



# MAC420/5744: Introdução à Computação Gráfica

---

Marcel P. Jackowski  
[mjack@ime.usp.br](mailto:mjack@ime.usp.br)

Aula #3: Visualização e WebGL

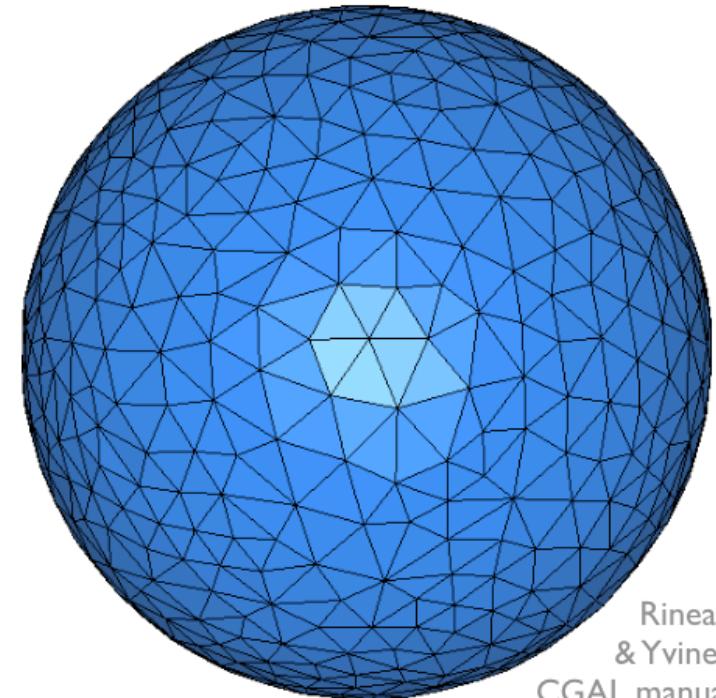
# Desenho de uma esfera

---



Andrzej Barabasz

**spheres**

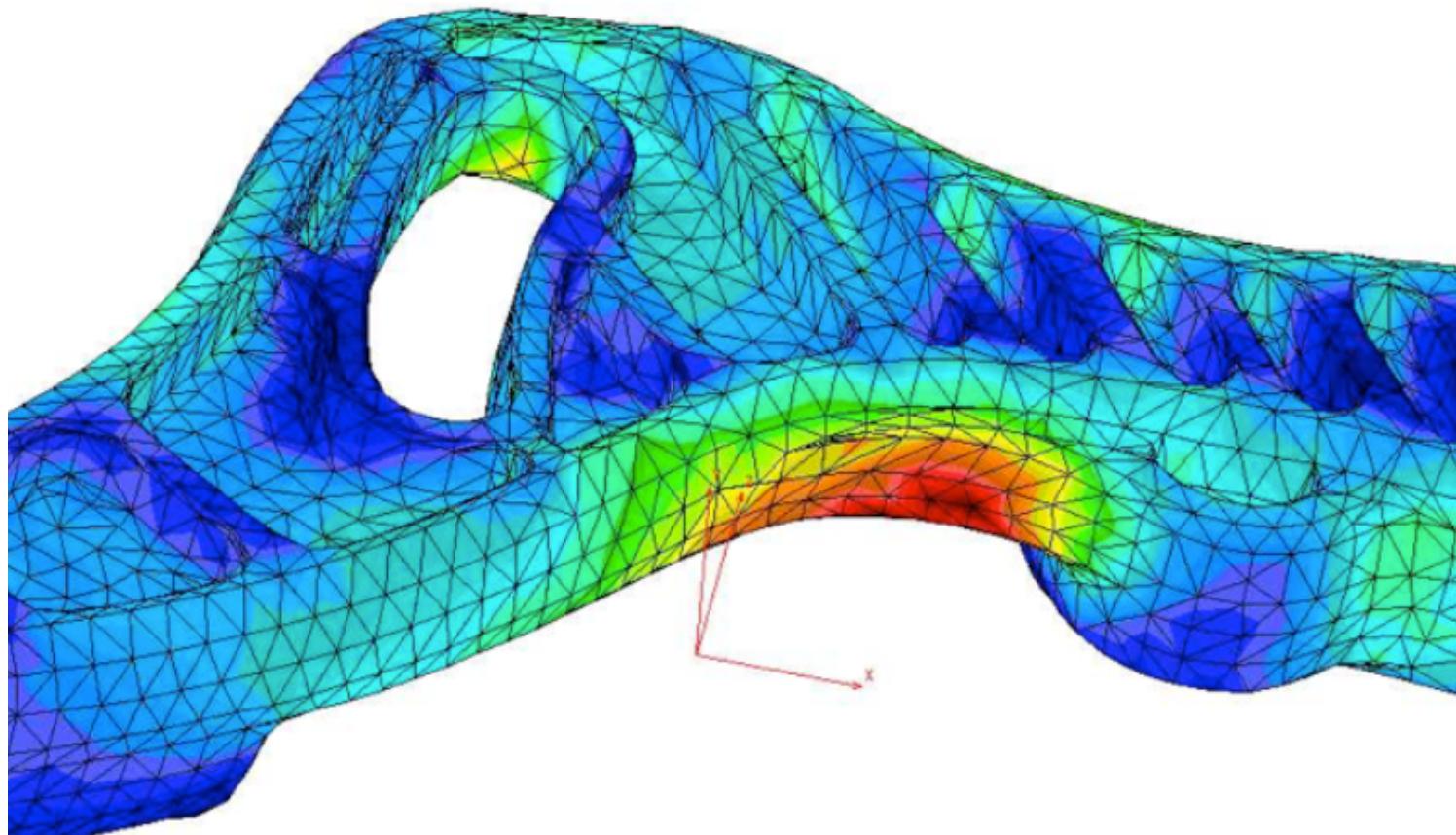


Rineau  
& Yvinec  
CGAL manual

**approximate  
sphere**

# Malha de triângulos

---

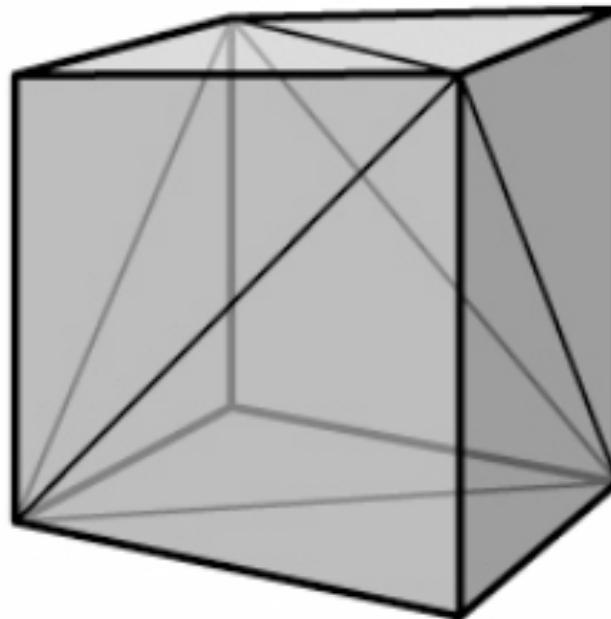


# Triângulos no mundo real



# Uma malha pequena

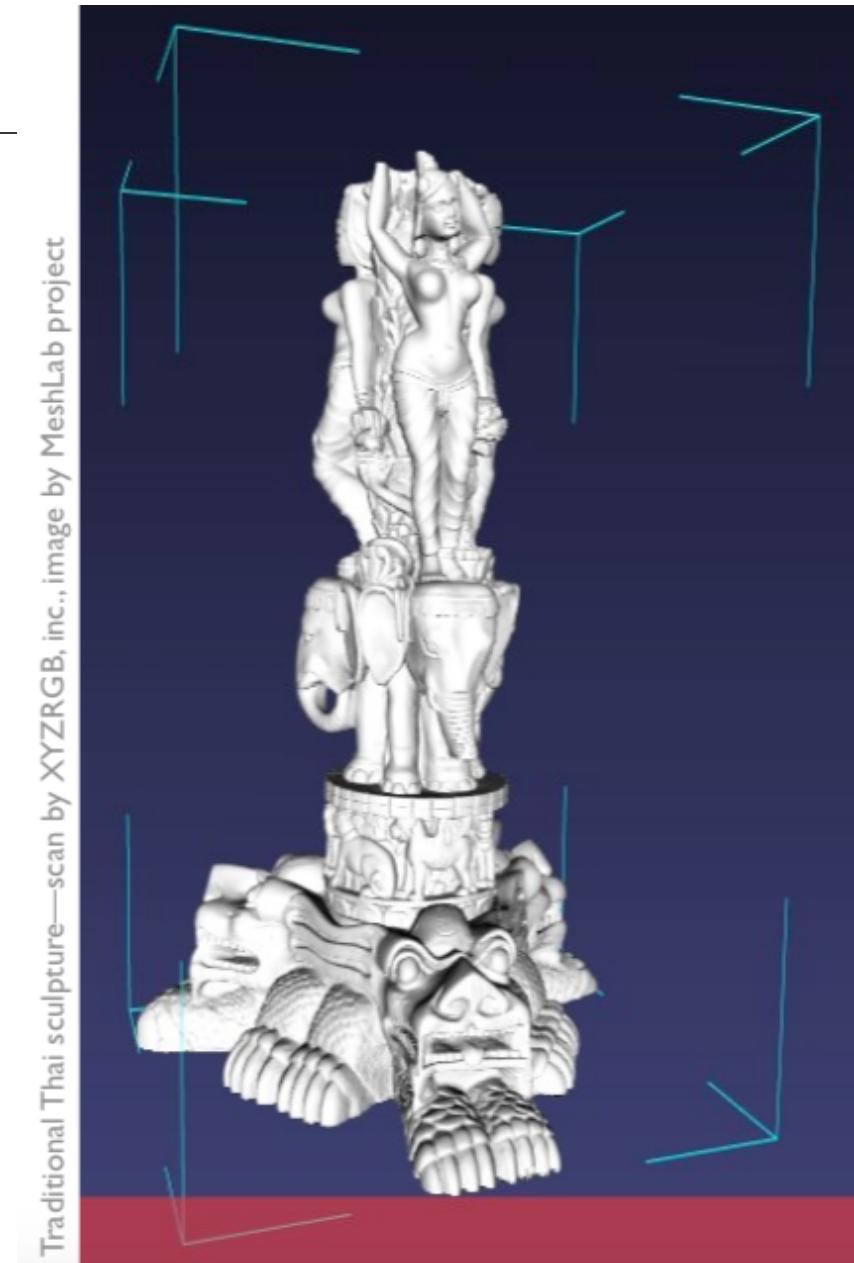
---



12 triangles, 8 vertices

