Auxiliar Nº5 – Programación paralela

CC7515 - Computación en GPU

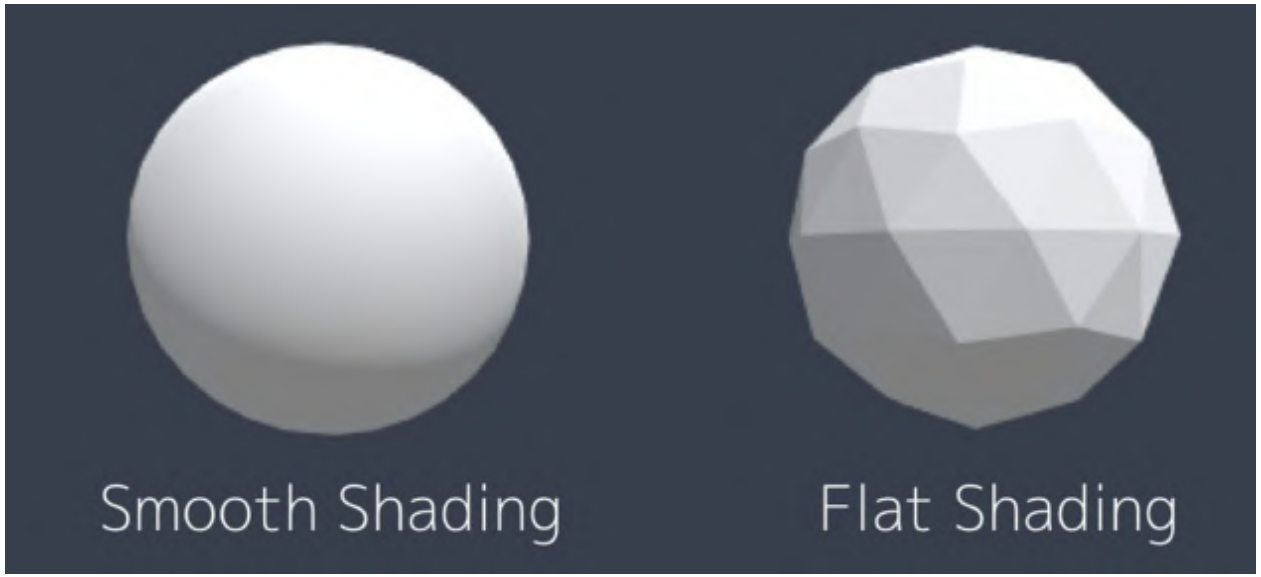
Pablo Pizarro R., pablo@ppizarror.com

Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

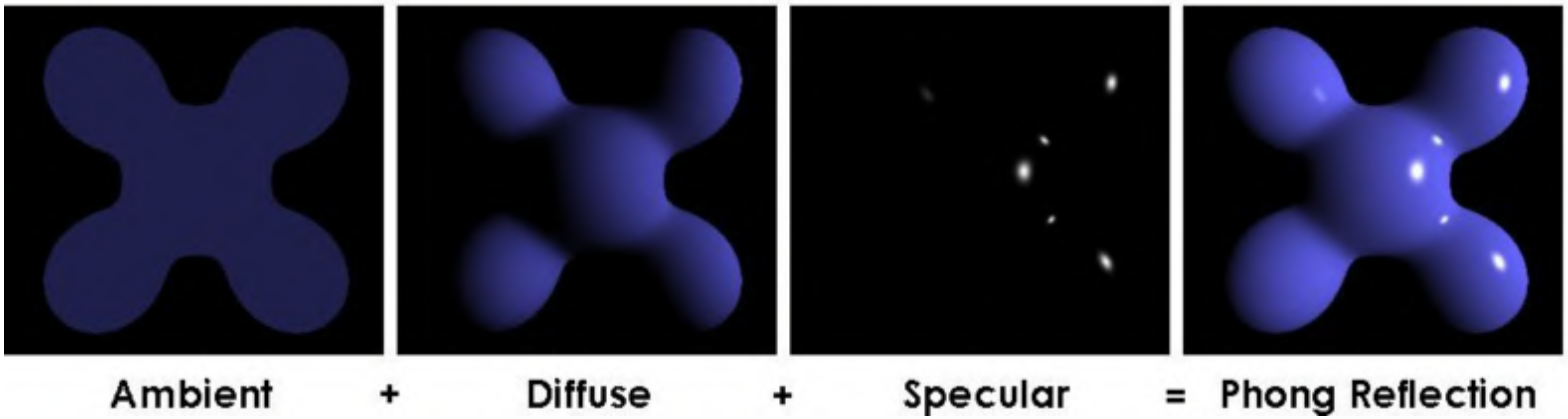
3 de abril de 2022

Contenidos

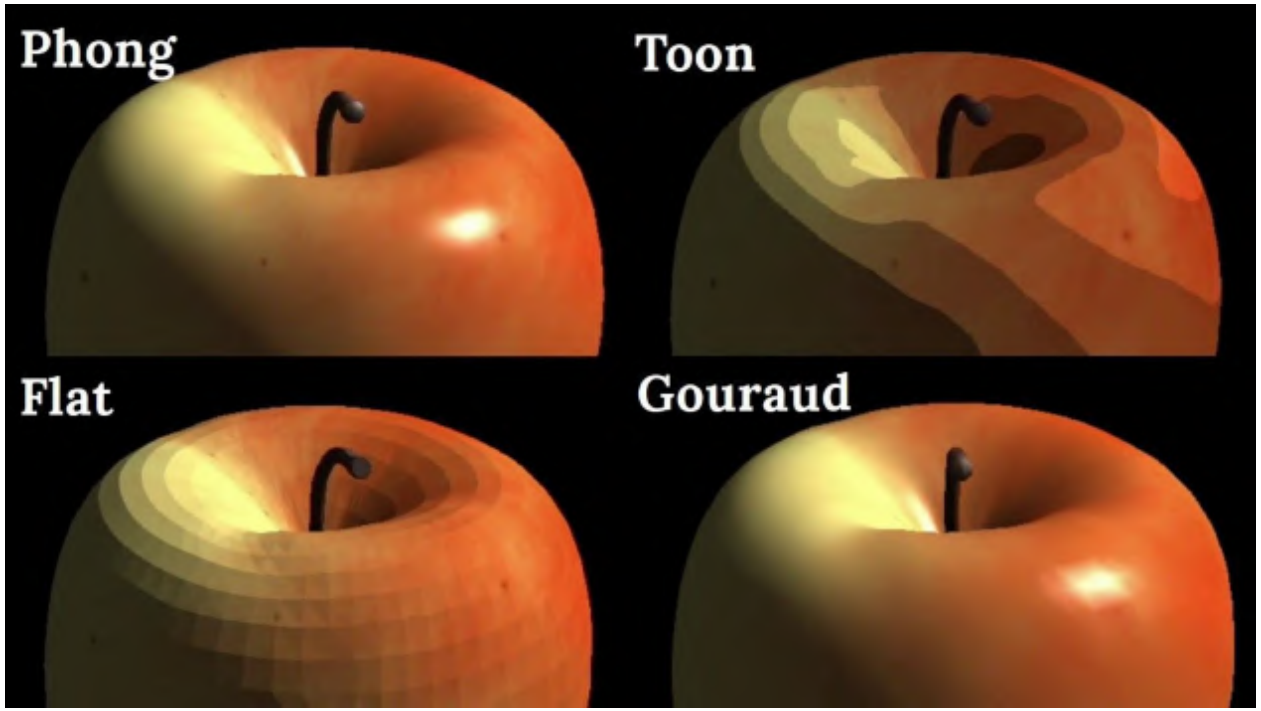
Shaders



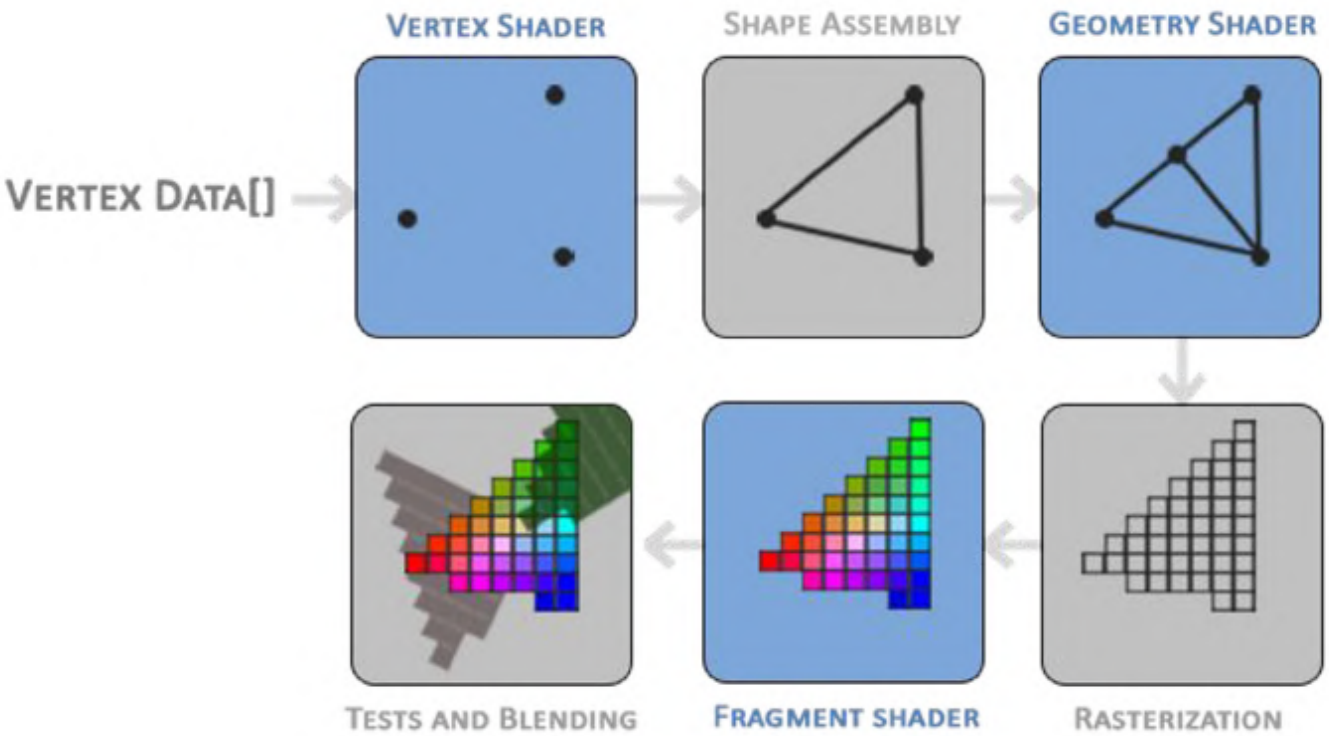
Phong shading



Gouraud shading



Rendering pipeline

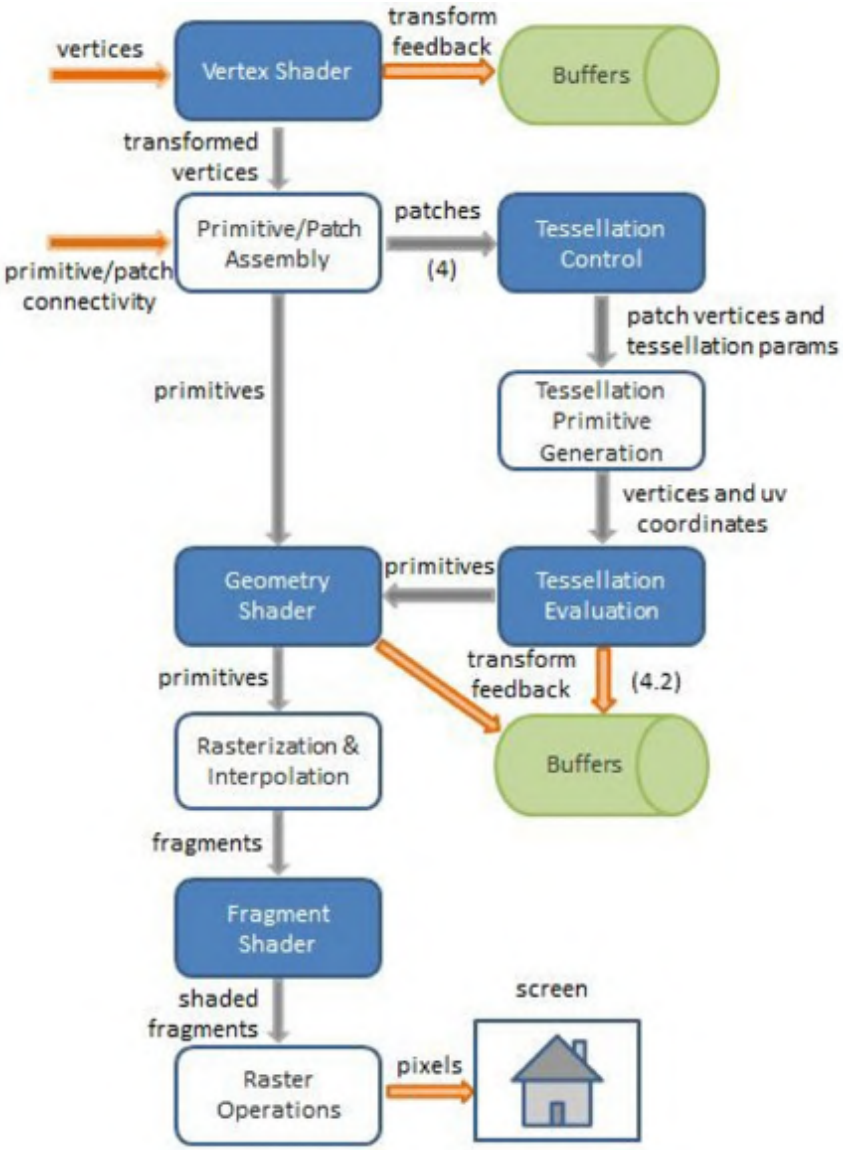


Rendering pipeline

Vertex shader runs once for every vertex

Fragment shader runs once for every pixel rendered in the scene, with vertex data interpolated

Pipeline

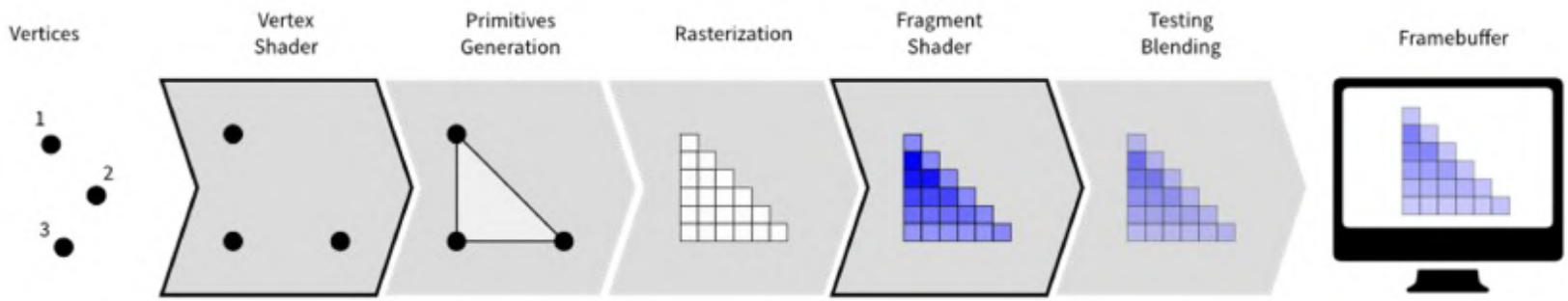

vertices
Vertex Shader
transform feedback
Buffers
transformed vertices
Primitive/Patch Assembly
