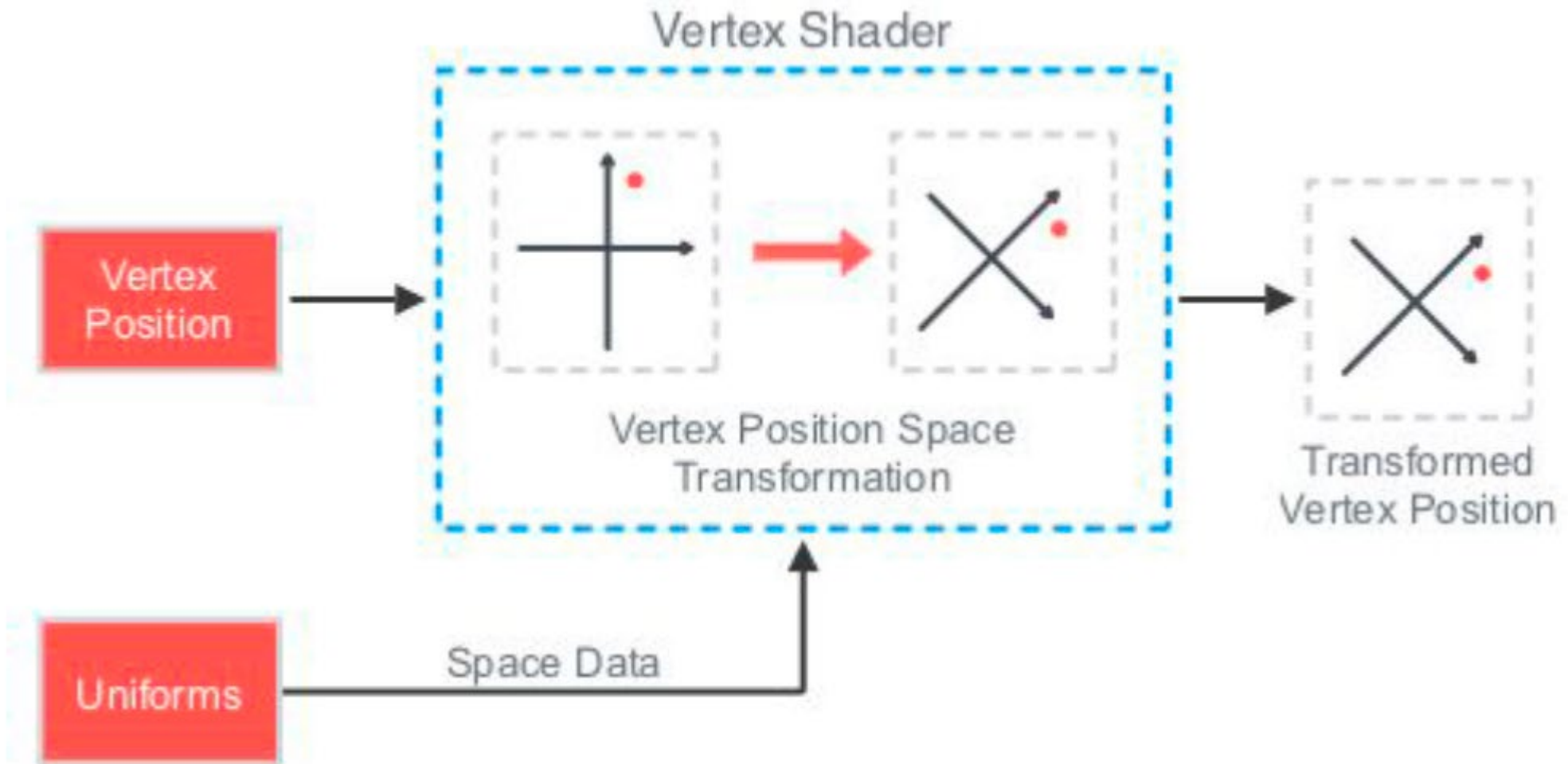
# Shaders



# ¿Qué vimos la clase pasada?



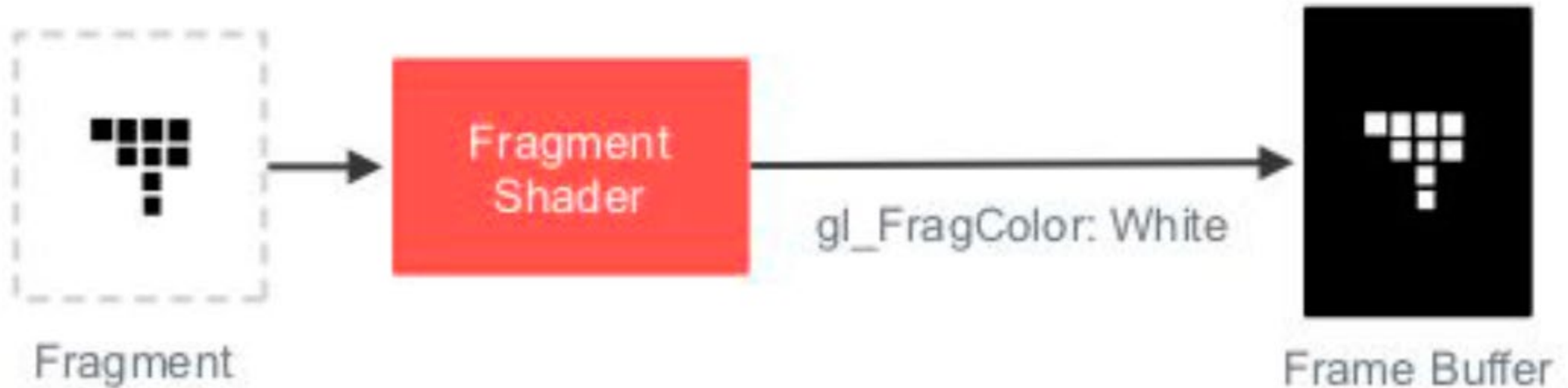
# ¿Qué vimos la clase pasada?



