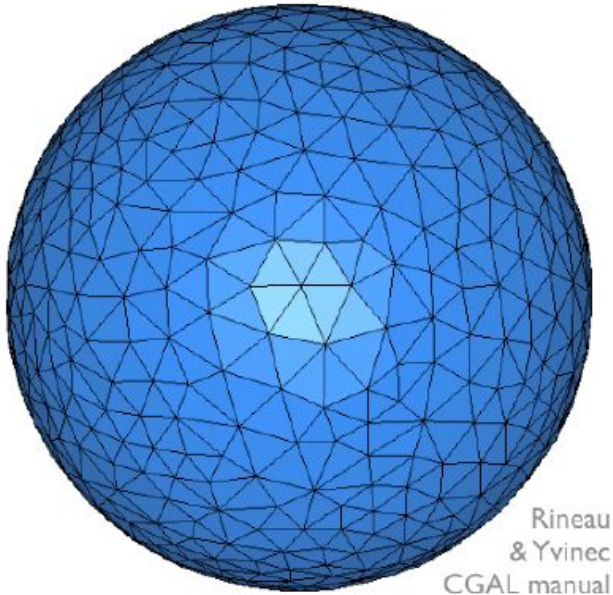
Introducción

Prof. Dr. Hans H. Ccacyahuillca Bejar

Diseño de una esfera

Andrzej Barabasz

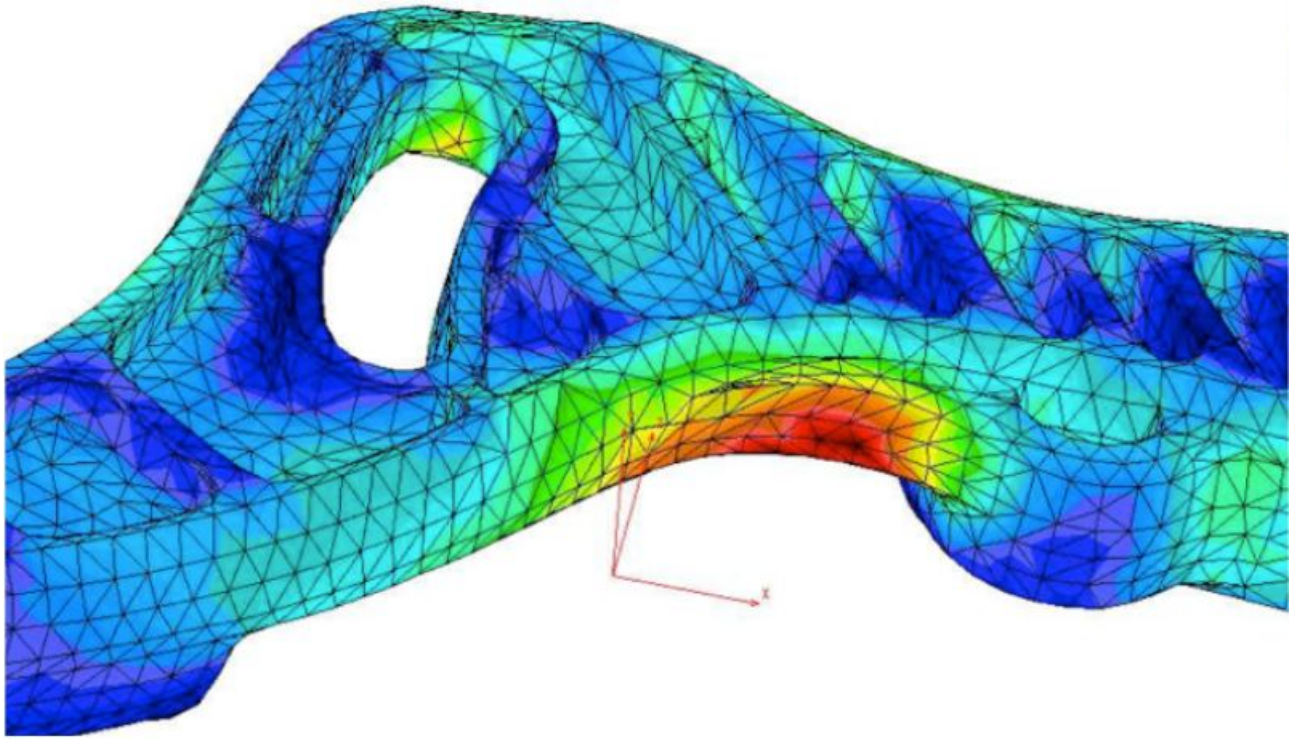
spheres



Rineau
& Yvinec
CGAL manual

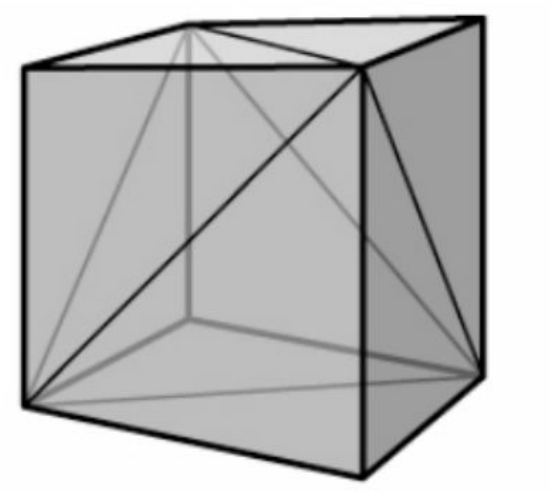
