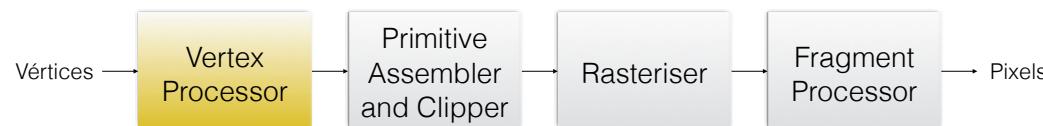


Pipeline de Visualização 3D

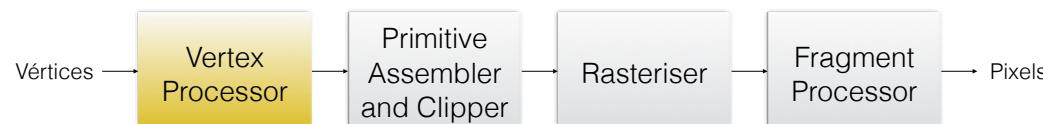
Processamento de vértices

- Uma grande parte do trabalho no pipeline gráfico consiste em efetuar mudanças de coordenadas dum referencial para outro
 - Coordenadas dos pontos em referenciais próprios do objeto
 - Coordenadas no referencial global da cena/mundo
 - Coordenadas no referencial da câmara
 - Coordenadas no referencial do ecrã
- Cada mudança de referencial equivale a uma operação matricial (mudança de base)
- O processamento de vértices também inclui a transformação de atributos (ex. cor), associados pela aplicação a cada vértice, em valores que irão variar durante o varrimento dos pixels gerados para as primitivas gráficas (triângulos na sua maioria)



Projeção

- A projeção permite combinar os objetos 3D e a forma como se observa a cena numa imagem 2D
 - Projeção perspetiva: as linhas projetantes encontram-se num ponto (centro de projeção)
 - Projeção paralela: as linhas projetantes são paralelas entre si (a mesma direção)
- A projeção é também realizada através duma operação matricial (multiplicação por uma matriz de projeção)



Vertex processor

```
#version 300 es

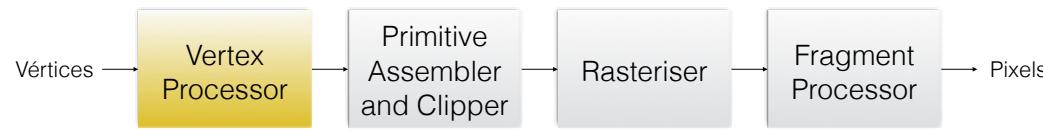
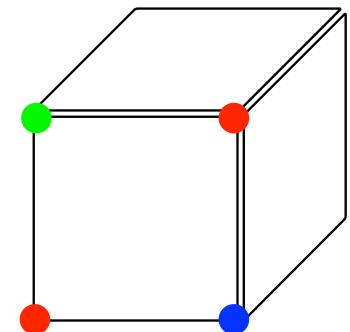
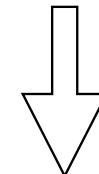
in vec4 a_position;
in vec4 a_color;

uniform mat4 u_projection;
uniform mat4 u_model_view;

out vec4 v_color;

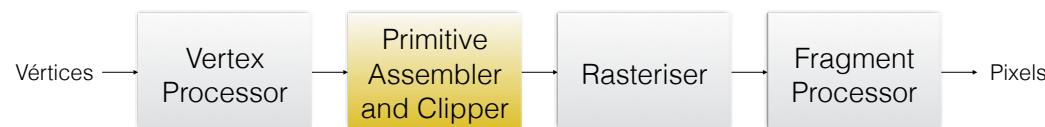
void main() {
    gl_Position = u_projection * u_model_view * a_position;
    v_color = a_color;
}
```

a_color	a_position
green	(-1,1,1)
red	(1,1,1)
red	(-1,-1,1)
blue	(1,-1,1)



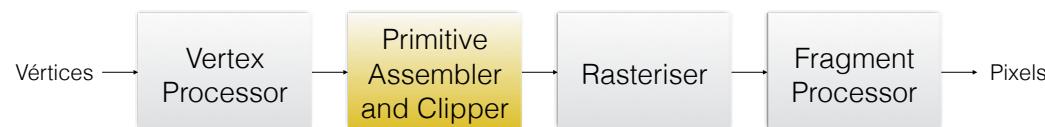
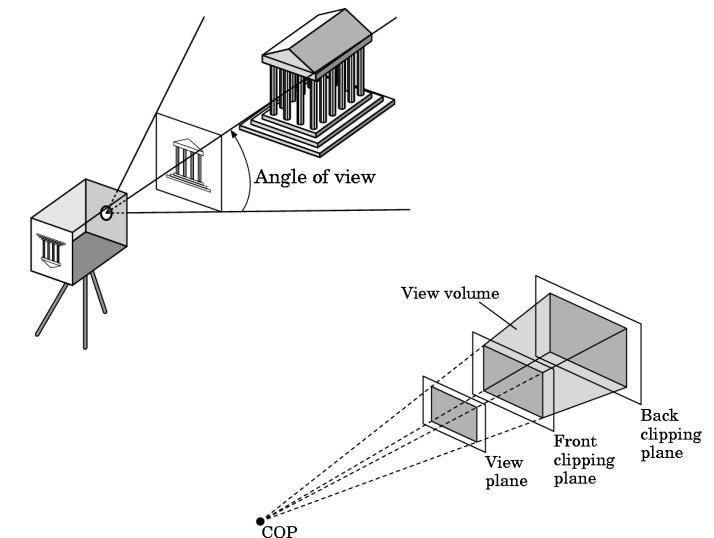
Primitive Assembly

- Os vértices submetidos ao pipeline são agrupados para a formação das primitivas, antes que se possa proceder ao seu varrimento (geração dos pixels que as constituem) e recorte (clipping)
 - Segmentos de reta
 - Polígonos (apenas triângulos no caso do WebGL)
 - Linhas curvas (discretizadas posteriormente em segmentos)
 - Superfícies curvas (discretizadas posteriormente em polígonos)



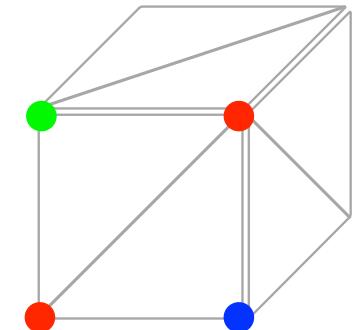
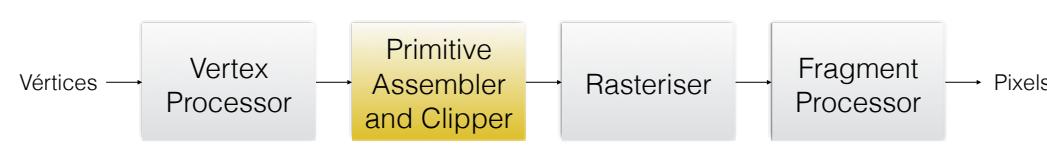
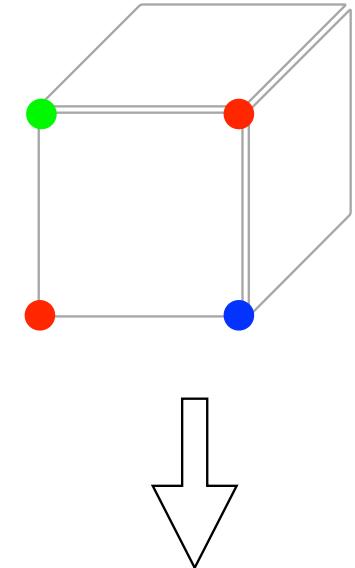
Clipper

- Uma câmara, quer seja real ou virtual, apenas consegue “capturar” parte do mundo
- Os objetos que não estão dentro dessa mesma parte visível (volume de visão) são recortados (total ou parcialmente)



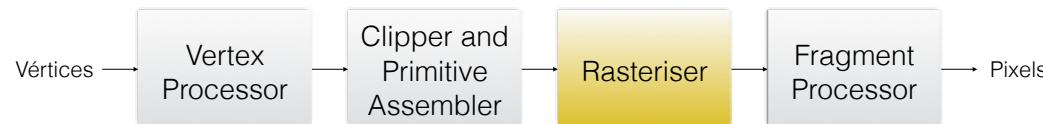
Primitive Assembly and Clipper

Neste exemplo vamos assumir que as primitivas a desenhar são triângulos e que nenhuma está fora do volume de visão, não havendo lugar a qualquer recorte.



