

Shaders - Básico

Abordagem de shaders em OpenGL 3.2 e GLSL 1.50

Filipi Nascimento Silva

Apresentação para o Curso de Computação Gráfica - ICMC (USP)

Resumo

- **História**
 - Origem
 - Shaders em GPUs
 - Shaders em APIs
- **Pipeline**
 - Fixo
 - Programável
 - Vantagens do pipeline programável
- **Shaders em OpenGL**
 - Versões
 - Core Profile vs Compatibility Profile
 - Requisitos
- **Linguagem GLSL**
 - Definições
 - Exemplo Simples
 - Exemplo menos Simples
 - Iluminação básica
 - Phong
 - Texturas
 - Geometry Shader
- **Usando GLSL no OpenGL**
 - Compilando os programas
 - Enviando dados à GPU
- **Futuro**
- **Bibliografia**

História

Origem

- O termo **shader** surgiu em 1980's na **LucasFilms**.
 - Ficou popularmente conhecido com o software **Renderman**.
- Inicialmente referia-se somente a **Pixel Shaders**.
 - O termo foi estendido para os outros shaders que usamos hoje.



Fontes

Cartaz Toy Story: http://en.wikipedia.org/wiki/File:Movie_poster_toy_story.jpg

História

Origem

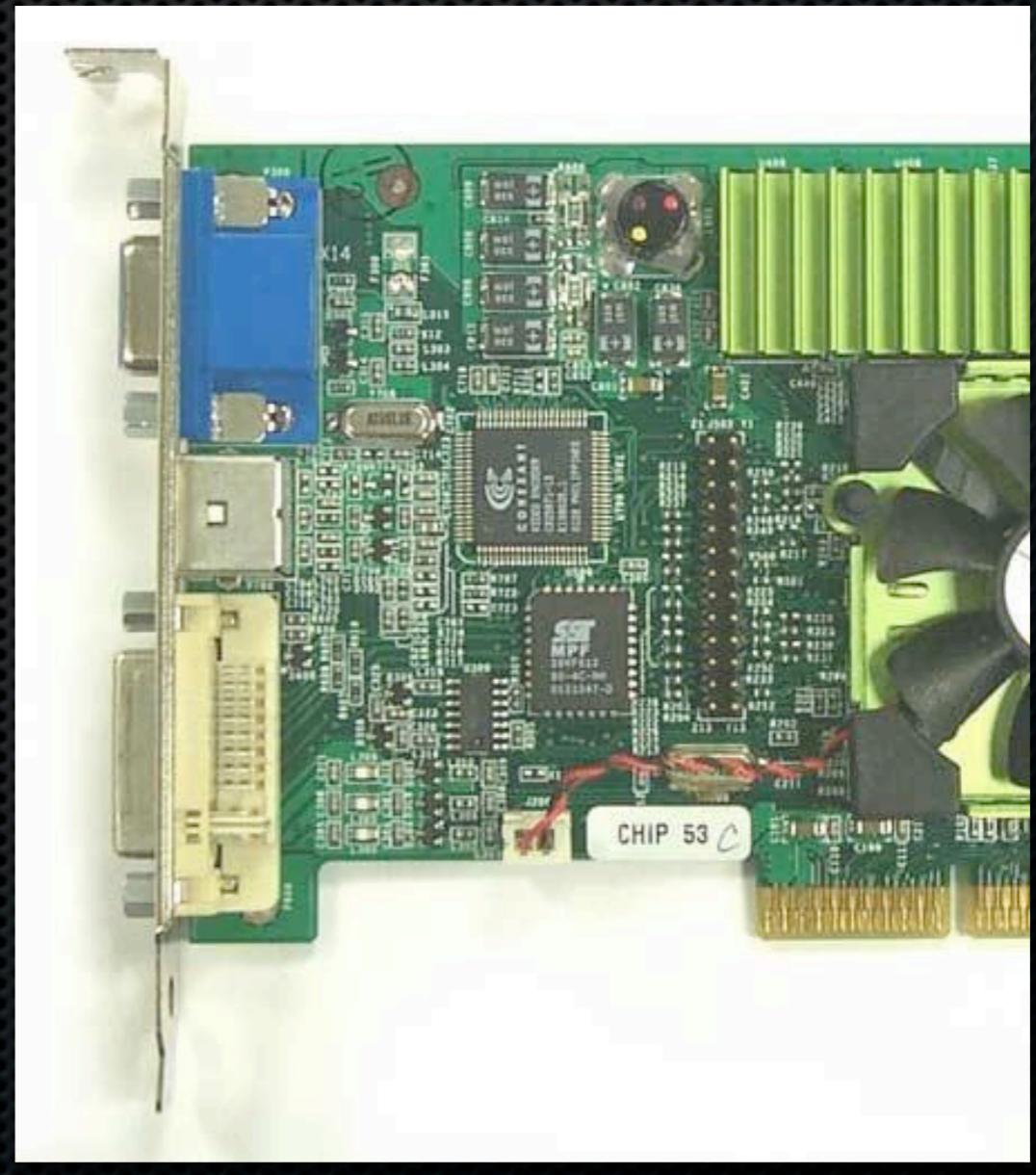
The **Renderman Shading Language** allows material definitions of surfaces to be described in not only a simple manner, but also **highly complex** and **custom manner** using a **C like language**. Using this method as **opposed** to a **pre-defined** set of **materials allows** for **complex procedural** textures, **new shading models** and **programmable lighting**. Another thing that sets the renderers based on the RISpec apart from many other renderers, is the ability to **output arbitrary variables** as an image—surface normals, separate lighting passes and **pretty much anything** else can be **output from the renderer** in one pass.

<http://wiki.cgsociety.org/index.php/Renderman>

História

Shaders em GPUs

- Em 2001, **Geforce 3** e a **Radeon 8500** foram as primeiras **GPUs** a suportarem algum tipo de **pipeline programável**.
 - Algumas desenvolvedoras de jogos adotaram a tecnologia.
 - Permitiu diversos novos efeitos que poderiam ser usados em realtime.
- Em 2002, **GeForce FX** e **Radeon 9700** foram as primeiras a suportar tanto **Vertex** quanto **Pixel Shaders**.



Fontes

Imagens: http://www.xbitlabs.com/articles/graphics/display/xabre400_3.html

História

Shaders em GPUs

- Em 2001, **Geforce 3** e a **Radeon 8500** foram as primeiras **GPUs** a suportarem algum tipo de **pipeline programável**.
 - Algumas desenvolvedoras de jogos adotaram a tecnologia.
 - Permitiu diversos novos efeitos que poderiam ser usados em realtime.
- Em 2002, **GeForce FX** e **Radeon 9700** foram as primeiras a suportar tanto **Vertex** quanto **Pixel Shaders**.



Fontes

Imagens: http://www.xbitlabs.com/articles/graphics/display/xabre400_3.html

História

Shaders em GPUs

- Em 2001, **Geforce 3** e a **Radeon 8500** foram as primeiras **GPUs** a suportarem algum tipo de **pipeline programável**.
 - Algumas desenvolvedoras de jogos adotaram a tecnologia.
 - Permitiu diversos novos efeitos que poderiam ser usados em realtime.
- Em 2002, **GeForce FX** e **Radeon 9700** foram as primeiras a suportar tanto **Vertex** quanto **Pixel Shaders**.



Fontes

Imagens: http://www.xbitlabs.com/articles/graphics/display/xabre400_3.html

História

Shaders em GPUs

- Em 2001, **Geforce 3** e a **Radeon 8500** foram as primeiras **GPUs** a suportarem algum tipo de **pipeline programável**.
 - Algumas desenvolvedoras de jogos adotaram a tecnologia.
 - Permitiu diversos novos efeitos que poderiam ser usados em realtime.
- Em 2002, **GeForce FX** e **Radeon 9700** foram as primeiras a suportar tanto **Vertex** quanto **Pixel Shaders**.

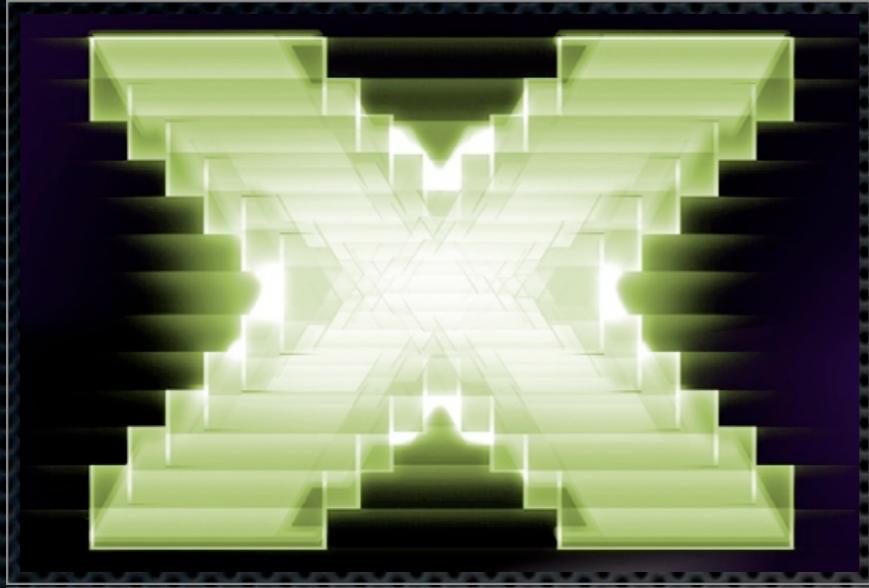


Fontes

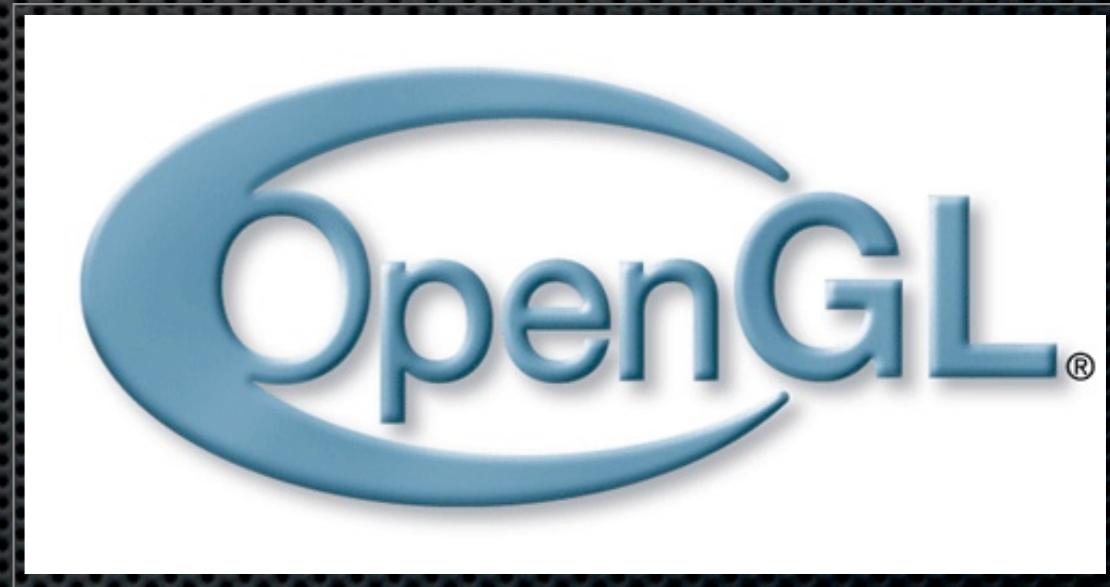
Imagens: http://www.xbitlabs.com/articles/graphics/display/xabre400_3.html

História

Shaders em APIs



- **OpenGL:** Suporta shaders **GLSL** desde a versão 2.0 (previamente em extensões).
 - A versão **3.1** suporta **Geometry Shaders**.
 - A versão **4.2** suporta **Tesselation Shaders**.



- **Cg:** Linguagem de shaders da **nvidia**, atualmente capaz de gerar tanto **GLSL** quanto **HLSL**.

Fontes

Logo DirectX : <http://en.wikipedia.org/wiki/File:Directx9.png> Logo OpenGL <http://kronos.org/> Logo CG: <http://developer.nvidia.com/cg-toolkit>

Resumo

- **História**
 - Origem
 - Shaders em GPUs
 - Shaders em APIs
- **Pipeline**
 - Fixo
 - Programável
 - Vantagens do pipeline programável
- **Shaders em OpenGL**
 - Versões
 - Core Profile vs Compatibility Profile
 - Requisitos
- **Linguagem GLSL**
 - Definições
 - Exemplo Simples
 - Exemplo menos Simples
 - Iluminação básica
 - Phong
 - Texturas
 - Geometry Shader
- **Usando GLSL no OpenGL**
 - Compilando os programas
 - Enviando dados à GPU
- **Futuro**
- **Bibliografia**

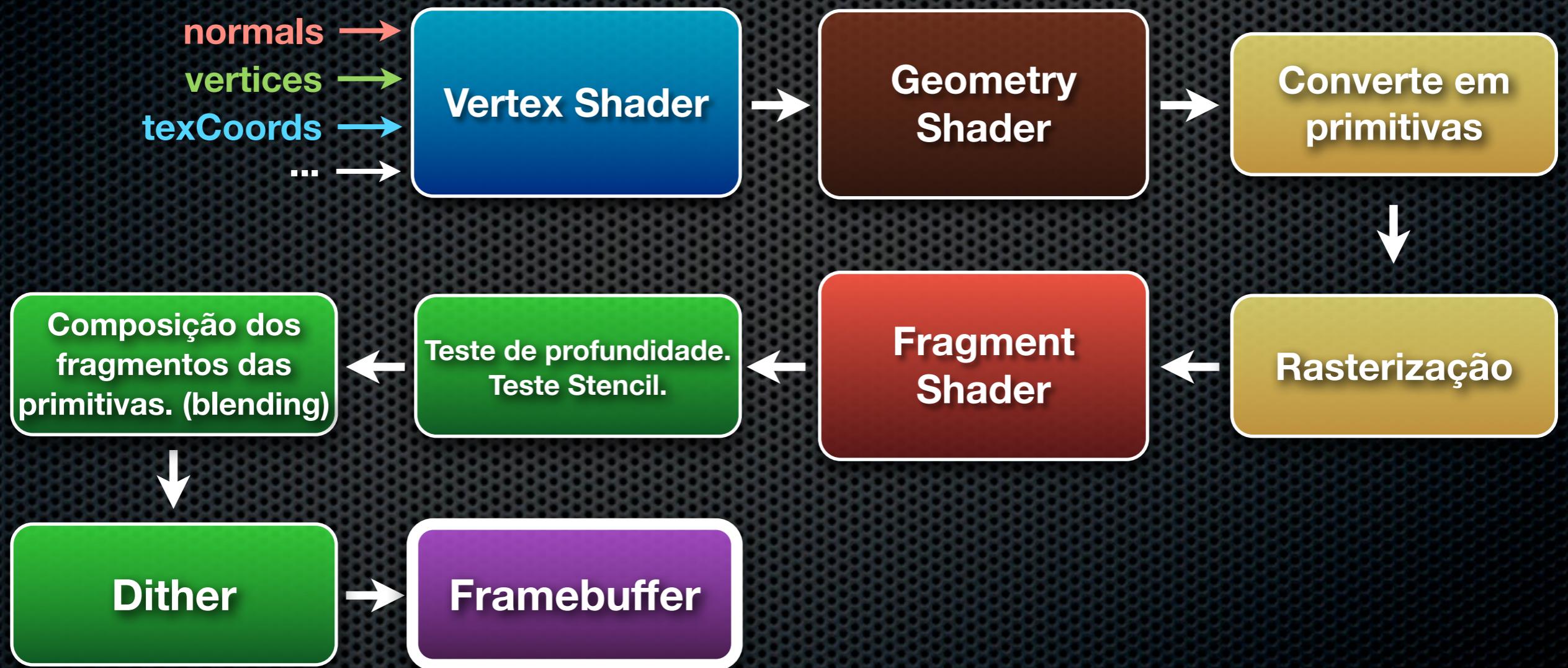
Pipeline

Pipeline fixo



Pipeline

Pipeline Programável



Pipeline

Pipeline Programável (OpenGL 4 e DirectX 11)



Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (OpenGL 4 e DirectX 11)

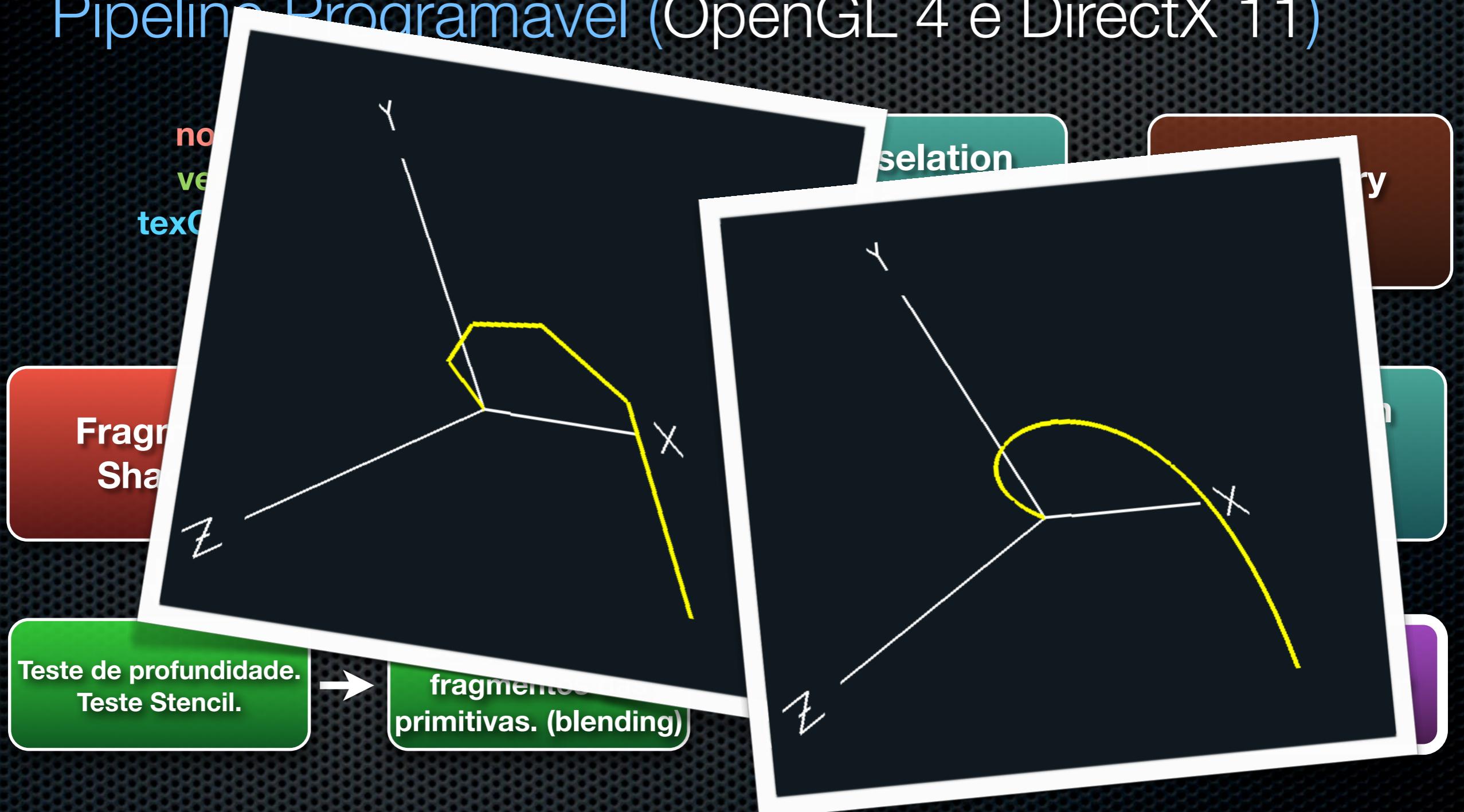


Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (OpenGL 4 e DirectX 11)

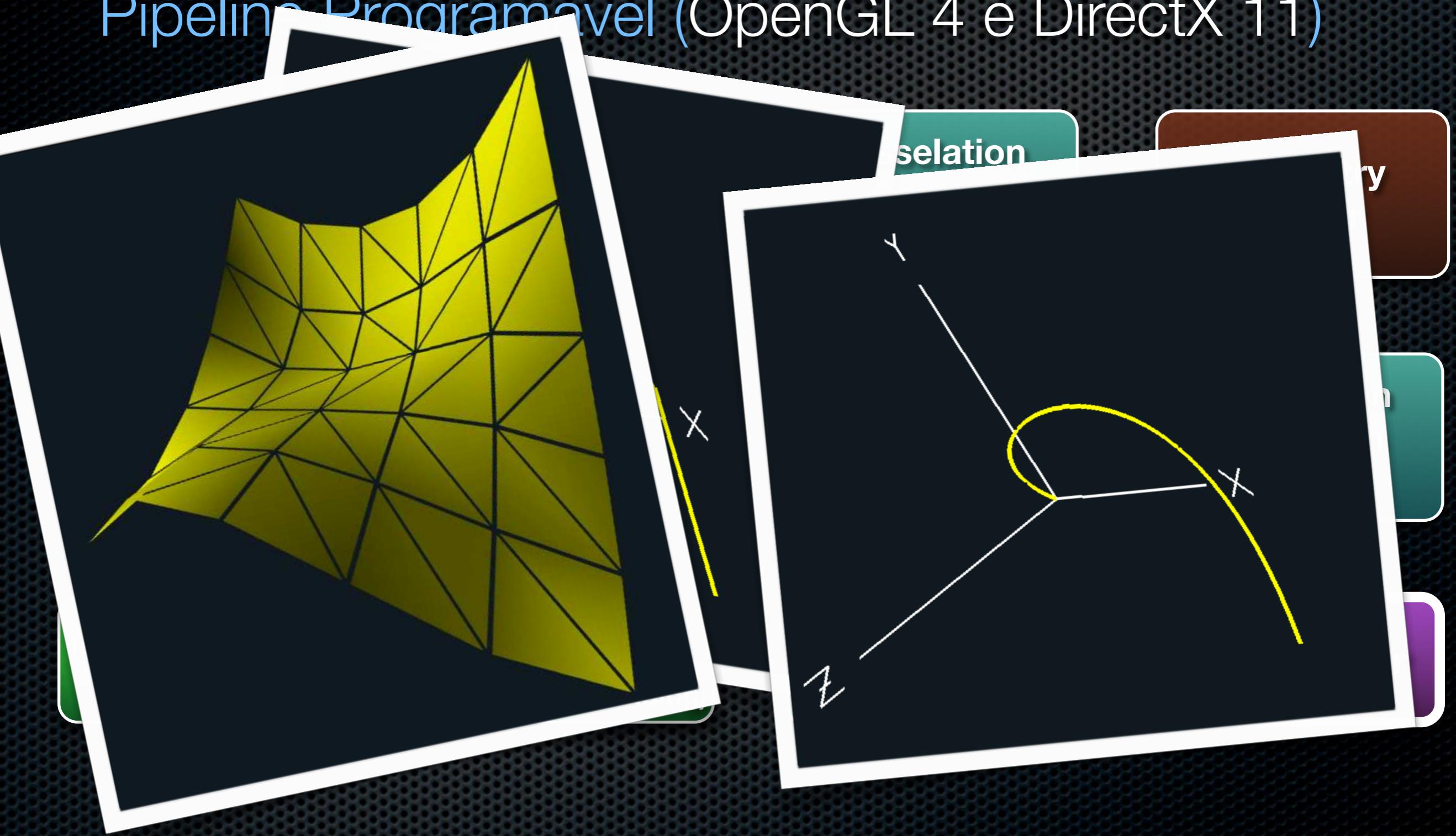


Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (OpenGL 4 e DirectX 11)

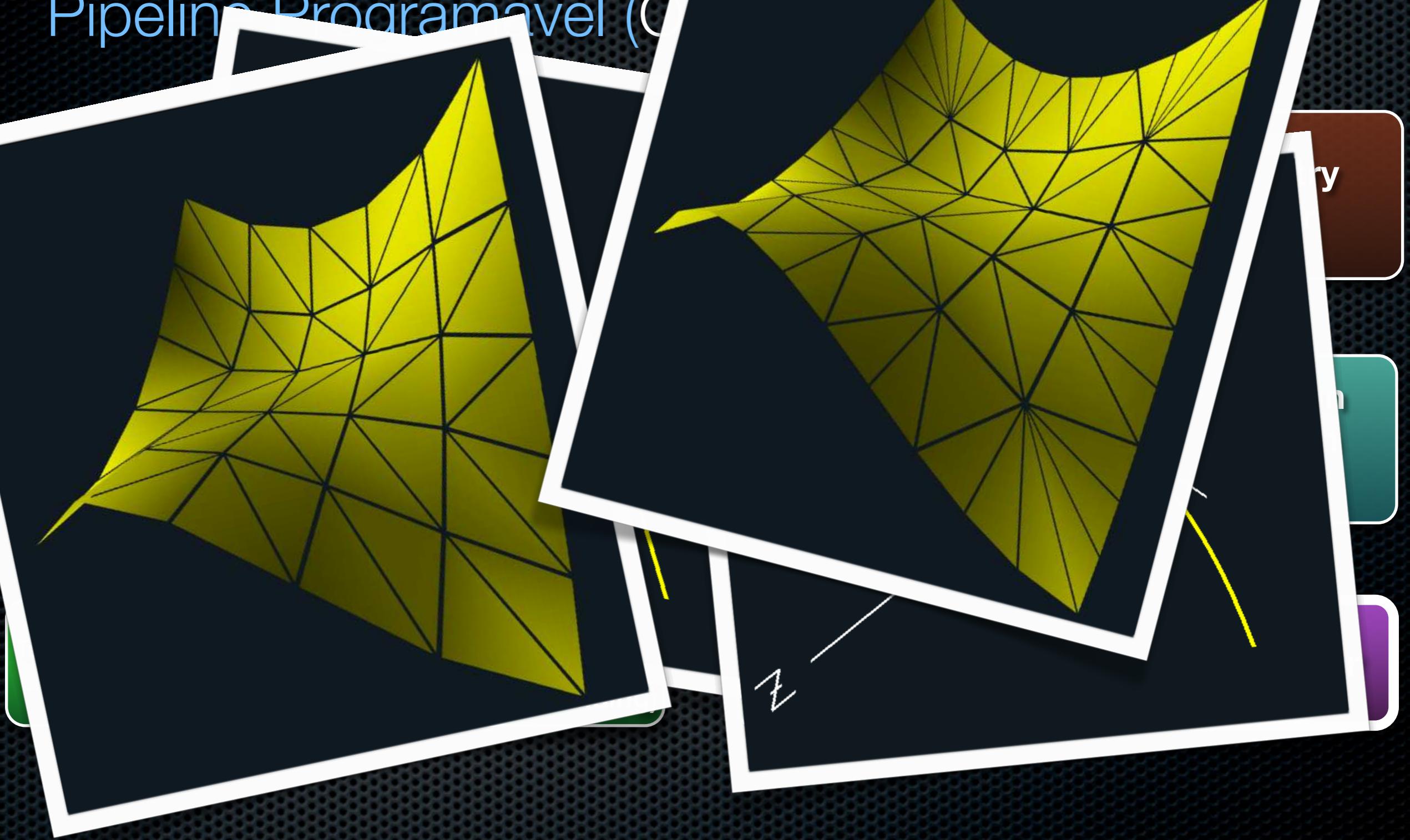


Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (C)