approximate sphere

Malha de triangulares



Triángulos en la mundo real

Una malla pequeña

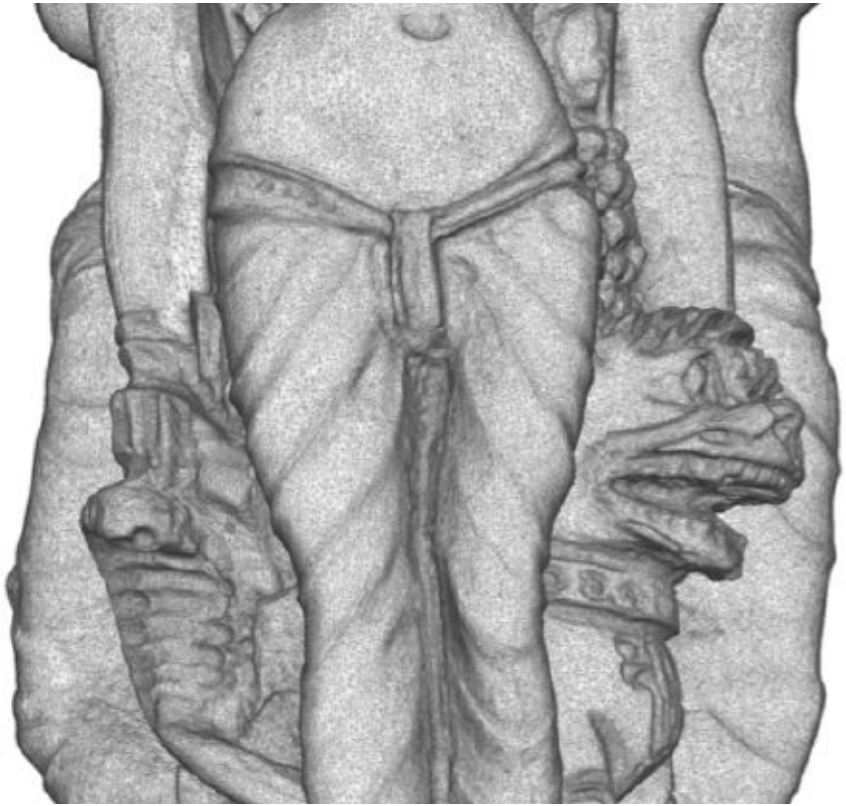


12 triangles, 8 vertices

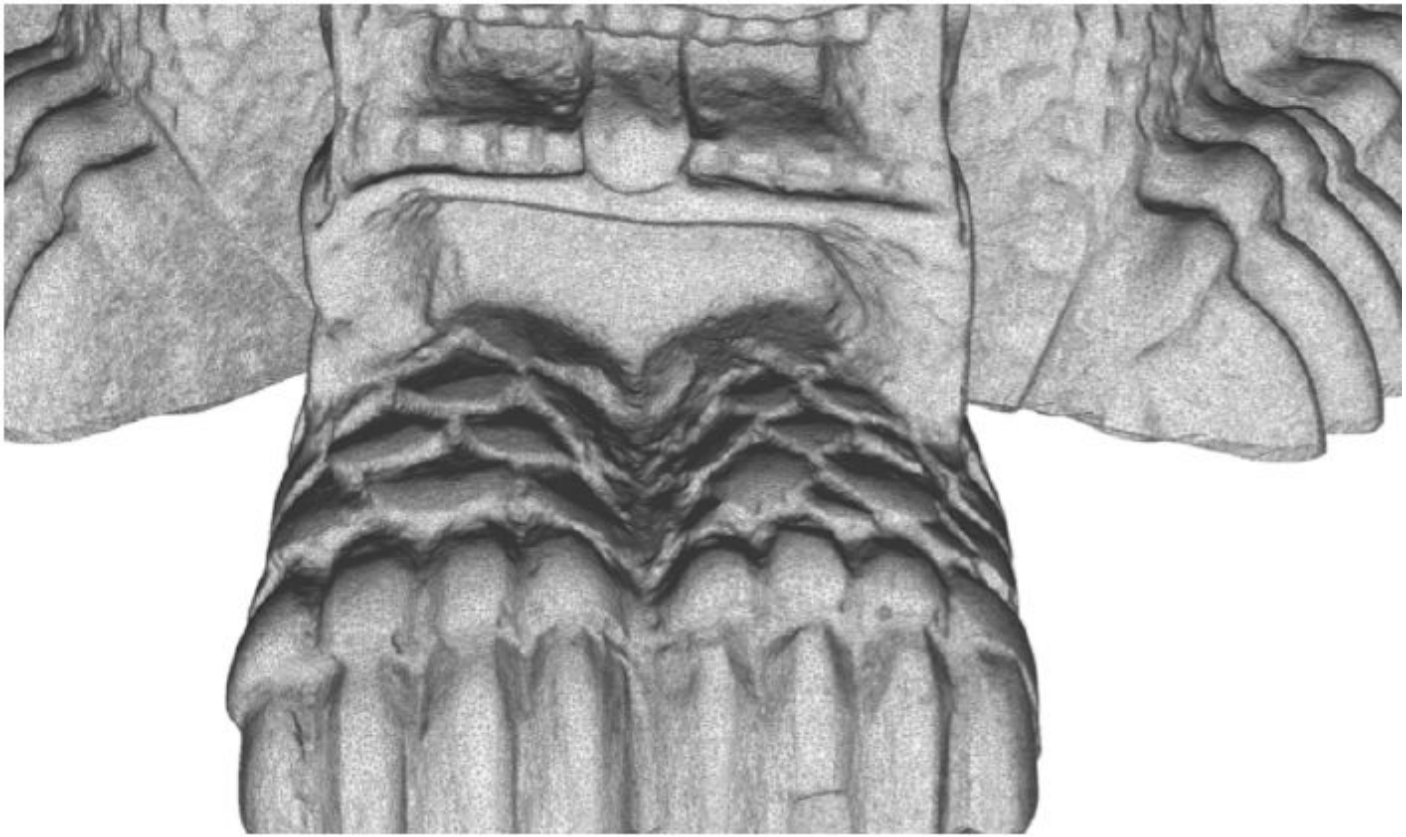
Una malla grande

- 10 millones de triángulos a partir de una adquisición 3D