patches
(4)
Tessellation Control
primitive/patch connectivity
patch vertices and tessellation params
primitives
Tessellation Primitive Generation
vertices and uv coordinates
Geometry Shader
primitives
Tessellation Evaluation
transform feedback
(4.2)
primitives
Rasterization & Interpolation
Buffers
fragments
Fragment Shader
shaded fragments
screen
Raster Operations
pixels

Tipos de shaders

- 2D
  - Fragment/Pixel Shader
- 3D
  - Vertex Shader
  - Geometry Shader
  - Tessellation Shader

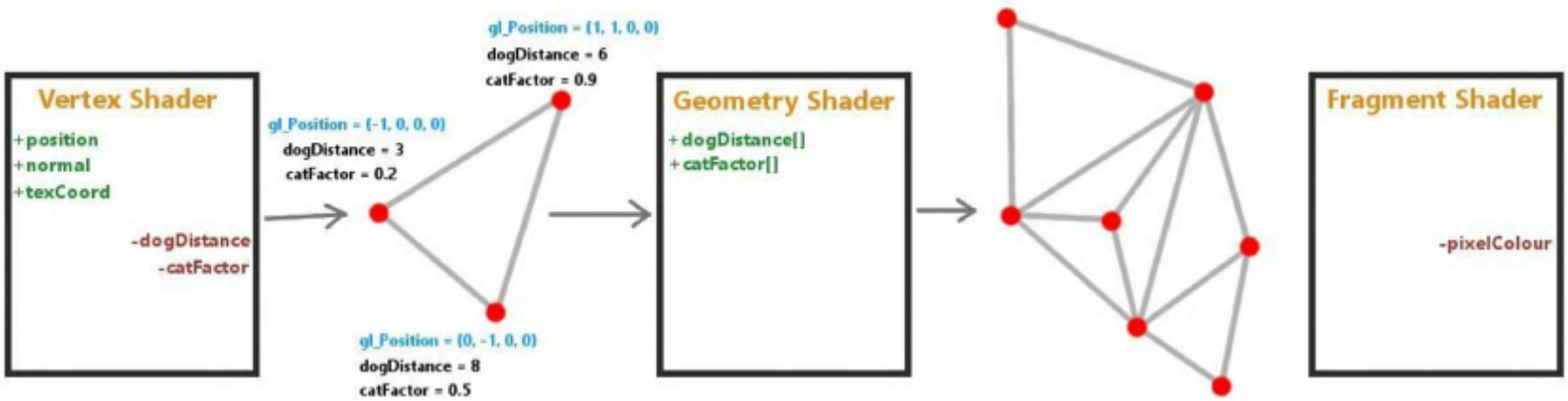
3D – Vertex Shader

Vertex shader es una herramienta capaz de trabajar con la estructura de vértices de figuras modeladas en 3D, y realizar operaciones matemáticas sobre ella para definir colores, texturas e incidencia de la luz.