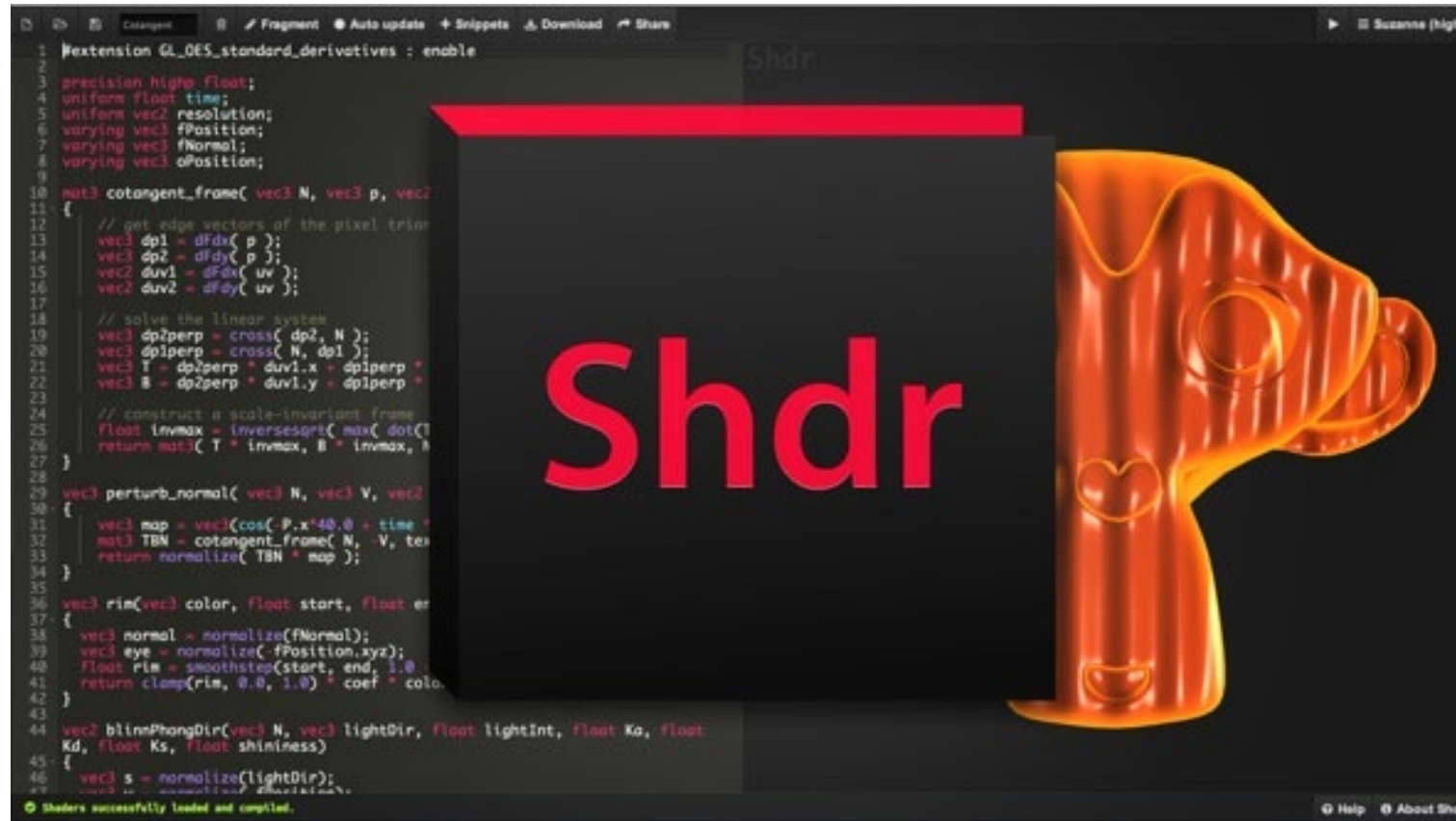


# ¿Qué vimos la clase pasada?

## Per-Fragment Stage

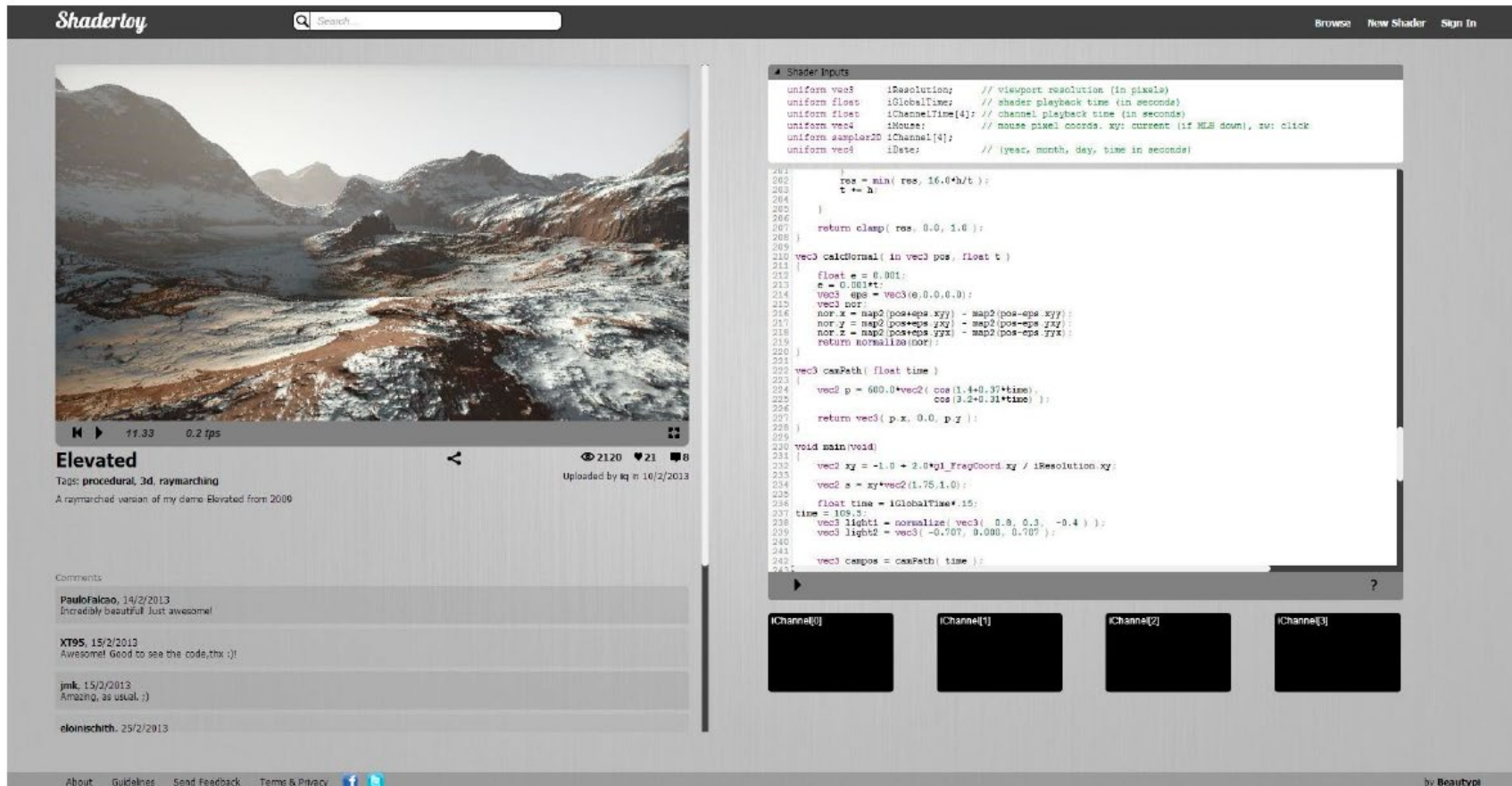


# Herramientas online



<https://shdr.bkcore.com>

# Herramientas online



The screenshot displays the ShaderToy website interface. On the left, a video player shows a procedural landscape with mountains and a body of water. Below the video, the title "Elevated" is displayed, along with tags "procedural, 3d, raymarching" and a description "A raymarched version of my dme Elevated from 2006". The video has 2120 views, 21 likes, and 6 comments. Comments from users like PauloFalcão, XT95, jmk, and eloisichth are visible.

On the right, the "Shader Inputs" section shows the GLSL code for the shader. The code includes uniform variables for resolution, global time, channel time, mouse, and date. The main function calculates a position vector and a color vector based on these inputs.

```
Shader Inputs
uniform vec3 iResolution; // viewport resolution (in pixels)
uniform float iGlobalTime; // shader playback time (in seconds)
uniform float iChannelTime[4]; // channel playback time (in seconds)
uniform vec4 iMouse; // mouse pixel coords. xy: current (if WEBGL down), zw: click
uniform sampler2D iChannel[4];
uniform vec3 iDate; // (year, month, day, time in seconds)

261
262   res = min( res, 16.0*h/t );
263   t += h;
264 }
265
266 return clamp( res, 0.0, 1.0 );
267
268
269
270 vec3 calcNormal( in vec3 pos, float t )
271 {
272   float e = 0.001;
273   s = 0.001*t;
274   vec3 eps = vec3(e,0.0,0.0);
275   vec3 nor;
276   nor.x = map2(pos+eps.xy) - map2(pos-eps.xy);
277   nor.y = map2(pos+eps.yx) - map2(pos-eps.yx);
278   nor.z = map2(pos+eps.zx) - map2(pos-eps.zx);
279   return normalize(nor);
280 }
281
282 vec3 cspPath( float time )
283 {
284   vec2 p = 600.0*vec2( cos(1.4+0.37*time),
285                       cos(3.2+0.31*time) );
286   return vec3( p.x, 0.0, p.y );
287 }
288
289
290 void main(void)
291 {
292   vec2 xy = -1.0 + 2.0*gl_FragCoord.xy / iResolution.xy;
293   vec2 s = xy*vec2(1.75,1.0);
294   float time = iGlobalTime*.15;
295   time = 109.5;
296   vec3 light1 = normalize( vec3( 0.8, 0.3, -0.4 ) );
297   vec3 light2 = vec3( -0.707, 0.000, 0.707 );
298
299   vec3 cspos = cspPath( time );
300 }
```

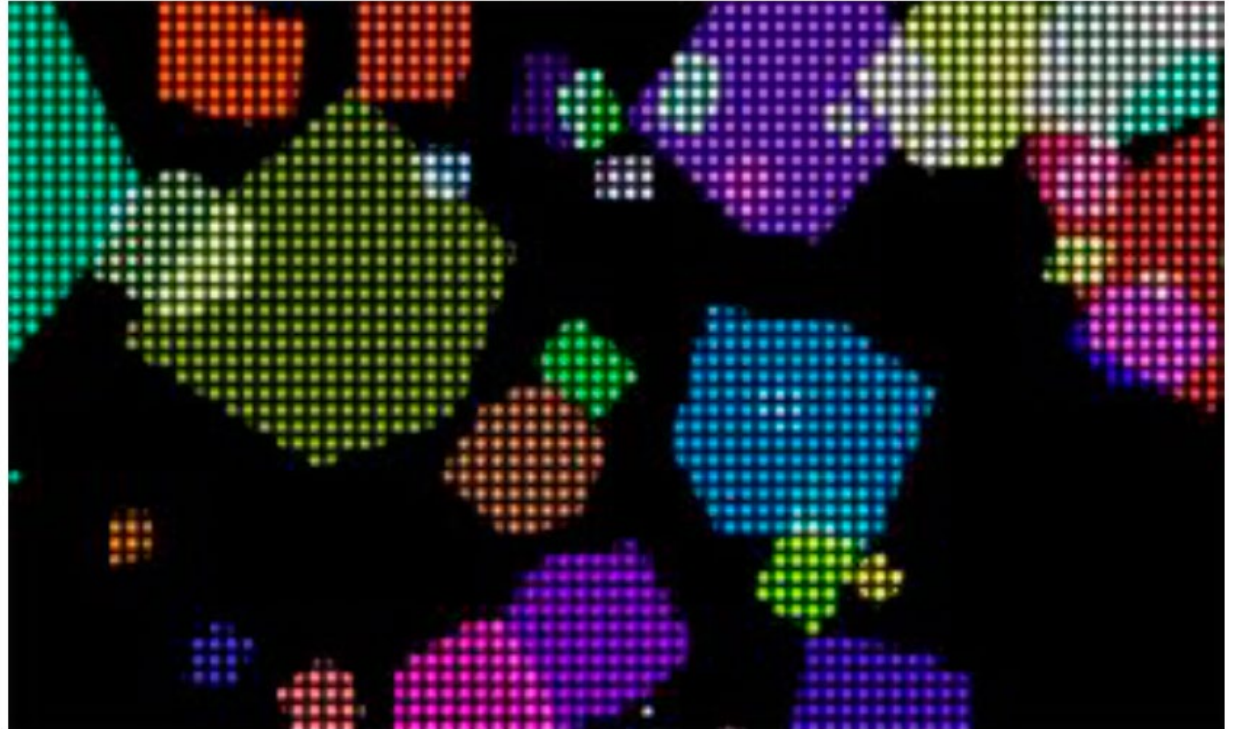
Below the code, there are four black boxes labeled "iChannel[0]", "iChannel[1]", "iChannel[2]", and "iChannel[3]".