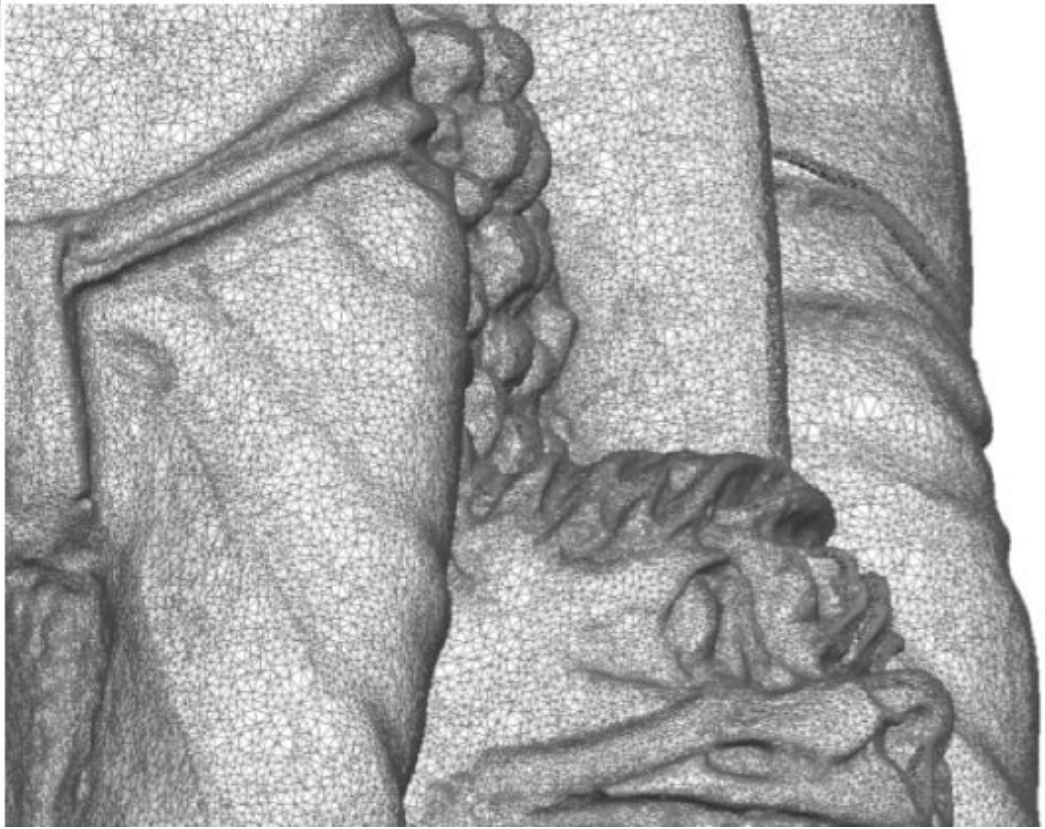
Malla de triángulos



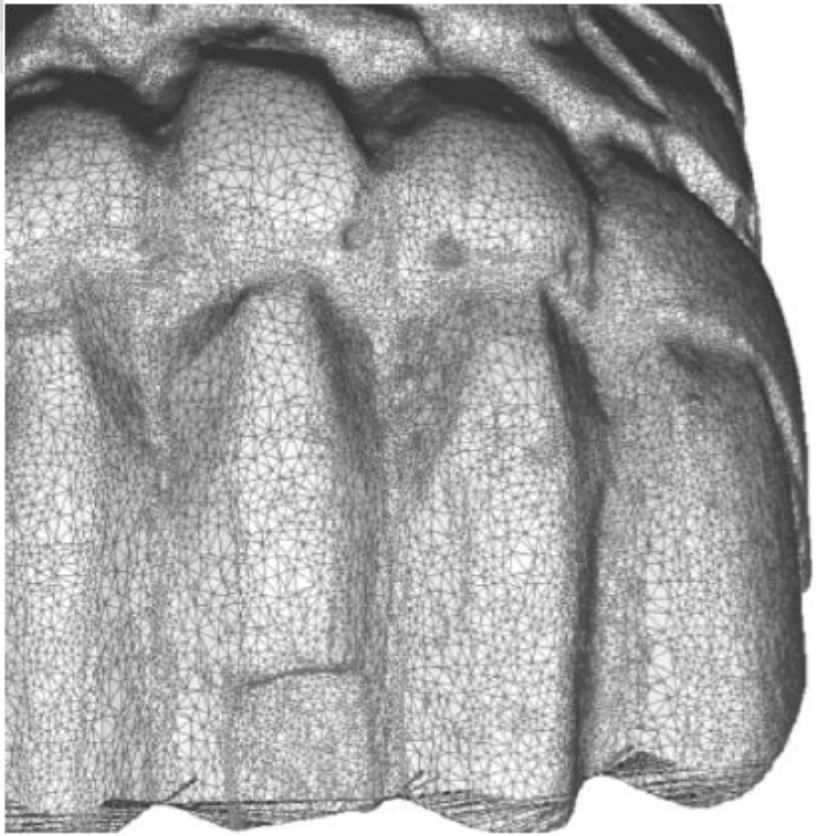
Malla de triángulos



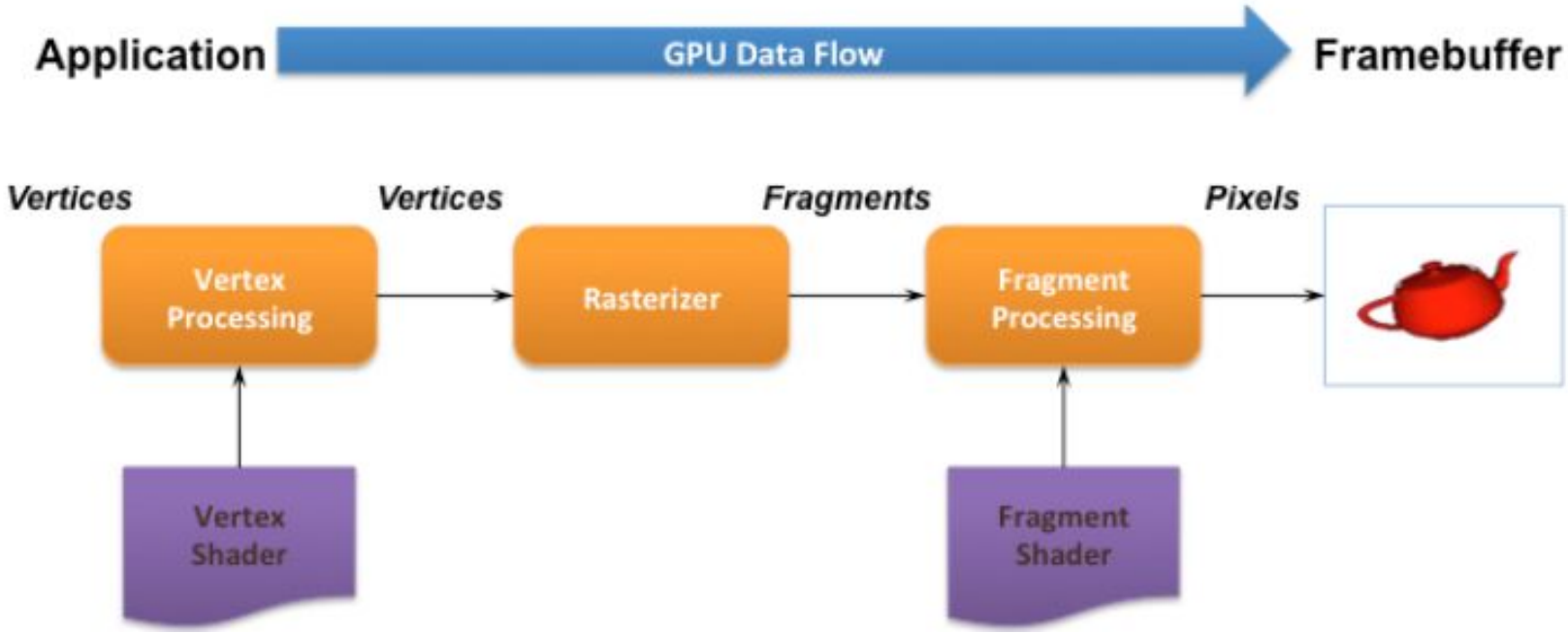
Malla de triángulos



Malla de triángulos



Pipeline (flujo) OpenGL