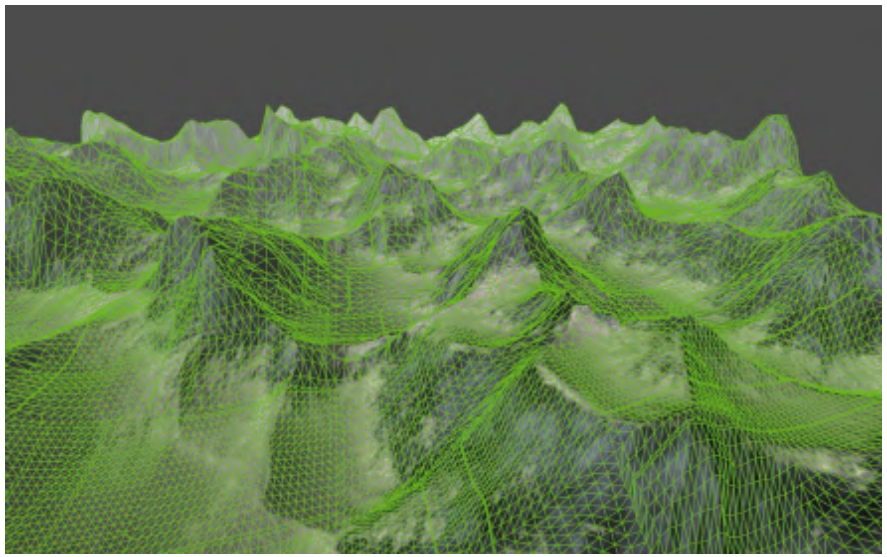
3D – Geometry shader

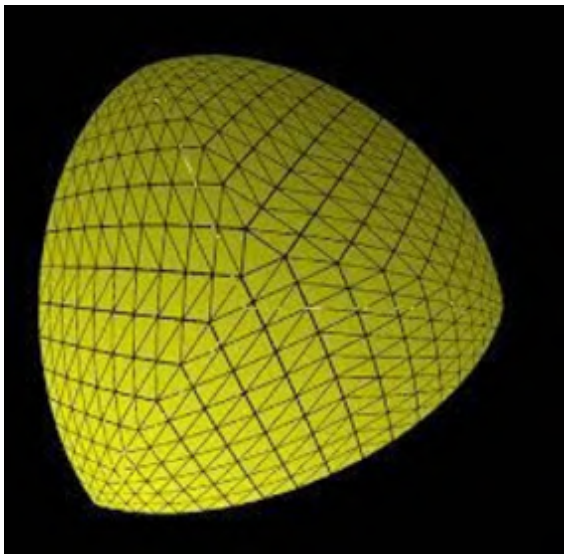
Geometry shader puede generar nuevas primitivas gráficas, como los puntos, las líneas o los triángulos, estas primitivas creadas son enviadas al principio del pipeline gráfico.



3D – Tessellation Shader

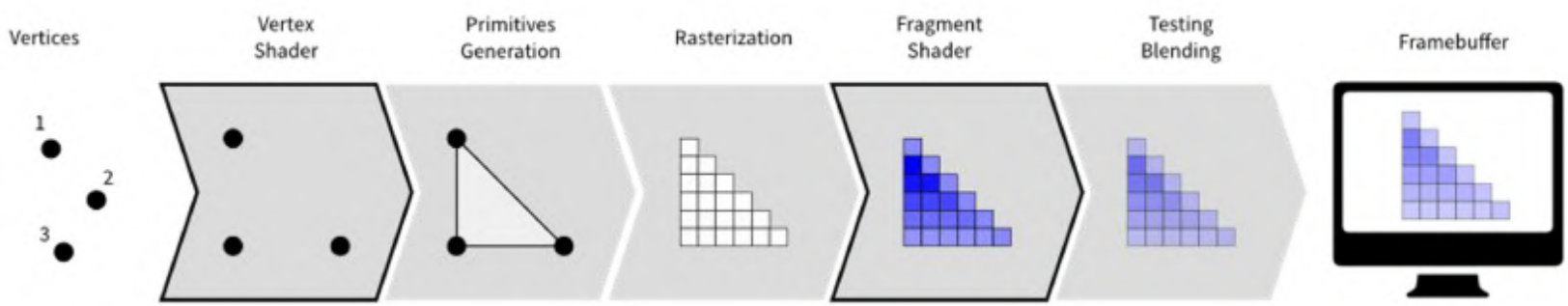
Tessellation Shader es la etapa de procesamiento de vértices en el proceso de renderizado de OpenGL en la que los datos de vértices se subdividen en primitivas más pequeñas.





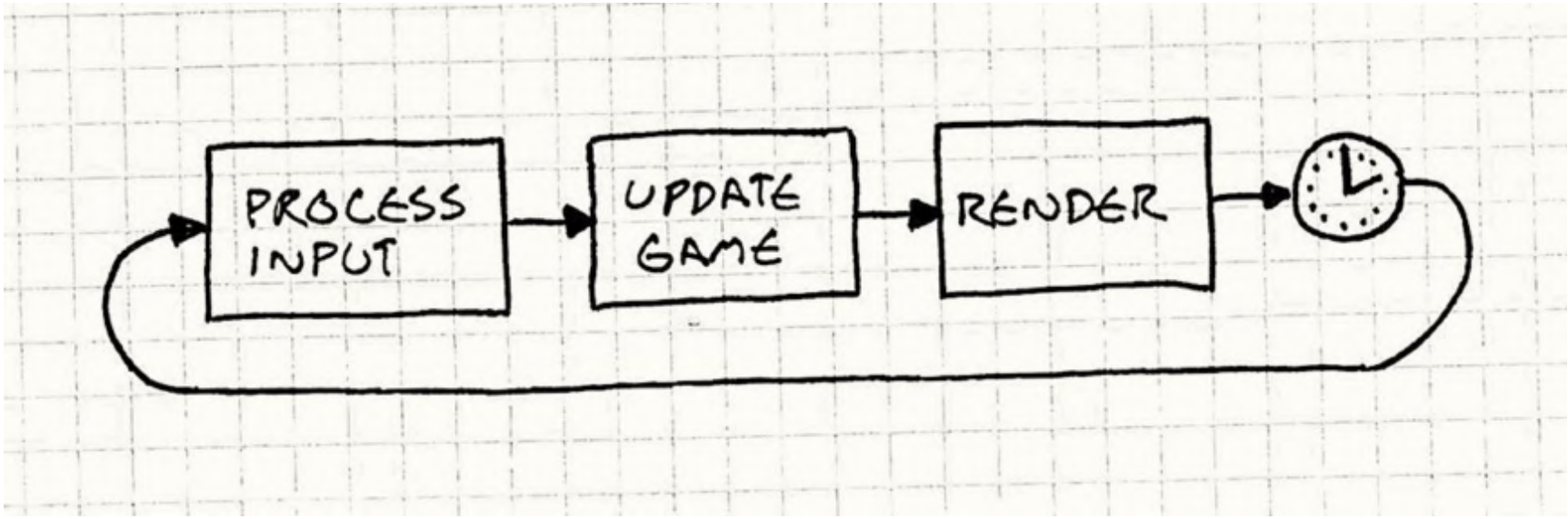
2D – Fragment/Pixel Shader

El **fragment shader** es la etapa del pipeline de OpenGL que se realiza después de rasterizar una primitiva. Para cada muestra de los píxeles cubiertos por una primitiva, se genera un "fragmento". Cada fragmento tiene una posición de espacio de ventana, algunos otros valores, y contiene todos los valores de salida interpolados por vértice de la última etapa de procesamiento de vértices.