<https://www.shadertoy.com>



# ¿Cómo funcionan estas herramientas?

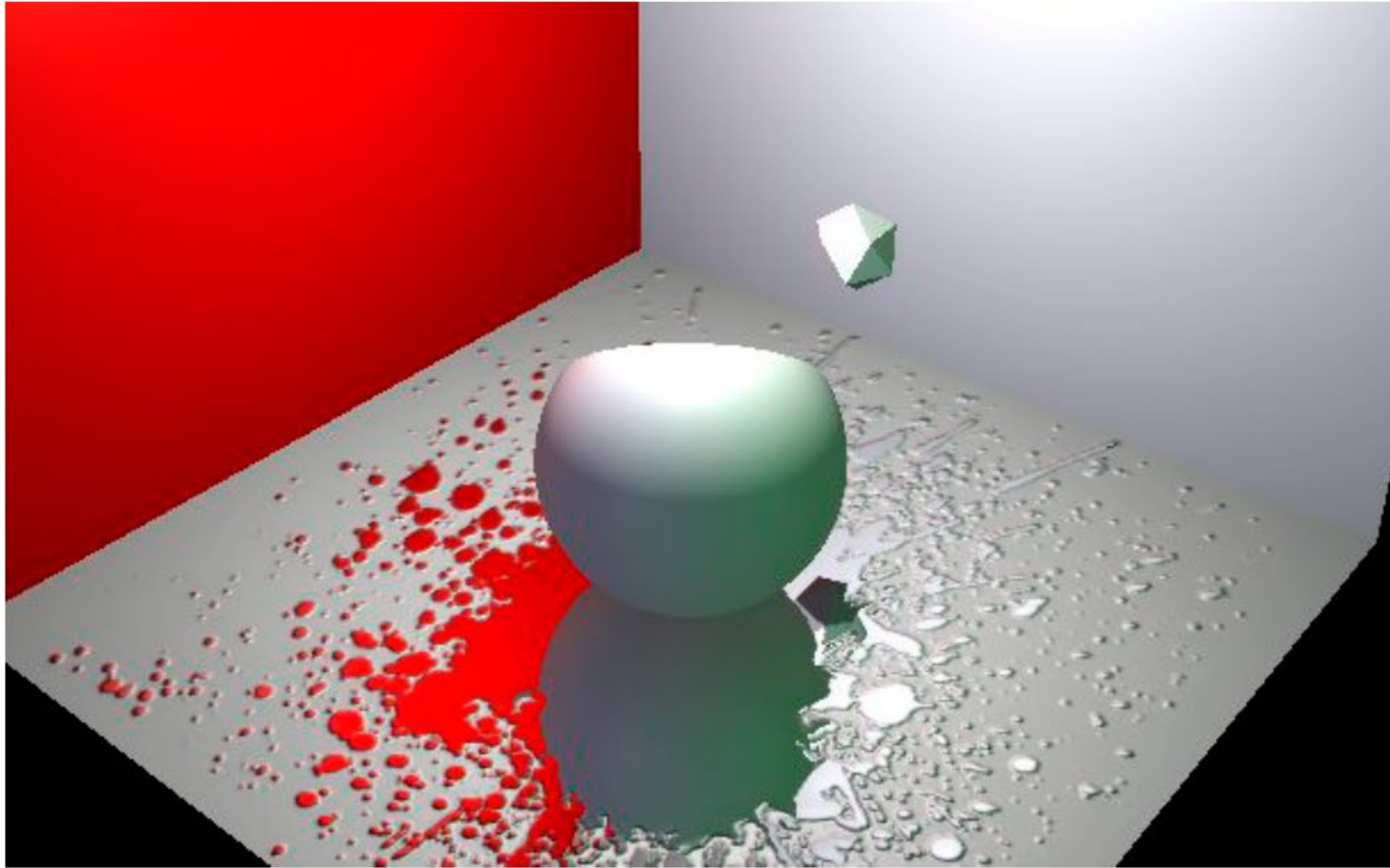
- WebGL
- Pixel Shader



# ¿Cómo funcionan estas herramientas?



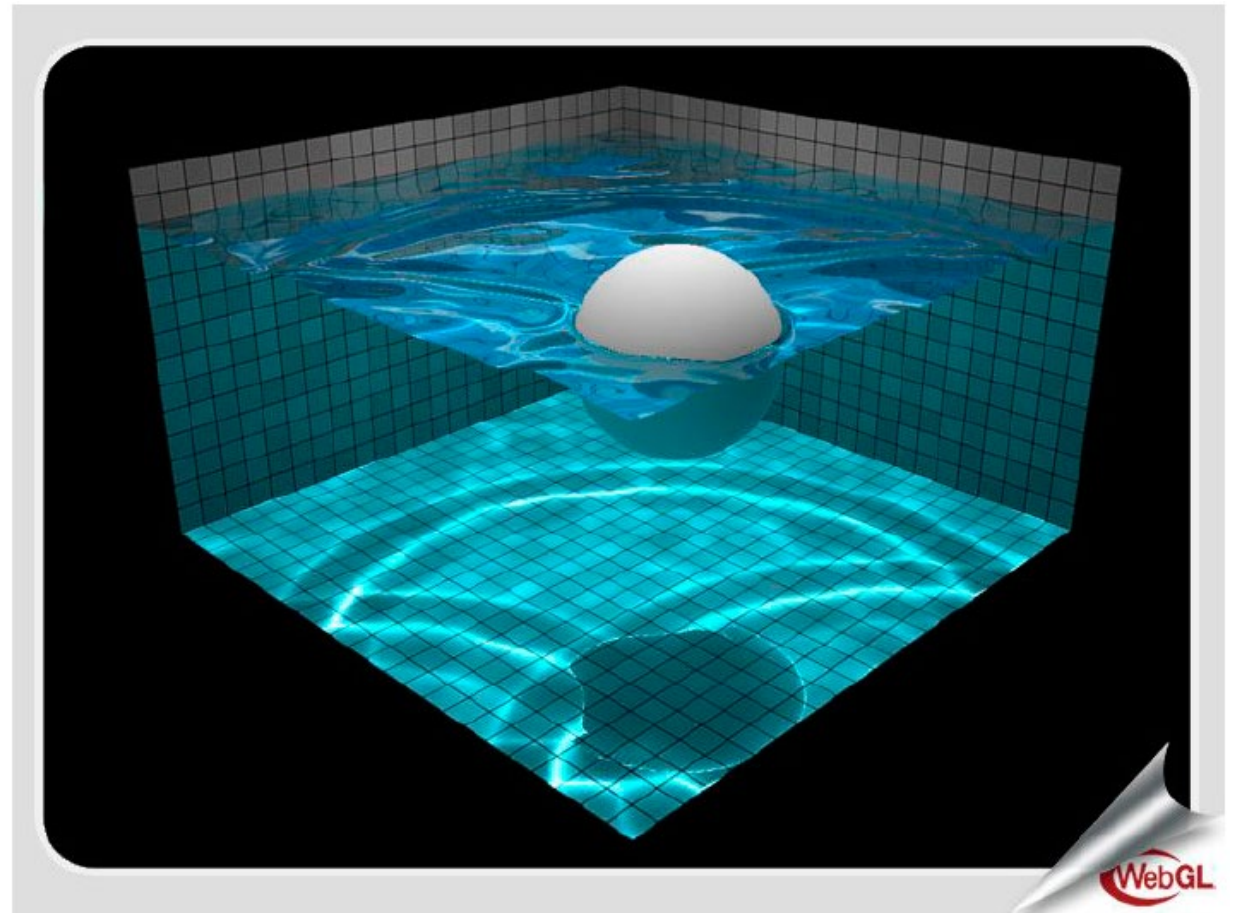
# Ejemplos Three.js



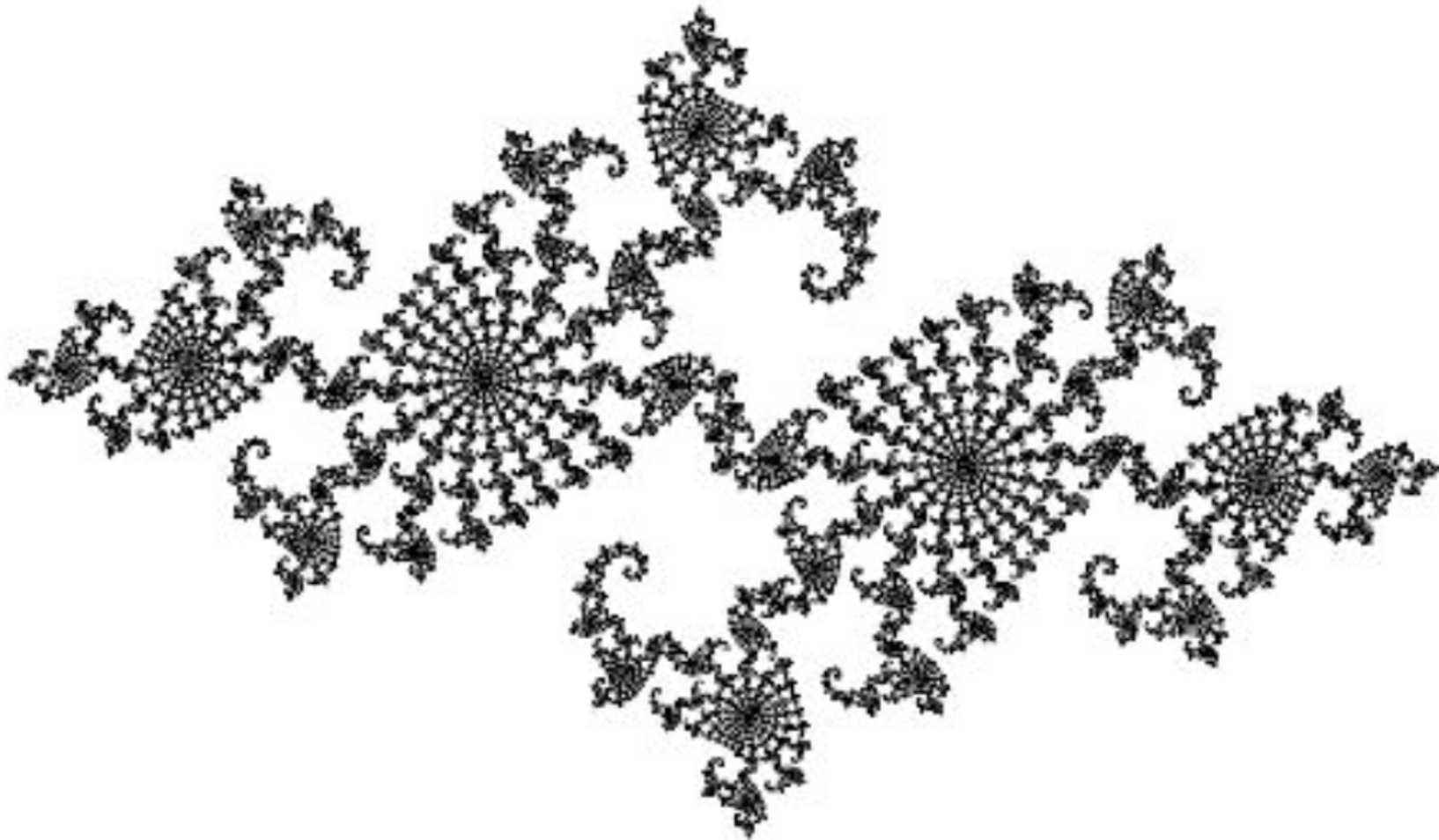
<https://threejs.org/examples/>

# ¿Cómo funcionan estas herramientas?

- VertexShader
- FragmentShader

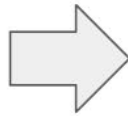
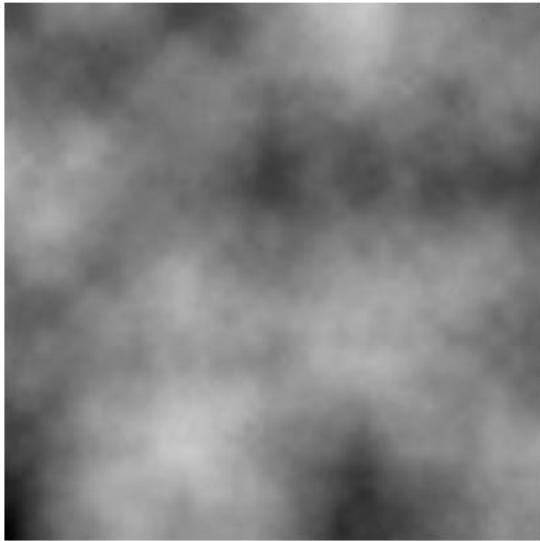


# Fractal de Julia

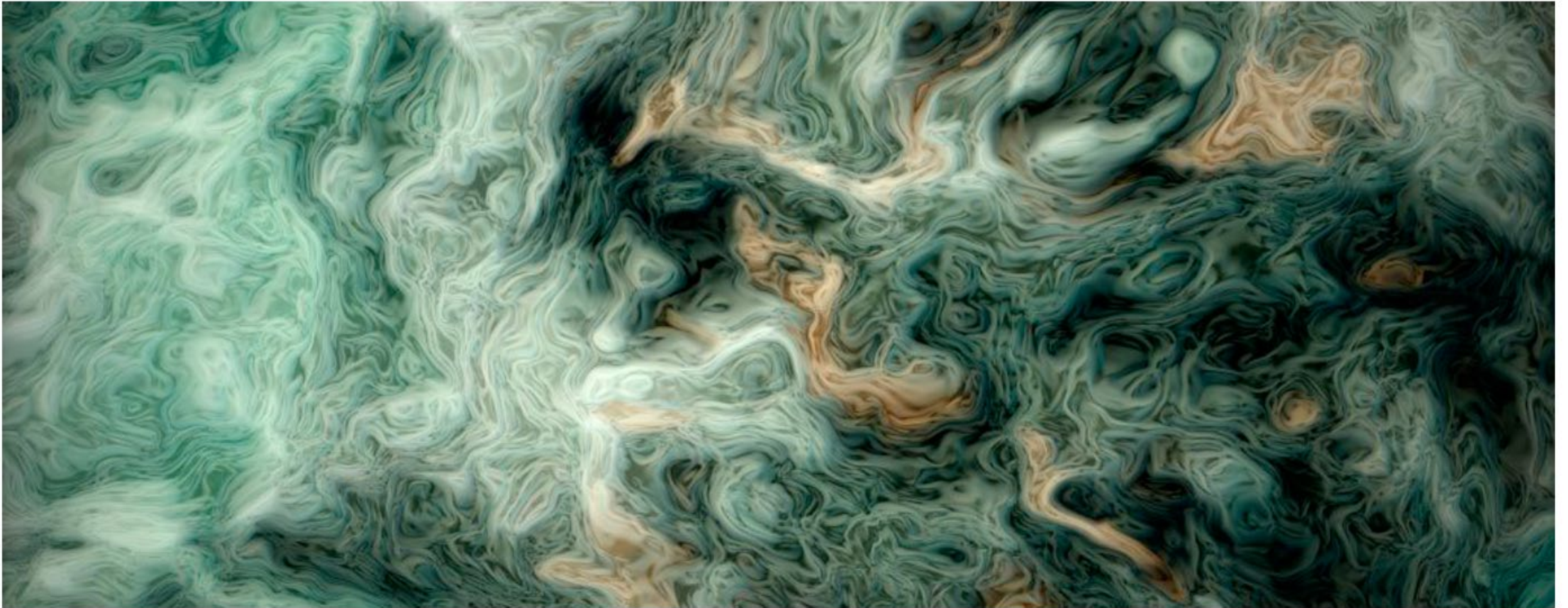




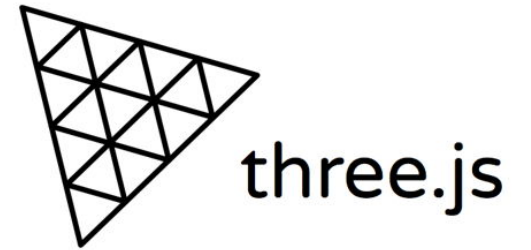
# Perlin Noise



# Domain Warping

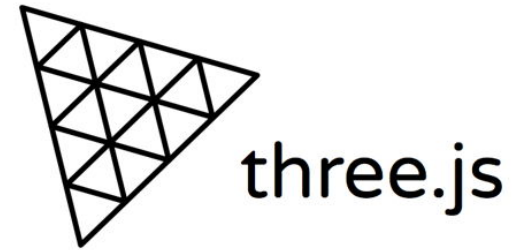


# Three.js



- Librería programada en **Javascript** que permite crear complejas aplicaciones en 3D utilizando WebGL.
- Permite desarrollar aplicaciones ejecutables desde cualquier navegador web moderno.
- Extensiva documentación: <https://threejs.org/docs/>
- Muchos ejemplos: <https://threejs.org/examples/>