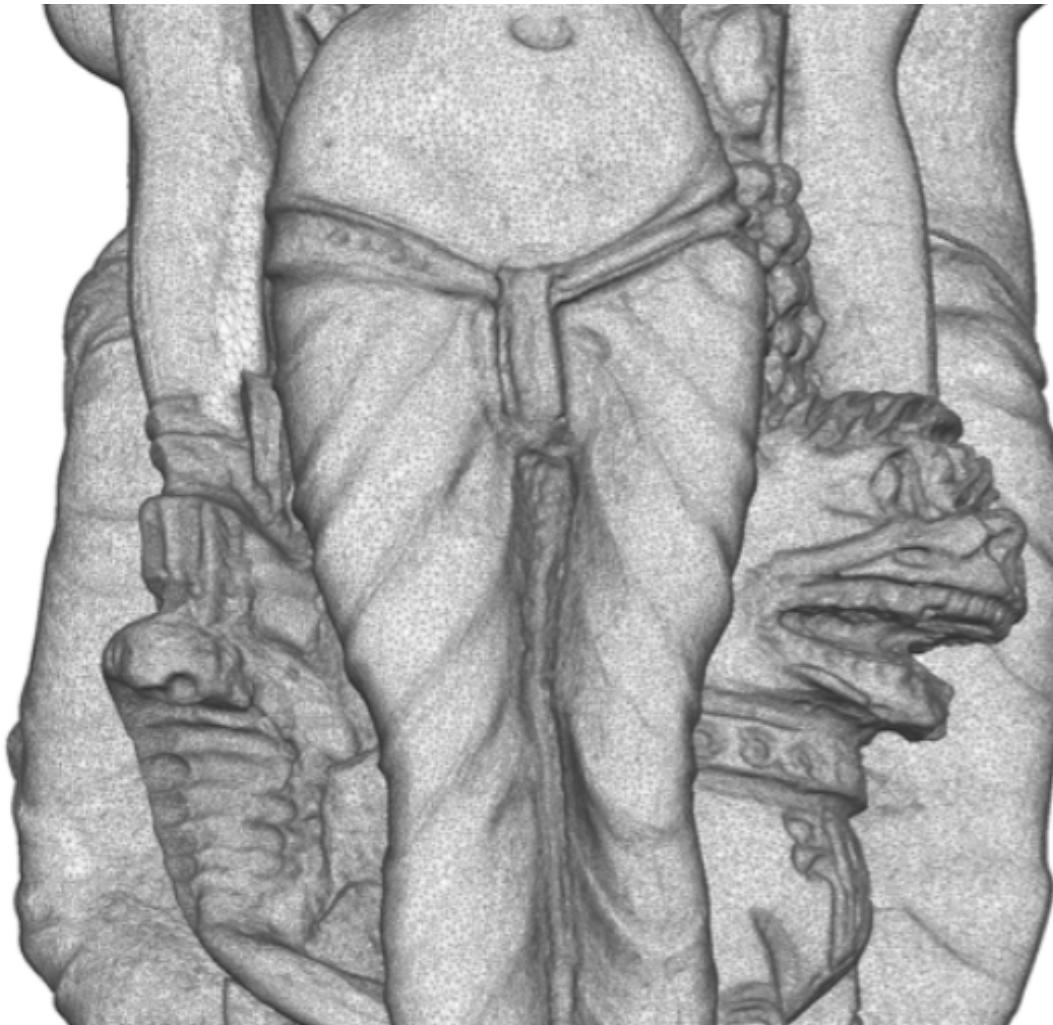
# Uma malha grande

- 10 milhões de triângulos a partir de uma aquisição em 3D



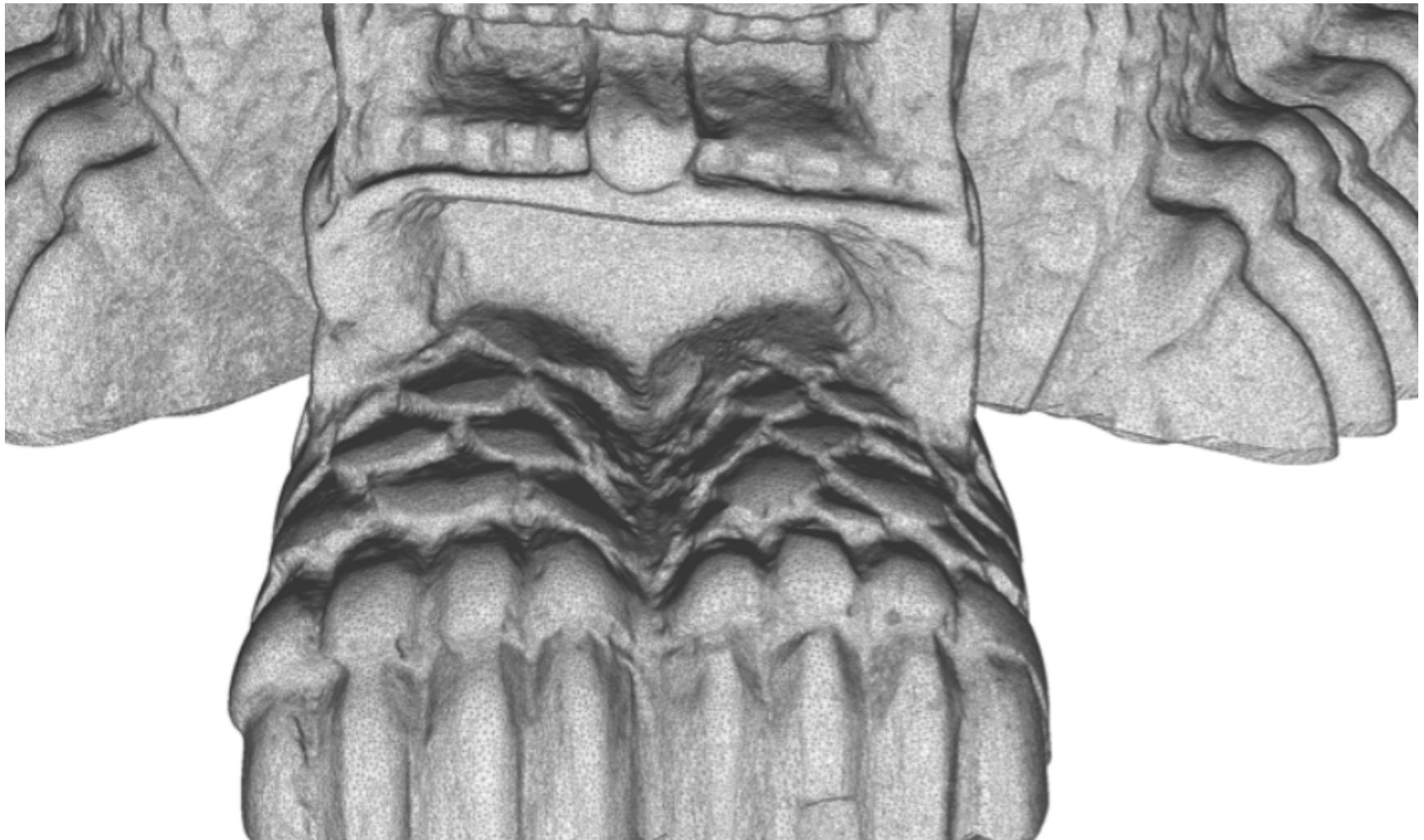
# Malha de triângulos

---



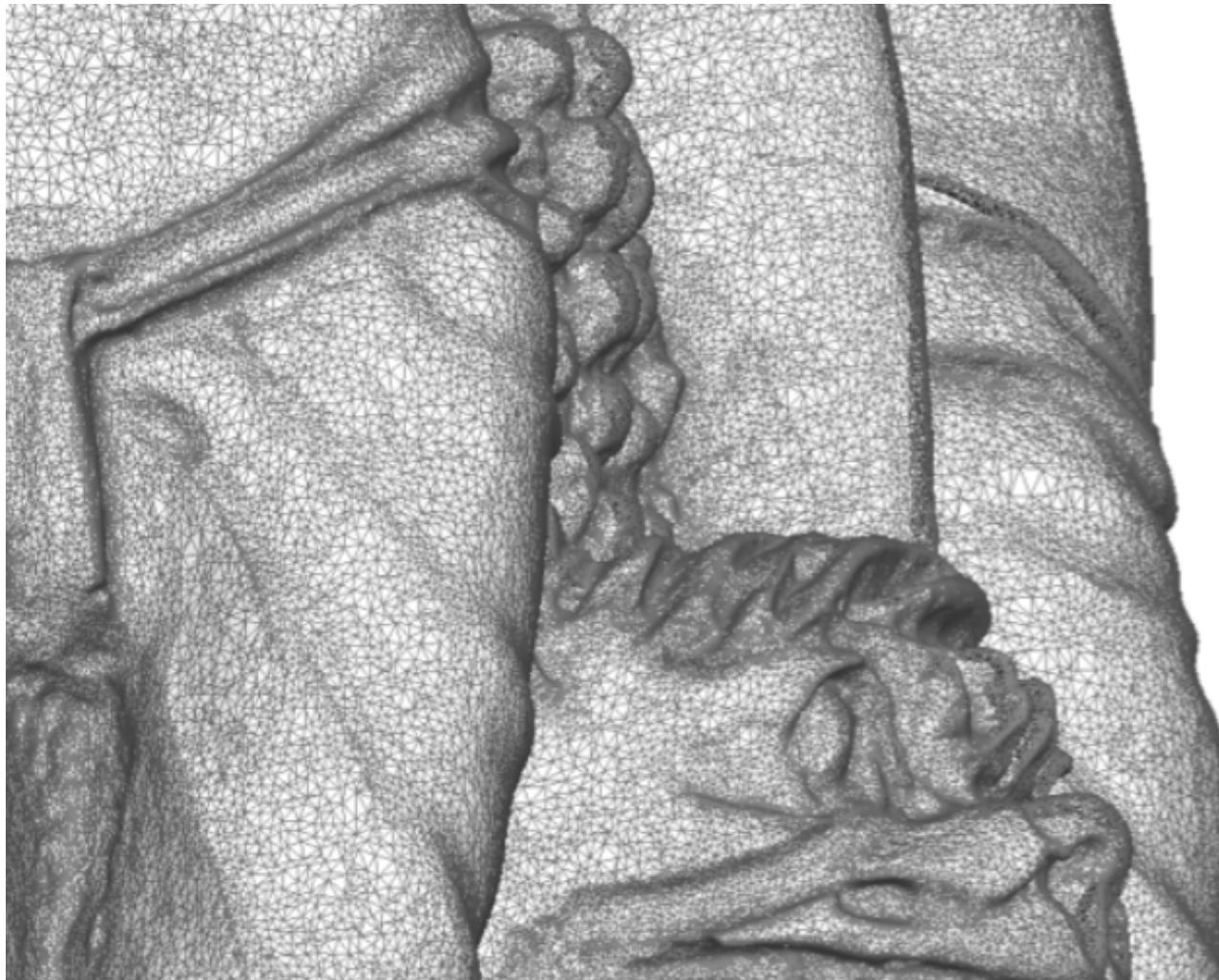
# Malha de triângulos

---



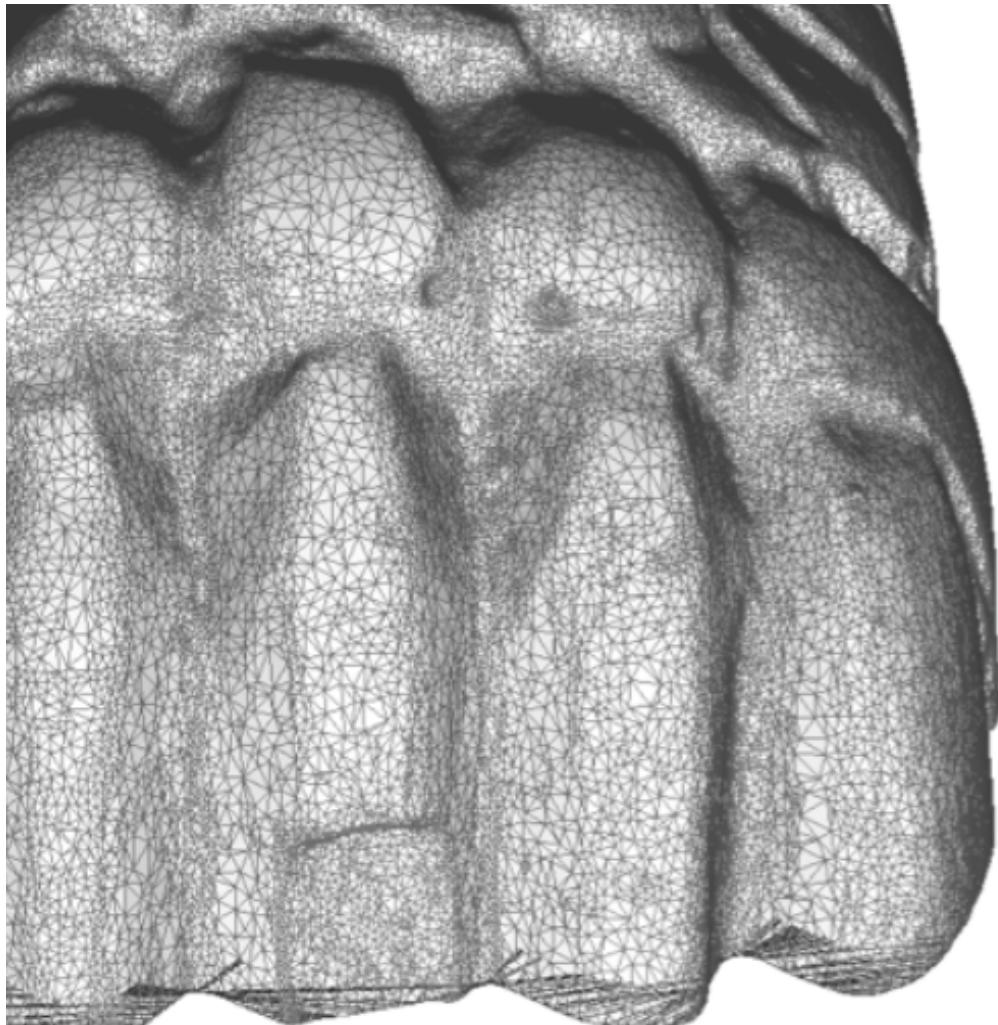
# Malha de triângulos

---

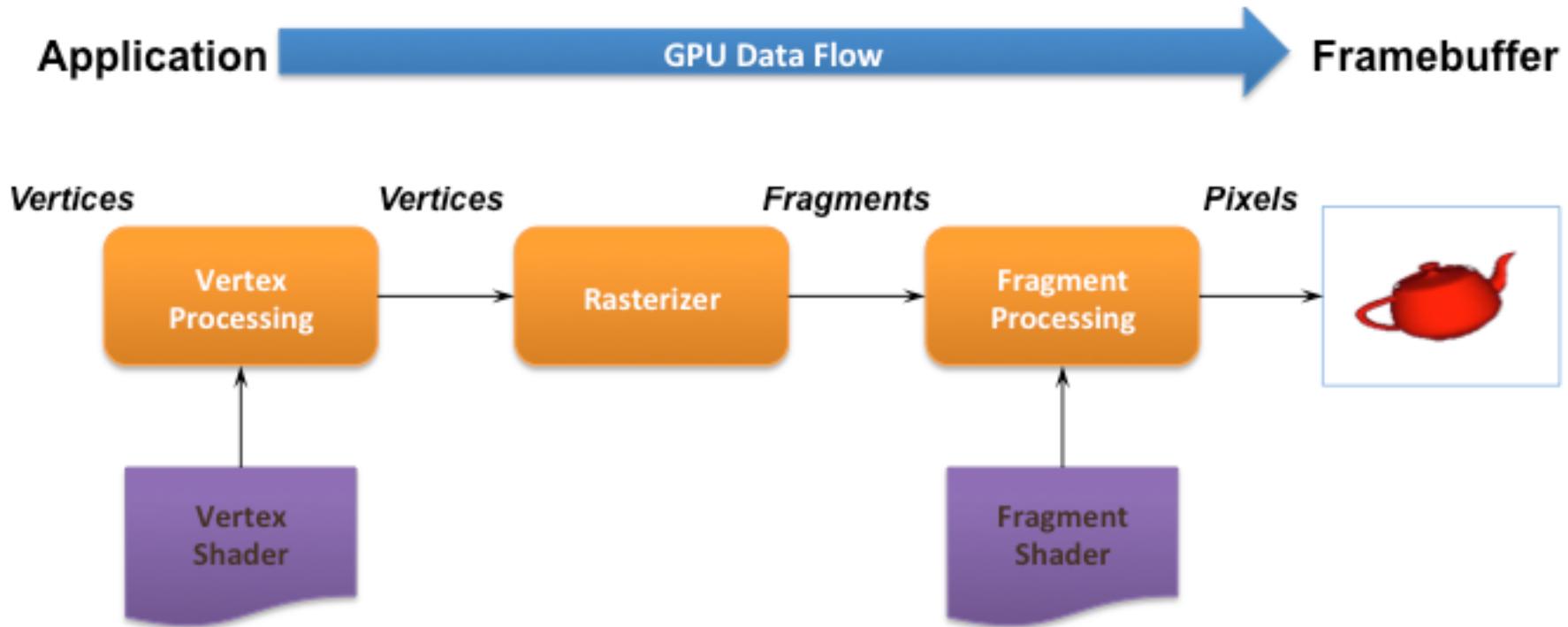


# Malha de triângulos

---



# Pipeline moderno do OpenGL



# Aplicações do vertex shader

---

- Movimentação de vértices
  - Movimentação não linear (e.g. ondulatória)
- Deformações
  - Morphing