Aplicaciones del vertex shader

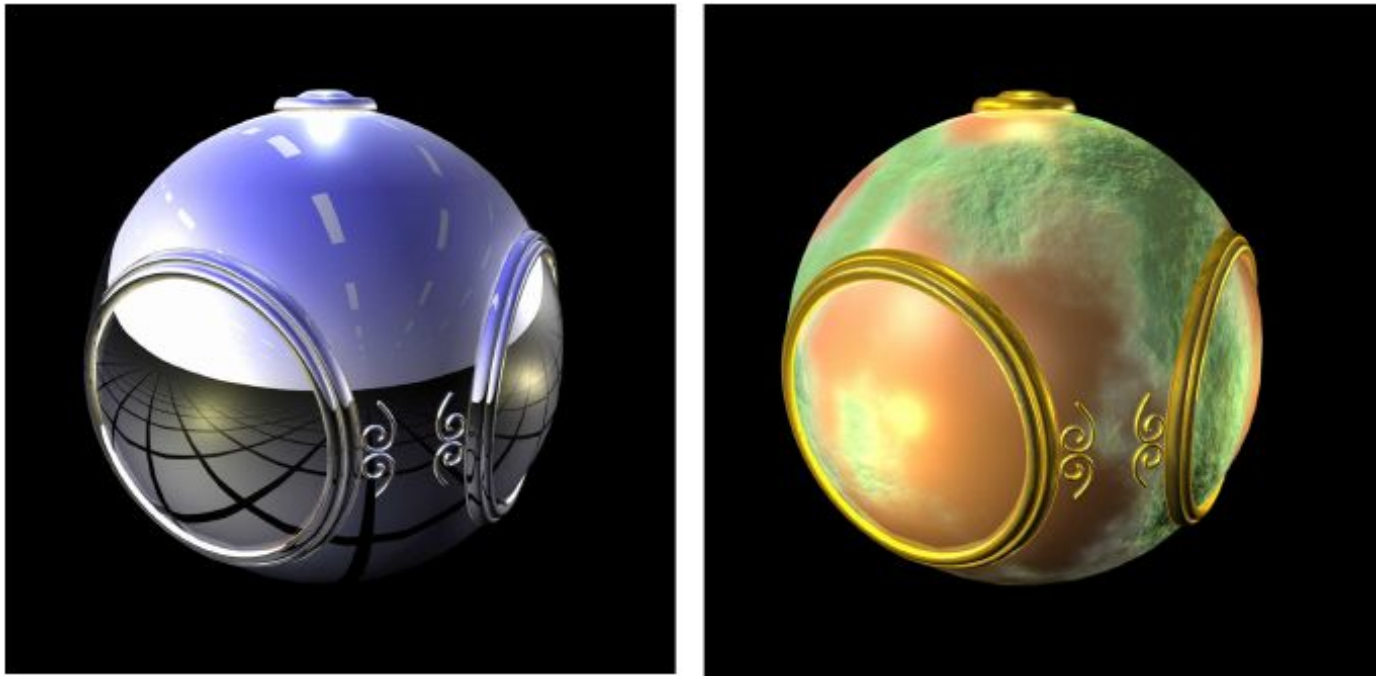
- **Movimiento** de vértices
  - Movimiento no lineal
- **Deformaciones**
  - Morphing
  - Generación de fractales
- **Iluminación**
  - Creación de modelos realistas
  - Shaders para diseños animados