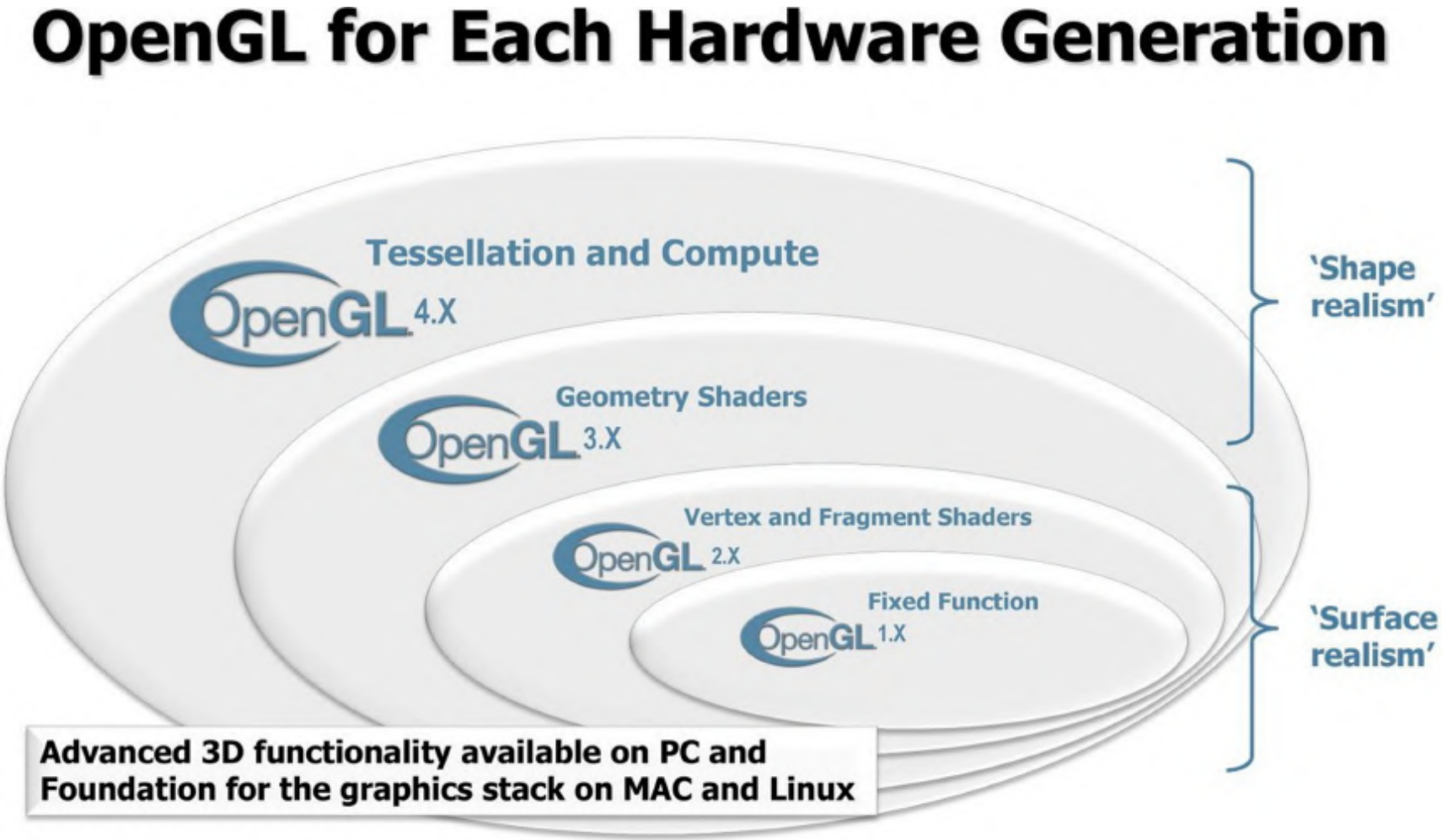


OpenGL

OpenGL – En juegos



Evolución de OpenGL



Alternativas a OpenGL

PyOpenGL 3.1.0

Inicialización

```
1  // 4x antialiasing
2  glfwWindowHint(GLFW_SAMPLES, 4);
3
4  // Define la versión de OpenGL
5  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
6  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
7
8  // Compatibilidad MacOS
9  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
10
11 // No queremos el viejo OpenGL
12 glfwWindowHint(GLFW_OPENGL_PROFILE,
       ↪ GLFW_OPENGL_CORE_PROFILE);
```

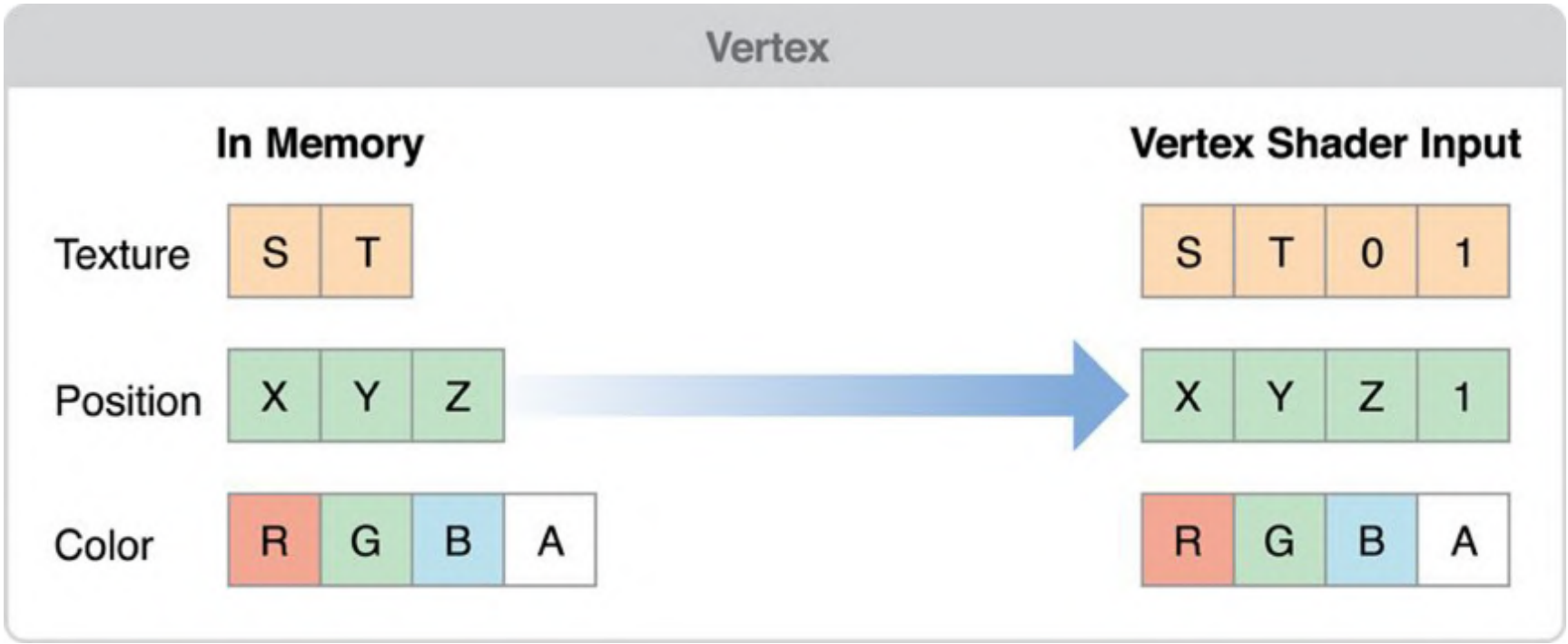
Inicialización

```
1  // Abre una ventana y crea el contexto de OpenGL
2  GLFWwindow *window;
3
4  // Variable global, para simplicidad
5  window = glfwCreateWindow(1024, 768, "Tutorial 01", NULL, NULL);
6  if (window == NULL)
7  {
8    fprintf(stderr, "Failed to open GLFW window.");
9    glfwTerminate();
10   return -1;
11 }
12
13 glfwMakeContextCurrent(window); // Inicializa GLEW
14 glewExperimental = true;       // Se requiere en el core
15 if (glewInit() != GLEW_OK)
16 {
17   fprintf(stderr, "Failed to initialize GLEW\n");
18   return -1;
19 }
```

Window loop

```
1  // Aseguramos poder capturar los eventos del teclado presionados
2  glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
3  do
4  {
5    // Borra la pantalla
6    glClear(GL_COLOR_BUFFER_BIT);
7    // Swap buffers
8    glfwSwapBuffers(window);
9    glfwPollEvents();
10 }
11
12 // Chequea que la tecla ESC haya sido presionada, si pasa eso
13 // se cierra la ventana
14 while (glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
15 glfwWindowShouldClose(window) == 0);
```

VAO (Vertex Array Object)