# Javascript

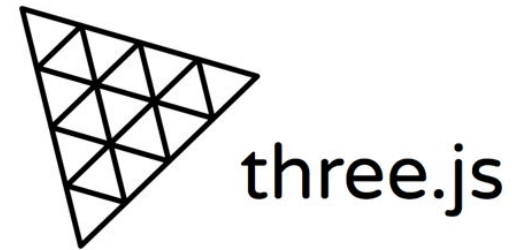


- Lenguaje de programación interpretado, orientado a objetos.
- Similar a Python.

```
function Object(){  
  this.variable = 1;  
  this.variable2 = "String";  
  this.lista = [1, 2, "String", false];  
  this.method = function(){  
    let a = 1; // Variable global  
    if (a === 1){  
      return 2;  
    } else {  
      return false;  
    }  
  }  
}
```

- Definición de objetos se realiza con *funciones*.
- Para declarar variables locales se utiliza **let** y globales **var**.
- Funcionamiento de listas es idéntico a Python.
- Para crear objetos utilizar **new Object(...)**

# Javascript



- La estructura de objetos en Javascript se denomina JSON (Javascript object notation).

```
let modelo = {  
  enabled: false,  
  mesh: {  
    "mesh-data": null,  
    "list-vertex": new VertexList(),  
    "list-faces": [[1,2,3], [4,5,6]],  
    "list-edges": edges[0],  
  }...  
}
```

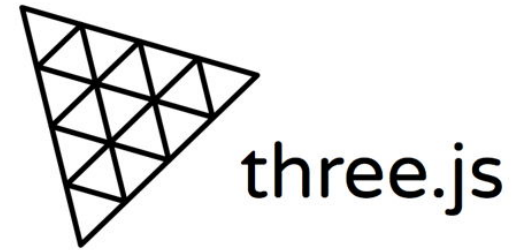
```
let a = modelo.enabled; // false  
let b = modelo["mesh"]["mesh-data"];  
modelo["mesh"]["list-edges"] = null;
```

- Objetos similares a los diccionarios de Python.

- Ver tutorial: <https://www.w3schools.com/js/>



# Filosofía de Three.js



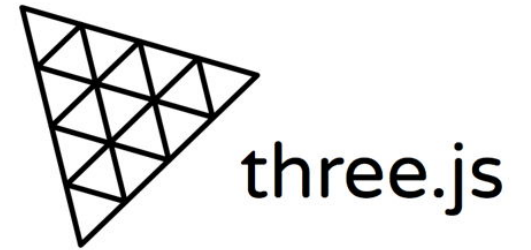
- Elementos basados en Geometrías (que heredan de *Shape*) y Materiales, los que conforman un Mesh.

```
geometry = new THREE.BoxGeometry( 0.2, 0.2, 0.2 );  
material = new THREE.MeshNormalMaterial();  
mesh = new THREE.Mesh( geometry, material );
```

- La escena está conformada por conjuntos de Meshes.
- La cámara junto con la escena son “pasados” al renderer que despliega la escena.



# Three.js en acción



```
camera = new THREE.PerspectiveCamera( 70, window.innerWidth / window.innerHeight, 0.01, 10 );
camera.position.z = 1;
scene = new THREE.Scene();

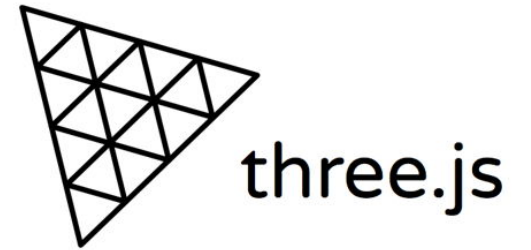
...

scene.add( mesh );
renderer = new THREE.WebGLRenderer( { antialias: true } );
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );

...

renderer.render( scene, camera );
```

# Three.js en acción



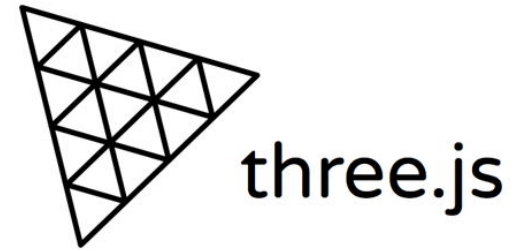
## Existen múltiples maneras de crear una geometría:

### 1. Utilizar funciones predefinidas:

- |                        |                    |
|------------------------|--------------------|
| - BoxGeometry          | Cubo               |
| - CircleGeometry       | Círculo            |
| - ConeGeometry         | Cono               |
| - CylinderGeometry     | Crea un cilindro   |
| - DodecahedronGeometry | Crea un dodecaedro |
| - IcosahedronGeometry  | Crea un icosaedro  |

Muchos más ...

# Three.js en acción

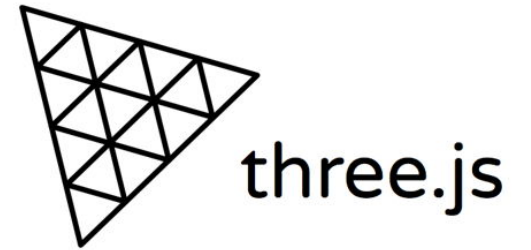


**Existen múltiples maneras de crear una geometría:**

2. Mediante listas de puntos:

- LatheGeometry                      Crea un mesh de geometría axial
- **ShapeGeometry**                      Crea un plano mediante una lista de puntos
  - Permite conectar puntos por rectas
  - Se pueden conectar puntos con curvas de bezier
- PlaneGeometry                      Crea un plano rectangular

# Three.js en acción



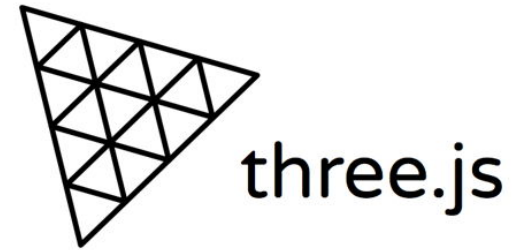
## Geometrías:

Las geometrías, al ser objetos poseen múltiples campos (posición, mesh, etc). Heredan de Object3D.

<https://threejs.org/docs/#api/en/core/Object3D>



# Three.js en acción



## Geometrías:

`.castShadow` : Boolean

Whether the object gets rendered into shadow map. Default is false.

`.children` : [Object3D](#)

Array with object's children. See [Group](#) for info on manually grouping objects.

`.customDepthMaterial` : [Material](#)

Custom depth material to be used when rendering to the depth map. Can only be used in context of meshes. When shadow-casting with a [DirectionalLight](#) or [SpotLight](#), if you are (a) modifying vertex positions in the vertex shader, (b) using a displacement map, (c) using an alpha map with `alphaTest`, or (d) using a transparent texture with `alphaTest`, you must specify a `customDepthMaterial` for proper shadows. Default is undefined.

`.customDistanceMaterial` : [Material](#)

Same as [customDepthMaterial](#), but used with [PointLight](#). Default is undefined.

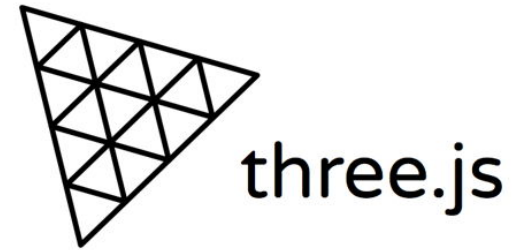
`.frustumCulled` : Boolean

When this is set, it checks every frame if the object is in the frustum of the camera before rendering the object. Otherwise the object gets rendered every frame even if it isn't visible. Default is true.

`.id` : Integer

readonly - Unique number for this object instance.

# Three.js en acción



## Geometrías:

[`.matrix`](#) : [`Matrix4`](#)

The local transform matrix.

[`.matrixAutoUpdate`](#) : Boolean

When this is set, it calculates the matrix of position, (rotation or quaternion) and scale every frame and also recalculates the `matrixWorld` property. Default is [`Object3D.DefaultMatrixAutoUpdate`](#) (true).

[`.matrixWorld`](#) : [`Matrix4`](#)

The global transform of the object. If the `Object3D` has no parent, then it's identical to the local transform [`.matrix`](#).

[`.matrixWorldNeedsUpdate`](#) : Boolean

When this is set, it calculates the `matrixWorld` in that frame and resets this property to false. Default is false.

[`.modelViewMatrix`](#) : [`Matrix4`](#)

This is passed to the shader and used to calculate the position of the object.

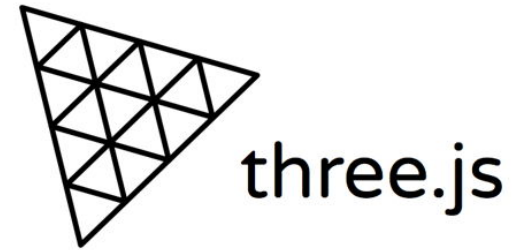
[`.name`](#) : String

Optional name of the object (doesn't need to be unique). Default is an empty string.

[`.normalMatrix`](#) : [`Matrix3`](#)

This is passed to the shader and used to calculate lighting for the object. It is the transpose of the inverse of the upper left 3x3 sub-matrix of this object's `modelViewMatrix`.

# Three.js en acción



## Geometrías:

[.parent](#) : [Object3D](#)

Object's parent in the [scene graph](#). An object can have at most one parent.

[.position](#) : [Vector3](#)

A [Vector3](#) representing the object's local position. Default is (0, 0, 0).

[.quaternion](#) : [Quaternion](#)

Object's local rotation as a [Quaternion](#).

[.receiveShadow](#) : Boolean

Whether the material receives shadows. Default is false.

[.renderOrder](#) : Number

This value allows the default rendering order of [scene graph](#) objects to be overridden although opaque and transparent objects remain sorted independently. When this property is set for an instance of [Group](#), all descendants objects will be sorted and rendered together. Sorting is from lowest to highest renderOrder. Default value is 0.

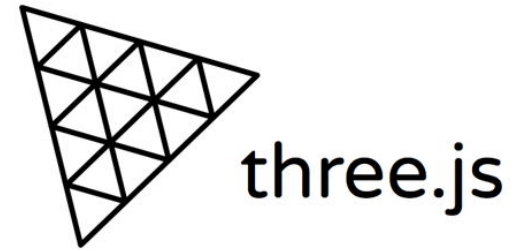
[.rotation](#) : [Euler](#)

Object's local rotation (see [Euler angles](#)), in radians.