Aplicaciones del fragment shader

- Cálculos de iluminación por fragmento

Aplicaciones del fragment shader

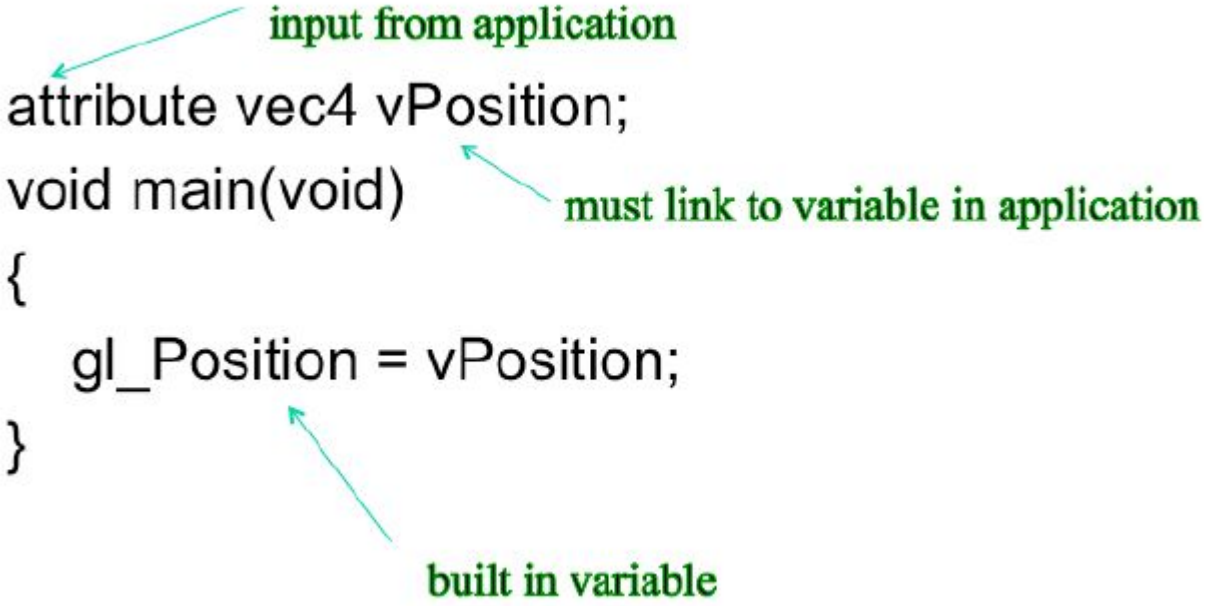
- Mapeamento de texturas



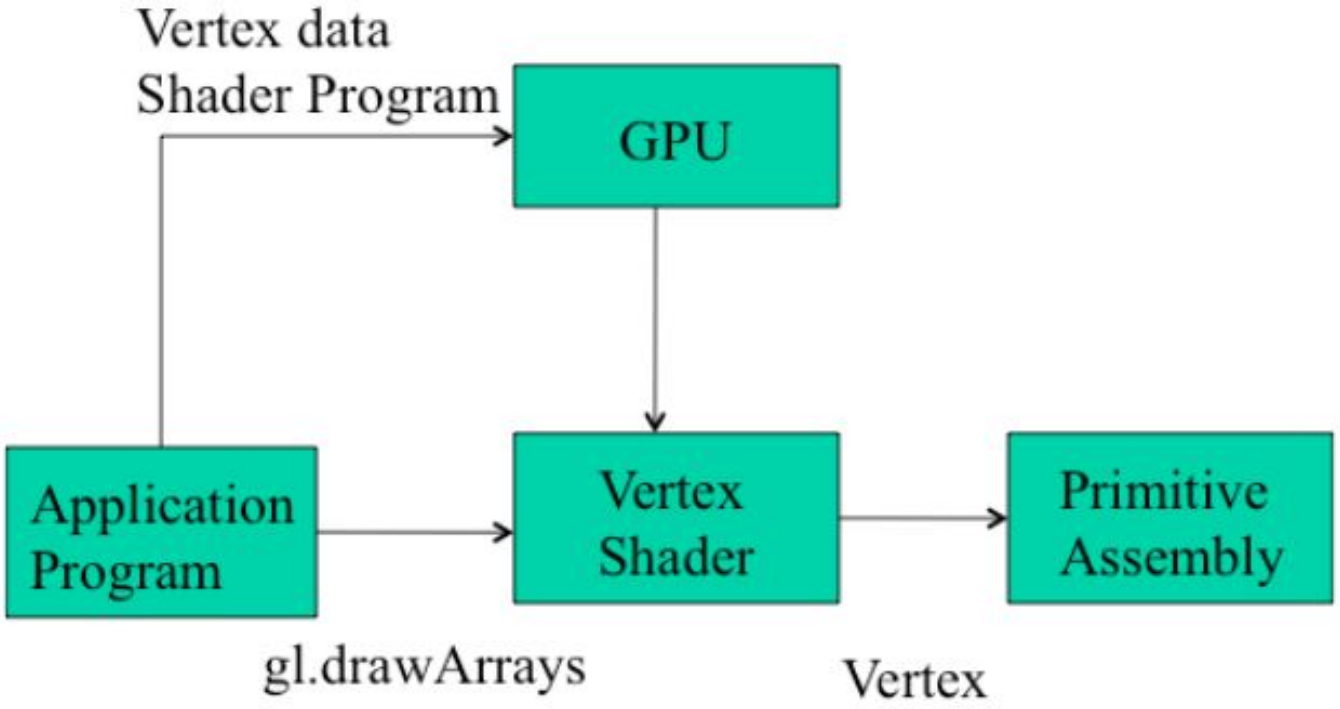
GLSL

- OpenGL Shading Language
  - Es un lenguaje de alto nivel parecido a “C”
  - tipo de datos:
    - vectores (arreglos)
    - Matrices
    - “Samplers” (texturas)
- A partir de OpenGL 3.1, una aplicación debe tener shaders

Un vertex shader simple



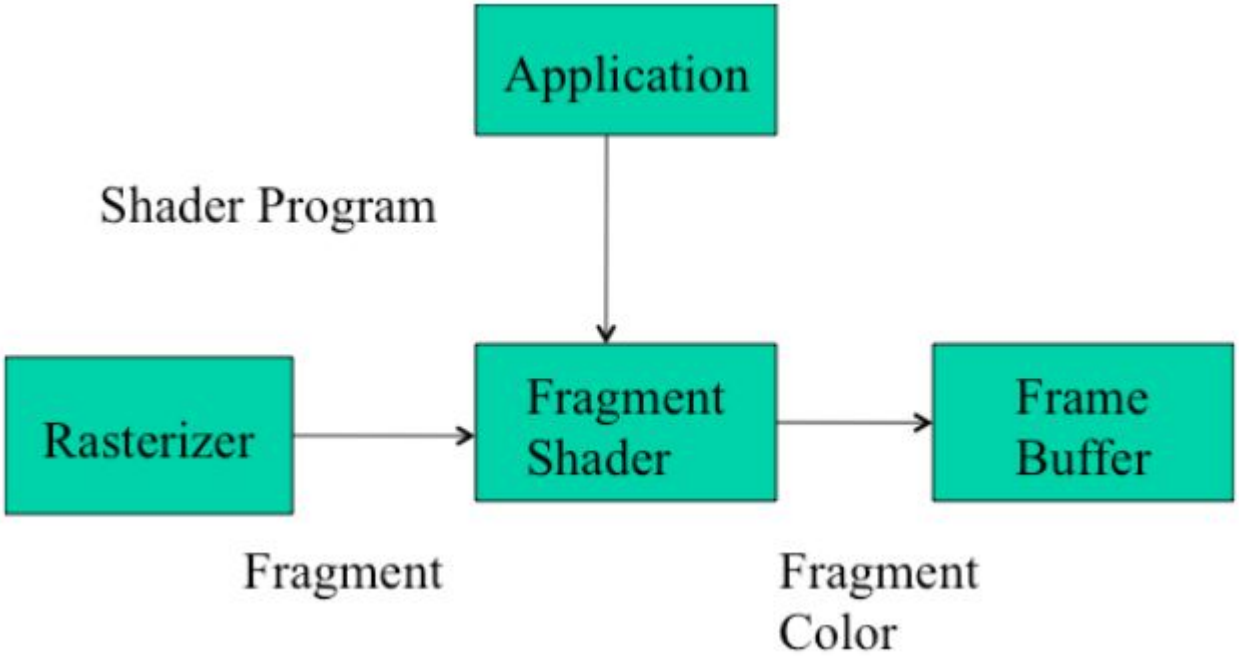
Modelo de ejecución (vertex shader)