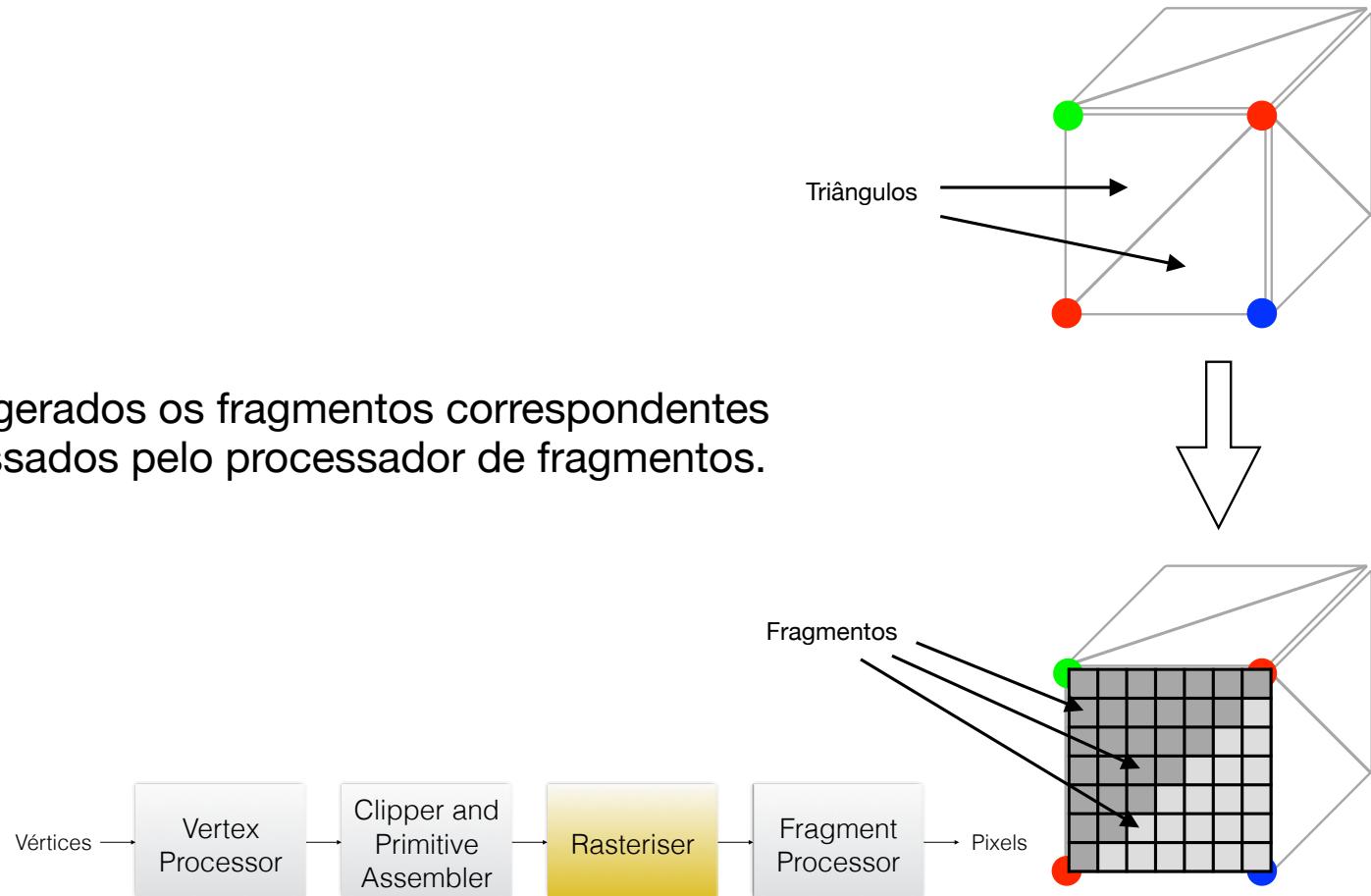
Rasterização - Varrimento

- Se um objeto não for recortado na sua totalidade (descartado), os pixels correspondentes no *frame buffer* terão que ser gerados (saber quais são)
- Um *rasteriser* produz um conjunto de fragmentos (potenciais pixels no final do pipeline) para cada objeto.
- Sendo potenciais pixels, os fragmentos possuem:
 - Localização no *frame buffer*
 - valores de cor e profundidade (coordenada z no referencial da câmara)
- Os atributos associados aos vértices (ou valores calculados a partir deles) poderão ser propagados para os fragmentos e interpolados sobre os objetos pelo módulo de varrimento.



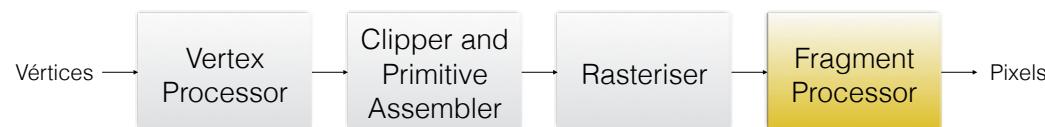
Rasteriser

Para cada primitiva são gerados os fragmentos correspondentes que serão depois processados pelo processador de fragmentos.



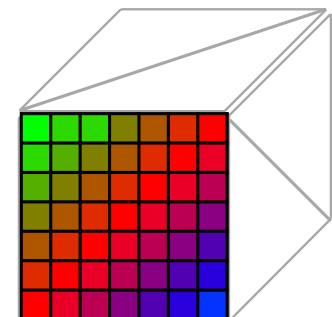
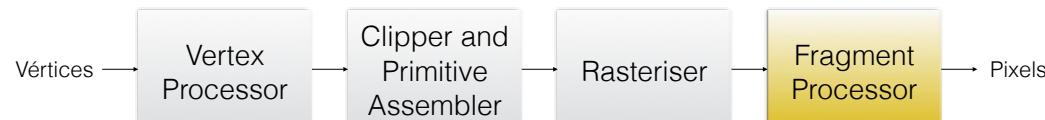
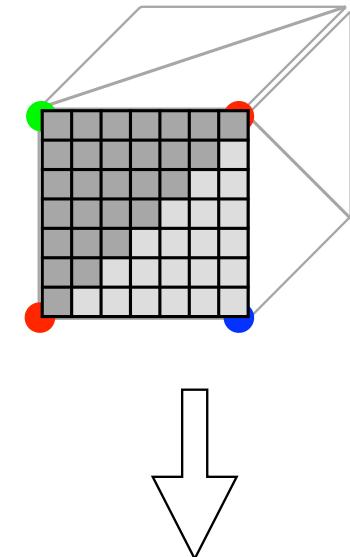
Processamento de fragmentos

- Os fragmentos são processados para determinar a cor final do pixel correspondente no *frame buffer*
- As cores podem ser determinadas por mapeamento de texturas e por interpolação de valores de atributos nos vértices
- Os fragmentos poderão ser “bloqueados” por outros mais perto da câmara
 - Remoção de superfícies ocultas (**Hidden Surface Removal**)



Fragment Processor

```
#version 300 es  
  
precision mediump float;  
  
in vec4 v_color;  
out vec4 color;  
  
void main() {  
    color = v_color;  
}
```



Introdução ao WebGL

Objetivos

- Componentes duma aplicação WebGL
- Convenções a usar (bibliotecas auxiliares e organização em ficheiros)
- Familiarização com *shaders*
- Familiarização com primitivas gráficas e sua invocação

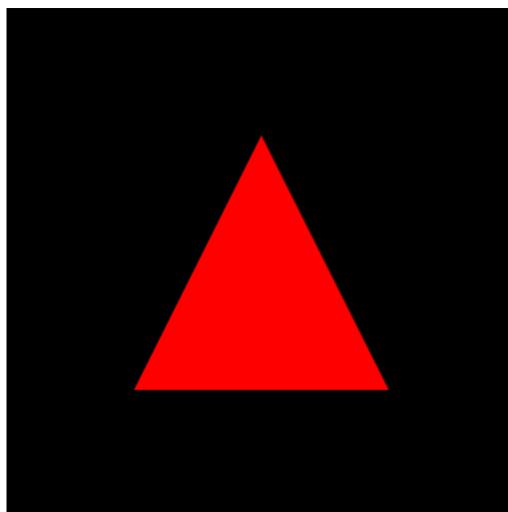
Componentes duma aplicação WebGL

- Uma aplicação WebGL tem duas componentes obrigatórias:
 - HTML + Javascript
- **HTML:**
 - descreve a página
 - efetua o carregamento de bibliotecas (javascript)
 - Pode incluir ainda os shaders (há outras alternativas para guardar os shaders)
- **Javascript:**
 - contém o código da aplicação + código de bibliotecas