  - Geração de fractais
- Iluminação
  - Criação de modelos mais realísticos
  - Shaders para desenhos animados

# Aplicações do fragment shader

---

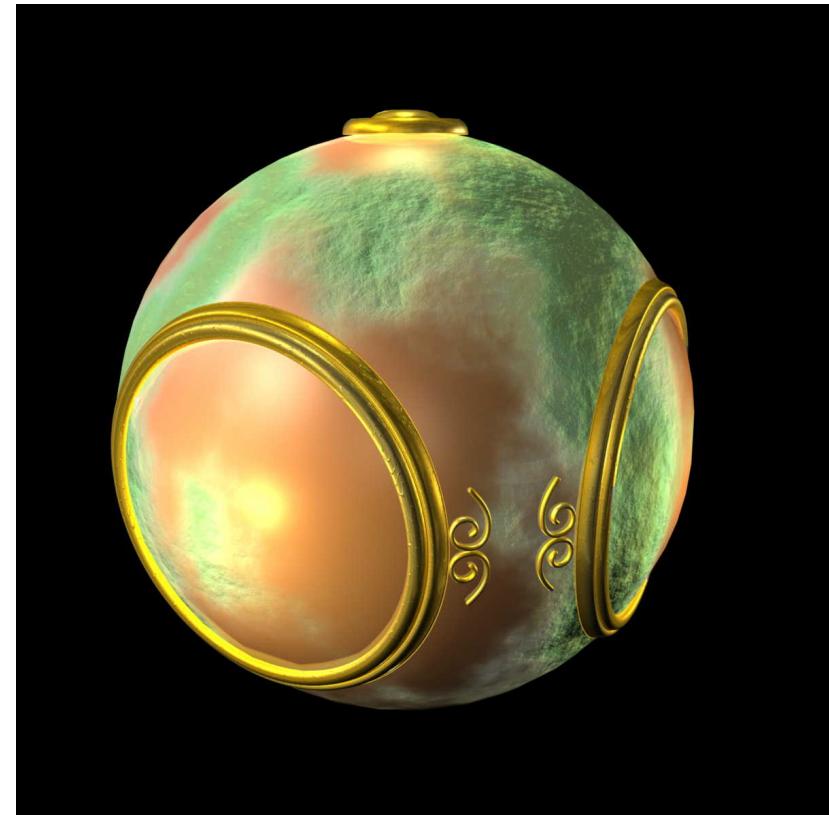
- Cálculos de iluminação por fragmento



# Aplicações do fragment shader

---

- Mapeamento de texturas



# Implementação de shaders

---

- Os primeiros shaders programáveis eram implementados usando linguagem parecida com assembly
- Extensões do OpenGL adicionaram funções para acesso aos shaders de vértice e fragmento
- Cg (C for graphics): linguagem tipo “C para programação de shaders