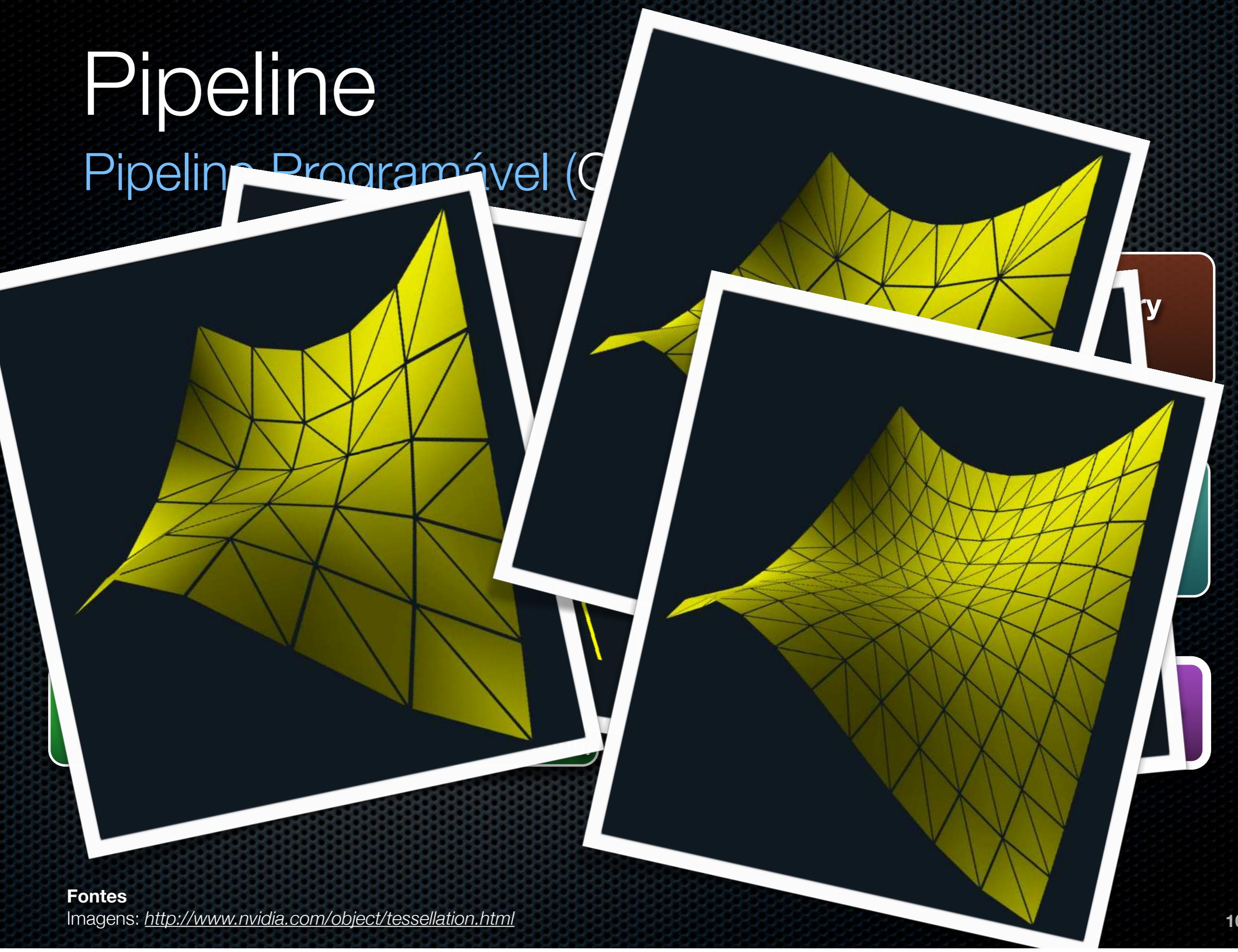


Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (C)

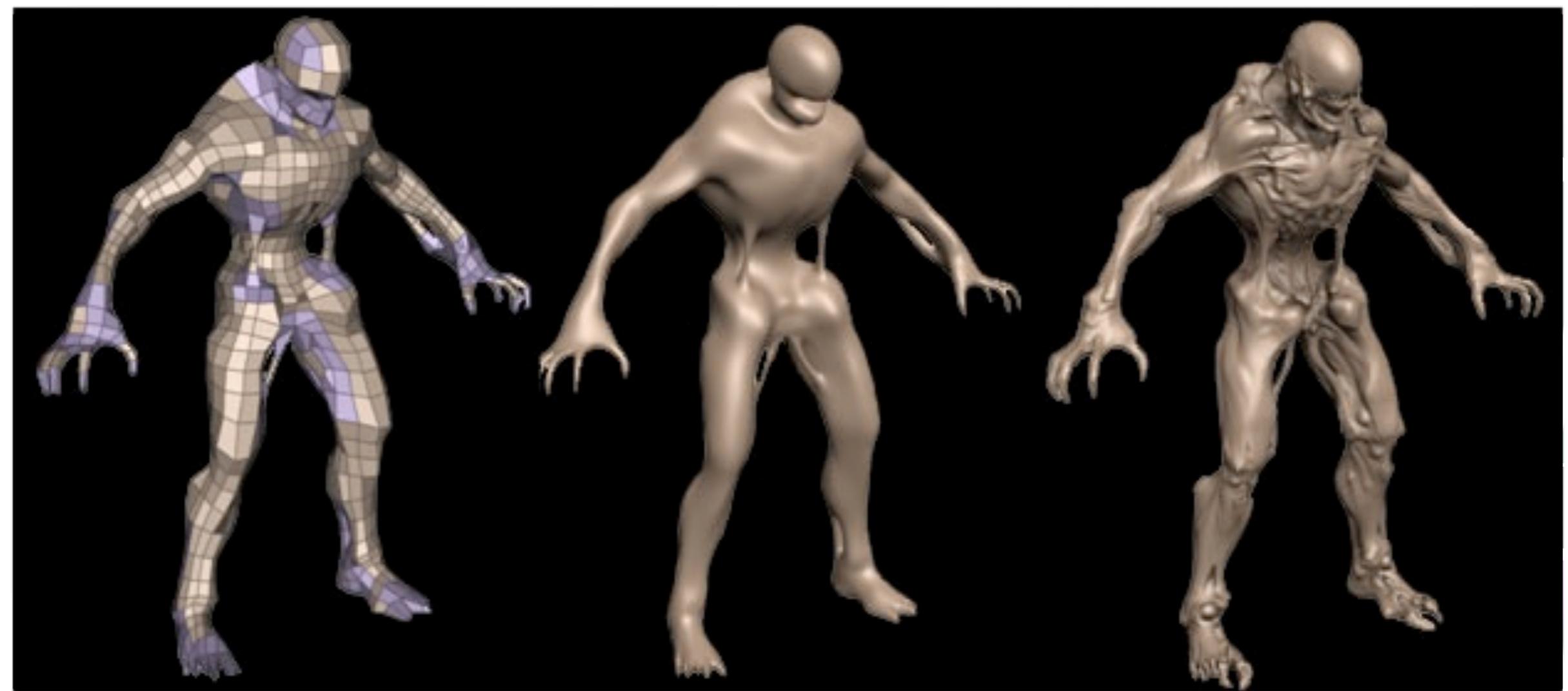


Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Pipeline Programável (C)



Fontes

Imagens: <http://www.nvidia.com/object/tessellation.html>

Pipeline

Vantagens de um Pipeline Programável

- Possibilita **APIs** mais **simples e eficientes** (**RISC** vs **CISC**).
- Permite maior **versatilidade** de uso. (**GPGPU** nasceu assim!).
- É mais **fácil** de **otimizar**.
- **Evita** o surgimento de **hacks** e recursos fora das especificações como **extensions**.
- É muito mais **divertido!!!**



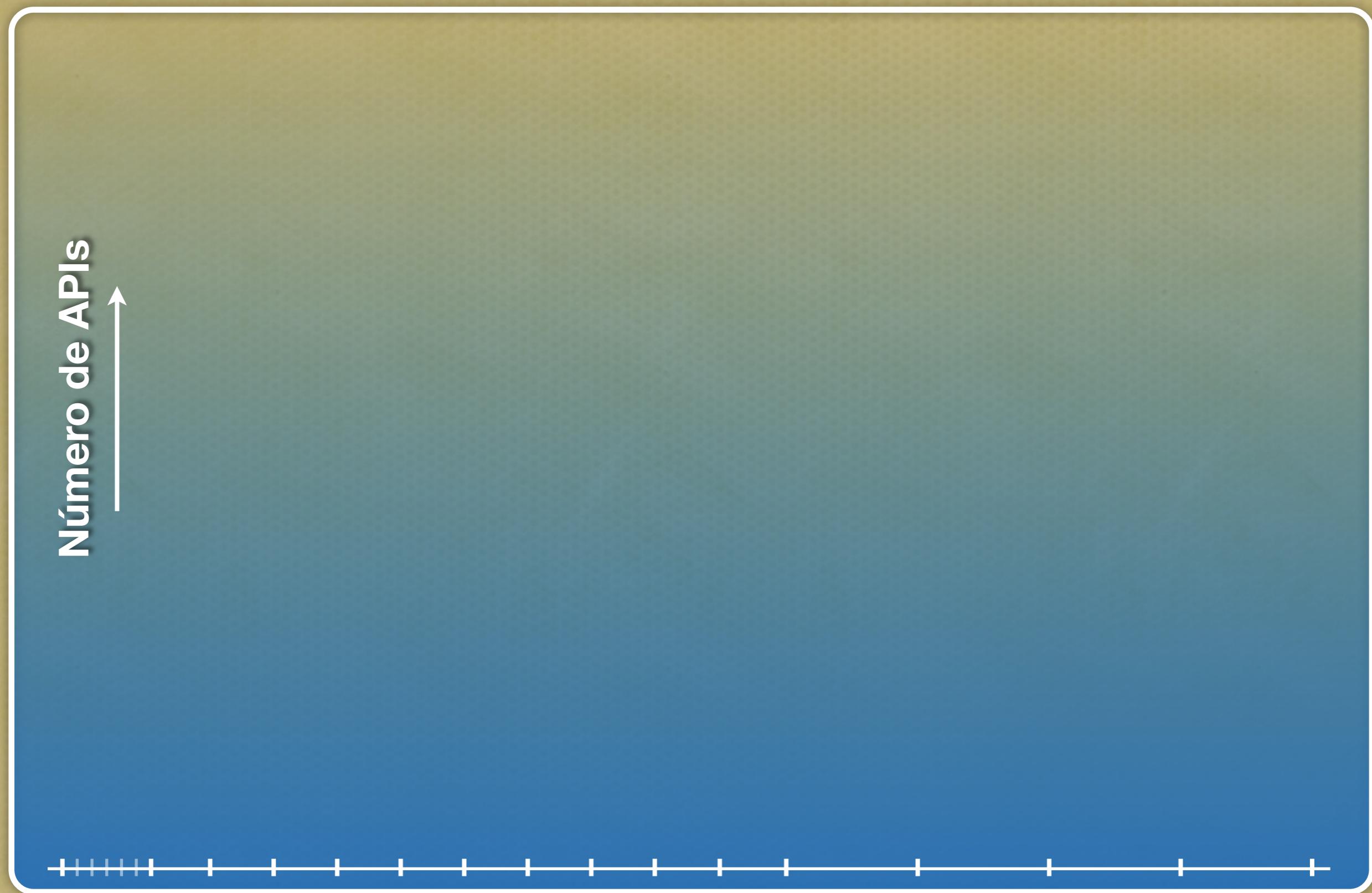
Fontes

Screenshot de Super Street Fighter IV Arcade Edition: STREET FIGHTER is Owned by CAPCOM. All Rights Reserved. STREET FIGHTER and all logos, character names and distinctive likenesses thereof are trademarks of CAPCOM. Image used in fair use.

Resumo

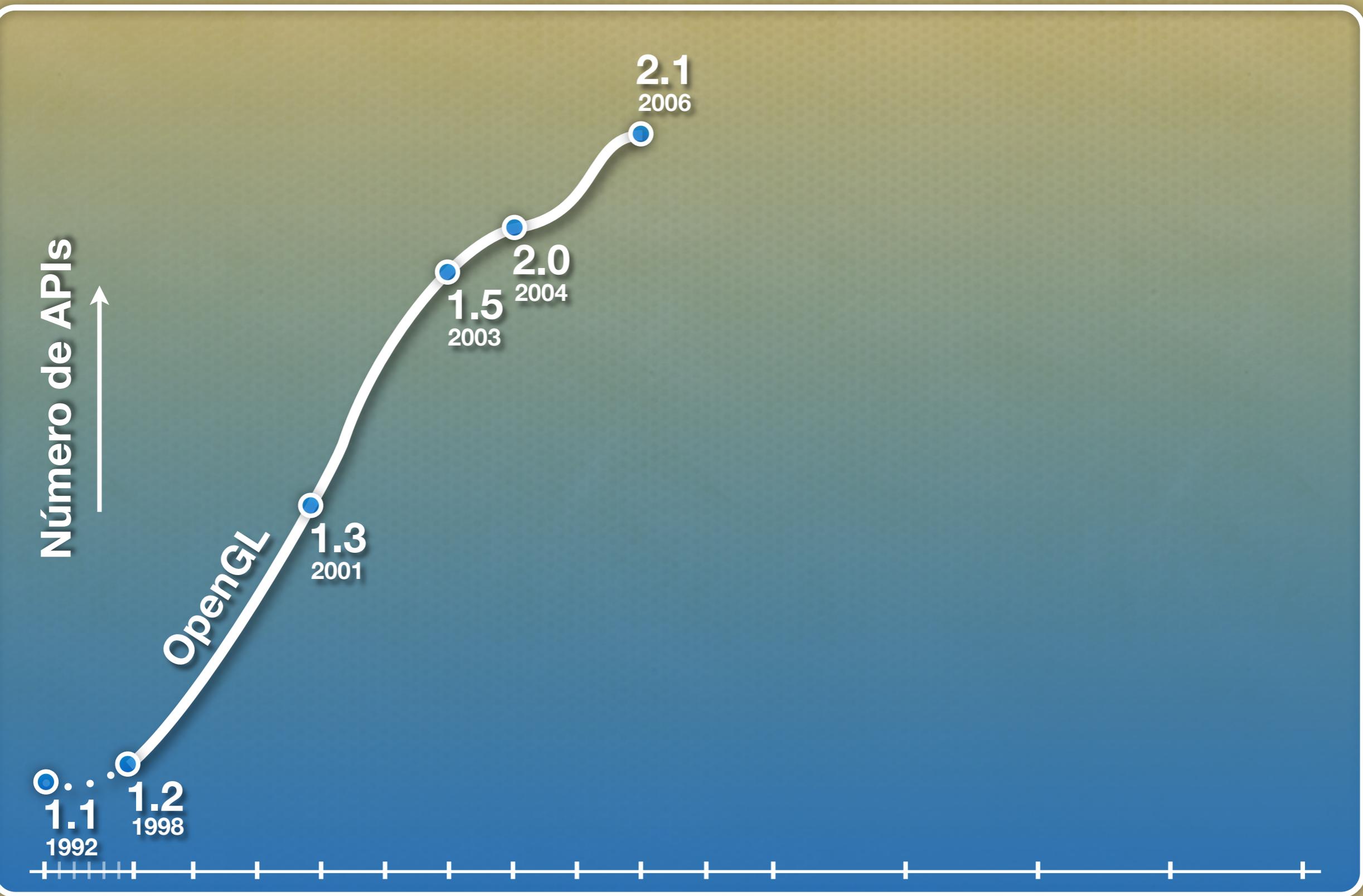
- **História**
 - Origem
 - Shaders em GPUs
 - Shaders em APIs
- **Pipeline**
 - Fixo
 - Programável
 - Vantagens do pipeline programável
- **Shaders em OpenGL**
 - Versões
 - Core Profile vs Compatibility Profile
 - Requisitos
- **Linguagem GLSL**
 - Definições
 - Exemplo Simples
 - Exemplo menos Simples
 - Iluminação básica
 - Phong
 - Texturas
 - Geometry Shader
- **Usando GLSL no OpenGL**
 - Compilando os programas
 - Enviando dados à GPU
- **Futuro**
- **Bibliografia**