[.scale](#) : [Vector3](#)

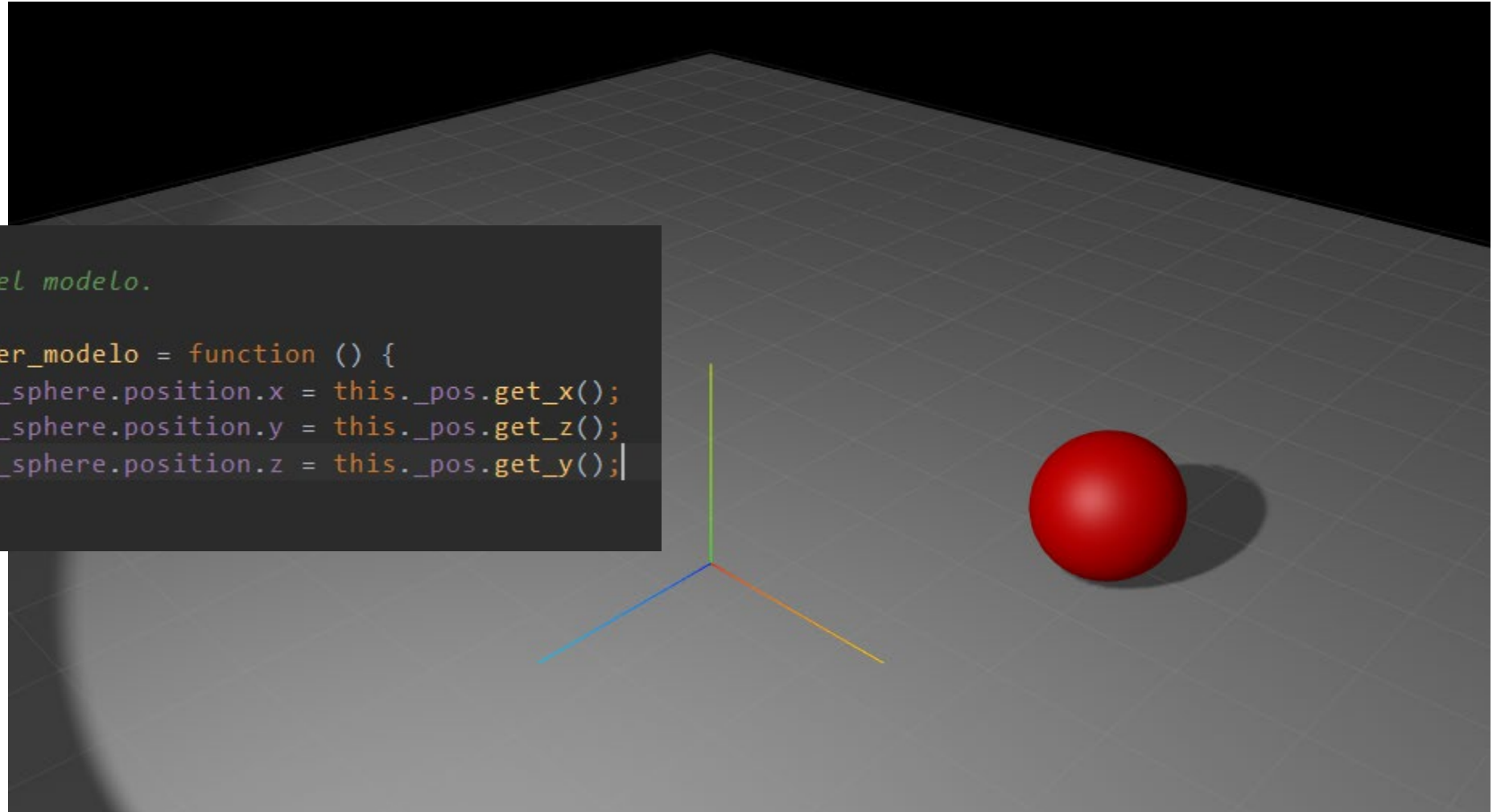
The object's local scale. Default is [Vector3](#)( 1, 1, 1 ).

# Three.js en acción

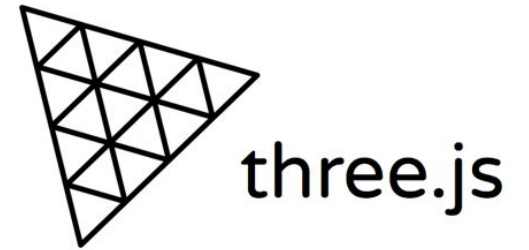


## Geometrías:

```
107  /**
108    * Mueve el modelo.
109    */
110    this._mover_modelo = function () {
111        this._sphere.position.x = this._pos.get_x();
112        this._sphere.position.y = this._pos.get_z();
113        this._sphere.position.z = this._pos.get_y();
114    };
115
```



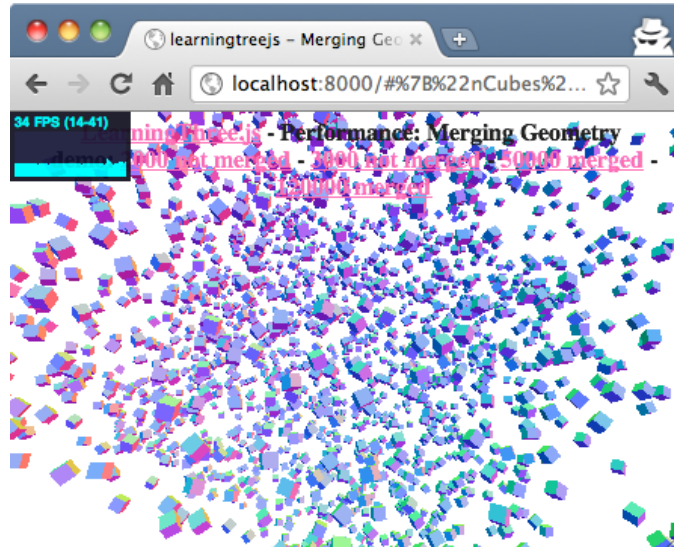
# Three.js en acción



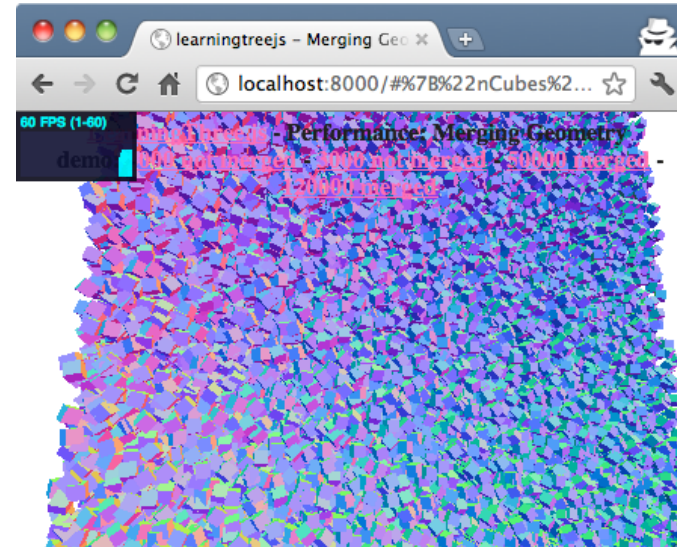
**Las geometrías pueden unirse (merging) lo cual permite optimizar objetos muy complejos.**

“Mientras menos datos sean intercambiados entre la CPU y la GPU mejor es el performance”

Sin merge:  
34FPS

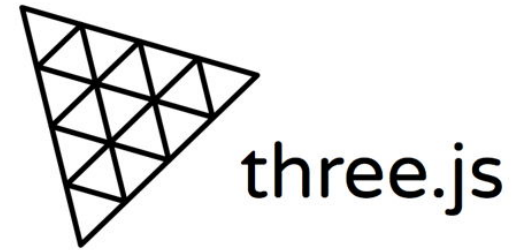


Con merge:  
60FPS





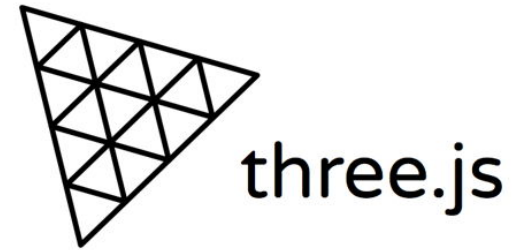
# Three.js en acción



## Existen múltiples materiales:

- MeshBasic
- MeshDepth
- MeshLambert
- MeshNormal
- MeshPhong
- MeshPhyscal
- MeshToon
- PointsMaterial
- RawShaderMaterial
- ShaderMaterial
- ShadowMaterial
- SpriteMaterial

# Three.js en acción

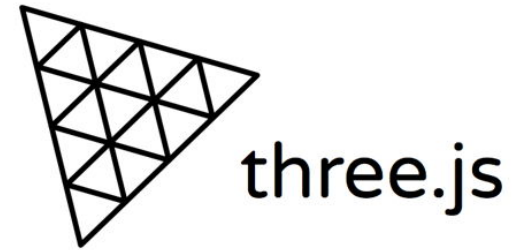


## Existen múltiples materiales:

```
43
44  /**
45   * Crea el modelo en three.js
46   */
47  this.crear_modelo = function (scene) {
48
49    /**
50     * Crea una esfera
51     * Basado en: https://threejs.org/docs/#api/en/geometries/SphereGeometry
52     */
53    let geometry = new THREE.SphereGeometry(this._radio, widthSegments: 32, heightSegments: 32);
54    let material = new THREE.MeshPhongMaterial({parameters: {color: this._color, shading: true}});
55    this._sphere = new THREE.Mesh(geometry, material);
56    this._sphere.castShadow = true;
57    scene.add(this._sphere);
58  };
59
```

Crea la geometría (Definición de vértices y normales, topología y conectividad)

# Three.js en acción

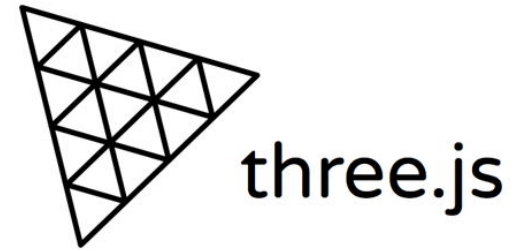


## Existen múltiples materiales:

```
43
44  /**
45   * Crea el modelo en three.js
46   */
47  this.crear_modelo = function (scene) {
48
49    /**
50     * Crea una esfera
51     * Basado en: https://threejs.org/docs/#api/en/geometries/SphereGeometry
52     */
53    let geometry = new THREE.SphereGeometry(this._radius, widthSegments: 32, heightSegments: 32);
54    let material = new THREE.MeshPhongMaterial( parameters: {color: this._color, dithering: true});
55    this._sphere = new THREE.Mesh(geometry, material);
56    this._sphere.castShadow = true;
57    scene.add(this._sphere);
58  };
59
```

Crea el material, usando Shading Phong, con parámetros.

# Three.js en acción

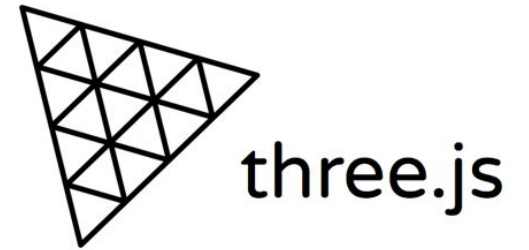


## Existen múltiples materiales:

```
43
44  /**
45   * Crea el modelo en three.js
46   */
47  this.crear_modelo = function (scene) {
48
49    /**
50     * Crea una esfera
51     * Basado en: https://threejs.org/docs/#api/en/geometries/SphereGeometry
52     */
53    let geometry = new THREE.SphereGeometry(this._radio, widthSegments: 32, heightSegments: 32);
54    let material = new THREE.MeshPhongMaterial( parameters: {color: this._color, lighting: true});
55    this._sphere = new THREE.Mesh(geometry, material);
56    this._sphere.castShadow = true;
57    scene.add(this._sphere);
58  };
59
```

Crea el mesh, asigna material (shader) a cada cara de la geometría.