  - Funciona com OpenGL e DirectX
  - Interfaceamento com OpenGL é complicado
- OpenGL Shading Language (GLSL)

# GLSL

---

- OpenGL Shading Language
  - Faz parte do OpenGL  $\geq 2.0$
  - Linguagem de alto nível parecida com o “C”
  - Novos tipos de dados
    - Vetores
    - Matrizes
    - “Samplers” (texturas)
- A partir do OpenGL 3.1, uma aplicação deve prover shaders

# Um vertex shader simples

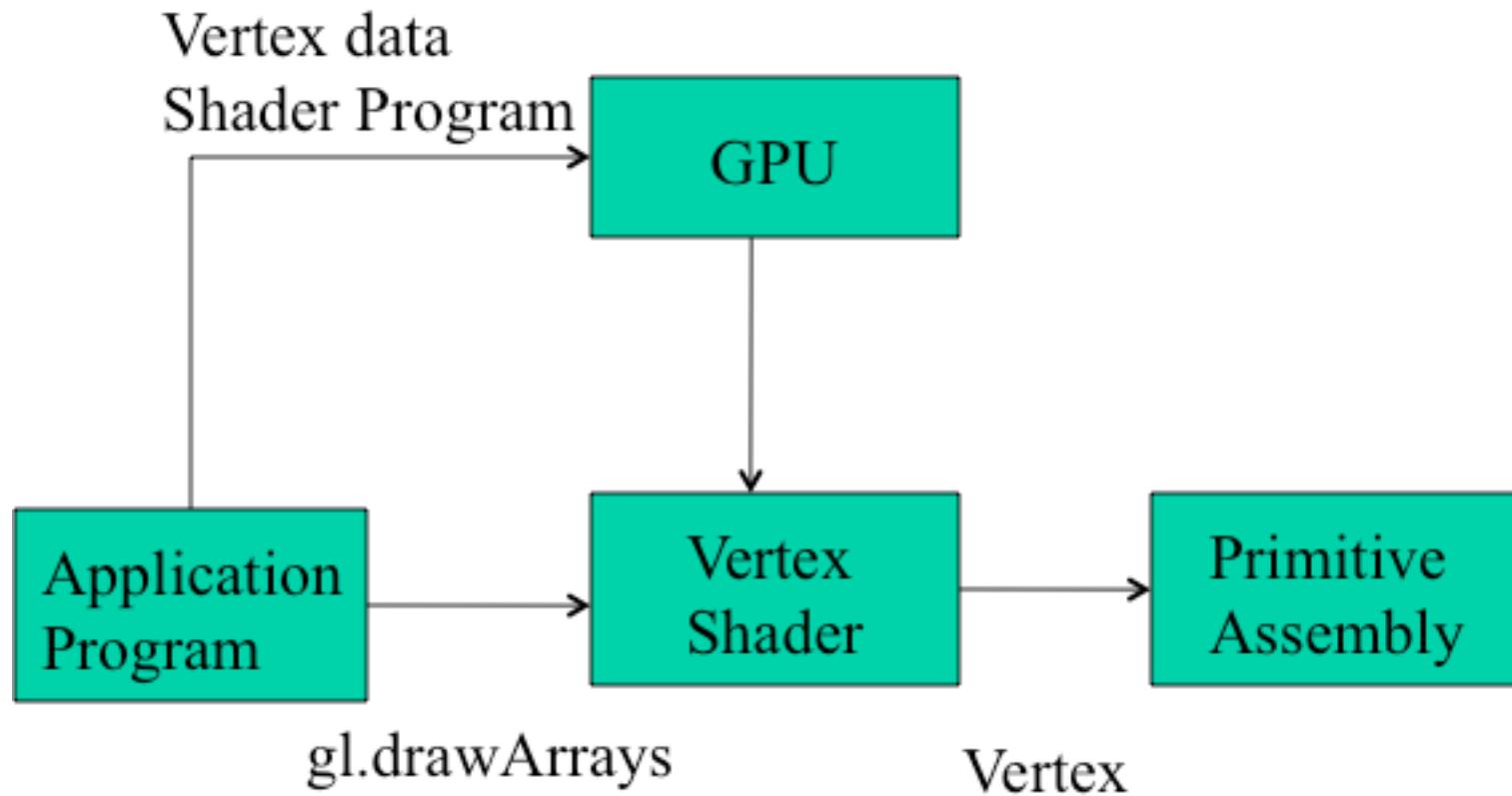
```
attribute vec4 vPosition;  
void main(void)  
{  
    gl_Position = vPosition;  
}
```