Vantagens/desvantagens em usar WebGL

- A aplicação corre num browser
 - dispensa instalação de ferramentas (compilador, bibliotecas)
 - é independente do sistema operativo
 - funciona em todos os browsers recentes
- O código é escrito em Javascript, é interpretado em vez de compilado
- **Obriga à utilização de shaders**
- **A API é bastante mais simples (resumida) do que a do OpenGL**

Exemplo: Desenhando um triângulo



HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Exemplo 01</title>
  </head>
  <body>
    <script type="module" src="./app.js"></script>
    <canvas id="gl-canvas" width="512" height="512"></canvas>
  </body>
</html>
```

Área de desenho

código JS da
aplicação

Javascript

```
import { loadShadersFromURLS, loadShadersFromScripts, setupWebGL, buildProgramFromSources, ... } from "../../libs/utils.js";
import { vec2, flatten, ... } from "../../libs/MV.js";
```

Importação das bibliotecas

```
loadShadersFromURLS(["shader.vert", "shader.frag"]).then(shaders => setup(shaders));

setup(loadShadersFromScripts(["shader.vert", "shader.frag"]));
```

Carregamento assíncrono, com semáforo para continuar

Carregamento síncrono, a partir de scripts na página HTML

Javascript

variáveis globais

```
var gl;  
var program;
```

```
function setup(shaders)  
{
```

// Setup

```
const canvas = document.getElementById("gl-canvas");  
gl = setupWebGL(canvas);
```

```
program = buildProgramFromSources(gl, shaders["shader.vert"], shaders["shader.frag"]);
```

```
const vertices = [ vec2(-0.5, -0.5), vec2(0.5, -0.5), vec2(0, 0.5) ];
```

As coordenadas dos vértices do triângulo

Função invocada na chamada colocada no final do script, após carregamento dos shaders

Um dicionário que associa o nome do shader ao seu código fonte.

id de <canvas> no HTML

Função auxiliar que devolve um programa GLSL a partir do código fonte dos seus shaders.

Javascript

```
// Setup the viewport  
gl.viewport(0, 0, canvas.width, canvas.height);
```

Definição do visor dentro do Canvas

```
// Setup the background color  
gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

Cor com que se limpa o ecrã

```
const aBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, aBuffer);  
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
```

Criação e preenchimento dum buffer no GPU com os dados dos vértices

Javascript

Descreve quais os dados associados aos vértices e como se encontram armazenados no buffer

```
vao = gl.createVertexArray();
gl.bindVertexArray(vao);

const a_position = gl.getAttribLocation(program, "a_position");
gl.vertexAttribPointer(a_position, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(a_position);

gl.bindVertexArray(null);

window.requestAnimationFrame/animate());
```

Pede ao browser para invocar a função `animate()` no próximo

Manda limpar o ecrã

```
function animate() {
    window.requestAnimationFrame/animate());

    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.useProgram(program);
    gl.bindVertexArray(vao);
    gl.drawArrays(gl.TRIANGLES, 0, 3);
    gl.bindVertexArray(null);
}
```

manda desenhar o triângulo

WebGL (Shaders)

Objetivos

- Aplicações típicas de:
 - vertex shaders
 - fragment shaders
- GLSL

Aplicações típicas de Vertex Shaders

- Modelação (Transformação de coordenadas) e Projeção
- “Mover” vértices
 - Sistemas de partículas
 - Morphing - Transformação/Deformação dum modelo noutro
 - movimentos ondulatórios
 - fractais
- Iluminação



iluminação por vértice

Aplicações típicas de Fragment Shaders

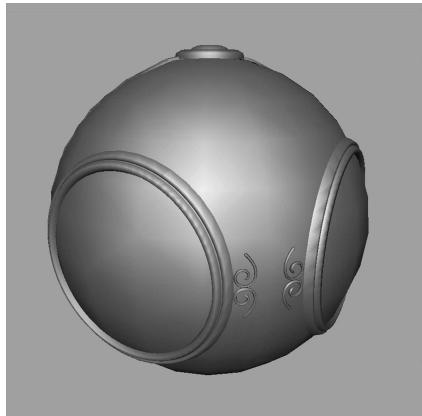
- Iluminação aplicada ao nível dos fragmentos



iluminação por fragmento

Aplicações típicas de Fragment Shaders

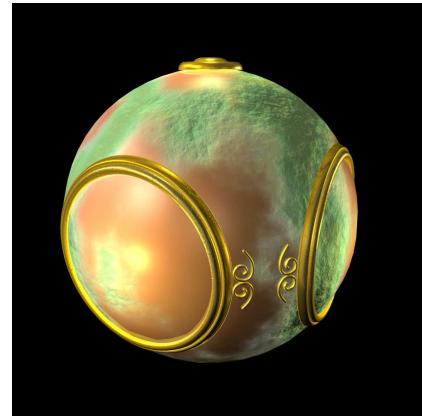
- Mapeamento de texturas



sem texturas



environment mapping



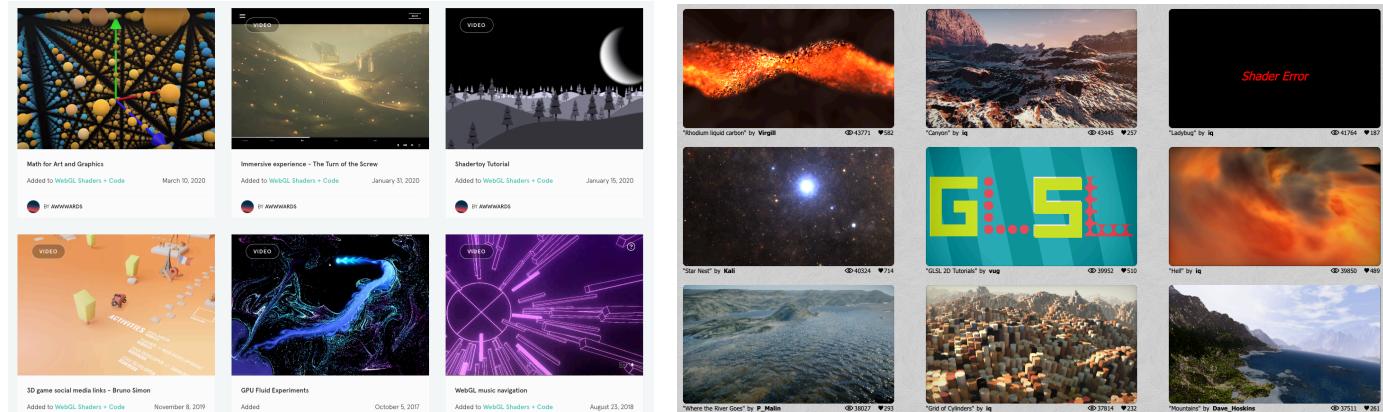
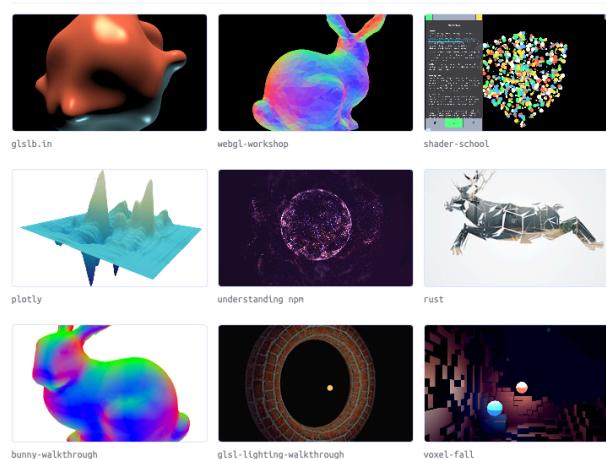
bump mapping

Shaders

- Os primeiros shaders eram programados diretamente em assembler (do GPU)
- Disponibilizados no OpenGL (recorrendo ao mecanismo de extensões)
- Atualmente podem ser escritos em vários “dialetos”:
 - Cg (C for Graphics) - linguagem da nVIDIA para a escrita de shaders, baseada em C
 - GLSL - OpenGL Shading Language
 - HLSL - High Level Shading Language (DirectX)