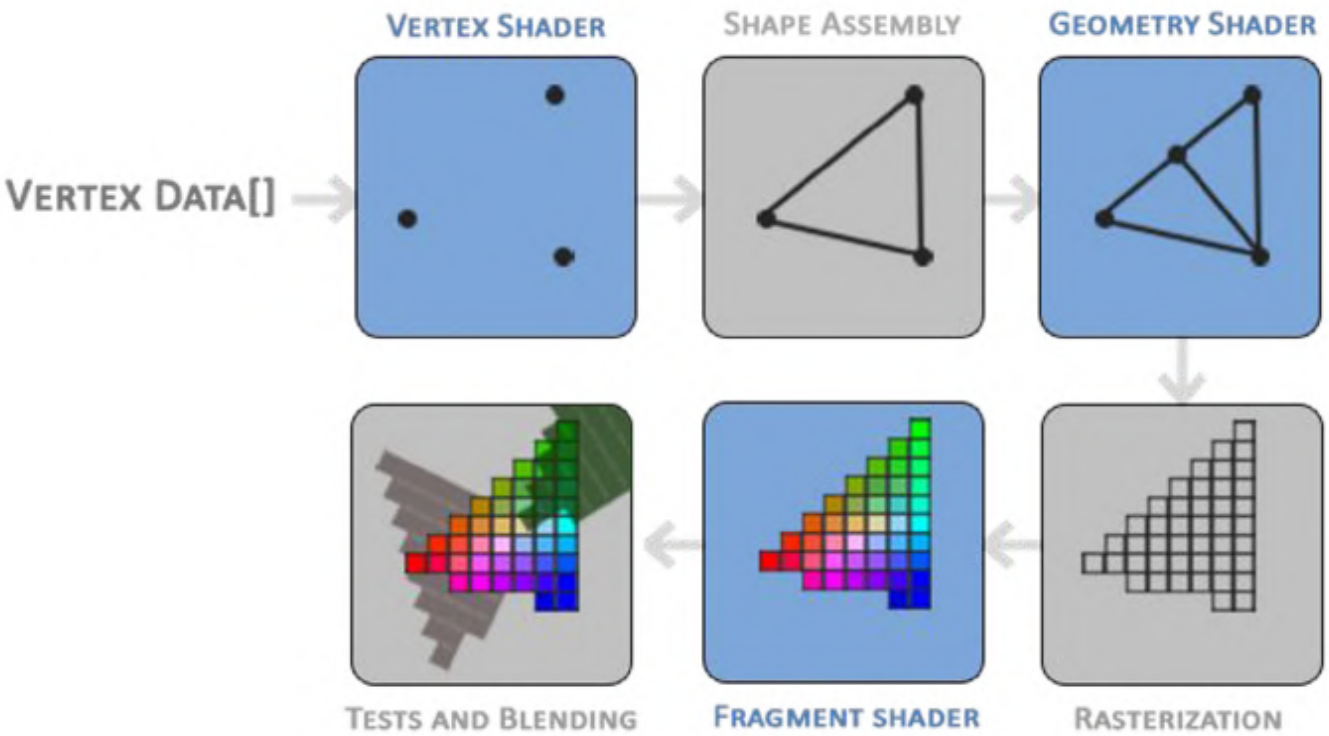
VAO (Vertex Array Object)

```
1  // Identifica el vertex buffer
2  GLuint vertexbuffer;
3
4  // Genera 1 buffer, pone el identificador resultante en vertexbuffer
5  glGenBuffers(1, &vertexbuffer);
6
7  // Los siguientes comandos se comunicarán con 'vertexbuffer'
8  glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
9
10 // Da los vértices a OpenGL
11 glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
12 g_vertex_buffer_data, GL_STATIC_DRAW);
```

VAO (Vertex Array Object)

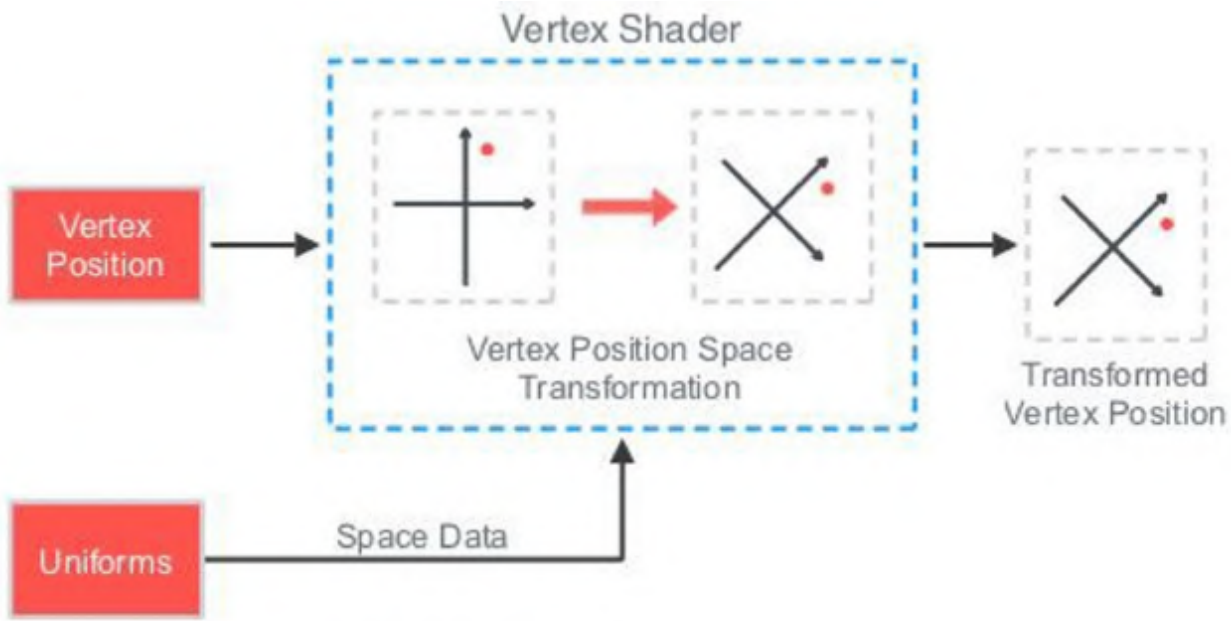
```
1  // 1º atributo del buffer: vértices
2  glEnableVertexAttribArray(0);
3  glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
4  glVertexAttribPointer(
5     0, // Atributo Nº0
6     3, // Tamaño
7     GL_FLOAT, // Tipo
8     GL_FALSE, // Normalizado?
9     0, // Stride
10    (void*)0 // Array buffer offset
11 );
12
13 // Dibuja el triángulo
14 glDrawArrays(GL_TRIANGLES, 0, 3);
15
16 // Partiendo desde el vértice 0, 3 vértices totales -> 1 triángulo
17 glDisableVertexAttribArray(0);
```

Volvamos al pipeline



Volvamos al pipeline

- El shader se ejecutará por cada vértice.
- Los shaders se programan con el lenguaje GLSL.



Vertex Shader – Código del Shader

```
#version 330 core

// Input vertex data, diferente por cada ejecución de este shader
layout(location = 0) in vec3 vertexPosition_modelspace;

void main() {
   gl_Position.xyz = vertexPosition_modelspace;
   gl_Position.w = 1.0;
}
```

Fragment Shader

```
1 #version 330 core
2
3 out vec3 color;
4 void main() {
5   color = vec3(1, 0, 0);
6 }
```

Vertex Shader – Carga

```
1  // Crea el shader
2  GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
3
4  // Lee el código desde un archivo
5  std::string VertexShaderCode;
6  std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
7  if (VertexShaderStream.is_open()) {
8     std::stringstream sstr;
9     sstr << VertexShaderStream.rdbuf();
10    VertexShaderCode = sstr.str();
11    VertexShaderStream.close();
12 }
```

Vertex Shader – Compilación

```
1 // Compila el shader
2 printf("Compiling shader : %s\n", vertex_file_path);
3 char const * VertexSourcePointer = VertexShaderCode.c_str();
4 glShaderSource(VertexShaderID, 1, &VertexSourcePointer, NULL);
5 glCompileShader(VertexShaderID);
```

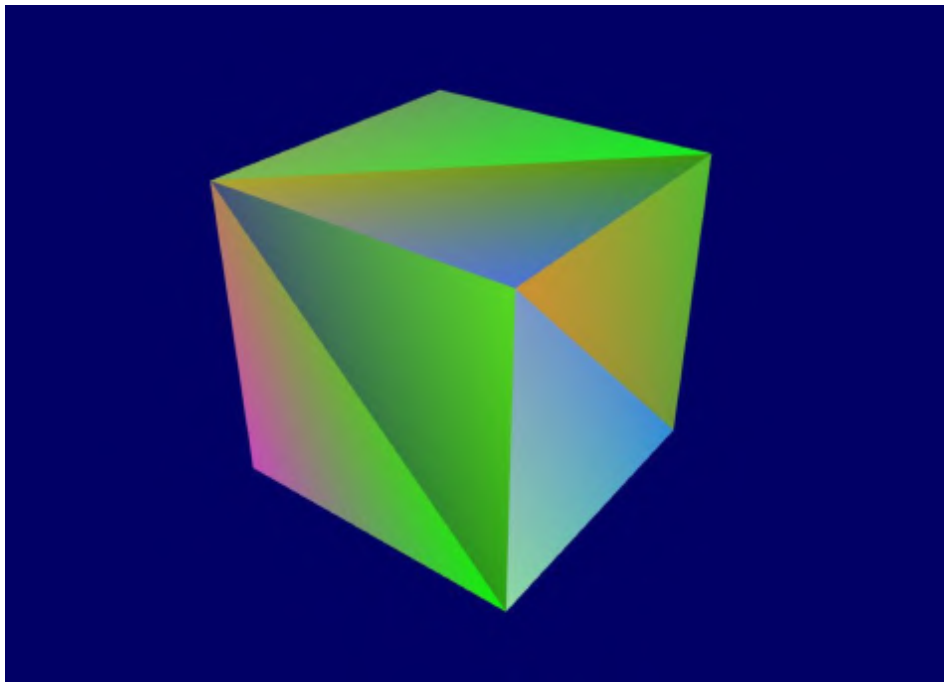
Vertex Shader – Chequeo

```
1  // Chequeo del shader
2  glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
3  glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
4
5  if (InfoLogLength > 0) {
6     std::vector<char> VertexShaderErrorMessage(InfoLogLength + 1);
7     glGetShaderInfoLog(
8        VertexShaderID,
9        InfoLogLength,
10       NULL,
11       &VertexShaderErrorMessage[0]
12    );
13    printf("%s\n", &VertexShaderErrorMessage[0]);
14 }
```