input from application

must link to variable in application

built in variable

# Modelo de execução

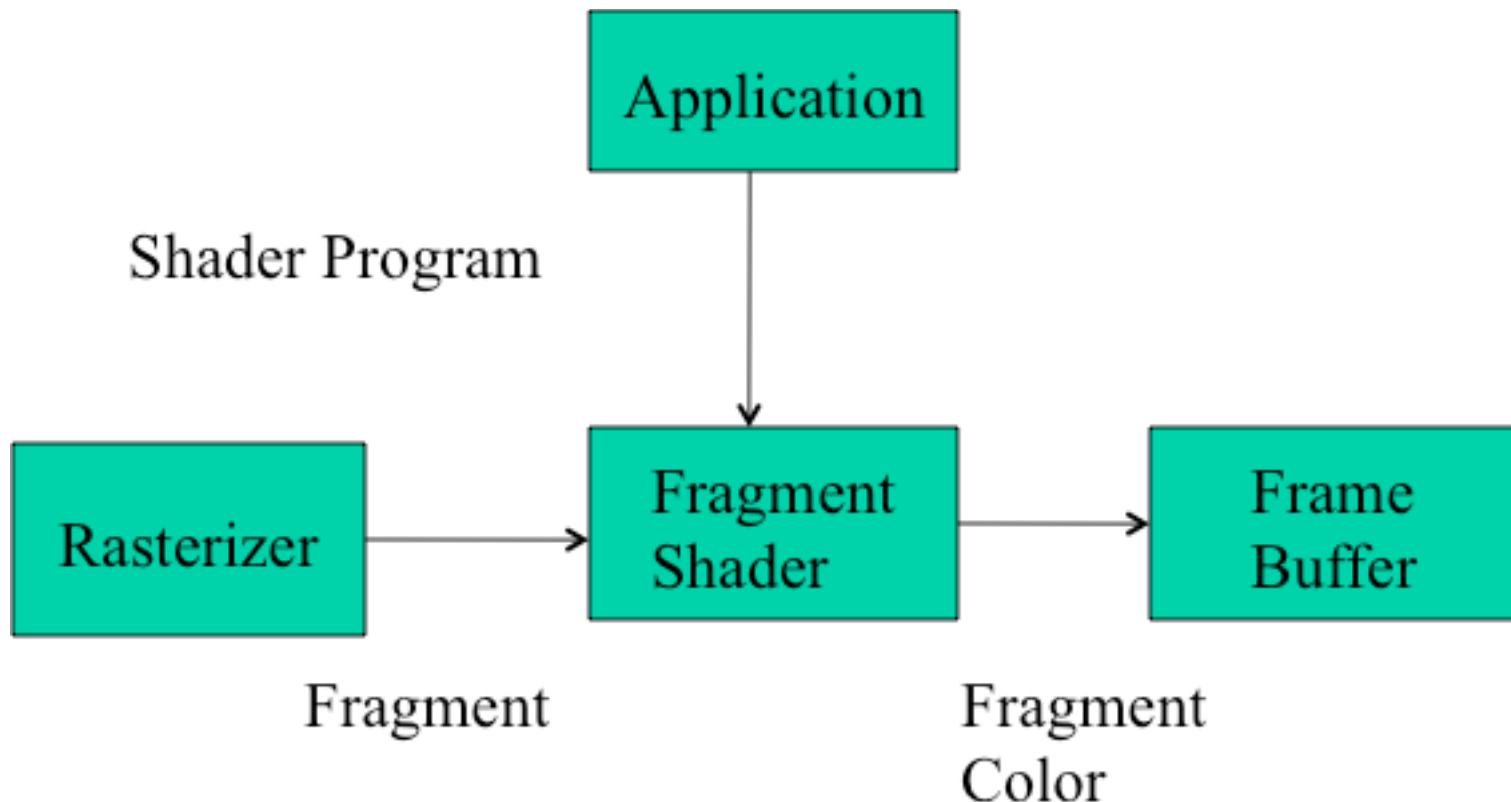


# Um fragment shader simples

---

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

# Modelo de execução



# Tipos de dados

---

- Tipos básicos:
  - float, int, bool
- Vetores:
  - vec2, vec3, vec4,
  - ivec2, ivec3, ivec4
  - bvec2, bvec3, bvec4
- Matrizes (armazenadas coluna a coluna):
  - mat2, mat3, mat4
- Construtores tipo C++
  - `vec3 a = vec3(1.0, 2.0, 3.0)`

# Valores passados por valor

---

- Ausência de ponteiros em GLSL
- Pode-se usar struct que podem ser devolvidas no retorno de funções
- Já que matrizes e vetores são tipos de dados básicos, eles podem ser enviados e retornados a partir de funções GLSL, e.g.

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

# Qualificadores

---

- GLSL utiliza muitos dos qualificadores do C/C++/Java (e.g. `const`)
- Outros são necessários devido ao modelo de execução
- Variáveis podem ter valores que mudam
  - Uma vez por vértice
  - Uma vez por primitiva
  - Uma vez por fragmento
  - A qualquer tempo dentro da aplicação
- Atributos de vértice são interpolados pelo rasterizador em atributos de fragmento

# Qualificador attribute

---

- Qualificador que indica que valores podem mudar no máximo uma vez por vértice
- Existem algumas variáveis especiais como `gl_Position`
- Definidas pelo usuário
  - `attribute float temperature`
  - `attribute vec3 velocity`
- Versões recentes do GLSL utilizam os qualificadores `in` e `out` para enviar e retornar dados de shaders.

# Qualificador uniform

---

- Variáveis que conservam um valor constante durante o processamento de uma primitiva
- Podem ser modificados na aplicação e enviados aos shaders
  - Não podem ser modificados nos shaders
- É utilizado para passar informações aos shaders (e.g. tempo, bounding box, matrizes de transformação, etc)

# Qualificador `varying`

---

- Variáveis que são passadas do vertex shader para o fragment shader
- Automaticamente interpoladas pelo rasterizador
- Versões mais recentes do WebGL utilizam `out` no vertex shader e `in` no fragment shader
  - `out vec4 color; // vertex shader`
  - `in vec4 color; // fragment shader`