OpenGL - História das versões



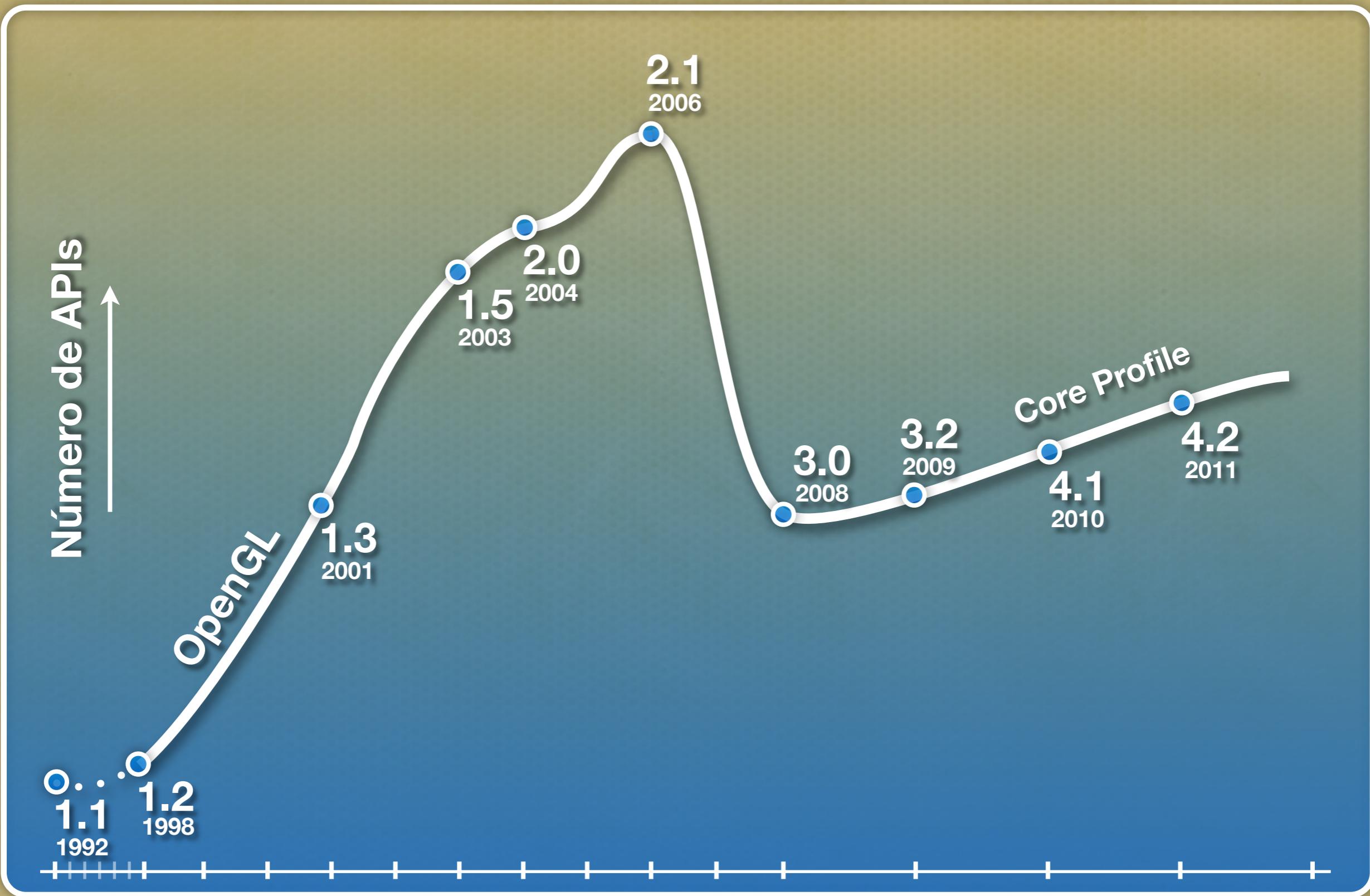
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



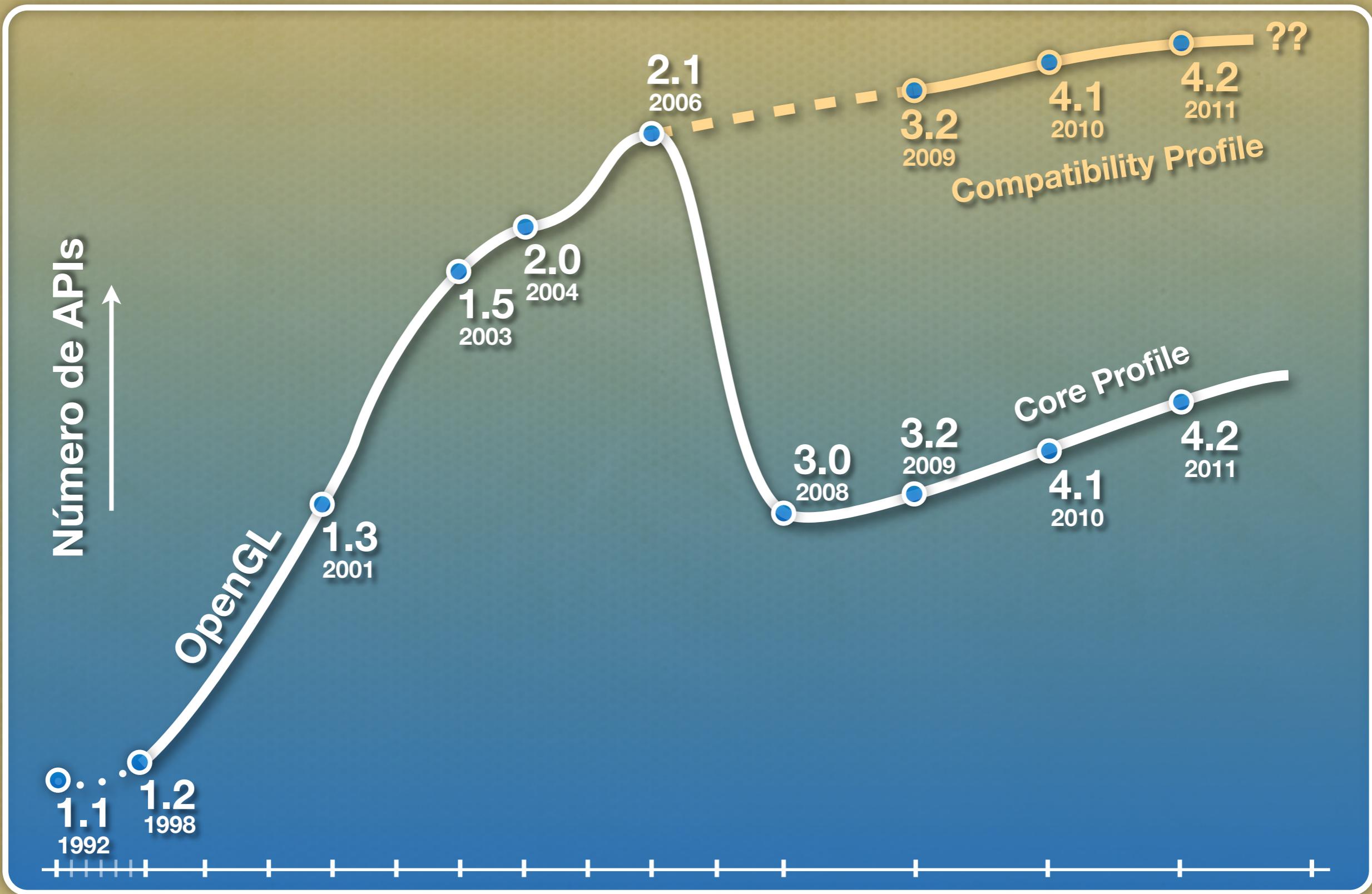
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



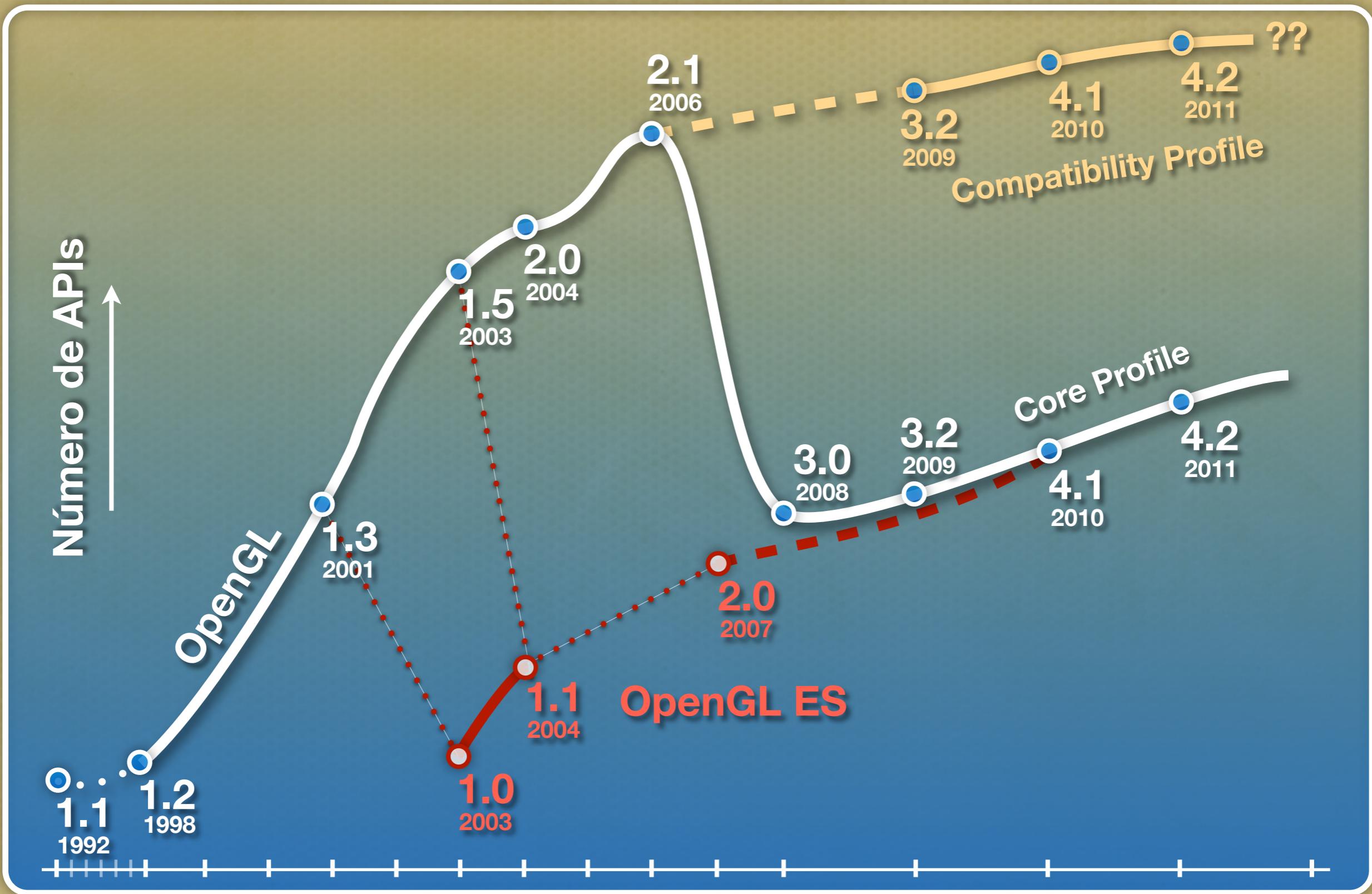
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



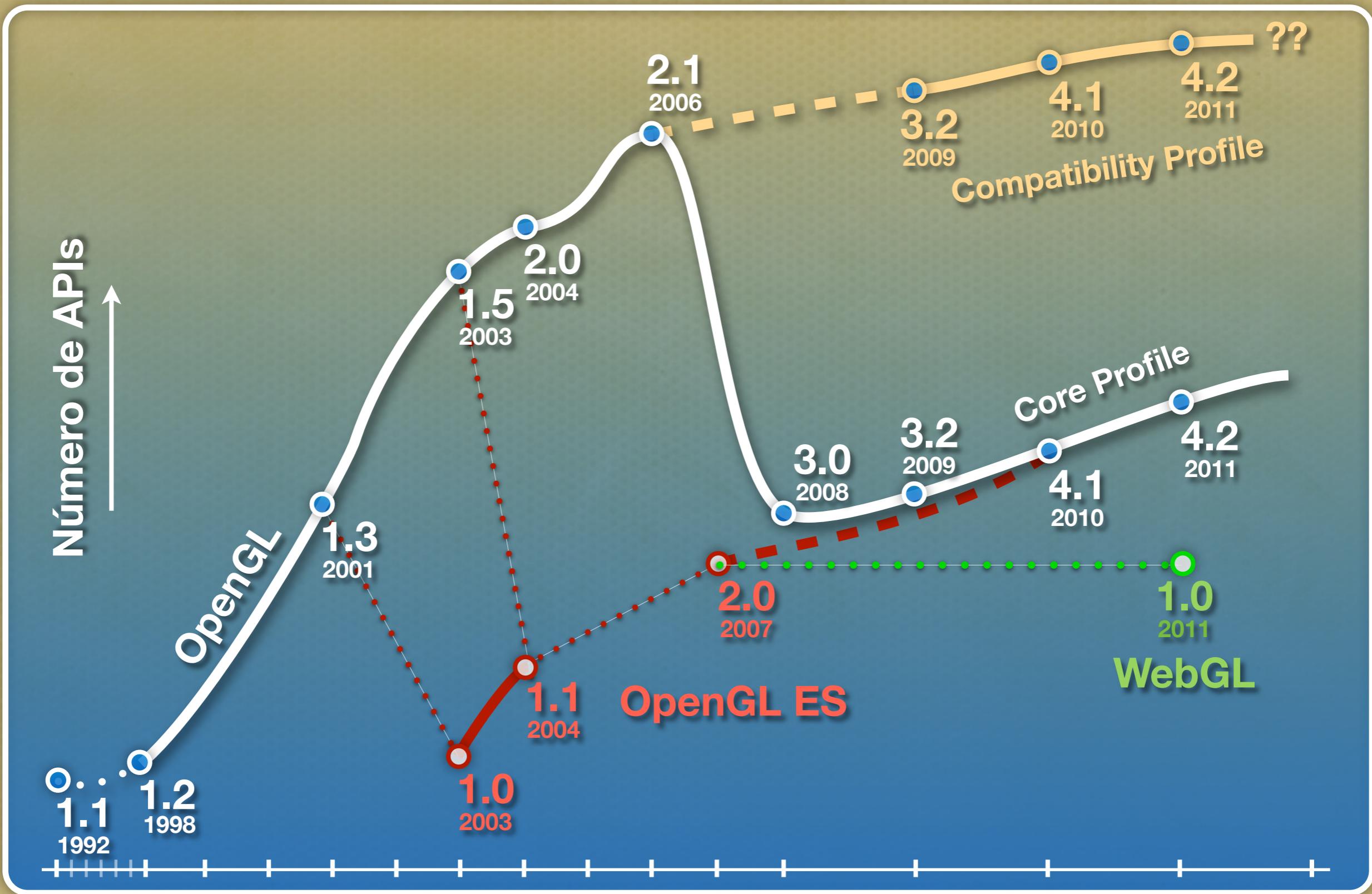
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