Un fragmento de shader simple

```
precision mediump float;
void main(void)
{
   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Modelo de ejecución (fragment shader)



Tipos de datos

- **Tipos básicos:**
  - float, int, bool
- **Vectores:**
  - vec2, vec3, vec4,
  - ivec2, ivec3, ivec4
  - bvec2, bvec3, bvec4
- **Matrices (almacenadas columna a columna):**
- mat2, mat3, mat4
- **Constructores tipo C++**
- vec3 a = vec3(1.0, 2.0, 3.0)

Valores pasados por valor

- **Ausencia de punteros en GLSL**
- **Se puede usar *struct* y pueden ser devueltas en el retorno de las funciones**
- **Como matrices y vectores son tipos de datos básicos, pueden ser enviados a partir de funciones GLSL, e.g.**

```
mat3 func(mat3 a)
```

Calificadores (Tags)

- GLSL utiliza muchos de los Tags de C/C++, Java (e.g. const)
- Los valores de las variables pueden cambiar:
  - Una vez por vértice
  - Una vez por primitiva
  - Una vez por fragmento
  - A cualquier hora dentro de la aplicación
- Atributos de vértices son interpolados por el rasterizador para atributos de fragmento

Calificadores (Tags)

- **GLSL** utiliza muchos de los Tags de C/C++, Java (e.g. **const**)
- Los valores de las variables pueden cambiar:
  - Una vez por **vértice**
  - Una vez por **primitiva**
  - Una vez por **fragmento**
  - A cualquier hora **dentro** de la **aplicación**
- Atributos de vértices son interpolados por el rasterizador para atributos de fragmento

Calificadores (Tags) attribute

- Calificador que indica que valores pueden cambiar una vez por vértice.
- Existen algumas variáveis especiales como:

  gl_Position

- Definidas por el usuario

  attribute float temperature

  attribute vec3 velocity

- También, GLSL utiliza los calificadores `in` y `out` para enviar e devolver datos de ***shaders***.

Calificadores (tags) uniform

- Valores que conservan un valor constante durante el procesamiento de una primitiva.
- Pueden ser modificados en la aplicación y enviados a los shaders
    - No pueden ser modificados en los shaders
- Es usado para pasar información a los shaders (e.g. tiempo, bounding box, matrices de transformación, etc)

Calificadores (tags) varying

- Valores que son pasadas del vertex shader para el fragment shader
- Automáticamente interpolada por el rasterizador
- WebGL utiliza *`out`* en el vertex shader y *`in`* en el fragment shader
  - *`out vec4 color; // vertex shader`*
  - *`in vec4 color; // fragment shader`*

Ejemplo de vertex shader

```
attribute vec4 vColor;
varying vec4 fColor;
void main()
{
  gl_Position = vPosition;
  fColor = vColor;
}
```

Fragment shader correspondiente

```
precision mediump float;

varying vec3 fColor;
void main()
{
  gl_FragColor = fColor;
}
```

Enviando atributos

```
var cBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer );
gl.bufferData( gl.ARRAY_BUFFER, flatten(colors),
                                gl.STATIC_DRAW );

var vColor = gl.getAttribLocation( program, "vColor" );
gl.vertexAttribPointer( vColor, 3, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vColor );
```

Enviando una variable uniform

```
// in application

vec4 color = vec4(1.0, 0.0, 0.0, 1.0);
colorLoc = gl.getUniformLocation( program, "color" );
gl.uniform4f( colorLoc, color);

// in fragment shader (similar in vertex shader)

uniform vec4 color;