# Exemplo de vertex shader

---

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

# Fragment shader corrispondente

---

```
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 uma variável 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 e funções

---

- Funções 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; // vetor coluna como um array 1D`
  - `d = a*b; // vetor linha como um array 1D`

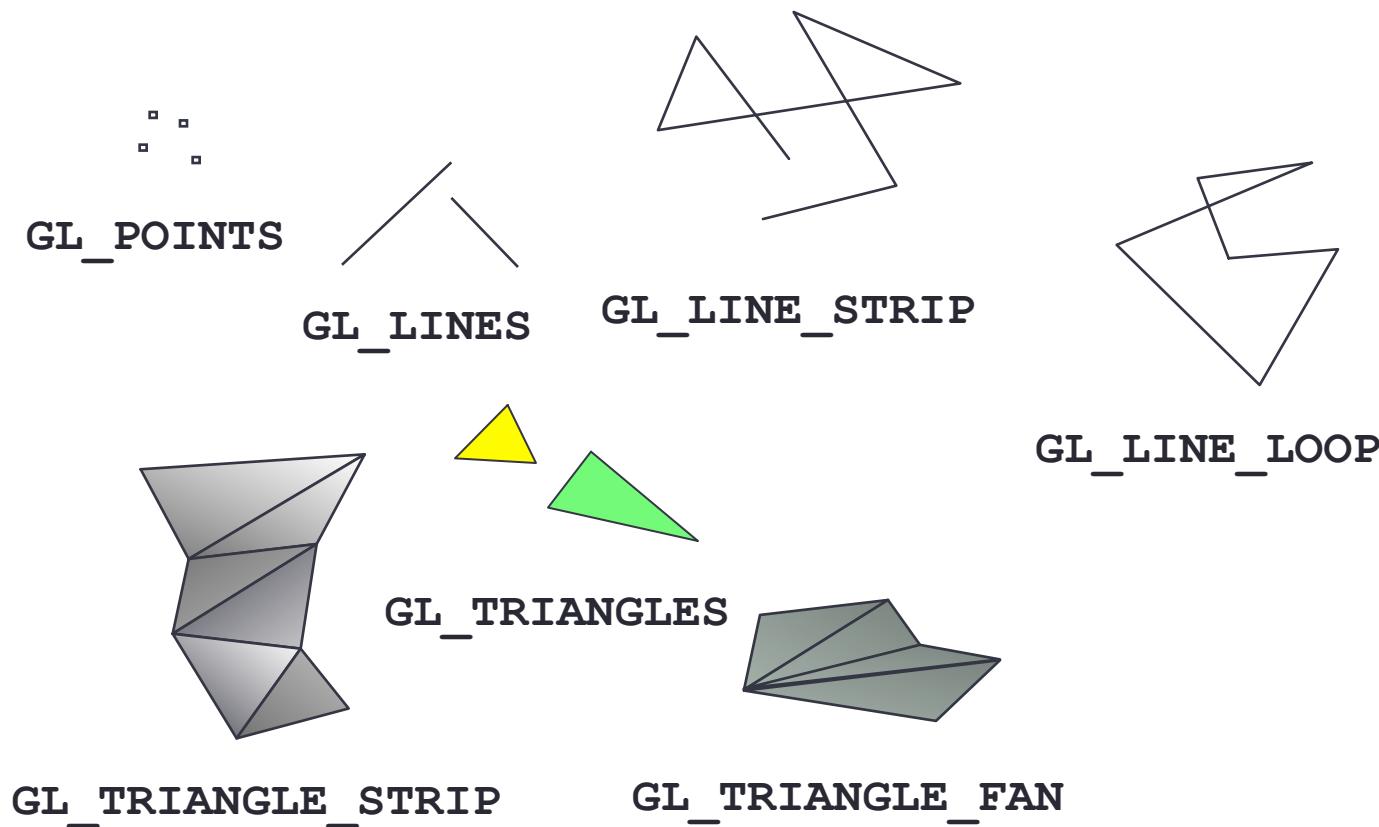
# Swizzling

---

- Elementos de arrays podem ser referenciados usando [] ou o operador de seleção (.) com:
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - a[2], a.b, a.z e a.p denotam o mesmo elemento
- O 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 em WebGL



# Polígonos

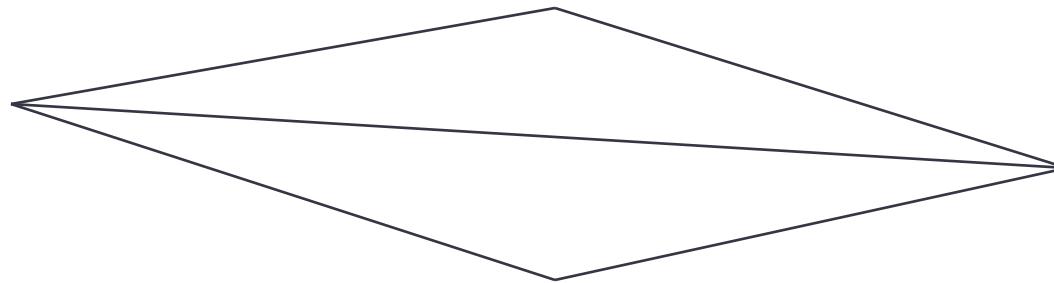
---

- OpenGL/WebGL só desenha triângulos:
  - **Simples:** arestas não se cruzam
  - **Convexos:** todos os pontos em um segmento de reta entre dois pontos dentro do polígono também estão dentro do polígono
  - **Planares:** todos os vértices estão no mesmo plano
- A aplicação deve converter um polígono em triângulos
  - OpenGL  $\geq 4.1$  disponibiliza shader para tal tarefa

# Bons triângulos

---

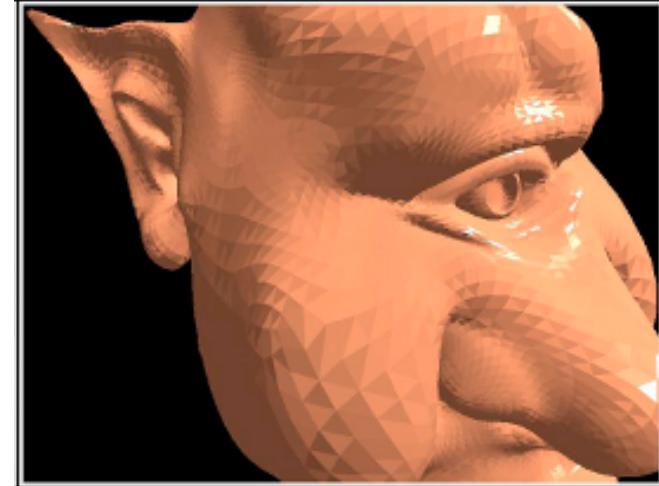
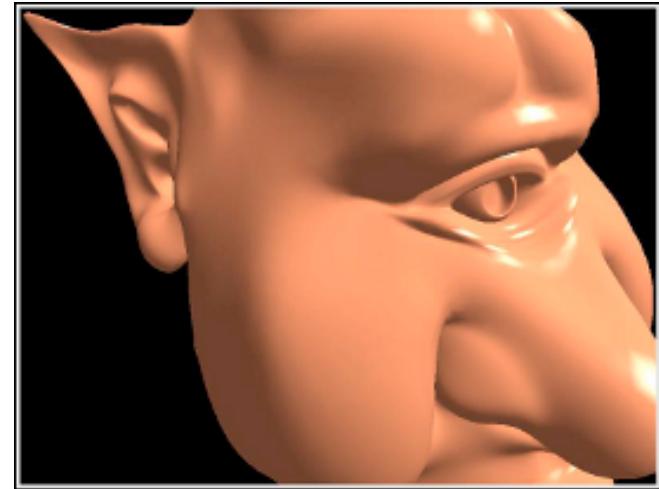
- A renderização de triângulos longos e finos é ruim



- Triângulos equilaterais renderizam bem
- Maximizar o menor ângulo
  - Triangulação de Delaunay

# Tonalização - Shading

- Tonalização suave
  - O rasterizador interpola valores dos vértices ao longo dos triângulos visíveis (default)
- Tonalização facetada
  - Cor do primeiro vértice determina a cor de preenchimento



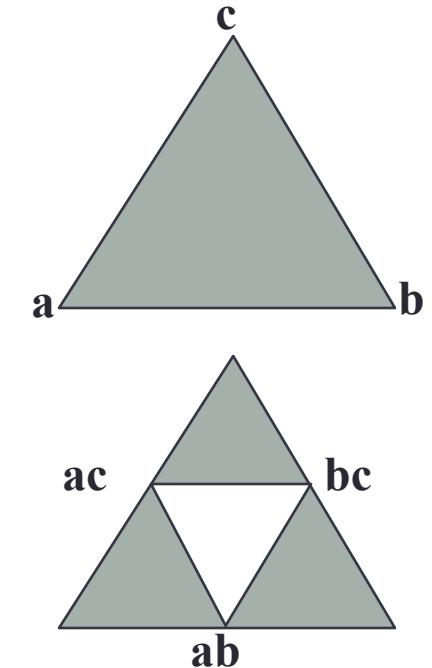
# Atribuição de cores

---

- As cores são finalmente atribuídas no fragment shader, porém podem ser definidas em qualquer shader ou na aplicação final
- Na aplicação:
  - Passa-se a cor ao vertex shader usando uma variável **uniform** ou **attribute**
- No vertex shader:
  - Passa-se a cor ao fragment shader através de uma variável **varying**
- No fragment shader:
  - Pode-se alterá-la com código do próprio shader.

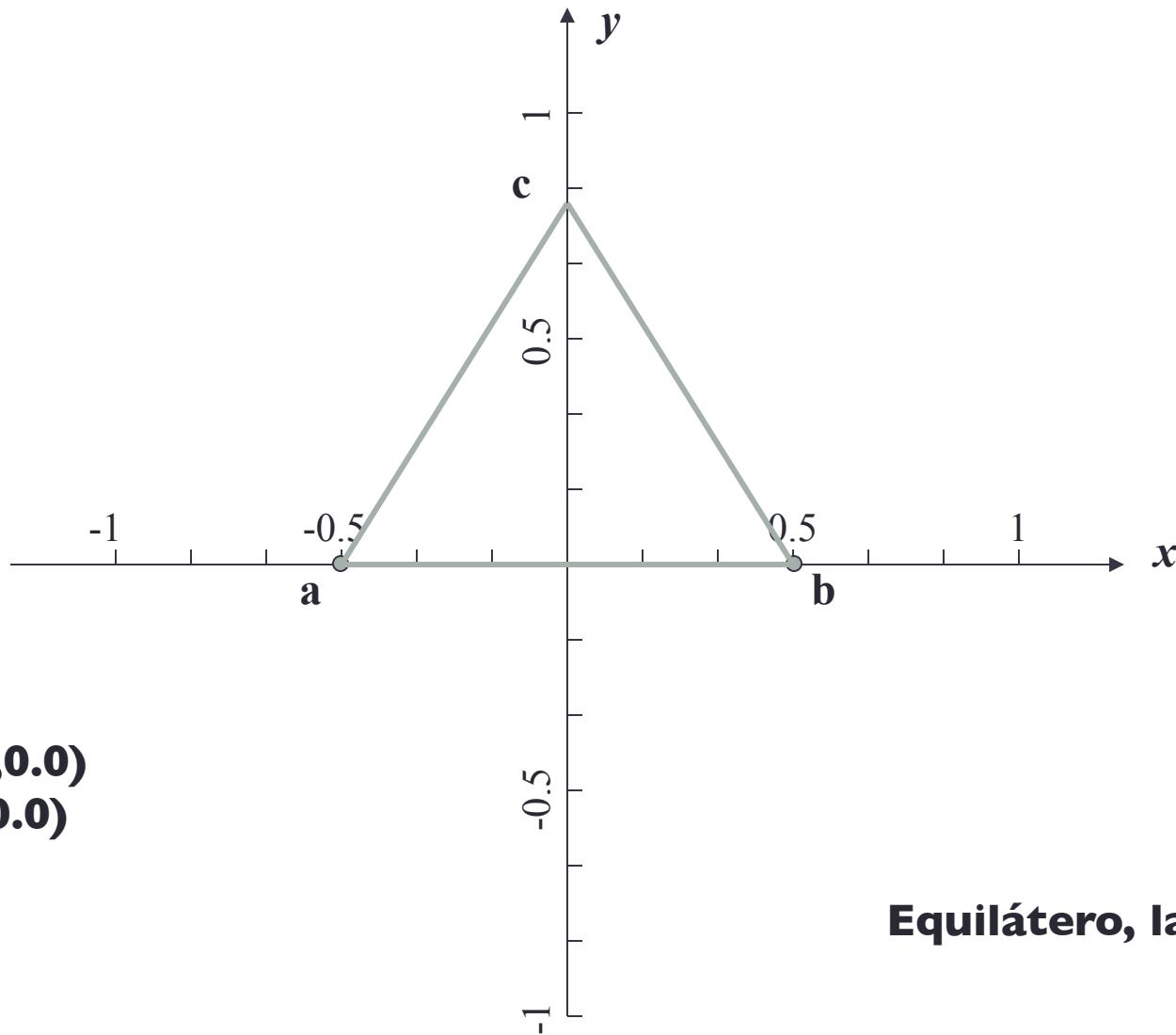
# Exemplo: triângulo de Sierpinski

- Inicia-se com um triângulo
- Conecta-se bisetores dos lados e remove-se o triângulo central
- Repete-se o processo



Waclaw Sierpinski (1882 - 1969), matemático polonês

# Coordenadas



# gasket.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)
];

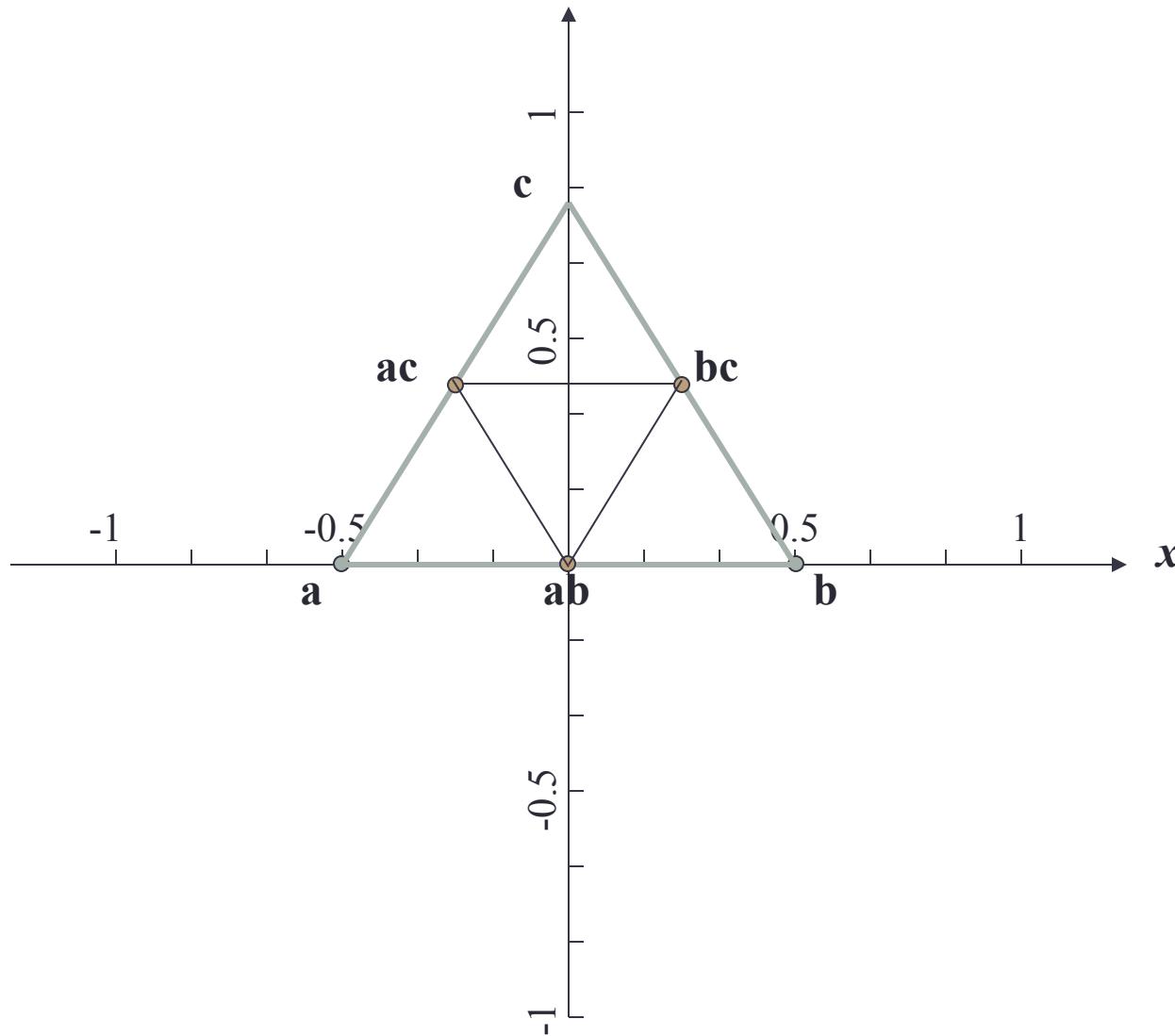
divideTriangle( vertices[0], vertices[1],
  vertices[2], NumTimesToSubdivide);
```