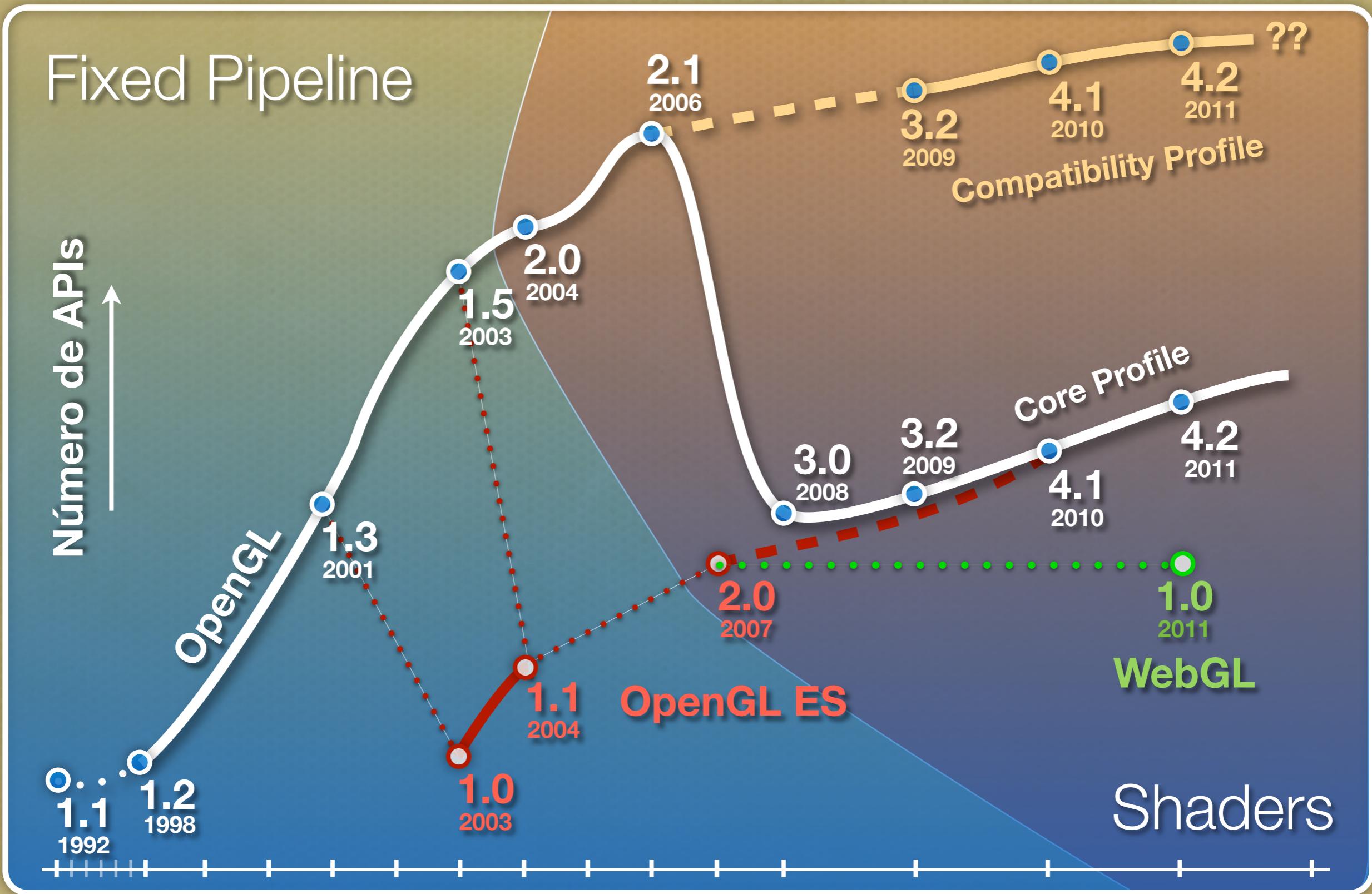
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



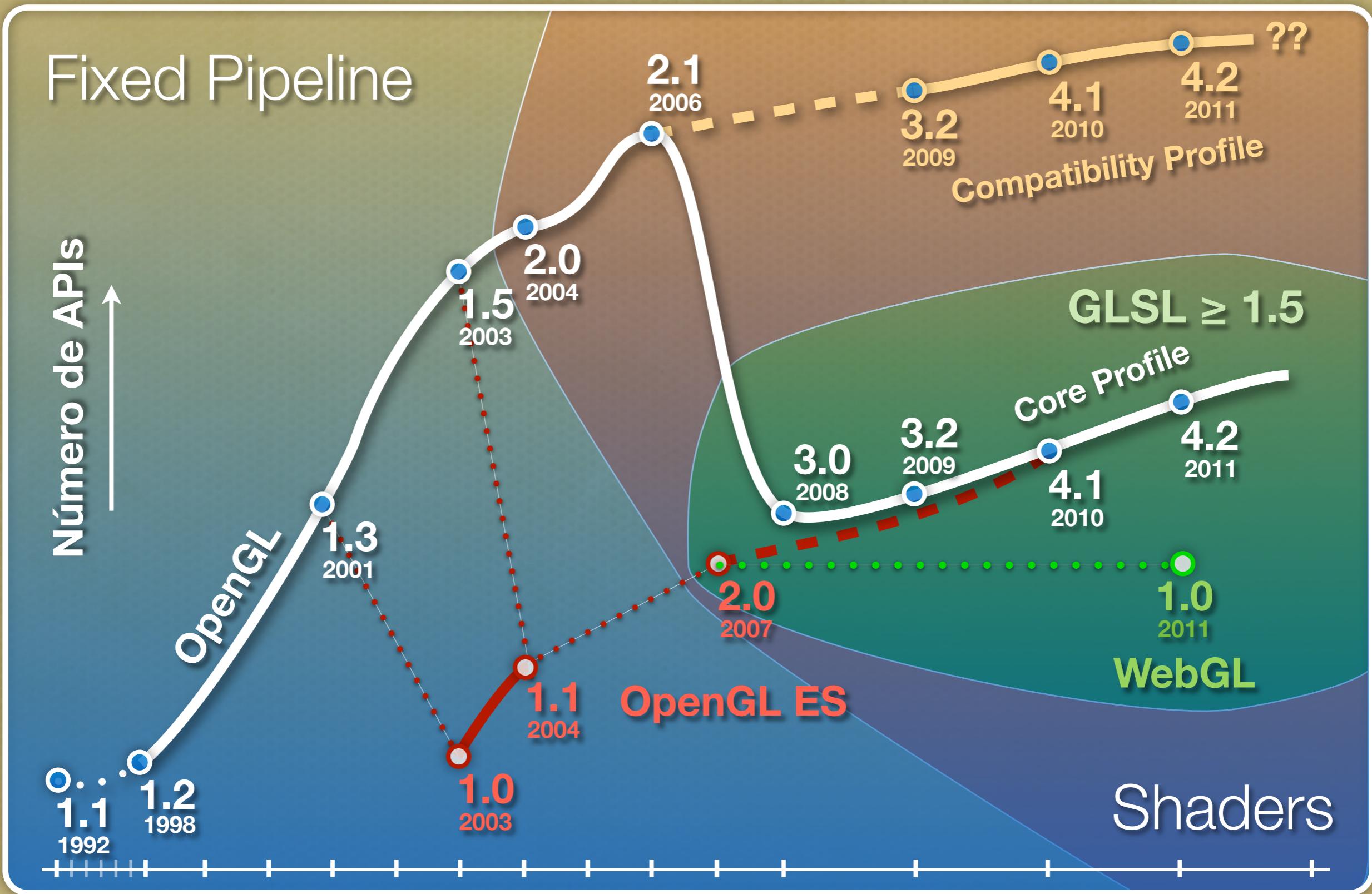
Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

OpenGL - História das versões



Parcialmente baseado em: http://www.opengl.org/wiki/History_of_OpenGL

Shaders em OpenGL

Compatibility Profile vs Core Profile

Compatibility Profile:

- Permanece com as **funções deprecadas** do OpenGL.
- Permite o uso de **pipeline fixo**.
- Permite o **modo imediato**.
- Acumula mais de **20 anos** de **história** em **APIs**.
- Implementação **Opcional**
- Tanto **Nvidia** quanto **AMD** não tem **planos** para **remove-lo**.

Core Profile:

- Elimina todas as **funções redundantes** e relacionadas ao **pipeline fixo**.
- Somente permite o uso de **pipeline programável**.
- Modo imediato e **operações matriciais removidas**.
- Mais **fácil** de **implementar**.
- **Compatibilidade** com **OpenGL ES 2.0** e **WebGL**.

Shaders em OpenGL

Requisitos

- Apesar de **OpenGL 2.0** já **suportar** shaders a linguagem mudou muito na **3.0**.
 - **OpenGL 3.0** exige **ao menos**:
 - **Geforce 8 series**
 - **Radeon HD**
 - **Intel HD 2000**
 - **OpenGL 4.0** e **Tesselation** shaders exigem:
 - **Geforce 400 series**
 - **Radeon HD 5000**
- **OpenGL ES 2.0** e **WebGL** suportam apenas Pixel Shaders e Vertex Shaders.
 - **iPad, iPhone, Android, etc.**
 - **WebGL**: Mozilla, Chrome, Safari, Opera.



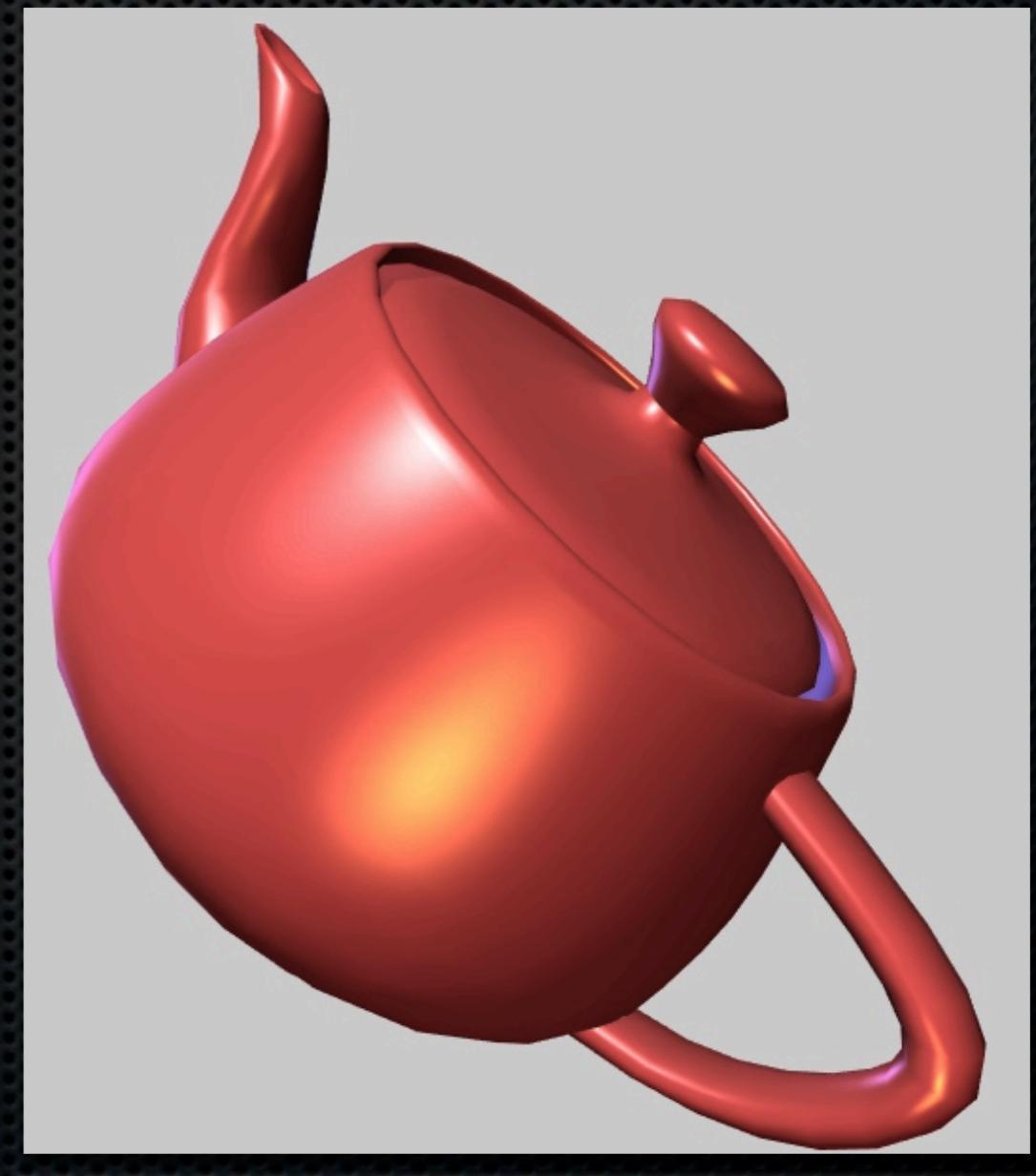
Fontes

Imagen: <http://www.legitreviews.com/article/593/1/>

Shaders em OpenGL

Requisitos (Software)

- **Drivers atualizados.**
- **Bibliotecas compatíveis** com **OpenGL 3.0**:
 - **GLFW**
<http://www.glfw.org/>
 - **freeGLUT**
<http://freeglut.sourceforge.net/>
 - É importante notar que **GLUT** **não é compatível** ainda.