Exemplos online

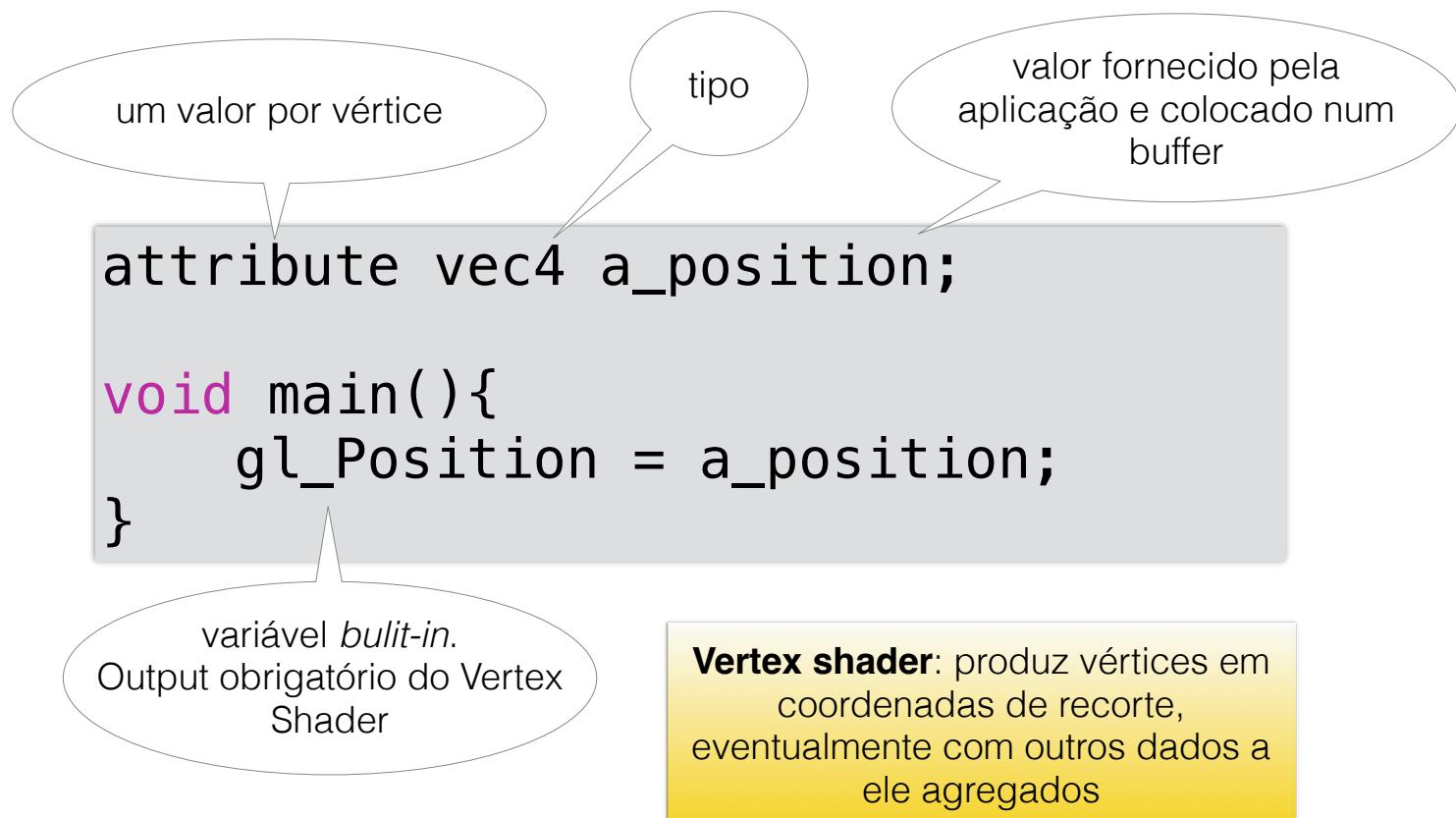
- <https://webgl-shaders.com>
- <https://www.awwwards.com/awwwards/collections/webgl-shaders-code/>
- <https://www.shadertoy.com>
- <http://stack.gl>



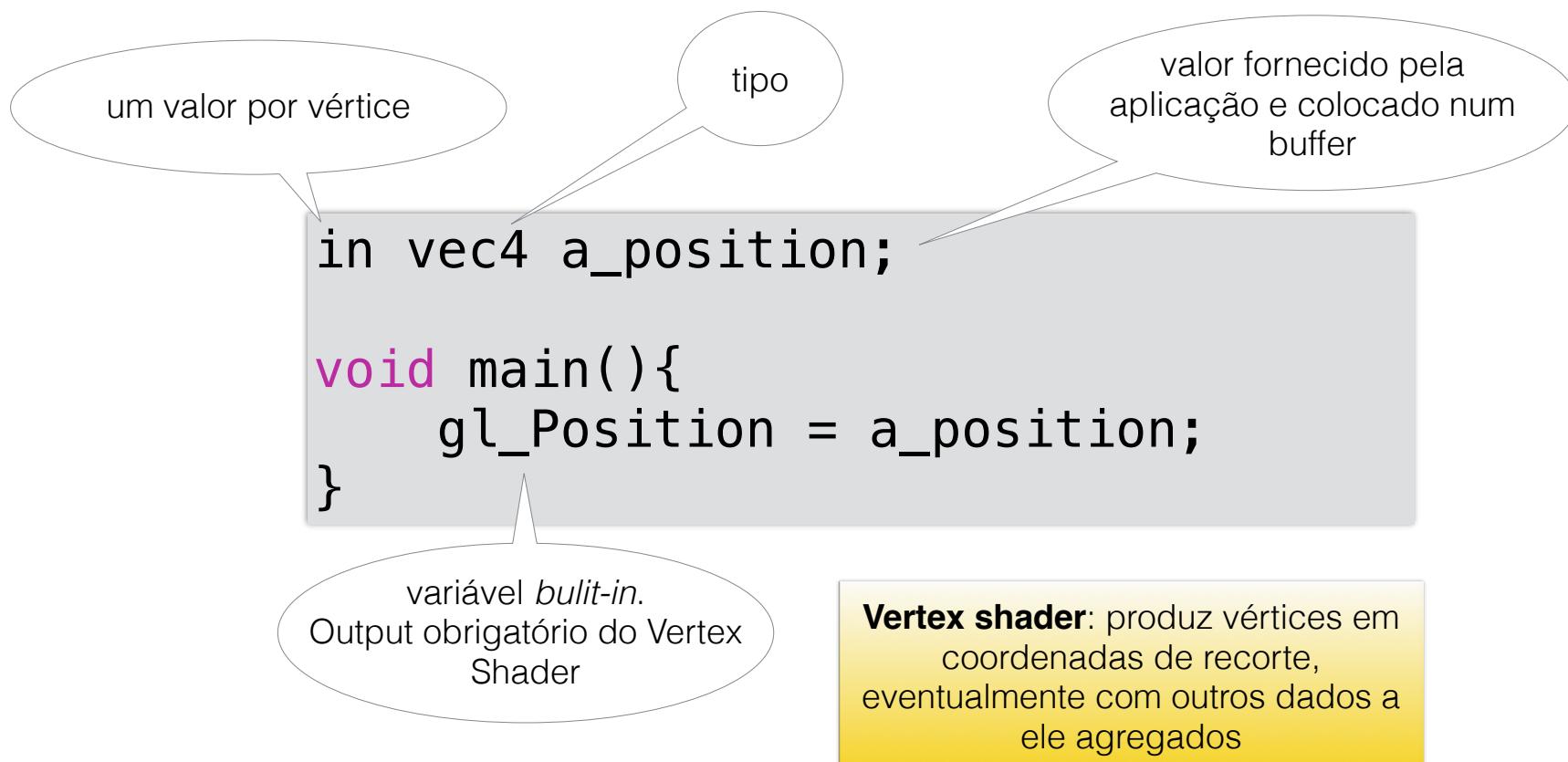
GLSL

- OpenGL Shading Language
- OpenGL 2.0 (e posteriores)
- Linguagem de alto nível (ao estilo do C)
- Tipos de dados úteis para CG:
 - vetores
 - matrizes
 - samplers (para aceder e filtrar texturas)
- A partir do OpenGL 3.1, todas as apps têm que fornecer os seus shaders