# Desenha um triângulo

---

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

# Subdivisão



$$ab = ?$$

$$ac = ?$$

$$bc = ?$$

# Subdivisão

---

```
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 );  
    }  
}
```

# Demo

---

# Em três dimensões

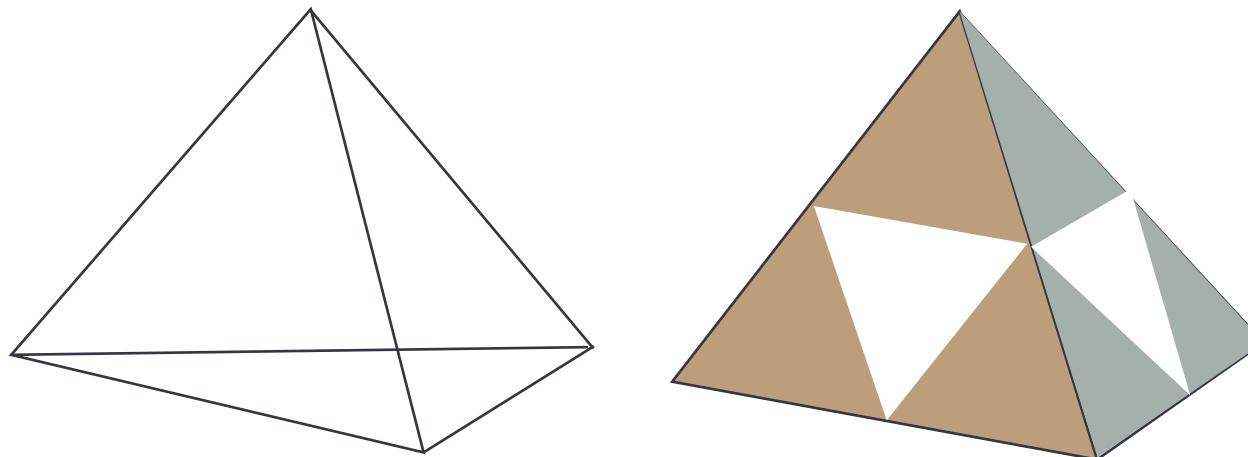
---

- Em OpenGL/WebGL, gráficos em 2D são um caso especial de gráficos em 3D
- Deve-se considerar a ordem em que os polígonos serão desenhados ou usar remoção de superfícies escondidas
- Polígonos devem ser simples, convexos e planares
- Deve-se se considerar a ordenação dos vértices (sentido horário ou anti-horário) para cada face
- Projeções paralela (ortogonal) ou perspectiva.