Resumo

- **História**
 - Origem
 - Shaders em GPUs
 - Shaders em APIs
- **Pipeline**
 - Fixo
 - Programável
 - Vantagens do pipeline programável
- **Shaders em OpenGL**
 - Versões
 - Core Profile vs Compatibility Profile
 - Requisitos
- **Linguagem GLSL**
 - Definições
 - Exemplo Simples
 - Exemplo menos Simples
 - Iluminação básica
 - Phong
 - Texturas
 - Geometry Shader
- **Usando GLSL no OpenGL**
 - Compilando os programas
 - Enviando dados à GPU
- **Futuro**
- **Bibliografia**

Linguagem GLSL

Definições

- Os **shaders** são pequenos **programas** que **rodam** em **unidades de processamento** da **GPU**.
- **GLSL** é um **subset** de **C**:
 - com **tipos** e **operações matriciais** e **vetoriais**.
 - com **biblioteca matematica**.
 - sem **ponteiros**.
 - sem **recursão**.
 - sem **alocação dinamica** de **memória**.
 - com **algumas outras restrições**.

Linguagem GLSL

Linguagem GLSL

Vertex Program

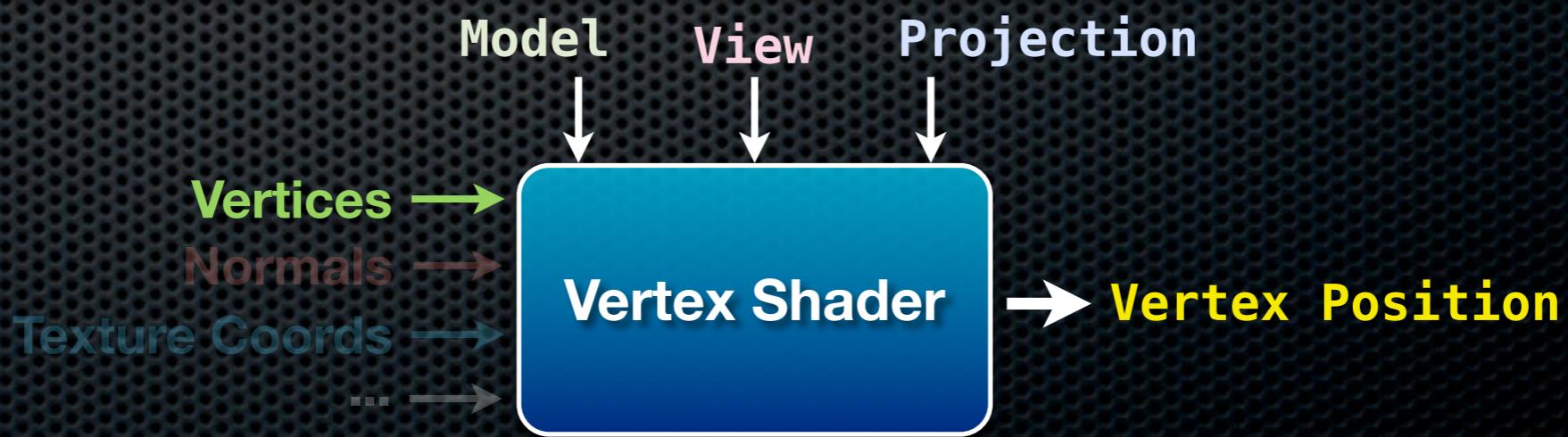
```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;

    gl_Position = u_projectionMatrix * vertex;
}
```



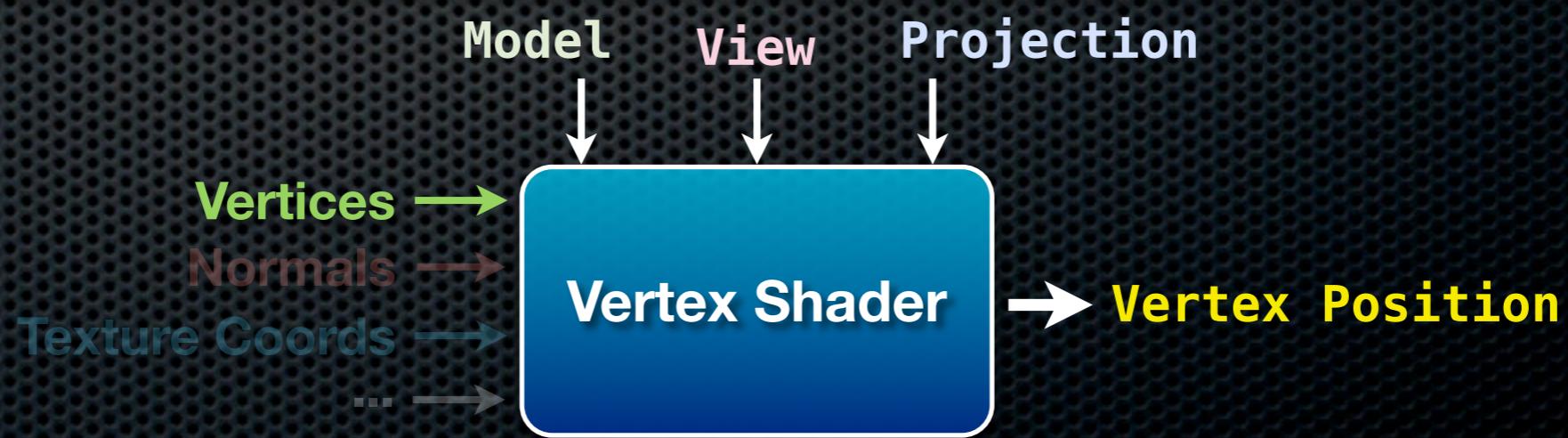
Linguagem GLSL

Vertex Program

```
#version 150
uniform mat4 u_projectionMatrix;           → Matriz de projeção na tela.
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;

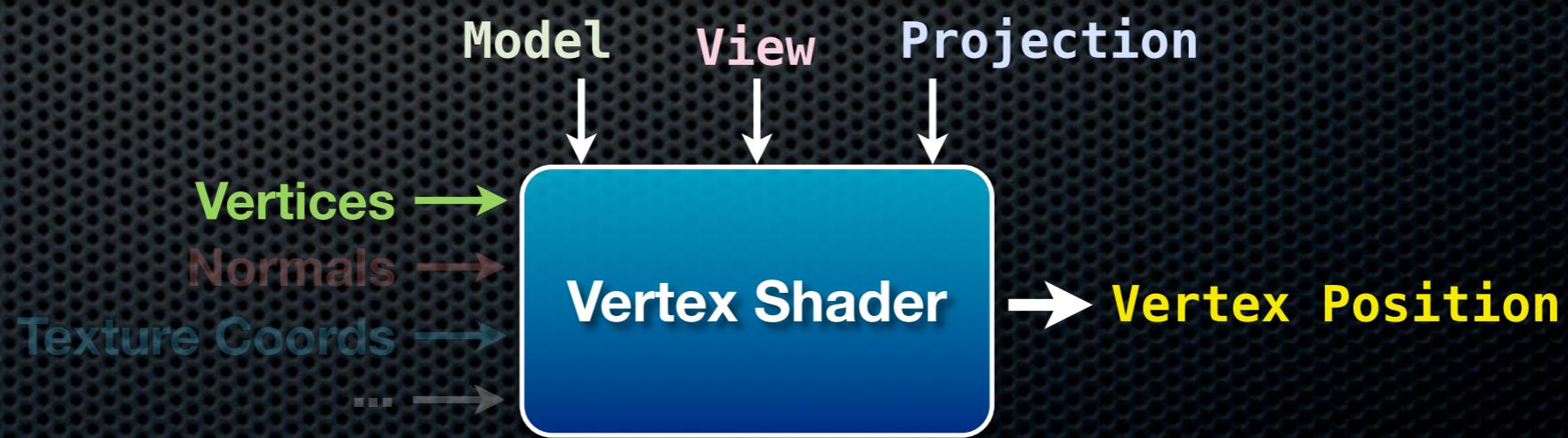
void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

Vertex Program

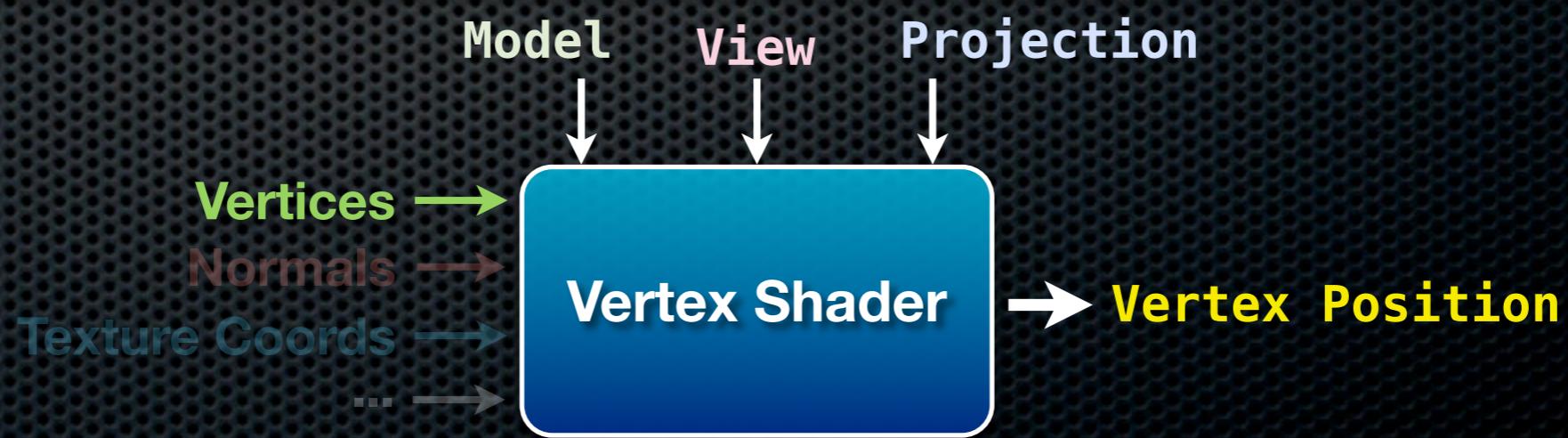
```
#version 150  
  
uniform mat4 u_projectionMatrix; ← Matriz de projeção na tela.  
uniform mat4 u_viewMatrix; ← Matriz de transformação para câmera.  
uniform mat4 u_modelMatrix;  
  
in vec4 a_vertex;  
  
void main(){  
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;  
  
    gl_Position = u_projectionMatrix * vertex;  
}
```



Linguagem GLSL

Vertex Program

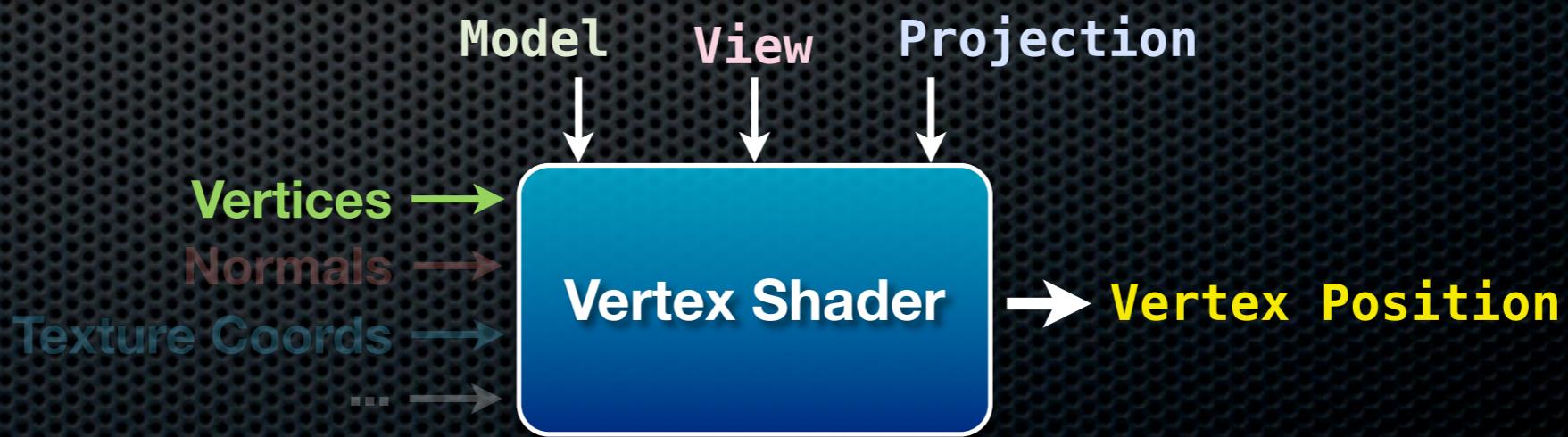
```
#version 150  
  
uniform mat4 u_projectionMatrix; ← Matriz de projeção na tela.  
uniform mat4 u_viewMatrix; ← Matriz de transformação para câmera.  
uniform mat4 u_modelMatrix; ← Matriz de transformação do objeto.  
  
in vec4 a_vertex;  
  
void main(){  
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;  
  
    gl_Position = u_projectionMatrix * vertex;  
}
```



Linguagem GLSL

Vertex Program

```
#version 150  
  
uniform mat4 u_projectionMatrix; ← Matriz de projeção na tela.  
uniform mat4 u_viewMatrix; ← Matriz de transformação para câmera.  
uniform mat4 u_modelMatrix; ← Matriz de transformação do objeto.  
  
in vec4 a_vertex; ← Vertices de entrada.  
  
void main(){  
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;  
  
    gl_Position = u_projectionMatrix * vertex;  
}
```

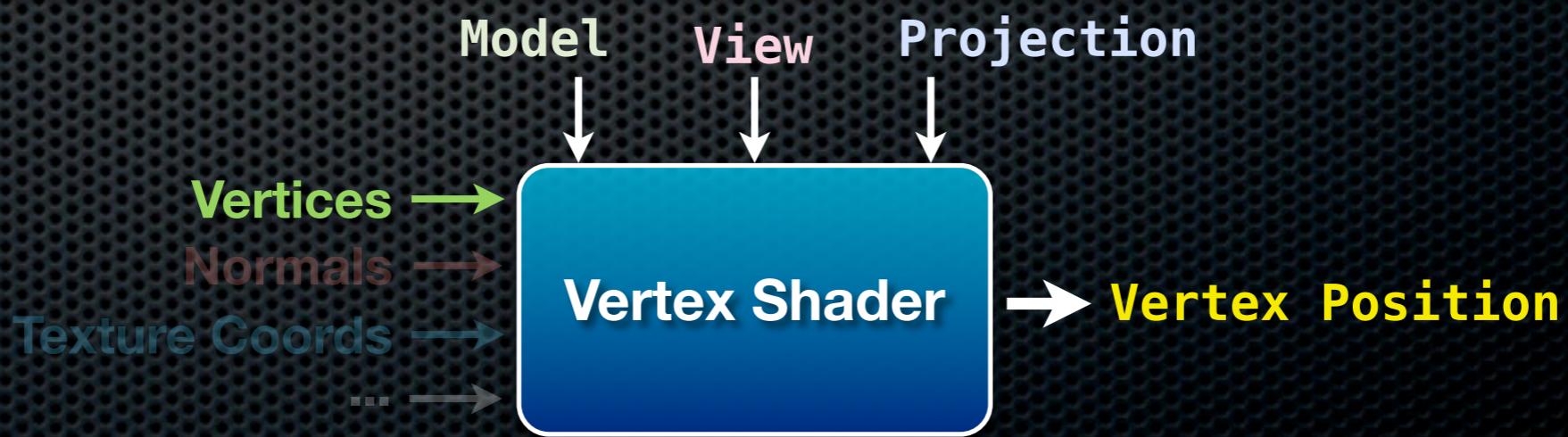


Linguagem GLSL

Vertex Program