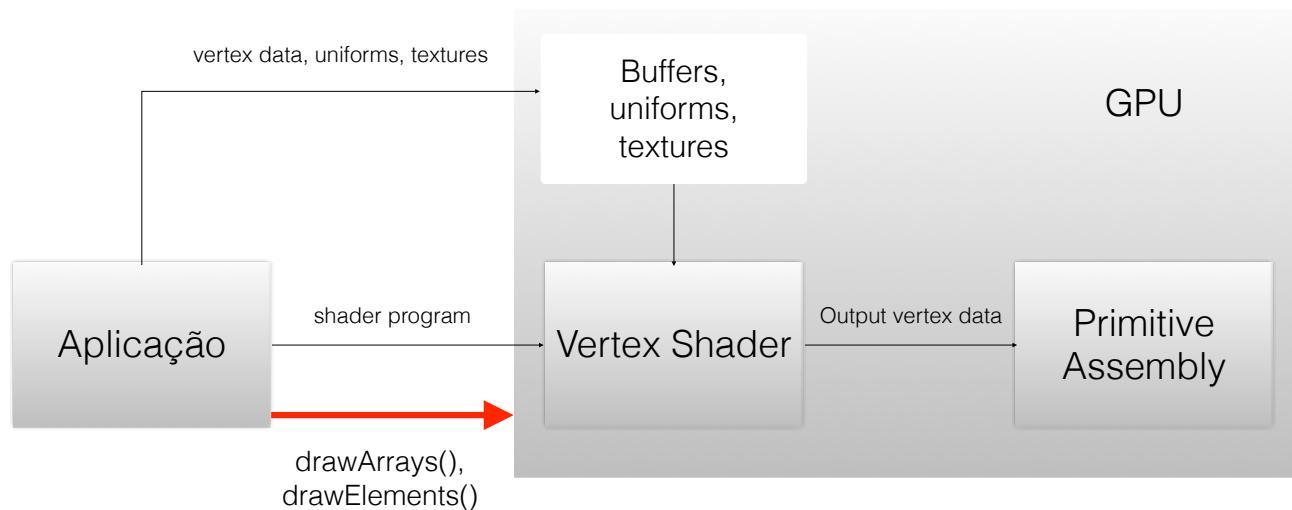
Vertex Shader (GLSL ES V1.00)



Vertex Shader (GLSL ES V3.00)



Modelo de Execução - Vertex Shader



Fragment Shader (GLSL ES V1.00)

precisão do frame buffer

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

variável *built-in*.
Output obrigatório* do
Fragment Shader

Fragment shader: produz cores
para pixels

*Exceto se for usada a instrução discard

Fragment Shader (GLSL ES V3.00)

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

precisão do frame buffer

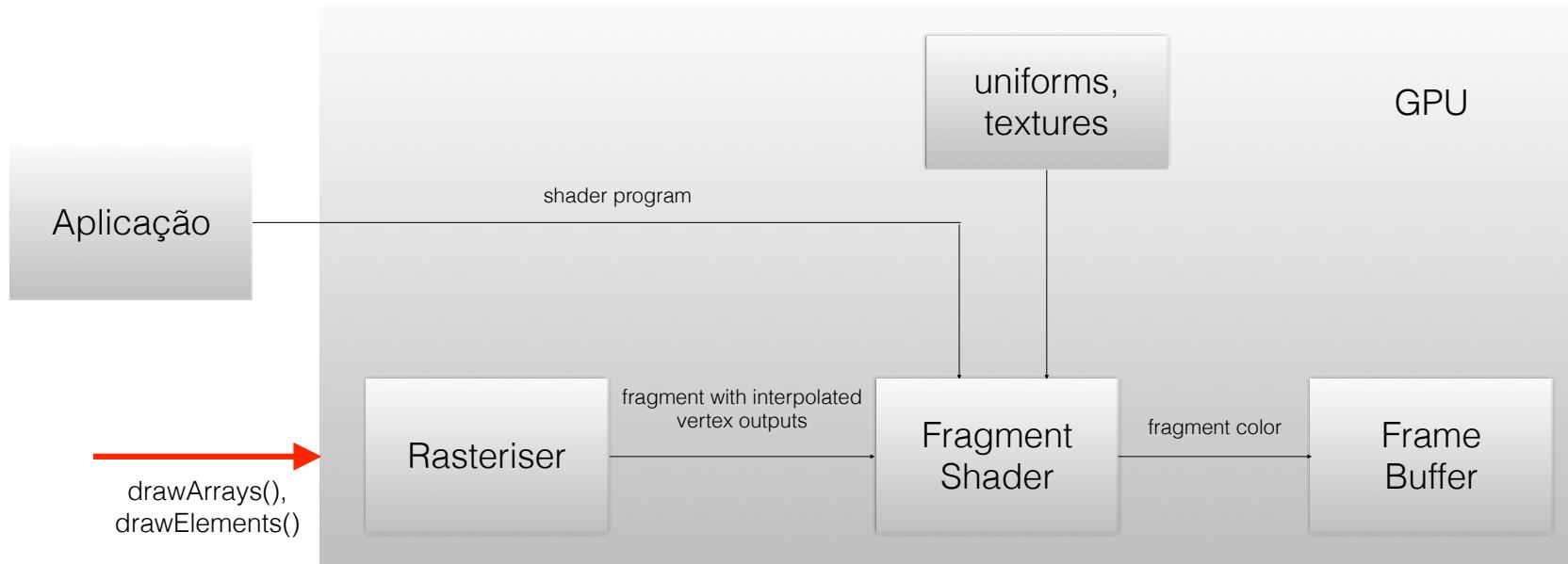
O
programador escolhe o
nome para a variável de
saída

Output obrigatório* do
Fragment Shader

Fragment shader: produz cores
para pixels

*Exceto se for usada a instrução discard

Modelo de Execução - Fragment Shader



Tipos de Dados

- Tipos C: int, float, bool
- Vetores:
 - float vec2, vec3, vec4
 - int (ivec) e boolean (bvec)
- Matrizes (mat2, mat3, mat4):
 - guardadas por colunas
 - acesso standard m[[linha]][[coluna]]
- Construtores ao estilo C++:
 - vec3 a = vec3(1.0, 2.0, 3.0)
 - vec2 b = vec2(a)

Passagem de parâmetros e resultados de funções

- Não há apontadores em GLSL
- Podem usar-se registos (struct) C como resultados de funções
- Os tipos `matN` e `vecN` podem ser passados e retornados de funções:
 - `mat3 inverse(mat3 a)`
- Passagem de parâmetros por valor (Não há apontadores!)

Qualificadores

- GLSL tem alguns dos qualificadores do C++, tal como `const`
- Há qualificadores novos devido ao modelo de execução ser diferente
- Variáveis podem variar:
 - Uma vez por chamada de desenho (`uniform`)
 - Uma vez por vértice (`attribute/in`)
 - Uma vez por fragmento (`varying/in/out`)
 - Em qualquer altura na aplicação (variáveis no código js)
- Os atributos dos vértices podem ser interpolados na fase de varrimento (rasterization) originando dados fornecidos aos fragmentos (`varying/in/out`)

Qualificador attribute

- Os inputs do vertex shader (as variáveis com o qualificador **attribute**) podem variar uma vez por cada vértice (são atributos dos vértices). Os seus valores são colocados em buffers pelo cliente javascript.
- O shader desconhece se os atributos estão guardados em apenas um ou vários buffers.
- Exemplos:

```
in float a_temperature  
in vec3 a_velocity
```

cada vértice, ao ser processado pelo vertex shader, terá disponível o valor do respetivo do atributo, previamente disponibilizado pela aplicação num buffer

Qualificador uniform

- As variáveis com o qualificador **uniform** são constantes durante o desenho da primitiva
- Podem ser modificadas pela aplicação e enviadas aos shaders (ao programa GLSL)
- Não podem ser modificadas pelos shaders (*read only*)
- Usadas para passar informação “global”. Por exemplo:
 - tempo (numa simulação)
 - posição da luz, cor da luz, matriz de projeção, etc.

```
uniform float u_time  
uniform vec3 u_light_pos
```

Os valores permanecem constantes durante todo o desenho das primitivas. Estão acessíveis quer no vertex shader, quer no fragment shader.