Vertex Shader – Link

```
1 // Link the program
2 printf("Linking program\n");
3 GLuint ProgramID = glCreateProgram();
4 glAttachShader(ProgramID, VertexShaderID);
5 glLinkProgram(ProgramID);
```

Cubo



Nuevo buffer con colores

```
1  GLuint colorbuffer;
2  glGenBuffers(1, &colorbuffer);
3  glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
4  glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data),
       ↪ g_color_buffer_data,
5  GL_STATIC_DRAW);
6  // 2nd attribute buffer : colores
7  glEnableVertexAttribArray(1);
8  glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
9  glVertexAttribPointer(
10   1, // Atributo N°1
11   shader.
12   3, // size
13   GL_FLOAT, // type
14   GL_FALSE, // normalized?
15   0, // stride
16   (void*)0 // array buffer offset
17 );
```

Usar Z-Búffer

```
1 // Activamos depth test
2 glEnable(GL_DEPTH_TEST);
3
4 // Activa profundidad entre la cámara y las superficies dibujadas con el
      ↪ fragment
5 glDepthFunc(GL_LESS);
6
7 // Borra la pantalla
8 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Vertex Shader – Cubo

```
#version 330 core

// Input del vertex, recordar las ubicaciones (0:posición, 1:colores)
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexColor;

// Output data, lo que recibirá el fragment shader
out vec3 fragmentColor;

// Valores constantes para el mesh
uniform mat4 MVP;

void main() {
  // Posición del vértice en la ventana de clipping
  gl_Position = MVP * vec4(vertexPosition_modelspace, 1);

  // El color de cada vértice será interpolado para producir el color en el fragment
  fragmentColor = vertexColor;
}
```

Fragment Shader – Cubo

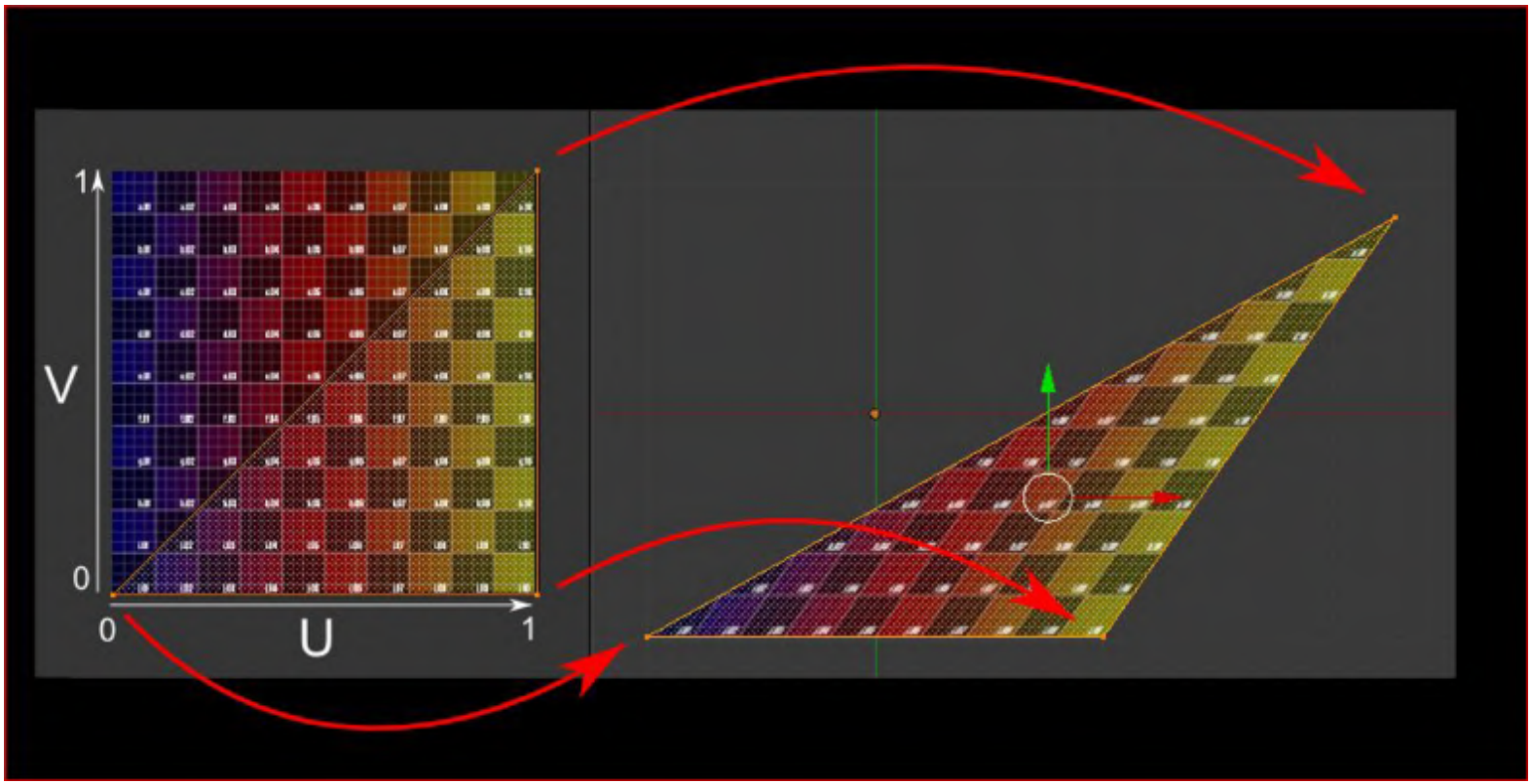
```
#version 330 core

// Color interpolado desde el vertex shader
in vec3 fragmentColor;

// Ouput data
out vec3 color;

void main() {
   // Color (output), se interpolará por los tres vértices de cada cara
   color = fragmentColor;
}
```

Cómo cargar un modelo - OBJ

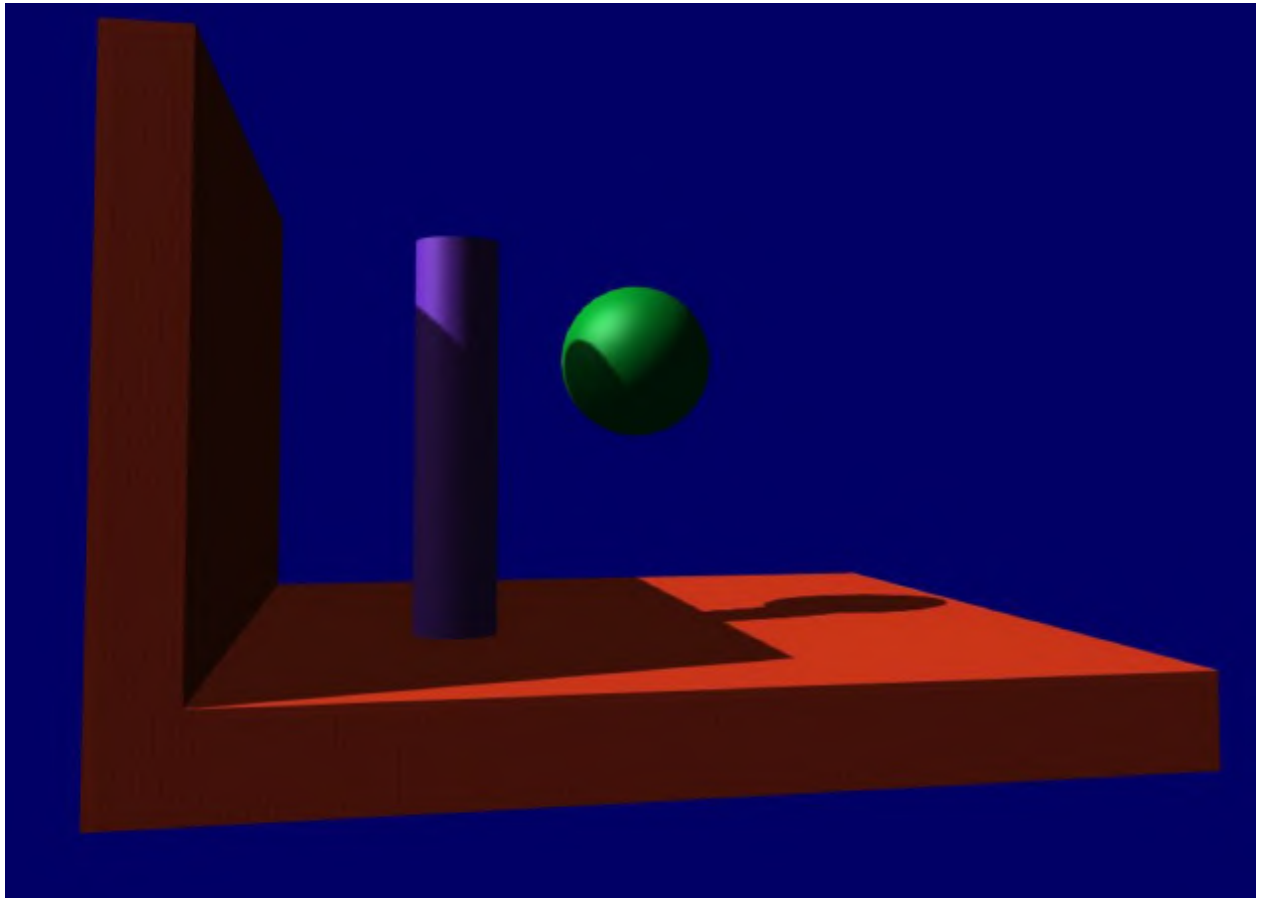
Cómo cargar un modelo - OBJ