```
#version 150  
  
uniform mat4 u_projectionMatrix; ← Matriz de projeção na tela.  
uniform mat4 u_viewMatrix; ← Matriz de transformação para câmera.  
uniform mat4 u_modelMatrix; ← Matriz de transformação do objeto.  
  
in vec4 a_vertex; ← Vertices de entrada.  
  
void main(){  
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;  
  
    gl_Position = u_projectionMatrix * vertex;  
}
```

Posição de saída.



Linguagem GLSL

Vertex Program

- O **vertex program** será executado para **todos os vertices** de **entrada**.
 - No exemplo o **vertice** de entrada possui um atributo **a_vertex** contendo a **posicao** do vertice.
- Todo vertex program **pode escrever** na **variável de saída gl_position**.
- Variáveis **in** (ou **attribute**) são **entradas** do shader.
- Variáveis **out** (ou **varying**) são **saídas** do shader.
- Variáveis **uniform** são **entradas constantes** ao longo da **geometria**.

Linguagem GLSL

Linguagem GLSL

Geometry Program

Linguagem GLSL

Geometry Program

Vamos deixar para mais tarde...

Linguagem GLSL

Linguagem GLSL

Fragment Program

```
#version 150

out vec4 fragColor;

void main(){
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Fragment
Shader

→ Cor do fragmento

Linguagem GLSL

Fragment Program

```
#version 150

out vec4 fragColor;

void main(){
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Vermelho (RGBA)



Linguagem GLSL

Fragment Program

```
#version 150

out vec4 fragColor;

void main(){
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Vermelho (RGBA)

Cor do fragmento

Fragment
Shader

→ Cor do fragmento

Linguagem GLSL

Fragment Program

- O **fragment program** será executado para **todos os fragmentos (pixels)** de cada primitiva.
- A linguagem é a mesma da usada no **vertex program**.
- Pode-se enviar dados do **vertex program** para o fragmento por meio de **out(varying)**.
- O **fragment program** tem como **saída** a **cor** do **fragmento**.

Linguagem GLSL

Fragment Program

- O **fragment** é o bloco de código que define os efeitos visuais de cada pixel.
- A linguagem GLSL é usada para escrever os **fragment programs**.
- Pode-se usar a linguagem GLSL para definir efeitos como o desenho de um teapot.
- O **fragment** é executado para cada pixel do teapot.



Linguagem GLSL

Linguagem GLSL

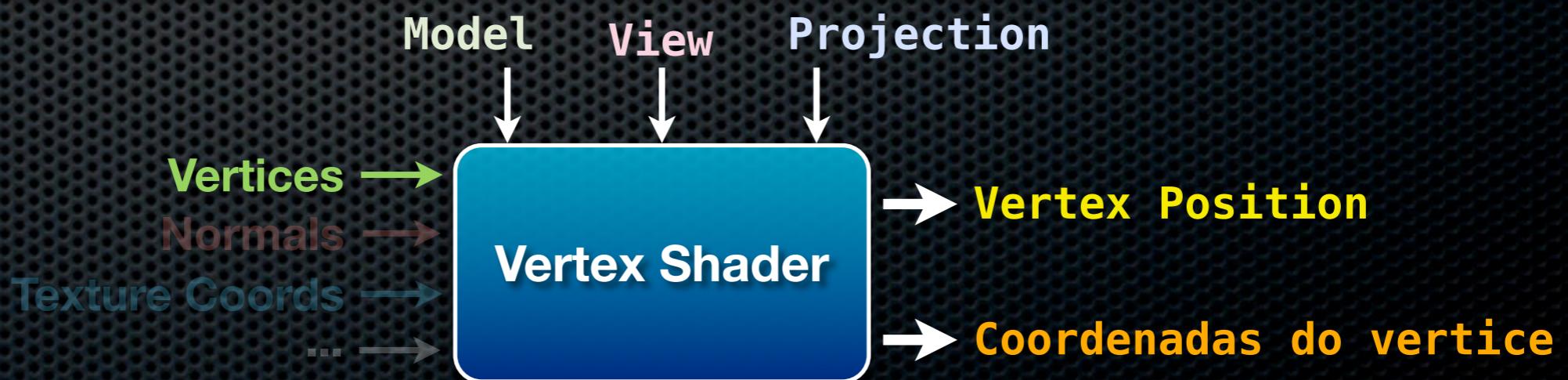
Vertex Program (*um pouco mais complicado*)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;
out vec4 v_vertex;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_vertex = a_vertex;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

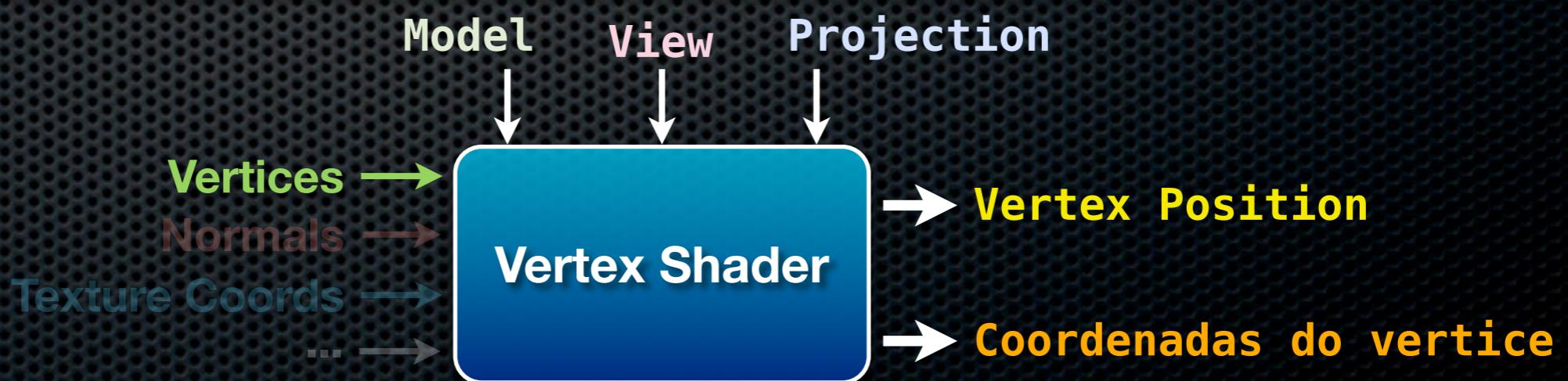
Vertex Program (*um pouco mais complicado*)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;
out vec4 v_vertex;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_vertex = a_vertex;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

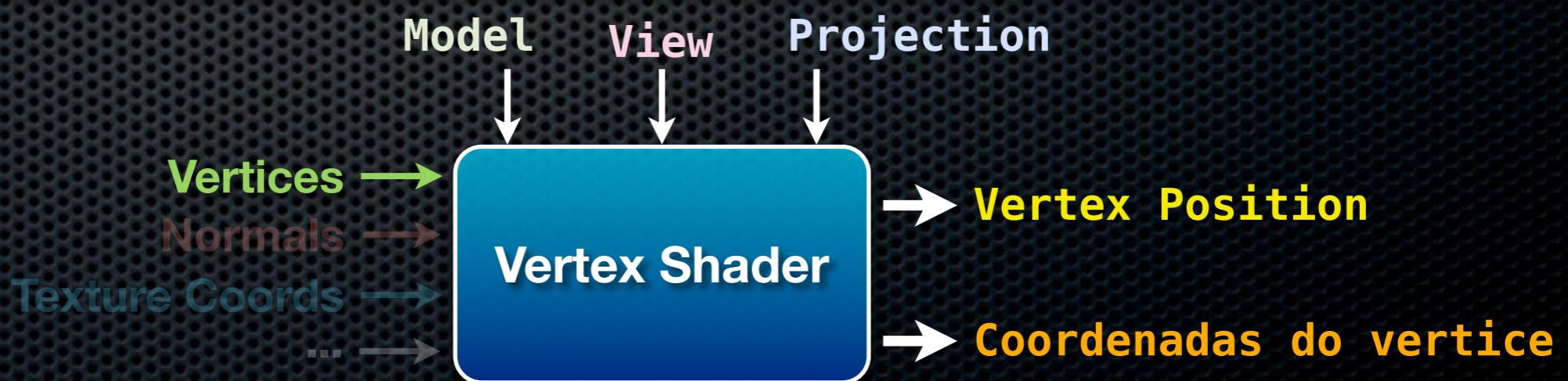
Vertex Program (*um pouco mais complicado*)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;
out vec4 v_vertex;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_vertex = a_vertex;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

Linguagem GLSL

Fragment Program (*um pouco mais complicado*)

```
#version 150

in vec4 v_vertex;

out vec4 fragColor;

void main(){
    fragColor = normalize(v_vertex);
}
```



Linguagem GLSL

Fragment Program (*um pouco mais complicado*)

```
#version 150

in vec4 v_vertex;

out vec4 fragColor;

void main(){
    fragColor = normalize(v_vertex);
}
```



Linguagem GLSL

Fragment Program (*um pouco mais complicado*)

```
#version 150

in vec4 v_vertex;

out vec4 fragColor;

void main(){
    fragColor = normalize(v_vertex);
}
```



Linguagem GLSL

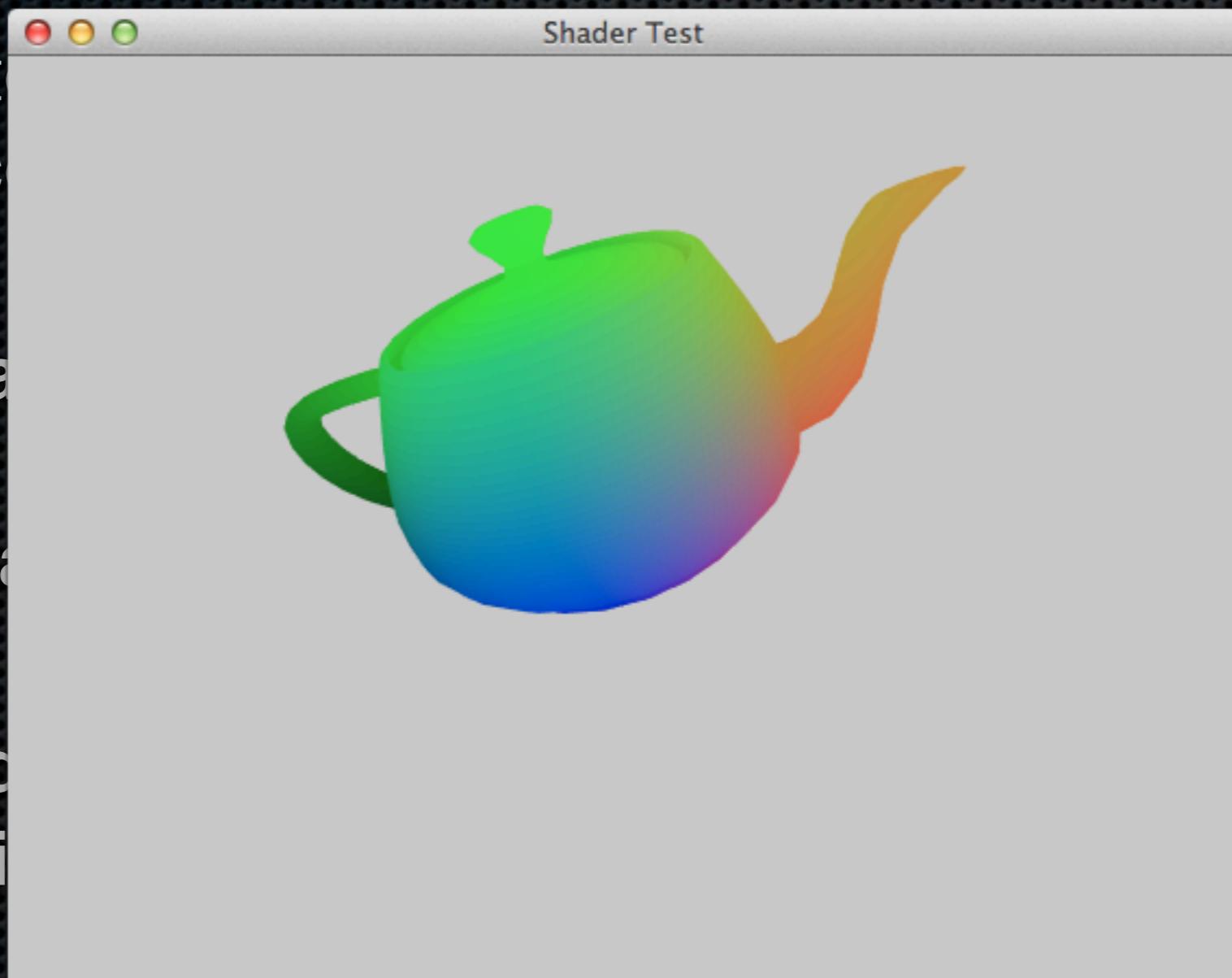
Fragment Program (*um pouco mais complicado*)

- O atributo **a_vertex** é **interpolado** para o fragment shader como **v_vertex**;
- Por **padrão** os valores são **interpolarados**, mas ao usar a palavra **flat**, os valores são tomados como **constantes** para cada **primitiva**.
- Para **cada saída** do **vertex program**, **deve haver** uma entrada **igual** no **fragment program**.