Qualificador `varying`

Os valores atribuídos no vertex shader a cada vértice determinam o valor acessível pelo fragment shader (por interpolação)

- variáveis que são passadas do vertex shader para o fragment shader
- São interpoladas automaticamente durante o varrimento
- No WebGL 1, o qualificador GLSL usado é o mesmo em ambos os shaders (**varying**). No WebGL 2, o qualificador varia consoante é uma entrada ou uma saída
- Exemplos:

```
/* Vertex shader */  
varying vec4 v_color;  
varying float v_temperature;
```

```
/* Vertex shader */  
out vec4 v_color  
out float v_temperature
```

```
/* Fragment shader */  
varying vec4 v_color;  
varying float v_temperature;
```

```
/* Fragment shader */  
in vec4 v_color  
in float v_temperature
```

WebGL 1

WebGL 2

Variáveis built-in dos shaders

Shader	Variável	Descrição	Unidades ou Coordenadas
Vertex [out]	<code>highp vec4 gl_Position</code>	Posição transformada do vértice	Coordenadas de recorte
Vertex [out]	<code>mediump float gl_PointSize</code>	Tamanho do ponto (apenas para primitiva gl.POINTS)	Pixels
Fragment [in]	<code>mediump vec4 gl_FragCoord</code>	Posição do fragmento no frame buffer	Coordenadas da janela
Fragment [in]	<code>bool gl_FrontFacing</code>	O fragmento pertence a uma face orientada positivamente (face da frente)	Boolean
Fragment [in]	<code>mediump vec2 gl_PointCoord</code>	Posição do fragmento dentro dum ponto (apenas para a primitiva gl.POINTS)	0.0 a 1.0 (para x e y)
Fragment [out]	<code>mediump vec4 gl_FragColor</code>	Cor do fragmento*	Cor RGBA
Fragment [out]	<code>mediump vec4 gl_FragData[n]</code>	Cor do fragmento para o anexo (plano) n*	Cor RGBA

* substituídos por variáveis definidas pelo programador em WebGL2

Sugestão de convenção para os nomes

- atributos passados ao vertex shader têm os nomes começados por a (a_position, a_color, a_temperature), quer na aplicação (js), quer no shader
- No shader (.glsl):

```
in vec4 a_position;
```

- Na aplicação (.js):

```
var a_position = gl.getAttribLocation(program, "a_position");
```

Convenção para os nomes

- Variáveis interpoladas (varyings), passadas do vertex shader para o fragment shader começam com v (v_color, v_temperature).
- Usa-se o mesmo nome (e tipo) nos dois shaders (**out** no vertex e **in** no fragment)

No vertex shader:

```
in float a_temperature;  
out float v_temperature;  
  
void main() {  
    v_temperature = a_temperature;  
    gl_Position = ...  
}
```

No fragment shader:

```
in float v_temperature;  
  
void main() {  
    color = colorFromTemp(v_temperature);  
}
```

função definida pelo
programador do shader numa
aplicação hipotética

Convenção para os nomes

- As variáveis **uniform** têm nomes começados por u
- Na aplicação (.js):

```
var u_twist = gl.getUniformLocation(program, "u_twist");
```

- Num shader (.glsl):

```
uniform float u_twist;  
  
void main() {  
    gl_Position = posFromTwist(u_twist);  
}
```

função definida pelo programador
do shader numa aplicação
hipotética

Passagem de valores para os atributos

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

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

The number of components per attribute.
Must be 1,2,3 or 4. Default is 4.

Specifies the data type of each component in the array.

Specifies the offset in bytes between the beginning of consecutive vertex attributes.

vertexAttribPointer(index, size, type, normalize, stride, offset)

Index of target attribute in the buffer bound to gl.ARRAY_BUFFER

if gl.TRUE, values are normalised when accessed.
Always use gl.FALSE

Specifies an offset in bytes of the first component of the first vertex attribute in the array.

Buffer layouts: stride & offset

- Atributos separados (com outros dados pelo meio)

Vertex 0				Vertex 1				Vertex 2				Vertex 3				Vertex 4										
x	y	r	g	b	x	y	r	g	b	x	y	r	g	b	x	y	r	g	b	x	y	r	g	b
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104

offset = 0 number = 2 stride = 20

```
gl.vertexAttribPointer(a_position, 2, gl.FLOAT, false, 20, 0);
```

```
gl.vertexAttribPointer(name, number, type, normalize, stride, offset);
```

Buffer layouts: stride & offset

- Atributos separados (com outros dados pelo meio)

Vertex 0					Vertex 1					Vertex 2					Vertex 3					Vertex 4						
x	y	r	g	b	x	y	r	g	b	x	y	r	g	b	x	y	r	g	b	x	y	r	g	b
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104

 offset = 8
  number = 3
  stride = 20

```
gl.vertexAttribPointer(a_color, 3, gl.FLOAT, false, 20, 8);
```

```
gl.vertexAttribPointer(name, number, type, normalize, stride, offset);
```

Buffer layouts: stride & offset

- Atributos juntos (sem outros dados pelo meio)

Vertex 0	Vertex 1	Vertex 2	Vertex 3	Vertex 4	Vertex 5	Vertex 0	Vertex 1	Vertex 2	Vertex 3											
x	y	x	y	x	y	x	y	r	g	b	r	g	b	r	g	b	r	
0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80

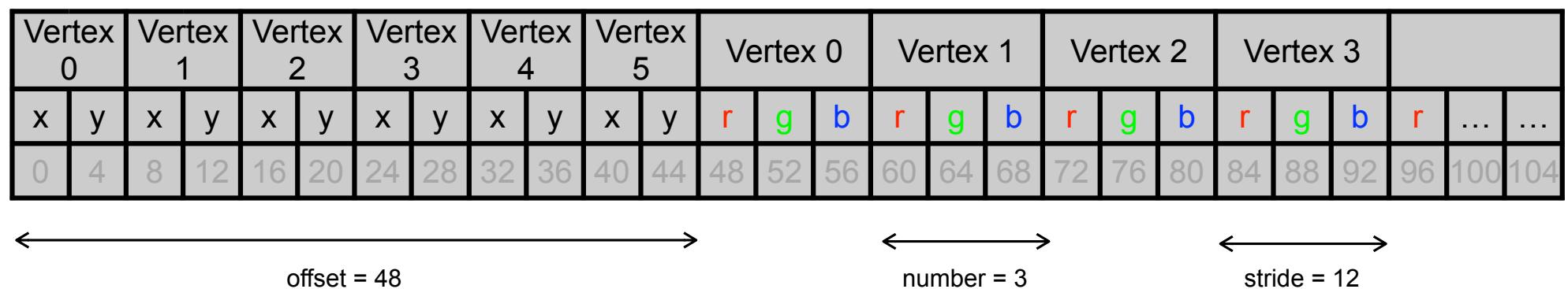
offset = 0 number = 2 stride = 8

```
gl.vertexAttribPointer(a_position, 2, gl.FLOAT, false, 8, 0);
```

```
gl.vertexAttribPointer(name, number, type, normalize, stride, offset);
```

Buffer layouts: stride & offset

- Atributos juntos (sem outros dados pelo meio)



```
gl.vertexAttribPointer(a_color, 3, gl.FLOAT, false, 12, 48);
```

```
gl.vertexAttribPointer(name, number, type, normalize, stride, offset);
```

Passagem de valores constantes (uniform)

Na aplicação:

```
var color = vec4(1.0, 0.0, 0.0, 1.0);
u_color = gl.getUniformLocation(program, "u_color");
gl.uniform4fv(u_color, color);
```

Pertence a uma família de
métodos: uniform*

usando o tipo de dados vec4

No fragment shader (de forma idêntica no vertex shader):

```
uniform vec4 u_color;
out vec4 color;

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

Passagem de valores constantes (uniform)

Na aplicação:

```
var color = [1.0, 0.0, 0.0, 1.0];
u_color = gl.getUniformLocation(program, "u_color");
gl.uniform4fv(u_color, color);
```

Alternativa usando um array javascript diretamente

Pertence a uma família de métodos: uniform*

No fragment shader (de forma idêntica no vertex shader):

```
uniform vec4 u_color;
out vec4 color;

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

GLSL: operadores e funções

- Funções C standard:
 - trigonométricas
 - aritméticas
- vetoriais: normalize, reflect, length, dot
- overloading de operadores para vetores e matrizes

```
mat4 a;  
vec4 b, c, d, e;  
c = a*b; // vetor coluna  
d = b*a; // vetor linha  
e = c*d; // component-wise multiplication
```

GLSL: swizzling e seleção

- Podemos nos referir aos elementos dos tipos vetor e matriz 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 $a.p$ são exatamente a mesma componente
- Swizzling permite manipular as componentes:

```
vec4 a, b;  
  
a.yz = vec2(1.0, 2.0);  
  
b = a.yxzw;
```