# Three.js en acción

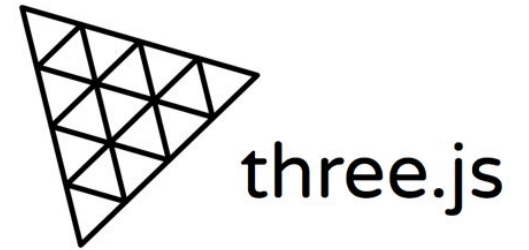


## Existen múltiples materiales:

```
43
44  /**
45   * Crea el modelo en three.js
46   */
47  this.crear_modelo = function (scene) {
48
49    /**
50     * Crea una esfera
51     * Basado en: https://threejs.org/docs/#api/en/geometries/SphereGeometry
52     */
53    let geometry = new THREE.SphereGeometry(this._radio, widthSegments: 32, heightSegments: 32);
54    let material = new THREE.MeshPhongMaterial( parameters: {color: this._color, dithering: true});
55    this._sphere = new THREE.Mesh(geometry, material);
56    this._sphere.castShadow = true;
57    scene.add(this._sphere);
58  };
59
```

Añade el mesh (Object3D) a la escena.

# Three.js en acción



## Añadir luces es muy sencillo.

Three.js ofrece varias formas de crear luces, todas ellas se añaden a la escena sólo 1 vez. LUZ+MATERIAL+GEOMETRIA = Respuesta visual.

- AmbientLight
- DirectionalLight
- HemisphereLight
- PointLight
- RectAreaLight
- SpotLight

```
279  
280 // noinspection JSUnusedGlobalSymbols  
281 this._cameralight = new THREE.PointLight();  
282 this._cameralight.color.setHex(this.objects_props.camera.light.color);  
283 this._cameralight.decay = this.objects_props.camera.light.decay;  
284 this._cameralight.distance = this.objects_props.camera.light.distance;  
285 this._cameralight.intensity = this.objects_props.camera.light.intensity;  
286 this._three_camera.add(this._cameralight);
```



# Three.js en acción

## Concepto importante: Renderizador

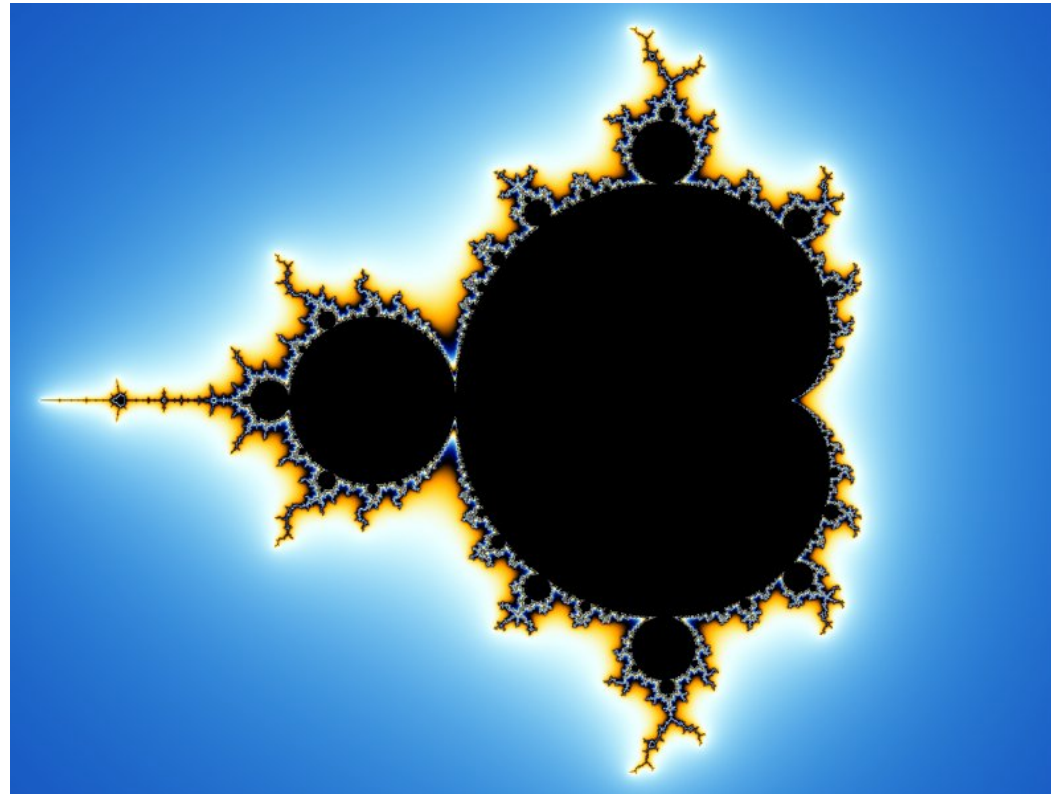
Three.js requiere definir el renderizador, ello es, la herramienta que utilizará para dibujar todo en pantalla. Se pueden usar distintos parámetros dependiendo del renderizador.

```
* -----  
* Inicia el render de Three.js  
* -----  
*/  
this._renderer = new THREE.WebGLRenderer( parameters: {  
  
    // Activa las transparencias  
    alpha: false,  
  
    // Antialias  
    antialias: true,  
  
    // Tiene un búffer de profundidad de 16 bits  
    depth: true,  
  
    // Búffer de profundidad logarítmico, usado cuando hay mucha diferencia en la escena  
    logarithmicDepthBuffer: false,  
  
    // Preferencia de WebGL, puede ser "high-performance", "low-power" ó "default"  
    powerPreference: "default",  
  
    // Precisión  
    precision: 'highp',  
  
    // Los colores ya tienen incorporado las transparencias  
    premultipliedAlpha: true,  
  
    // Para capturas, si molesta deshabilitar  
    preserveDrawingBuffer: false,  
  
    // El búffer de dibujo tiene un stencil de 8 bits  
    stencil: false,  
  
});
```

# Programar en OpenGL

## Ejemplos – Tarea N°2 2018

Construir un fractal utilizando shaders



# Programar en OpenGL

## Ejemplos – Tarea N°2 2018

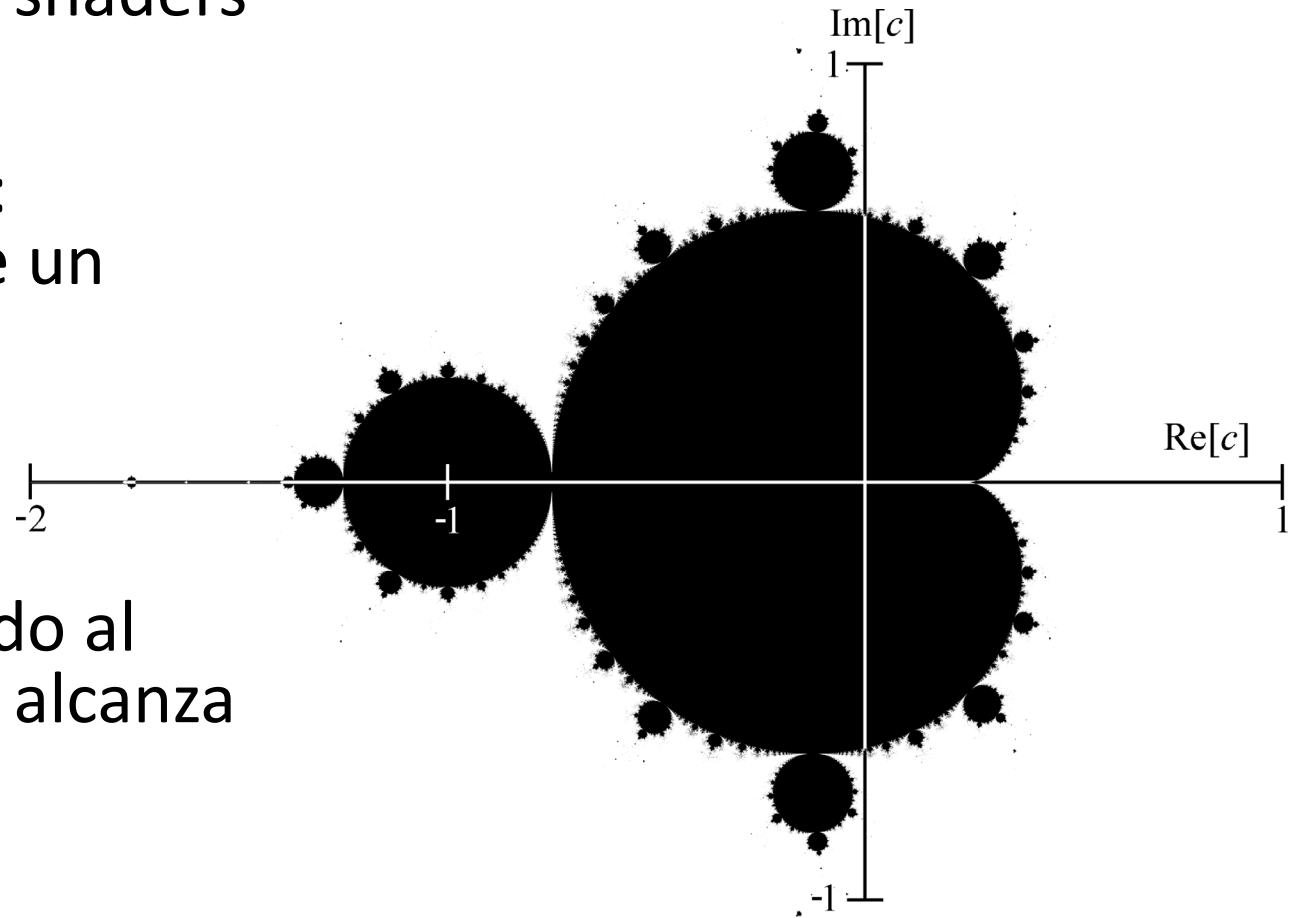
Construir un fractal utilizando shaders

¿Ideas?