void main()
{
    gl_FragColor = color;
}
```

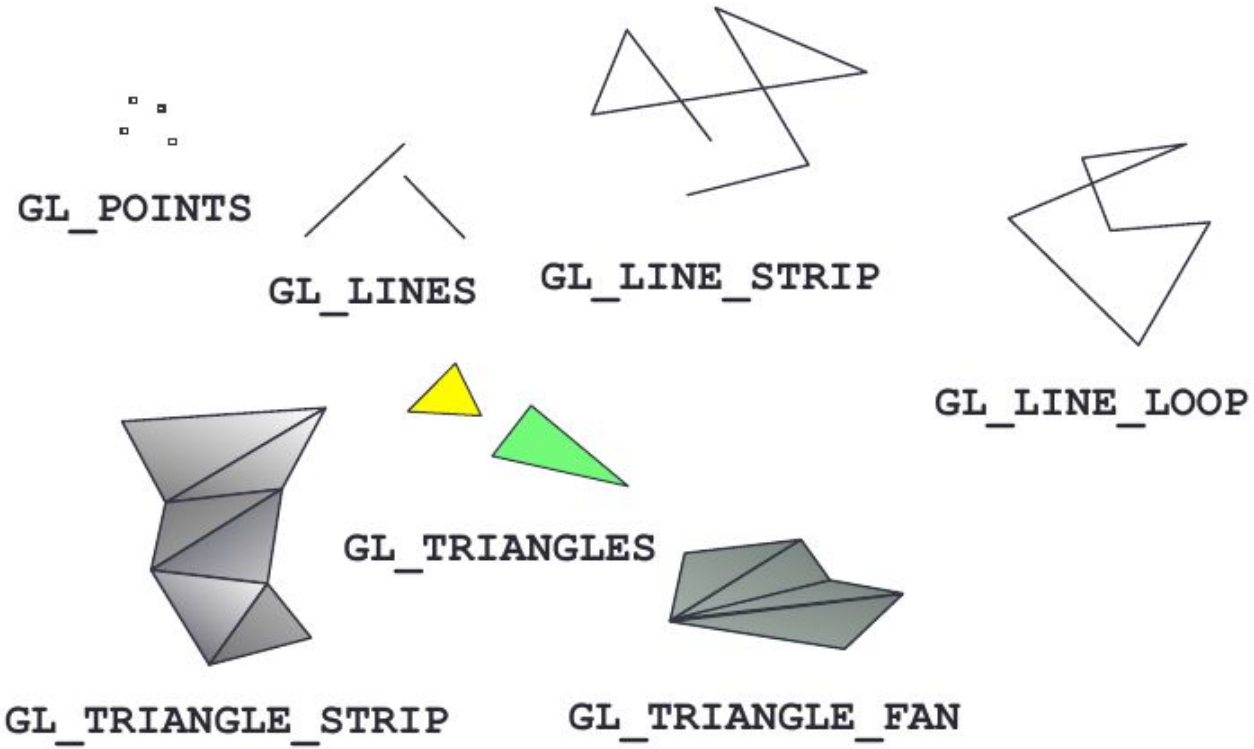
Operadores y funciones

- **Funciones matemáticas**
  - **Trigonométricas**: `sin(), cos(), asin(),tan(),` etc
  - **Aritméticas**: `sqrt(), power(), abs()`
  - **Gráficas**: `reflect(), length()`
- **Sobrecarga de operadores**
  - `mat4 a;`
  - `vec4 b, c, d;`
  - `c = b*a; // vector columna como un array 1D`
  - `d = a*b; // vector fila como un array 1D`

Swizzling

- **Elementos de arrays pueden ser referenciados usando [] el operador de seleción (.) con:**
  - `x, y, z, w`
  - `r, g, b, a`
  - `s, t, p, q`
  - `a[2], a.b, a.z y a.p` **denotan el mismo elemento**
- **El operador de swizzling nos permite manipular elementos:**
  - `vec4 a, b;`
  - `a.yz = vec2(1.0, 2.0);`
  - `b = a.yxzw;`
  - `a.xy = b.yx;`

Primitivas en WebGL

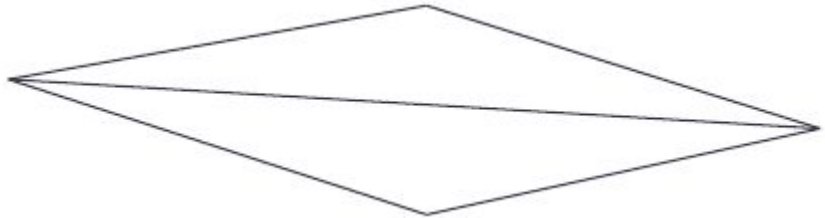


Polígonos

- OpenGL/WebGL solo diseña triángulos:
  - Simples: aristas no se cruzan
  - Convexos: todos los puntos en un segmento de recta entre dos puntos del polígono, también están dentro del polígono
  - Planares: todos los vértices están en el mismo plano
- La aplicación debe convertir un polígono en triángulos
  - OpenGL >= 4.1 disponibiliza shader para esa tarea

Buenos triángulos

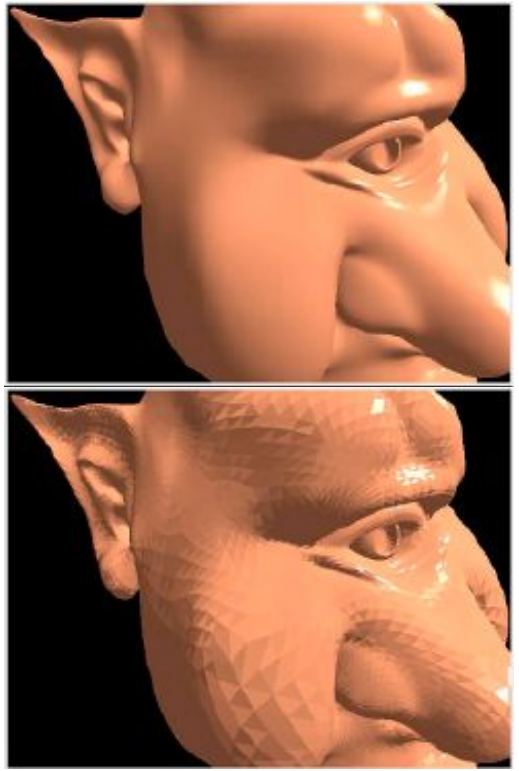
- La renderización de triángulos largos y delgados son pésimos



- Triángulos equiláteros renderizan bien

Tonificación - “shading”

- Tonificación suave
  - EL rasterizador interpola valores dos vértices a lo largo de los triángulos visibles (default)
- Mala tonificación
  - Color del primer vértice determina el color del relleno

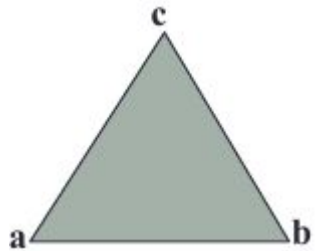


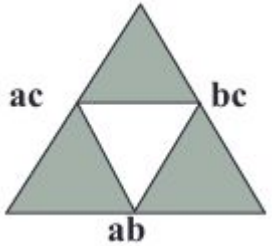
Atributos de colores