```
# List of geometric vertices, with(x, y, z[, w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in(u, v[, w]) coordinates, these will vary between 0 and 1, w is optional and defaults to 0.
vt 0.500 1[0]
vt ...
...
# List of vertex normals in(x, y, z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
# Parameter space vertices in(u[, v ][, w]) form; free form geometry statement(see below)
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element(see below )
f 1 2 3
f 3 / 1 4 / 2 5 / 3
f 6 / 4 / 1 3 / 5 / 3 7 / 6 / 5
f 7//1 8//2 9//3
f ...
...
# Line element(see below)
l 5 8 1 2 4 9
```

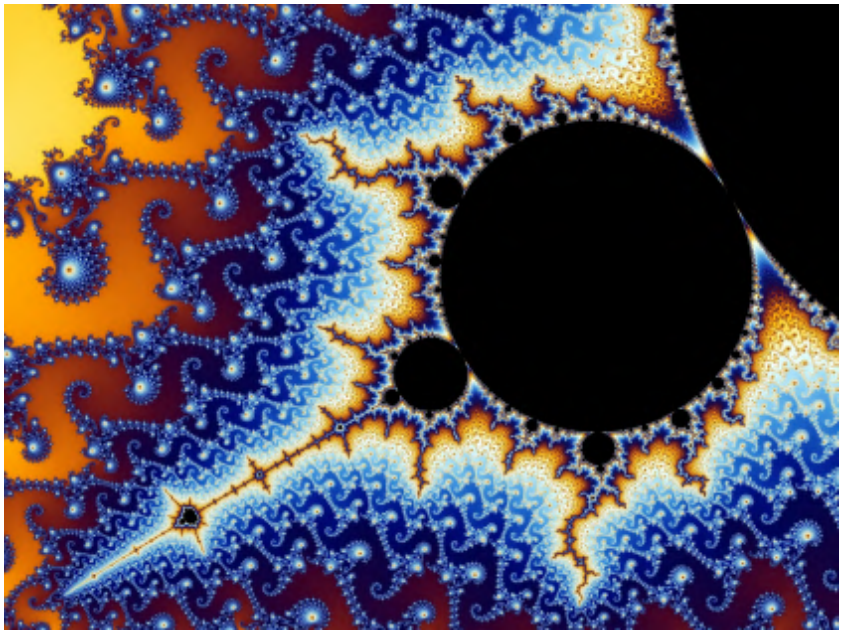
Otros conceptos – Coordenadas UV



Ejemplos

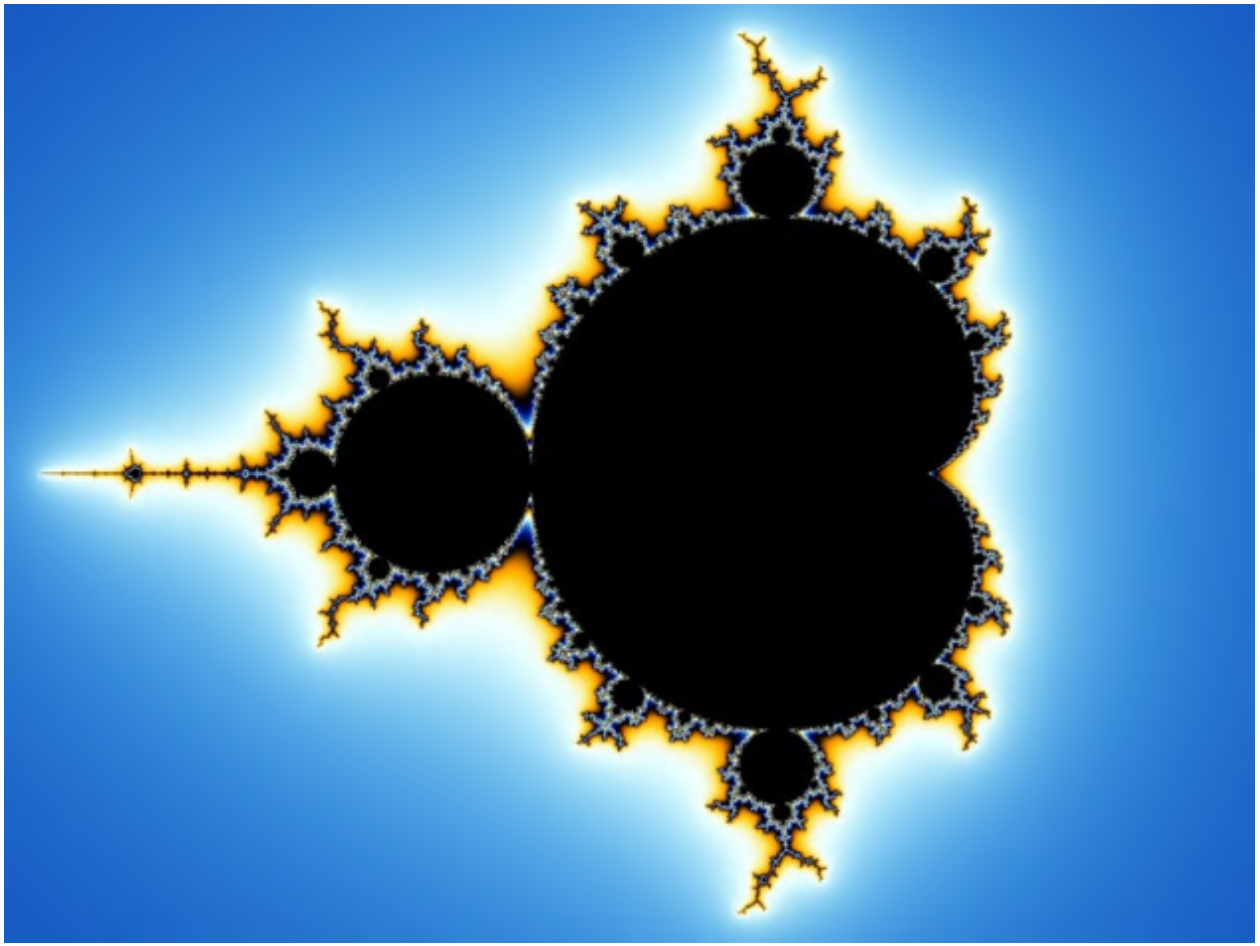


Ejemplos



Ejemplos – Tarea N°2 2018

Construir un fractal utilizando shaders:



Ejemplos – Tarea N°2 2018

- Construir un fractal utilizando shaders

Ejemplos – Tarea N°2 2018

- Construir un fractal utilizando shaders
- ¿Ideas?

Ejemplos – Tarea N°2 2018

- Construir un fractal utilizando shaders
- ¿Ideas?
- Funcionamiento de un fractal: Ecuación de recurrencia sobre un plano

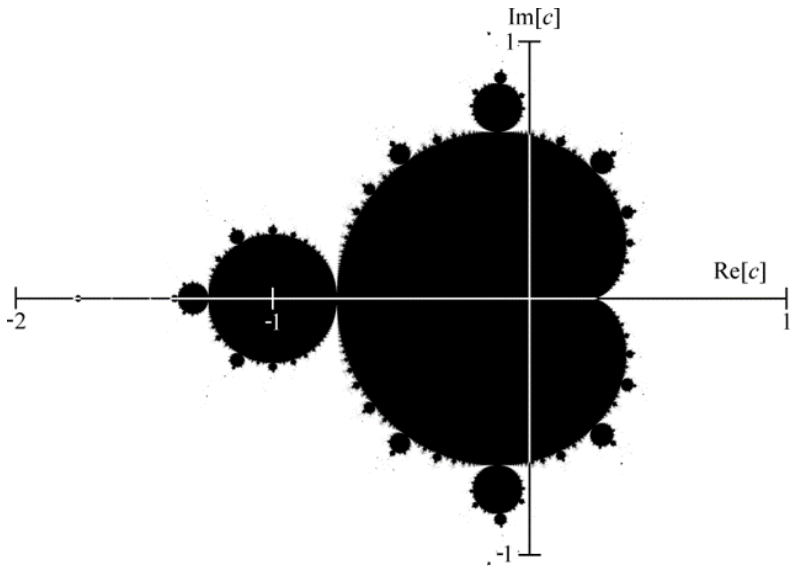
$$\begin{cases} ll z_0 = 0 \in \mathbb{C} & \text{(término inicial)} \\ z_{n+1} = {z_n}^2 + c & \text{(sucesión recursiva)} \end{cases}$$