Funcionamiento de un fractal:  
Ecuación de recurrencia sobre un plano

$$\begin{cases} z_0 = 0 \in \mathbb{C} & (\text{término inicial}) \\ z_{n+1} = z_n^2 + c & (\text{sucesión recursiva}) \end{cases}$$

El color del fractal está asociado al número de iteraciones que se alcanza antes de diverger



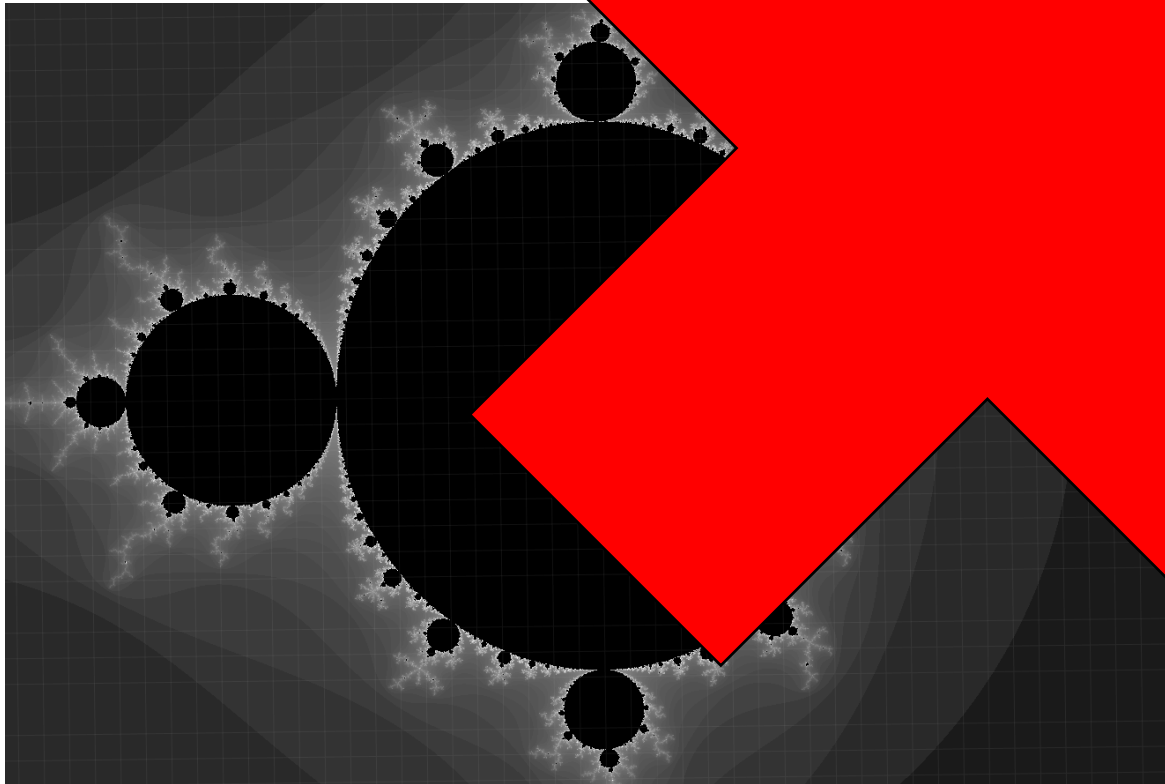
# Programar en OpenGL

## Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros

R: Modelar un plano  
grilla es un vértice

cada intersección de la



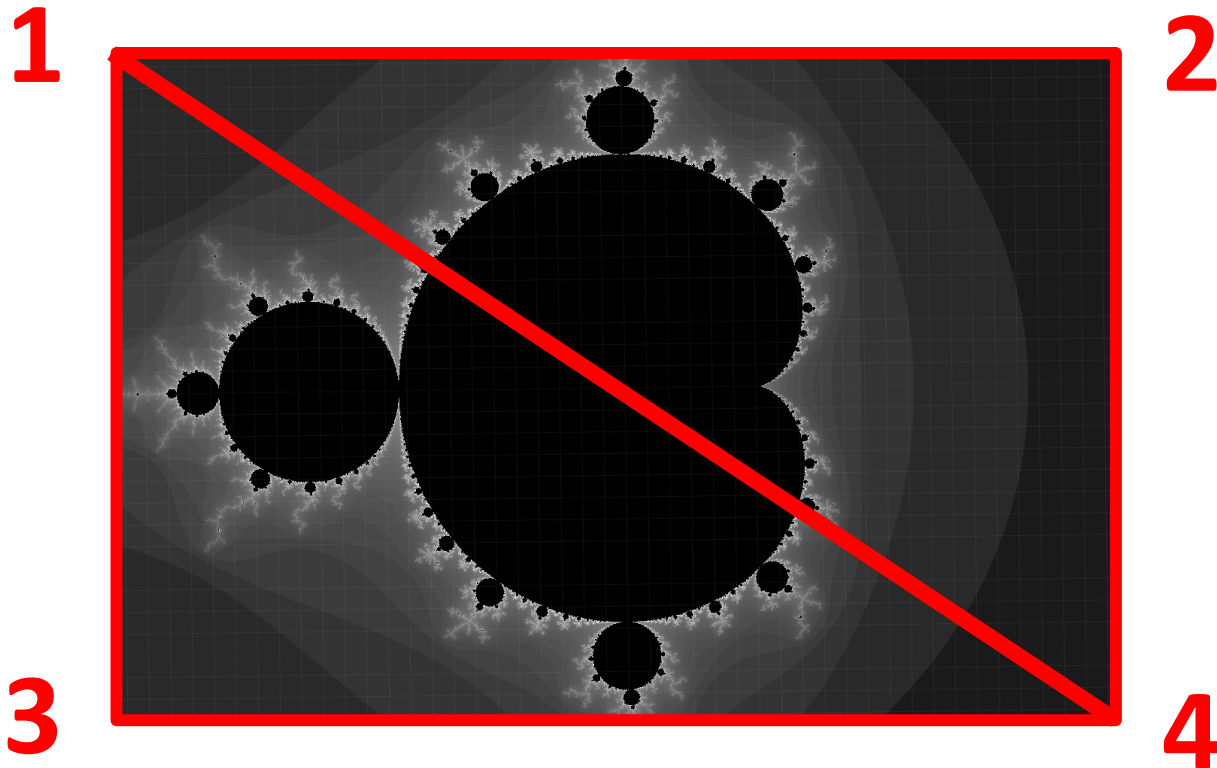
Un punto  $(x, y)$ . X: Parte real, Y: parte imaginaria

# Programar en OpenGL

## Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros vértices?

R: Una aproximación mas lenta es usar dos triángulos. Cada vértice (4) recogerá la información de su posición (x,y) y los parámetros del fractal.



T1: (1,2,4)

T2: (4,3,1)

# Programar en OpenGL

## Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros vértices?
- ¿Cómo modelamos los colores?  
R: Fragment shader



# Aplicación web – Uso HTML+CSS

The image shows the 'Julia-Shaders' web application interface. On the left is a control panel with the following sections:

- Julia-Shaders**: A circular preview of a fractal.
- Shader**: A dropdown menu showing 'Julia f(z) = z^2 + C'.
- Plano complejo**: A box containing 'Re(z): -2, 2', 'Im(z): -2, 2', 'Largo: 2', and 'Nivel zoom: 1'.
- Color del shader**: Six sliders for 'R min', 'R max', 'G min', 'G max', 'B min', and 'B max'.
- N° iteraciones max**: A label at the bottom of the control panel.

The main area displays a large, detailed fractal visualization with a dark red background and a bright yellow/orange fractal structure. Three callout boxes with arrows point to specific features:

- Top box: 'Cambiar el archivo de shader, uso de AJAX para cargar archivos de texto.' (Change the shader file, use of AJAX to load text files.) - points to the Shader dropdown.
- Middle box: 'Visualización de la información de la app.' (Visualization of the app's information.) - points to the Plano complejo section.
- Bottom box: 'Permite variar los parámetros del shader (vertex shader)' (Allows varying the shader parameters (vertex shader)) - points to the Color del shader sliders.

# Aplicación web – Uso HTML+CSS

The screenshot shows the 'Julia-Shaders' web application. On the left is a control panel with the following sections:

- Julia-Shaders**: A circular thumbnail of a fractal.
- Shader**: A dropdown menu showing 'Julia  $f(z) = z^2 + C$ '.
- Plano complejo**: A grey box containing 'Re(z): -2, 2', 'Im(z): -2, 2', 'Largo: 2', and 'Nivel zoom: 1'.
- Color del shader**: Six sliders for 'R min', 'R max', 'G min', 'G max', 'B min', and 'B max'.
- N° iteraciones max**: A label at the bottom of the control panel.

The main area displays a large fractal image. Three callout boxes provide technical details:

- Plano opaco, no se dibuja con shaders sino con la api de Three.js.**: Points to the dark red background plane.
- Color del fractal (TEXTURA) dibujada con shaders**: Points to the yellow fractal pattern.
- Grilla, dibujo con la api de Three.js**: Points to the grid overlay on the fractal.

# Aplicación web – Uso HTML+CSS

Los shaders se dibujan como **TEXTURAS** de una geometría definida por vértices. El shader se compone del string con el código fuente (cargado con Ajax) y los parámetros (varying/in-out) que en este caso son la posición en el plano complejo, el color del shader (r,g,b max) y las constantes de julia.

# Aplicación web – 1) Definición de shaders

```
9
10 /**
11  * Contiene la información de los shaders disponibles
12  * @global
13  * @const
14  * @since 0.1.6
15  */
16 let shader_lib = {
17   "julia-z2": {
18     "files": {
19       "frag": "shaders/julia-z2.frag",
20       "vert": "shaders/julia.vert",
21     },
22     "julia": {
23       "re": -0.450,
24       "im": 0.600,
25     },
26     "name": "Julia  $f(z) = z^2 + C$ ",
27   },
28   "julia-z3": {
29     "files": {
30       "frag": "shaders/julia-z3.frag",
31       "vert": "shaders/julia.vert",
32     },
33     "julia": {
34       "re": 0.400,
35       "im": 0.000,
36     },
37     "name": "Julia  $f(z) = z^3 + C$ ",
```

src/shaderviewer/loader.js

# Aplicación web – 2) Carga del shader

```
136  /**
137   * Carga un shader desde los archivos y luego llama a callback.
138   *
139   * @param {string} vertex - Archivo vertex shader
140   * @param {string} fragment - Archivo fragment shader
141   * @param {function=} callback - Función a llamar para pasar [vertex,shader]
142   * @since 0.1.3
143   */
144  function load_shader(vertex, fragment, callback) {
145
146    /**
147     * Almacena las respuestas de cada caso
148     * @type {{fragment: string, vertex: string}}
149     */
150    let data = {
151      fragment: '',
152      vertex: '',
153    };
154
155    // noinspection JSUnresolvedFunction,JSCheckFunctionSignatures
156    /**
157     * Se crea la consulta para el vertex shader
158     */
159    let $loadVertex = $.ajax( url: {
160      crossOrigin: cfg_ajax_cors,
161      timeout: cfg_href_ajax_timeout,
162      type: 'get',
163      url: vertex,
164    });
```

src/shaderviewer/loader.js

# Aplicación web – 3) Inicio de Three.js

src/shaderviewer/viewer.js

```
520 * -----
521 * Inicia el render de Three.js
522 * -----
523 */
524 this._renderer = new THREE.WebGLRenderer( parameters: {
525
526     // Activa las transparencias
527     alpha: true,
528
529     // Antialias
530     antialias: true,
531
532     // Tiene un búffer de profundidad de 16 bits
533     depth: true,
534
535     // Búffer de profundidad logarítmico, usado cuando hay mucha diferencia en la escena
536     logarithmicDepthBuffer: false,
537
538     // Preferencia de WebGL, puede ser "high-performance", "low-power" ó "default"
539     powerPreference: 'high-performance',
540
541     // Precisión
542     precision: 'highp',
543
544     // Los colores ya tienen incorporado las transparencias
545     premultipliedAlpha: true,
546
547     // Para capturas, si molesta deshabilitar
548     preserveDrawingBuffer: false,
549
550     // El búffer de dibujo tiene un stencil de 8 bits
551     stencil: false,
```