- Los colores se atribuyen al final en el fragment shader, sin embargo pueden ser definidas en cualquier shader o en la aplicación final.
- En la aplicación:
  - Se pasa el color al vertex shader usando una variable `uniform` o `attribute`
- En el vertex shader:
  - Se pasa el color al fragment shader usando una variable `varying`
- En el fragment shader:
  - Se puede alterarlo con el código del propio shader

Ejemplo: triángulo de Sierpinski

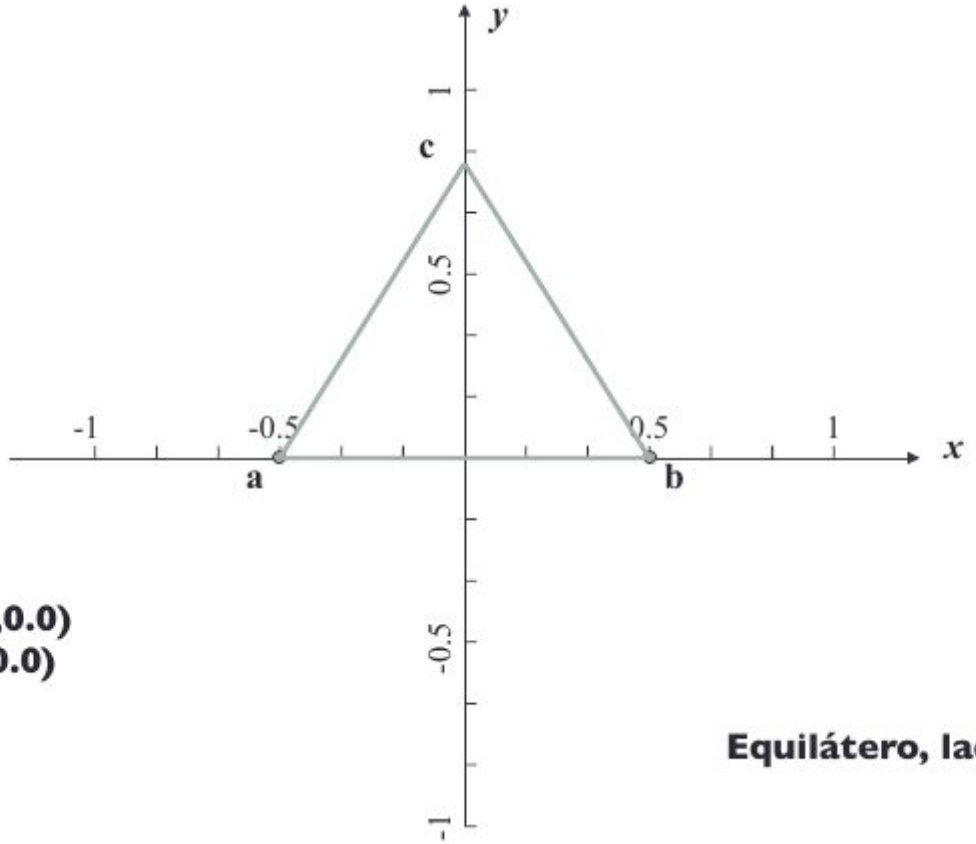
- Se inicia con un triángulo
- Se conecta bisectores de los lados y se elimina el triángulo central
- Se repite el proceso





Waclaw Sierpinski (1882 - 1969), matemático polonês

Coordenadas



a= (-0.5,0.0)
b= (0.5,0.0)
c=?

Equilátero, lado = 1cm

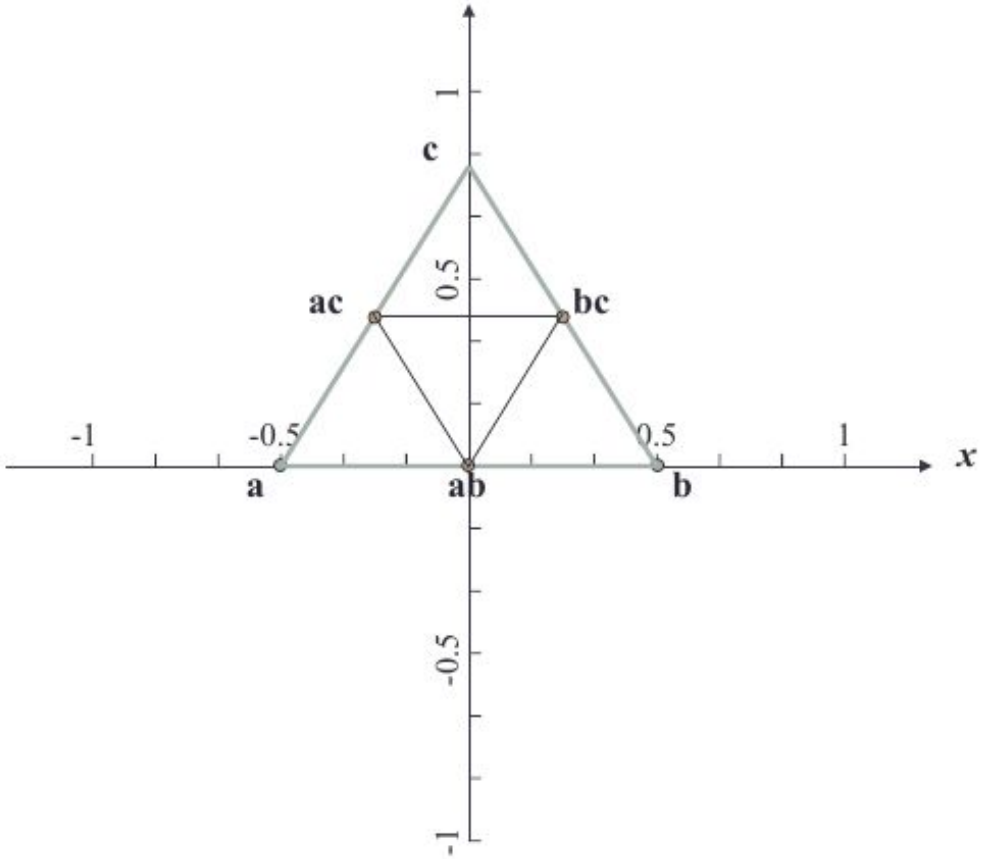
gasket2.js

```
var points = [];
var NumTimesToSubdivide = 5;

/* triângulo inicial */
var vertices = [
 vec2(-0.5, 0.0),
 vec2( 0.5, 0.0),
 vec2( 0.0, 0.866)
];

divideTriangle( vertices[0], vertices[1],
     vertices[2], NumTimesToSubdivide);
```

Dibuja el triángulo

```
function triangle(a, b, c) {
      points.push(a, b, c);
}
```

Subdivisión



ab= ?
ac= ?
bc= ?

Subdivisión

```
function divideTriangle(a, b, c, count) {

    // check for end of recursion
    if ( count === 0 ) {
       triangle( a, b, c );
    }
    else {
       // bisect the sides
       var ab = mix( a, b, 0.5 );
       var ac = mix( a, c, 0.5 );
       var bc = mix( b, c, 0.5 );
       --count;
       // three new triangles
       divideTriangle( a, ab, ac, count );
       divideTriangle( c, ac, bc, count );
       divideTriangle( b, bc, ab, count );
    }
}
```

Ejercicio

Crear un programa en WebGL que aproxime un círculo de radio unitario utilizando subdivisiones sucesiva de un cuadrado inscrito.