# Sierpinski em 3D

---

- Podemos subdividir as quatro faces triangulares de um tetraedro



- Aparecerá como se tivéssemos removido um tetraedro do centro

# Tetraedro

---

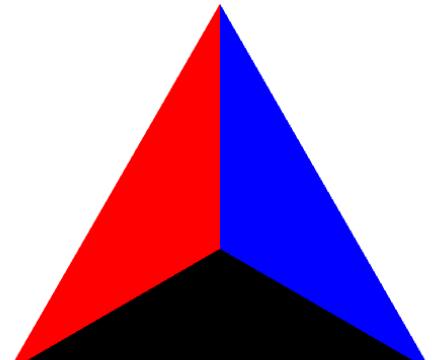
```
var points = [];
var NumTimesToSubdivide = 5;

// Four vertices on unit circle
// Initial tetrahedron with equal length sides
var vertices = [
    vec3( 0.0000, 0.0000, -1.0000 ),
    vec3( 0.0000, 0.9428, 0.3333 ),
    vec3( -0.8165, -0.4714, 0.3333 ),
    vec3( 0.8165, -0.4714, 0.3333 )
];

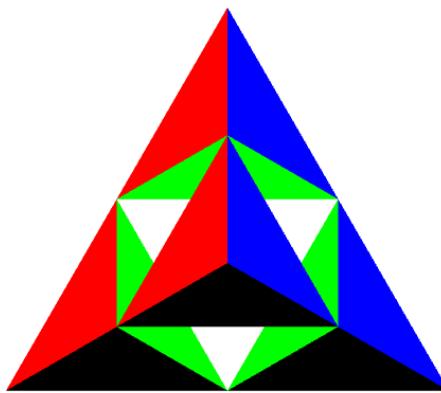
tetra( vertices[0], vertices[1], vertices[2],
       vertices[3], NumTimesToSubdivide );
```

# Exemplos

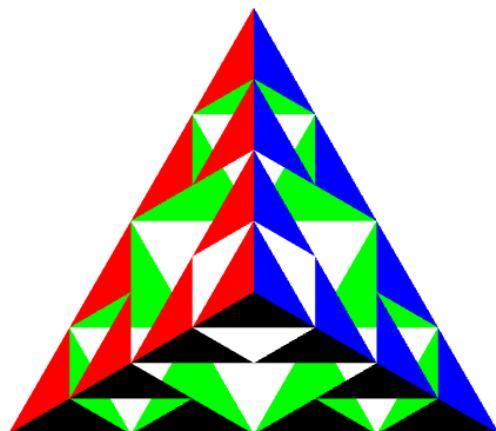
---



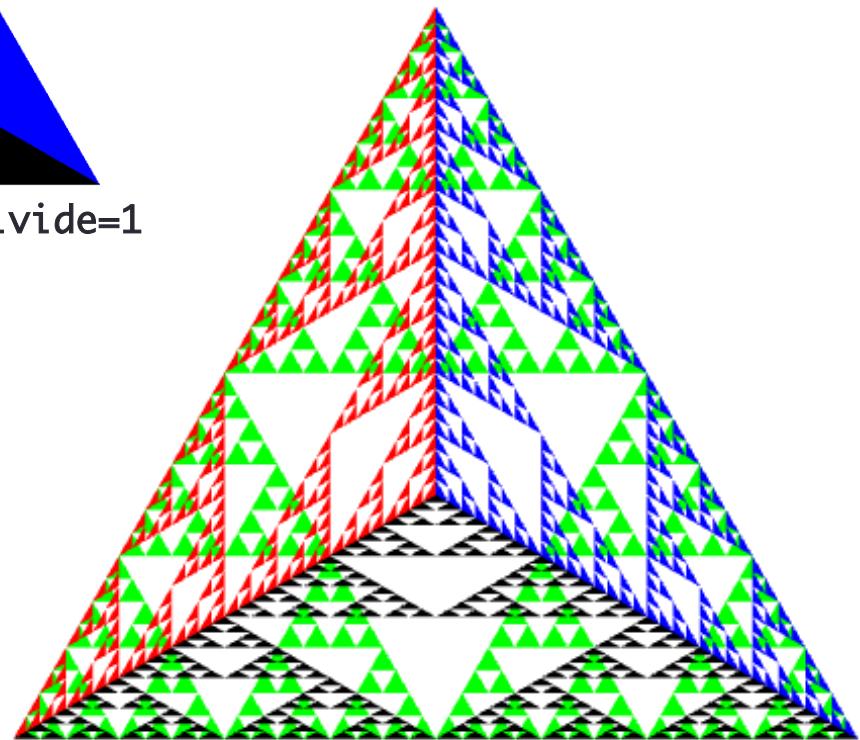
NumTimesToSubdivide=0



NumTimesToSubdivide=1



NumTimesToSubdivide=2



NumTimesToSubdivide=5

# Demo

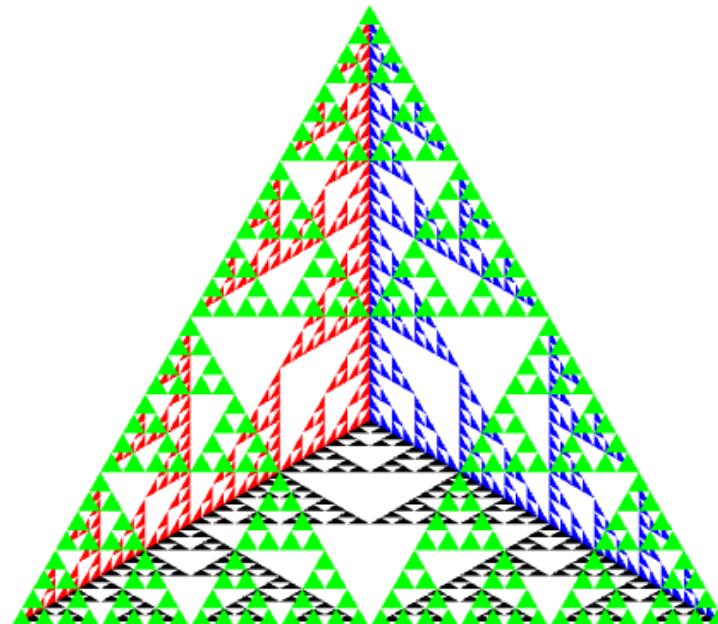
---

# Quase lá

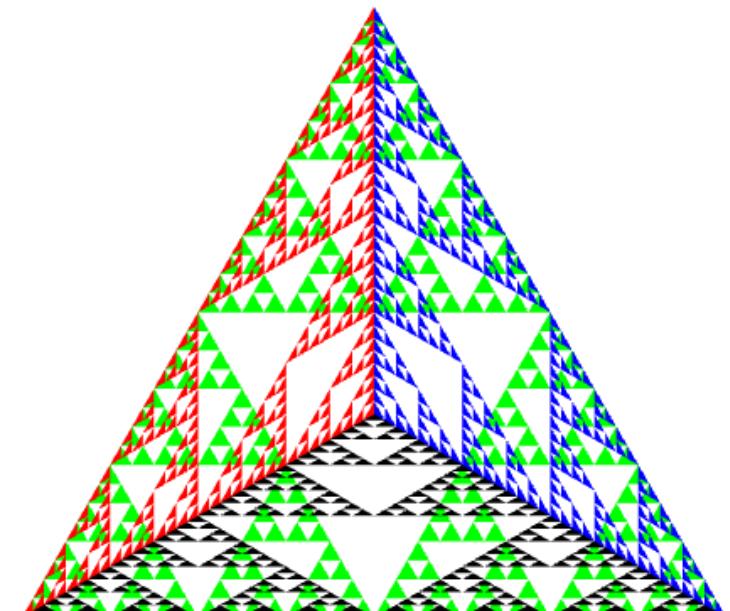
---

- Os triângulos são desenhados na ordem em que são definidos no programa

**obtido**

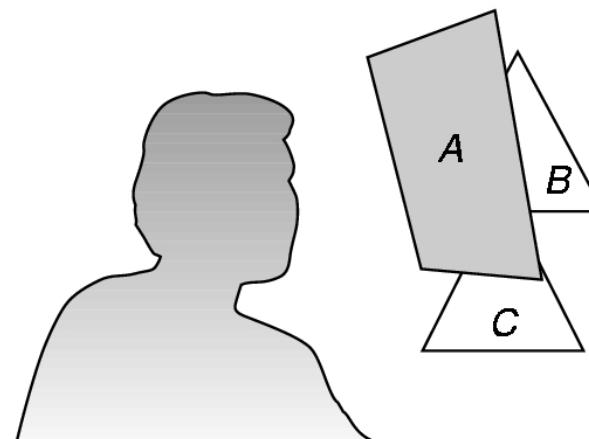


**desejado**



# Remoção de superfícies escondidas

- Desejamos ver somente superfícies que estão realmente na frente de outras
- OpenGL utiliza o método chamado de “z-buffer” que salva a profundidade de cada ponto de modo que somente objetos que estão na frente são visíveis.



# Utilizando o z-buffer

---

- Precisa ser habilitado antes do uso:
  - `gl.enable(gl.DEPTH_TEST)`
- Deve ser limpo a cada renderização:
  - `gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)`

# Demo

---

# Exercício

---

- Criar um programa em WebGL que aproxime um círculo de raio unitário utilizando subdivisões sucessivas de um quadrado inscrito
- Faça a mesma tarefa em 3D, mas de forma a aproximar uma esfera utilizando como base um octaedro regular.