# Aplicación web – 4) Escena, Luces, Cámara y Acción

```
555  /**
556  * -----
557  * Crea la escena
558  * -----
559  */
560  this._scene = new THREE.Scene();
561  this._scene.name = 'VIEWER-3D-SCENE';
562
```

src/shaderviewer/viewer.js

# Aplicación web – 4) Escena, Luces, Cámara y Acción

```
573  /**
574  * -----
575  * Crea la cámara
576  * -----
577  */
578  this._three_camera = new THREE.PerspectiveCamera(
579      self.objects_props.camera.angle,
580      self.objects_props.camera.aspect,
581      self.objects_props.camera.near,
582      self.objects_props.camera.far,
583  );
584  this._three_camera.zoom = this.objects_props.camera.zoom;
585
586  // noinspection JSUnusedGlobalSymbols
587  /**
588  * -----
589  * Crea la luz, se añade a la cámara
590  * -----
591  */
592  this._cameralight = new THREE.PointLight();
593  this._cameralight.color.setHex(this.objects_props.camera.light.color);
594  this._cameralight.decay = this.objects_props.camera.light.decay;
595  this._cameralight.distance = this.objects_props.camera.light.distance;
596  this._cameralight.intensity = this.objects_props.camera.light.intensity;
597  this._three_camera.add(this._cameralight);
```

src/shaderviewer/viewer.js

# Aplicación web – 5) Bucle principal de la aplicación

```
1191  /**
1192  * Inicia el visualizador.
1193  *
1194  * @param {string} parentElement - Contenedor del visualizador
1195  */
1196  this.init = function (parentElement) {
1197      self.id = parentElement;
1198      self._canvasParent = $(parentElement);
1199      self._initThree();
1200      self._initWorldObjects();
1201      self._initEvents();
1202      self._animateFrame();
1203  };
1204
```

src/shaderviewer/viewer.js

# Aplicación web – 5) Bucle principal de la aplicación

```
736  /**
737   * Actualiza controles y renderiza.
738   */
739  this._animateFrame = function () {
740
741      /**
742       * Actualiza los controles
743       */
744      this._controls.update();
745
746      /**
747       * Renderiza un cuadro
748       */
749      this._render();
750
751  };
```

src/shaderviewer/viewer.js

```
724  /**
725   * Renderiza el contenido de Three.js.
726   */
727  this._render = function () {
728
729      /**
730       * Renderiza
731       */
732      self._renderer.render(self._scene, self._three_camera);
733
734  };
```

# Aplicación web – 6) Creación de la geometría y del shader

Creamos la geometría en Three.js

Los valores de cada vértice son flotantes, 6 (uno por cada vértice)

src/shaderviewer/viewer.js

```
1273      /**
1274       * Inicia el objeto de dibujo del shader.
1275       */
1276      this._initShaderObject = function () {
1277
1278          /**
1279           * Si el objeto ya existía se elimina
1280           */
1281          if (notNullUndf(self._shaderObject.mesh)) {
1282              this._scene.remove(self._shaderObject.mesh);
1283          }
1284          if (notNullUndf(self._bound.mesh)) {
1285              this._scene.remove(self._bound.mesh);
1286          }
1287
1288          /**
1289           * Crea la geometría
1290           */
1291          self._shaderObject.geometry = new THREE.BufferGeometry();
1292
1293          /**
1294           * Inicia los valores iniciales del shader complejo
1295           */
1296          self._shaderObject.vertex.zi = new Float32Array( length: 6);
1297          self._shaderObject.vertex.zr = new Float32Array( length: 6);
1298      }
```

# Aplicación web – 6) Creación de la geometría y del shader

Se definen los vértices de cada figura

```
1299  /**
1300   * Dibuja la figura
1301   */
1302   let vertices = new Float32Array( length: 18);
1303   vertices[0] = -1.0;
1304   vertices[1] = -1.0;
1305   vertices[2] = self._shaderObject.plotz;
1306
1307   vertices[3] = 1.0;
1308   vertices[4] = 1.0;
1309   vertices[5] = self._shaderObject.plotz;
1310
1311   vertices[6] = -1.0;
1312   vertices[7] = 1.0;
1313   vertices[8] = self._shaderObject.plotz;
1314
1315   // Segundo triángulo
1316   vertices[9] = -1.0;
1317   vertices[10] = -1.0;
1318   vertices[11] = self._shaderObject.plotz;
1319
1320   vertices[12] = 1.0;
1321   vertices[13] = 1.0;
1322   vertices[14] = self._shaderObject.plotz;
1323
1324   vertices[15] = 1.0;
1325   vertices[16] = -1.0;
1326   vertices[17] = self._shaderObject.plotz;
1327
```

src/shaderviewer/viewer.js



# Aplicación web – 6) Creación de la geometría y del shader

El shader corresponde al material de la figura

Los “uniform” son los parámetros (constantes) de cada vértice. Se leen en el vertex shader.

src/shaderviewer/viewer.js

```
1328  /**
1329   * Crea el material
1330   */
1331  try {
1332    self._shaderObject.material = new THREE.ShaderMaterial( parameters: {
1333      'fragmentShader': self._shaderObject.datashader.f,
1334      'side': THREE.DoubleSide,
1335      'uniforms': {
1336        'r_min': {
1337          'type': 'f',
1338          'value': self._shaderObject.color.r_min,
1339        },
1340        'r_max': {
1341          'type': 'f',
1342          'value': self._shaderObject.color.r_max,
1343        },
1344        'g_min': {
1345          'type': 'f',
1346          'value': self._shaderObject.color.g_min,
1347        },
1348        'g_max': {
1349          'type': 'f',
1350          'value': self._shaderObject.color.g_max,
1351        },
1352        'b_min': {
1353          'type': 'f',
1354          'value': self._shaderObject.color.b_min,
1355        },
1356        'b_max': {
1357          'type': 'f',
1358          'value': self._shaderObject.color.b_max,
1359        },
1360      },
1361    });
```

# Aplicación web – 6) Creación de la geometría y del shader

```
1380  /**
1381   * Crea la geometría
1382   * @type {BufferGeometry}
1383   */
1384  self._shaderObject.geometry = new THREE.BufferGeometry();
1385  self._shaderObject.geometry.addAttribute('position', new THREE.BufferAttribute(vertices, itemSize: 3));
1386  self._shaderObject.geometry.addAttribute('vertex_z_r', new THREE.BufferAttribute(self._shaderObject.vertex.zr, itemSize: 1));
1387  self._shaderObject.geometry.addAttribute('vertex_z_i', new THREE.BufferAttribute(self._shaderObject.vertex.zi, itemSize: 1));
1388
1389  // Rota la geometría para dejarla coplanar
1390  self._shaderObject.geometry.rotateX(-Math.PI / 2);
1391  self._shaderObject.geometry.rotateY(-Math.PI / 2);
1392
1393  /**
1394   * Crea el mesh
1395   * @type {Mesh}
1396   */
1397  self._shaderObject.mesh = new THREE.Mesh(
1398    self._shaderObject.geometry,
1399    self._shaderObject.material
1400  );
1401  self._addMeshToScene(self._shaderObject.mesh, self._globals.shader, collidable: true);
```

# Aplicación web – ¿Cómo actualizamos los parámetros del shader?

```
1451
1452  /**
1453   * Obtiene los atributos del shader para modificarlos
1454   */
1455   let attrib_z_r = self._shaderObject.mesh.geometry.attributes.vertex_z_r.array;
1456   let attrib_z_i = self._shaderObject.mesh.geometry.attributes.vertex_z_i.array;
1457
1458  /**
1459   * Primer triángulo
1460   */
1461   attrib_z_r[0] = z_r - range;
1462   attrib_z_i[0] = z_i - range;
1463
1464   attrib_z_r[1] = z_r + range;
1465   attrib_z_i[1] = z_i + range;
1466
1467   attrib_z_r[2] = z_r - range;
1468   attrib_z_i[2] = z_i + range;
```

1) Escribir

```
1488  /**
1489   * Anima un nuevo cuadro
1490   */
1491   self._shaderObject.mesh.geometry.attributes.vertex_z_r.needsUpdate = true;
1492   self._shaderObject.mesh.geometry.attributes.vertex_z_i.needsUpdate = true;
1493   this._printCoords();
1494   this._animateFrame();
1495
```

2) Actualizar y dibujar

# Programar en OpenGL

## Ejemplos – Tarea N°2 – Vertex Shader

```
/*
MANDELBROT
VERTEX SHADER

Ejecuta mandelbrot,  $C = c_r + i*c_i$  se pasa por
cada (x,y) del plano complejo.

@author Pablo Pizarro R. @ppizarror.com
@license MIT
@since 0.1.0
*/

// Activa precisión alta
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float;
#else
precision mediump float;
#endif
precision mediump int;
```

```
// (x,y) de cada valor del plano
attribute float vertex_z_r;
attribute float vertex_z_i;
```

```
varying float c_r;
varying float c_i;
```

```
void main() {
```

```
// El valor complejo corresponde a (x,y)
    c_r = vertex_z_r;
    c_i = vertex_z_i;
```

```
    gl_Position = projectionMatrix *
modelViewMatrix * vec4(position, 1.0);
```

```
}
```

# Aplicación web – 6) Creación de la geometría y del shader

```
1380  /**
1381   * Crea la geometría
1382   * @type {BufferGeometry}
1383   */
1384  self._shaderObject.geometry = new THREE.BufferGeometry();
1385  self._shaderObject.geometry.addAttribute('position', new THREE.BufferAttribute(vertices, itemSize: 3));
1386  self._shaderObject.geometry.addAttribute('vertex_z_r', new THREE.BufferAttribute(self._shaderObject.vertex.zr, itemSize: 1));
1387  self._shaderObject.geometry.addAttribute('vertex_z_i', new THREE.BufferAttribute(self._shaderObject.vertex.zi, itemSize: 1));
1388
1389  // Rota la geometría para dejarla coplanar
1390  self._shaderObject.geometry.rotateX(-Math.PI / 2);
1391  self._shaderObject.geometry.rotateY(-Math.PI / 2);
1392
1393  /**
1394   * Crea el mesh
1395   * @type {Mesh}
1396   */
1397  self._shaderObject.mesh = new THREE.Mesh(
1398    self._shaderObject.geometry,
1399    self._shaderObject.material
1400  );
1401  self._addMeshToScene(self._shaderObject.mesh, self._globals.shader, collidable: true);
```

# Programar en OpenGL

## Ejemplos – Tarea N°2 – Fragment Shader

```
/*
MANDELBROT
FRAGMENT SHADER

Ejecuta mandelbrot,  $C = c_r + i*c_i$  se pasa
por cada (x,y) del plano complejo.

@author Pablo Pizarro R. @ppizarror.com
@license MIT
@since 0.1.0
*/

// Activa precisión alta
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float;
#else
precision mediump float;
#endif

precision mediump int;

// Pasa C de cada (x,y)
varying float c_r;
varying float c_i;

// Interacciones máximas
uniform int max_iterations;

// Rango de colores
uniform float r_min;
uniform float r_max;
uniform float g_min;
uniform float g_max;
uniform float b_min;
uniform float b_max;
```



# Programar en OpenGL

## Ejemplos – Tarea N°2 – Fragment Shader

```
// Inicio del shader
void main() {
    float r;
    float g;
    float b;
    float t;
    float w_r;
    float w_i;
    float u;
    float v;

    w_r = 0.0;
    w_i = 0.0;

    // Si converge es negro
    r = 0.0;
    g = 0.0;
    b = 0.0;

    for (int i = 0; i < 65536; i++) {
        u = w_r;
```

```
        v = w_i;
        w_r = u*u - v*v + c_r;
        w_i = 2.0*u*v + c_i;

        if (w_r*w_r + w_i*w_i > 4.0) { // |z| > 2
            // Computa el rojo
            t = log(float(i + 1)) /
log(float(max_iterations + 1));
            r = t*r_max + (1.0 - t)*r_min;
            // Computa el verde
            g = t*g_max + (1.0 - t)*g_min;
            // Computa el azul
            b = t*b_max + (1.0 - t)*b_min;
            break;
        }
        if (i >= max_iterations) {
            break;
        }
    }
    gl_FragColor = vec4(r, g, b, 1.0);
}
```

Muchas gracias por su  
atención

¿Preguntas?