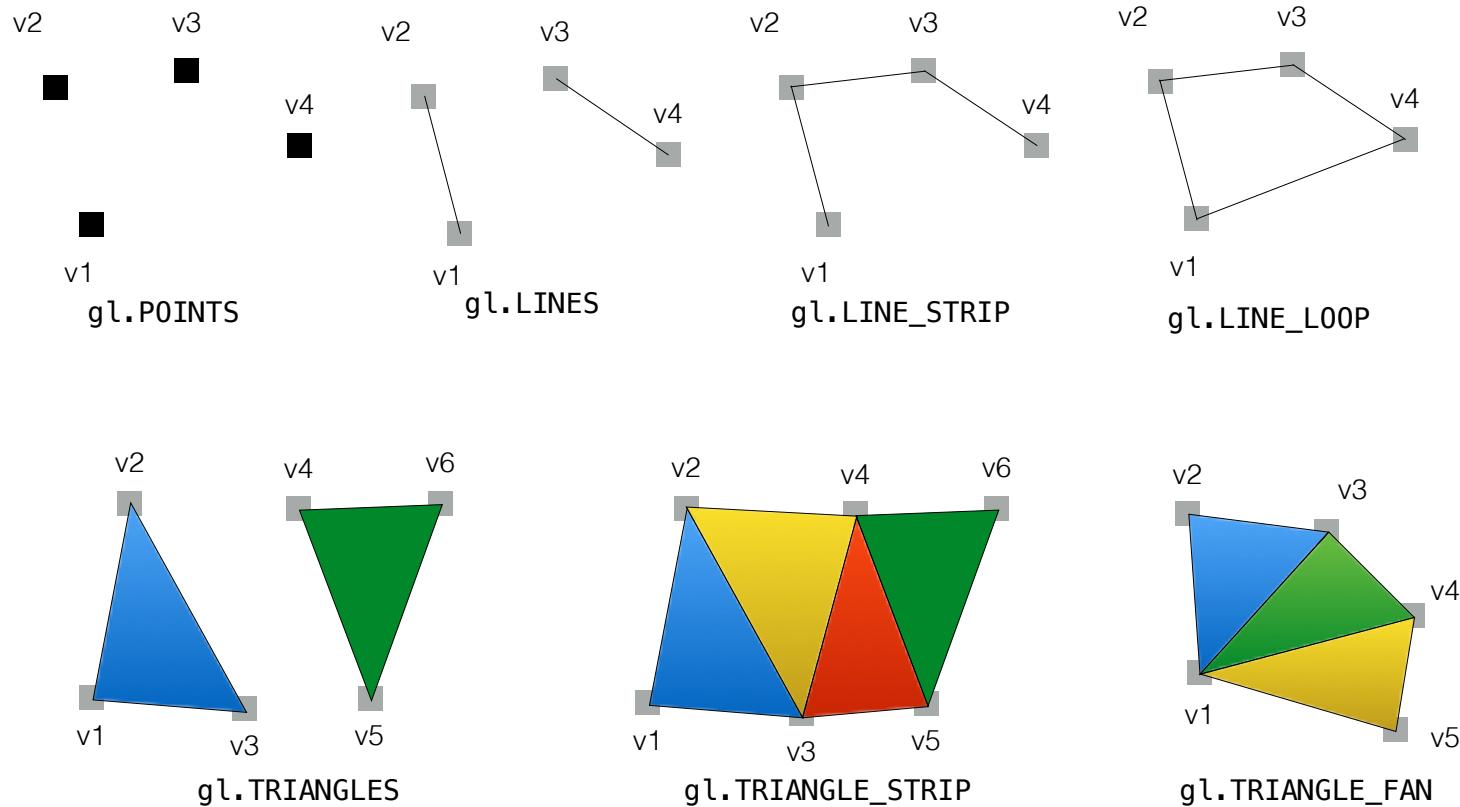
[WebGL 1.0 Reference Card](#)

WebGL Primitivas

Objetivos

- Conhecer as primitivas WebGL
- Converter polígonos em triângulos
- Interpolação de atributos durante o varrimento

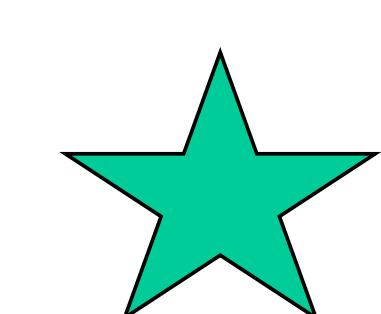
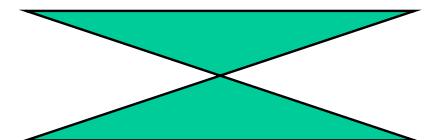
Primitivas WebGL



Polígonos (>3 lados)

- WebGL apenas suporta triângulos porque:
 - são simples: as arestas não se cruzam
 - são convexos: todos os pontos num segmento de reta entre dois pontos no polígono também pertencem ao polígono
 - são planos: todos os vértices (e pontos no seu interior) pertencem ao mesmo plano
- As aplicações são obrigadas a decompor (*tessellation*) polígonos em triângulos (triangulação)
- OpenGL 4.1 inclui um *tessellator*, mas não o WebGL

Exemplos de polígonos:

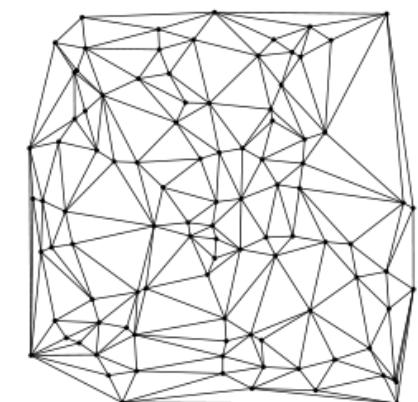


Teste de Polígonos

- Conceptualmente é fácil testar um polígono quanto à simplicidade e convexidade...
 - ... mas é dispendioso
 - versões anteriores assumiam simplicidade e convexidade e deixavam o teste para a aplicação
- a versão atual apenas desenha triângulos
- é necessário um algoritmo para triangular um polígono arbitrário

Bons e maus triângulos

- Triângulos longos e finos podem gerar artefactos
 - Triângulos equiláteros são os mais adequados
 - objetivo: maximizar o ângulo mínimo
-
- Delaunay triangulation: para triangular uma malha de pontos não estruturados



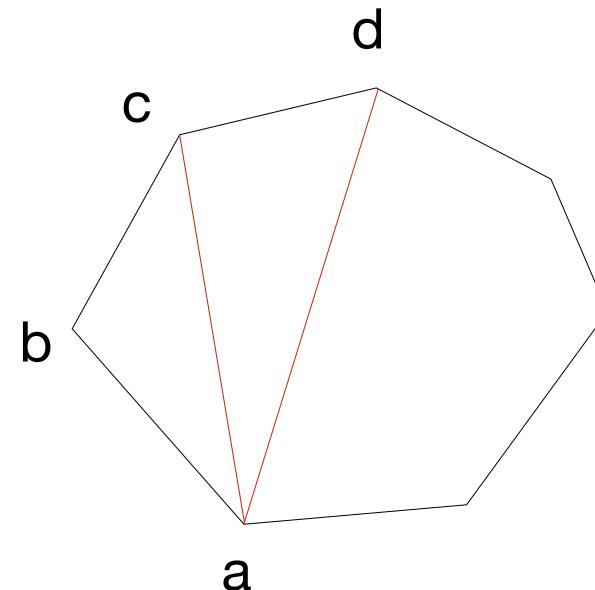
Delaunay triangulation

Triangulação de polígonos convexos

Dado um polígono convexo (a,b,c,d,...),
Gerar os triângulos: [a,b,c], [a,c,d], ...

Algoritmo

1. Escolhe-se um vértice pivot, neste caso a.
2. Formam-se os triângulos varrendo os vértices no sentido horário ou no seu inverso, usando sempre a e o último vértice do triângulo anterior.