Ejemplos – Tarea N°2 2018

- Construir un fractal utilizando shaders
- ¿Ideas?
- Funcionamiento de un fractal: Ecuación de recurrencia sobre un plano

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

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



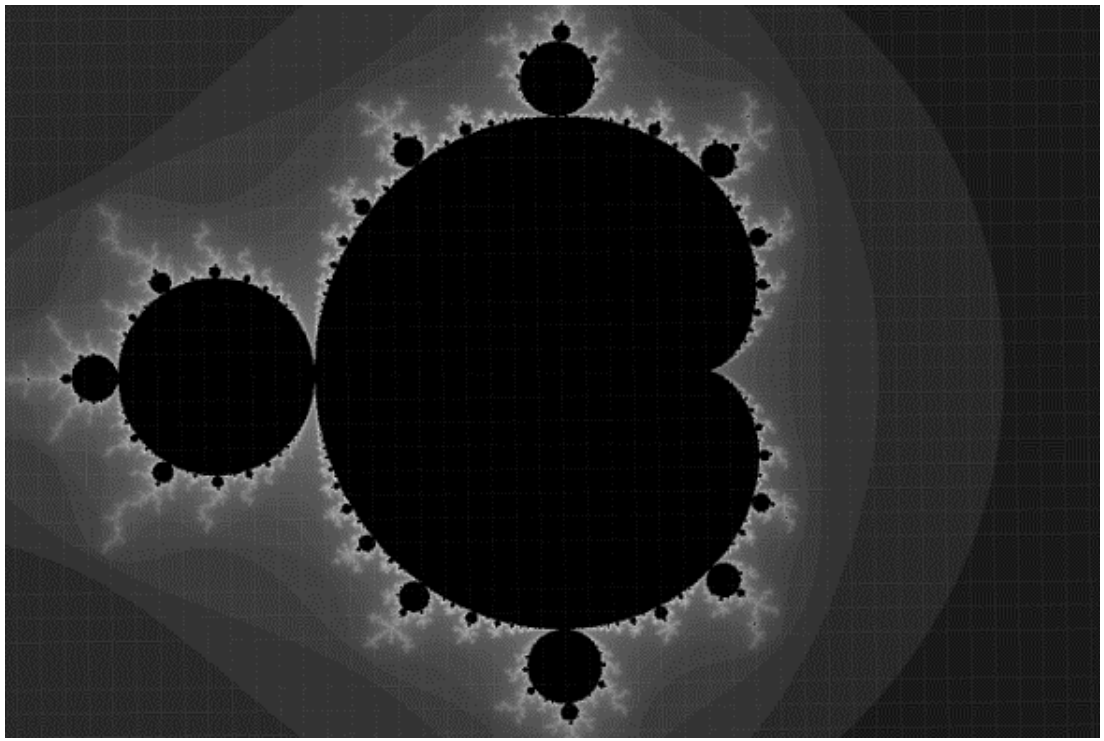
Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros vértices?

Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros vértices?

R: Modelar un plano cartesiano como una grilla (plano), cada intersección de la grilla es un vértice



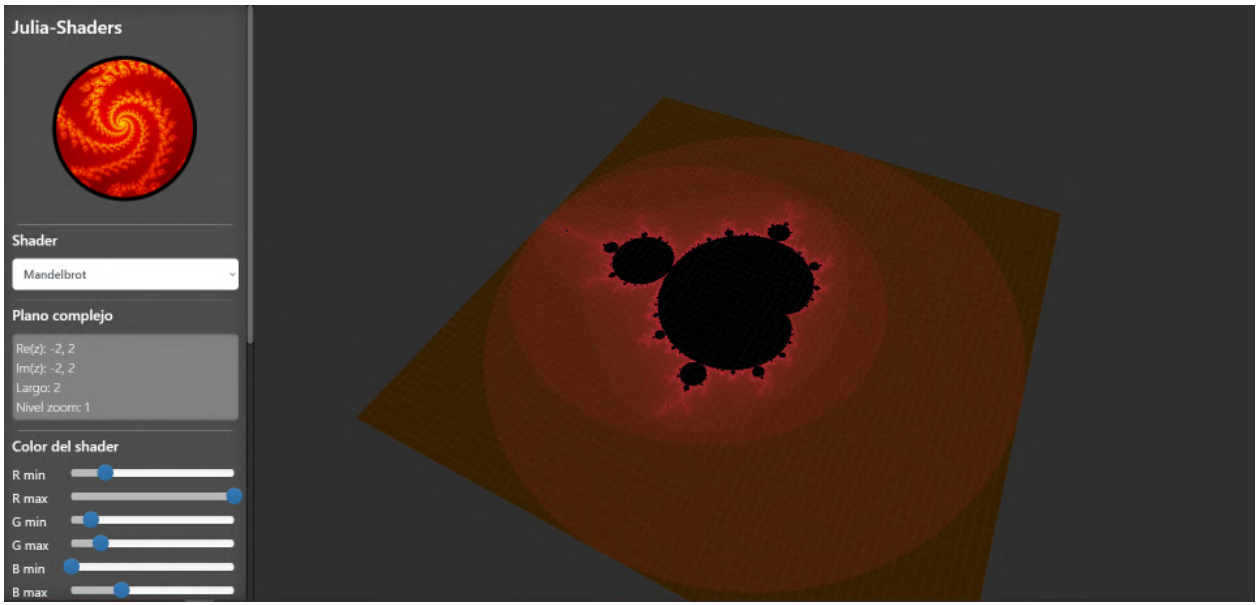
Ejemplos – Tarea N°2 2018

- ¿Cuáles son nuestros vértices?
- ¿Cómo modelamos los colores?

Ejemplos – Tarea N°2 2018

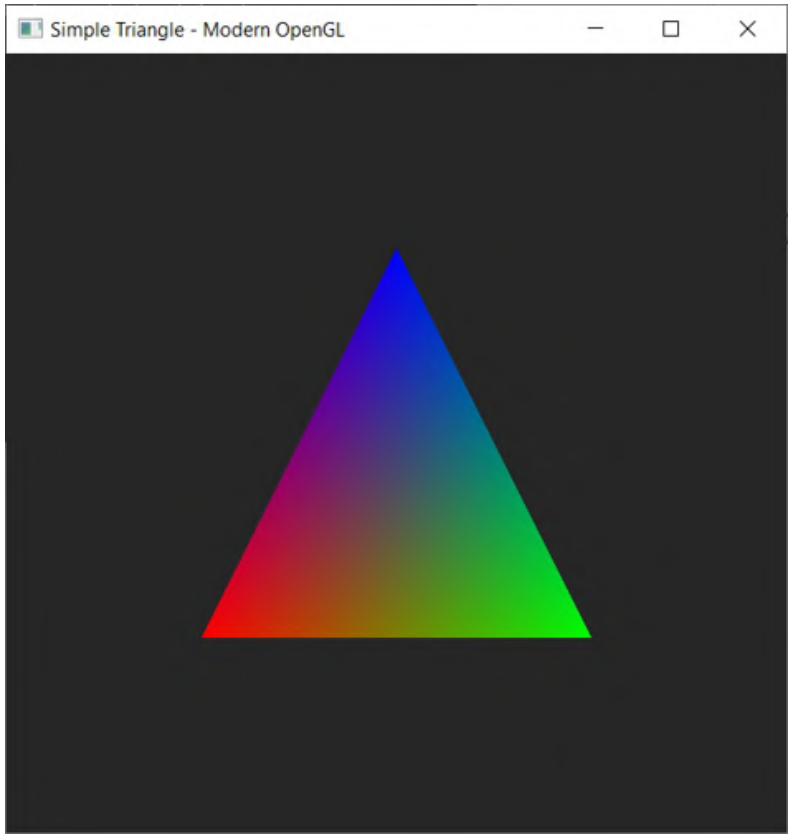
- ¿Cuáles son nuestros vértices?
- ¿Cómo modelamos los colores?
  R: Fragment shader

Ejemplos – Tarea N°2 2018



https://github.ppizarror.com/julia-shaders-threejs/

Ejemplos – Multiplataforma – Ejemplo en Python



ex_triangle.py

Ejemplos – Multiplataforma – Ejemplo en Python

Mismo esquema:

1. Iniciar OpenGL
2. Crear la ventana
3. Definir la geometría u el objeto (En este caso un triángulo)
4. Definir los shaders
5. Crear el VAO
6. Compilar los shaders
7. Bucle del programa

Gracias por su atención