Nota: Pode implementar-se em WebGL
usando a primitiva gl.TRIANGLE_FAN

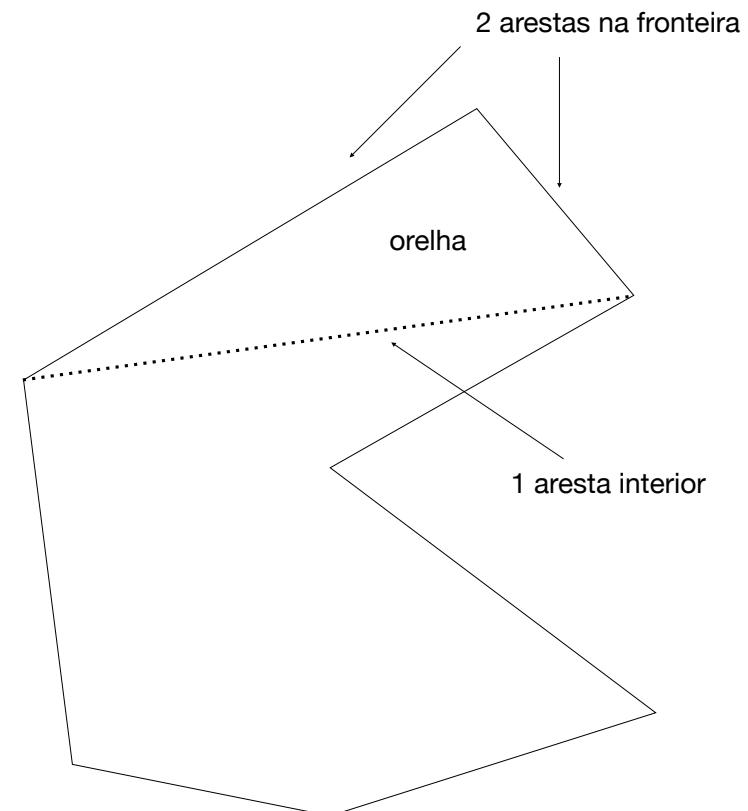
Triangulação de polígonos côncavos

Pode aplicar-se o algoritmo *earcut*.

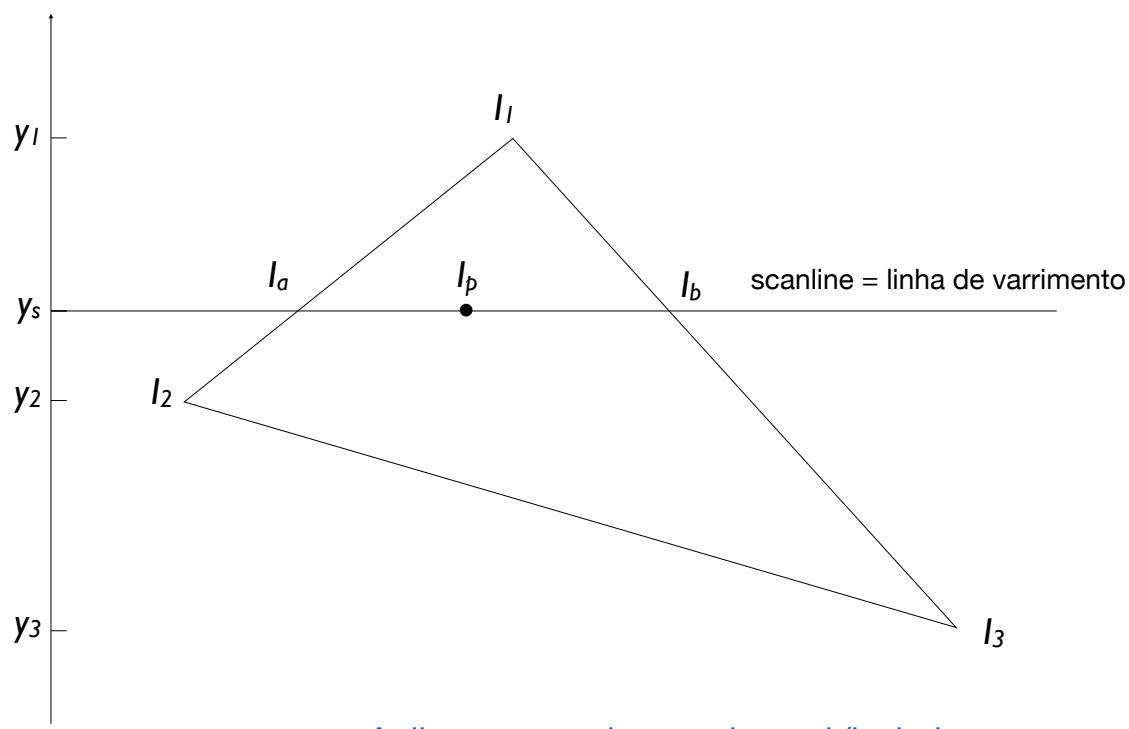
Os polígonos simples (sem buracos) e côncavos possuem pelo menos 2 orelhas.

Uma orelha é um triângulo formado por duas arestas consecutivas (do polígono original), estando a aresta que fecha o triângulo completamente no interior do polígono.

- Encontrar uma das orelhas e usá-la para decompor o polígono.
- Repetir o passo anterior ao polígono restante da remoção da orelha.



Interpolação de atributos (no espaço da imagem)



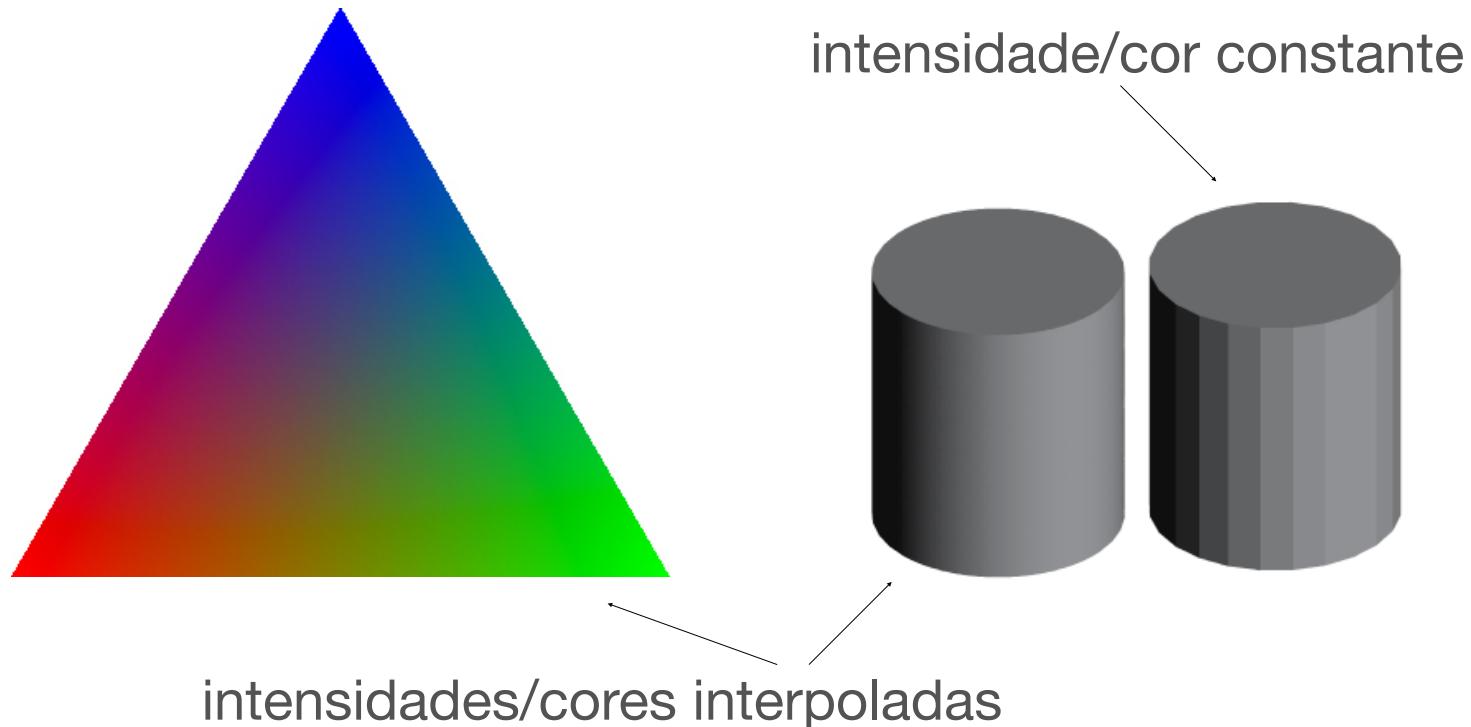
$$I_a = I_2 + (I_1 - I_2) \frac{y_s - y_2}{y_1 - y_2}$$

$$I_b = I_3 + (I_1 - I_3) \frac{y_s - y_3}{y_1 - y_3}$$

$$I_p = I_a + (I_b - I_a) \frac{x_p - x_a}{x_b - x_a}$$

Aplica-se a qualquer valor real (isolado ou como componentes de vetores)

Interpolação de atributos



Exemplo de interpolação aplicado a cores (RGB) ou luminância