Linguagem GLSL

Fragment Program (*um pouco mais complicado*)

- O atributo `position` é usado no shader como constante.
- Por padrão, o fragment shader usa a constante `gl_FragColor` para armazenar o resultado.
- Para cada variável de entrada, é preciso declarar uma saída.



mento
nas ao
como
aver uma

Linguagem GLSL

Linguagem GLSL

Vertex Program (Iluminação)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;
uniform mat3 u_normalMatrix;

in vec4 a_vertex;
in vec3 a_normal;

out vec3 v_normal;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_normal = u_normalMatrix * a_normal;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

Linguagem GLSL

Fragment Program (Iluminação)

```
#version 150
uniform vec4 ambientLight;
uniform vec4 difuseLight;
uniform vec4 lightDirection;
uniform vec4 materialColor;

in vec3 v_normal;

out vec4 fragColor;

void main(){
    vec3 normal = normalize(v_normal);
    vec4 color = ambientLight * materialColor;
    vec3 cosTheta = max(dot(lightDirection,normal),0.0);
    color += difuseLight * materialColor * cosTheta;
    fragColor = color;
}
```

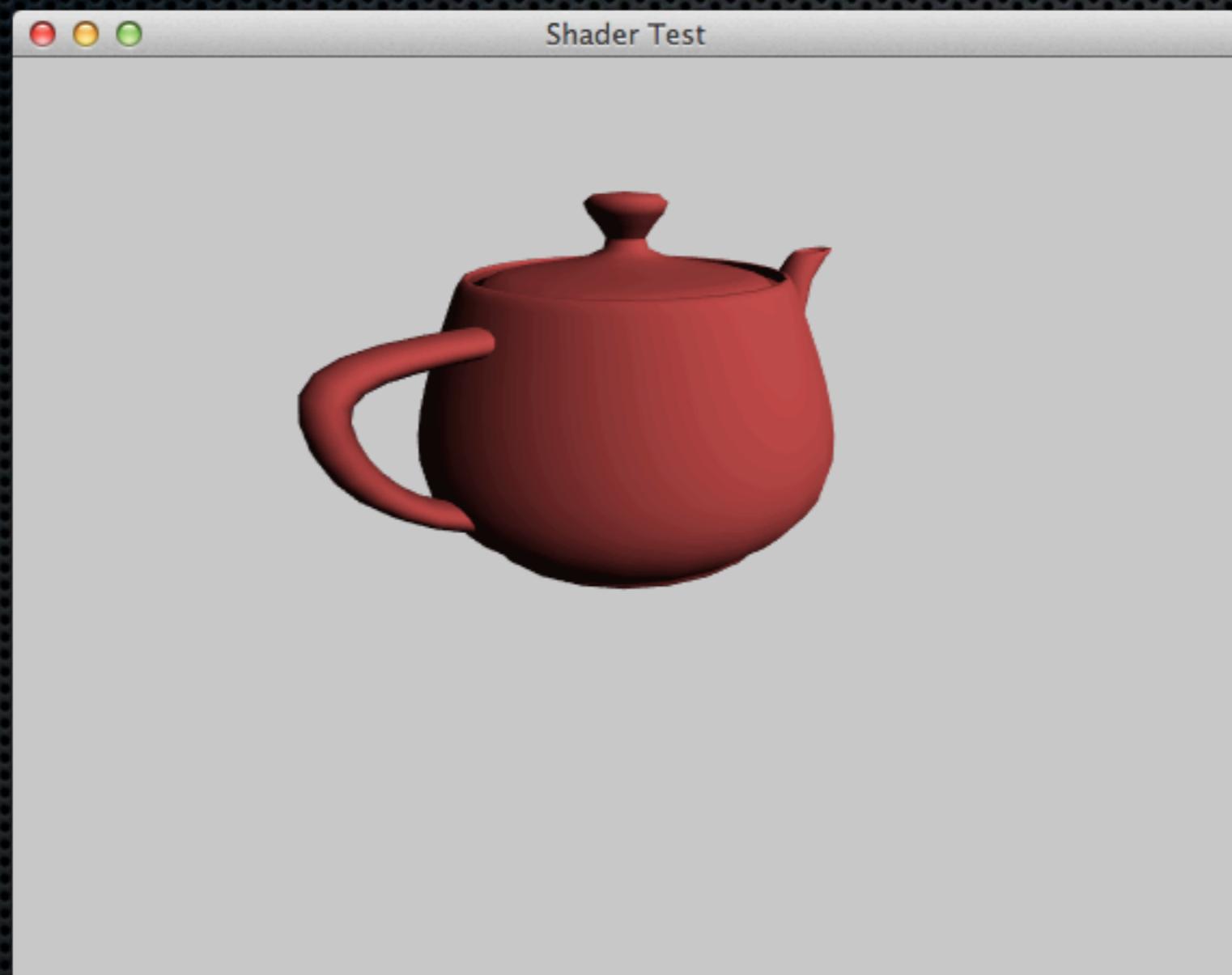


Linguagem GLSL

Fragment Program (*Iluminação*)

Linguagem GLSL

Fragment Program (*Iluminação*)



Linguagem GLSL

Linguagem GLSL

Vertex Program (Phong)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;
uniform mat3 u_normalMatrix;

in vec4 a_vertex;
in vec3 a_normal;

out vec3 v_normal;
out vec3 v_eye;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_normal = u_normalMatrix * a_normal;
    v_eye = -vec3(vertex);
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

Linguagem GLSL

Fragment Program (*Phong*)

```
#version 150
uniform vec4 ambientLight;
uniform vec4 difuseLight;
uniform vec4 specularLight;
uniform float specularExpoent;
uniform vec4 lightDirection;
uniform vec4 materialColor;

in vec3 v_normal;
in vec3 v_eye;

out vec4 fragColor;

void main(){
    vec3 normal = normalize(v_normal);
    vec3 eye = normalize(v_eye);
    vec4 color = ambientLight * materialColor;
    vec3 cosNormal = max(dot(lightDirection,normal),0.0);
    color += difuseLight * materialColor * cosNormal;
    vec3 reflection = reflect(-lightDirection, normal);
    float cosEye = max(dot(eye, reflection), 0.0);
    color += specularLight * materialColor * pow(cosEye, specularExpoent);
    fragColor = color;
}
```



Linguagem GLSL

Fragment Program (*Phong*)

Linguagem GLSL

Fragment Program (*Phong*)



Linguagem GLSL

Linguagem GLSL

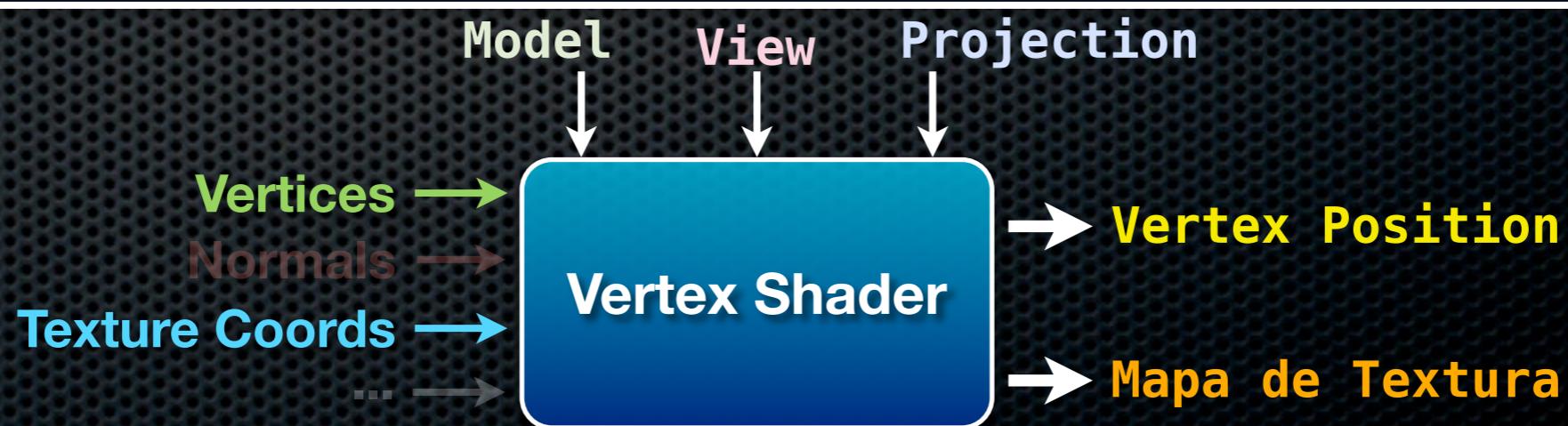
Vertex Program (*texturas*)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;

in vec4 a_vertex;
in vec2 a_texCoords;
out vec2 v_texCoords;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    v_texCoords = a_texCoords;
    gl_Position = u_projectionMatrix * vertex;
}
```



Linguagem GLSL

Linguagem GLSL

Fragment Program (*texturas*)

```
#version 150
uniform sampler2D u_texture;
in vec4 v_texCoords;

out vec4 fragColor;

void main(){
    fragColor = texture(u_texture, v_texCoords);
}
```

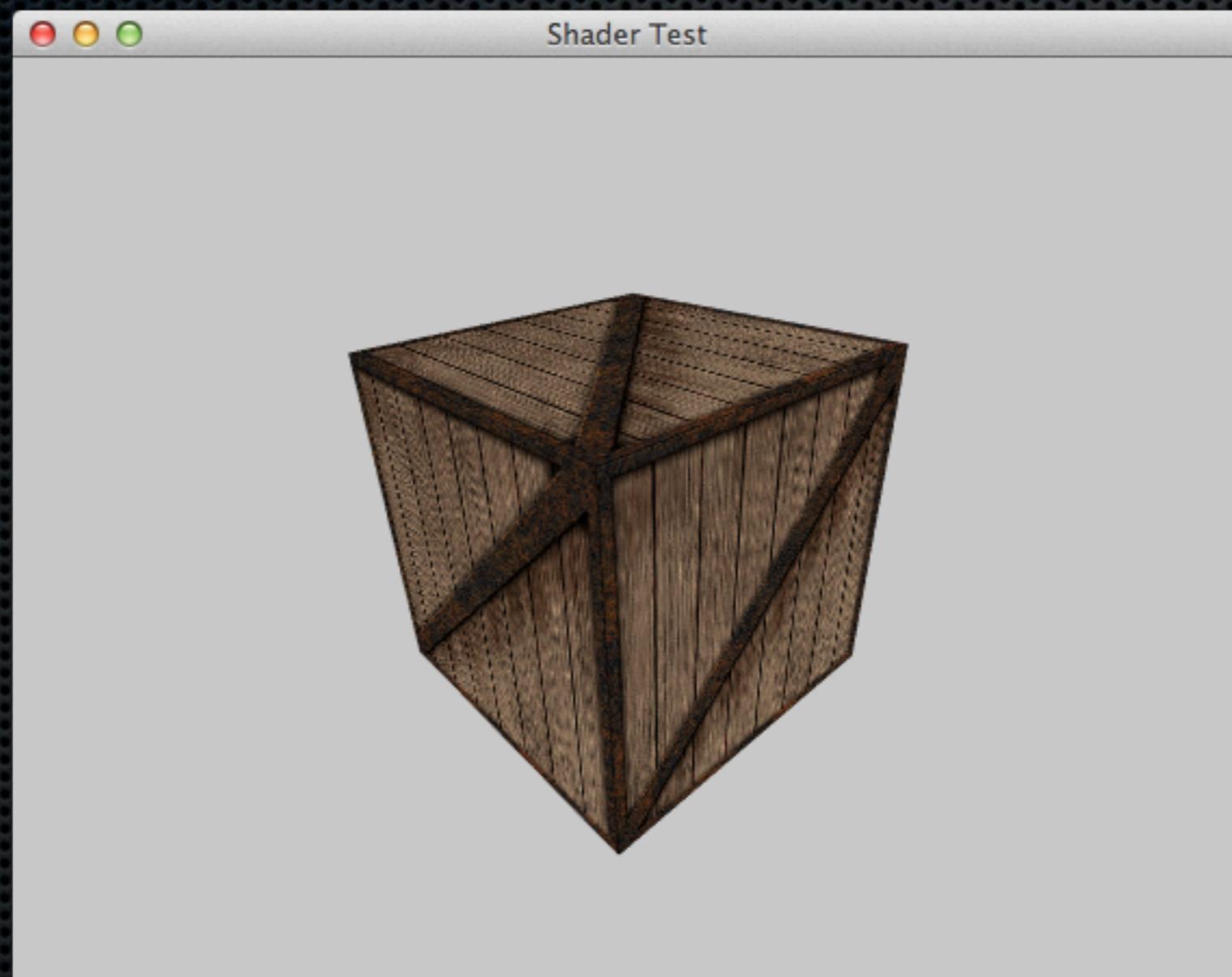


Linguagem GLSL

Fragment Program (*Textura*)

Linguagem GLSL

Fragment Program (*Textura*)



Linguagem GLSL

Geometry Program

- O **geometry program** é executado logo após o **vertex program**.
- Tem a finalidade de **emitir mais ou menos vértices** do que é fornecida.
- Também pode **converter** o **tipo** de **geometria**.
- Pode-se realizar todas as **operações** do **vertex program** diretamente no **geometry program**.

Linguagem GLSL

Linguagem GLSL

Vertex Program (wireframe)

```
#version 150

uniform mat4 u_projectionMatrix;
uniform mat4 u_viewMatrix;
uniform mat4 u_modelMatrix;
uniform mat3 u_normalMatrix;

in vec4 a_vertex;
in vec3 a_normal;

out vec3 g_normal;

void main(){
    vec4 vertex = u_viewMatrix * u_modelMatrix * a_vertex;
    g_normal = u_normalMatrix * a_normal;
    gl_Position = vertex;
}
```



Linguagem GLSL

Linguagem GLSL

Geometry Shader (wireframe)

```
#version 150

layout (triangles) in;
layout (line_strip, max_vertices=4) out;

in vec3 g_normal[3];
out vec3 v_normal;

void main(){
    int i;
    for (i=0; i < gl_in.length(); i++){
        gl_Position = gl_in[i].gl_Position;
        v_normal = g_normal[i];
        EmitVertex();
    }
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
    EndPrimitive();
}
```

Triângulo
-Posição[3]
-Normal[3]

Geometry
Shader

Linhas
-Posição[4]
-Normal[4]

Linguagem GLSL

Linguagem GLSL

Fragment Program (Iluminação)

```
#version 150
uniform vec4 ambientLight;
uniform vec4 difuseLight;
uniform vec4 lightDirection;
uniform vec4 materialColor;

in vec3 v_normal;

out vec4 fragColor;

void main(){
    vec3 normal = normalize(v_normal);
    vec4 color = ambientLight * materialColor;
    vec3 cosTheta = max(dot(lightDirection,normal),0.0);
    color += difuseLight * materialColor * cosTheta;
    fragColor = color;
}
```

Normal do Vértice →

Fragment
Shader

→ Cor do fragmento

Linguagem GLSL

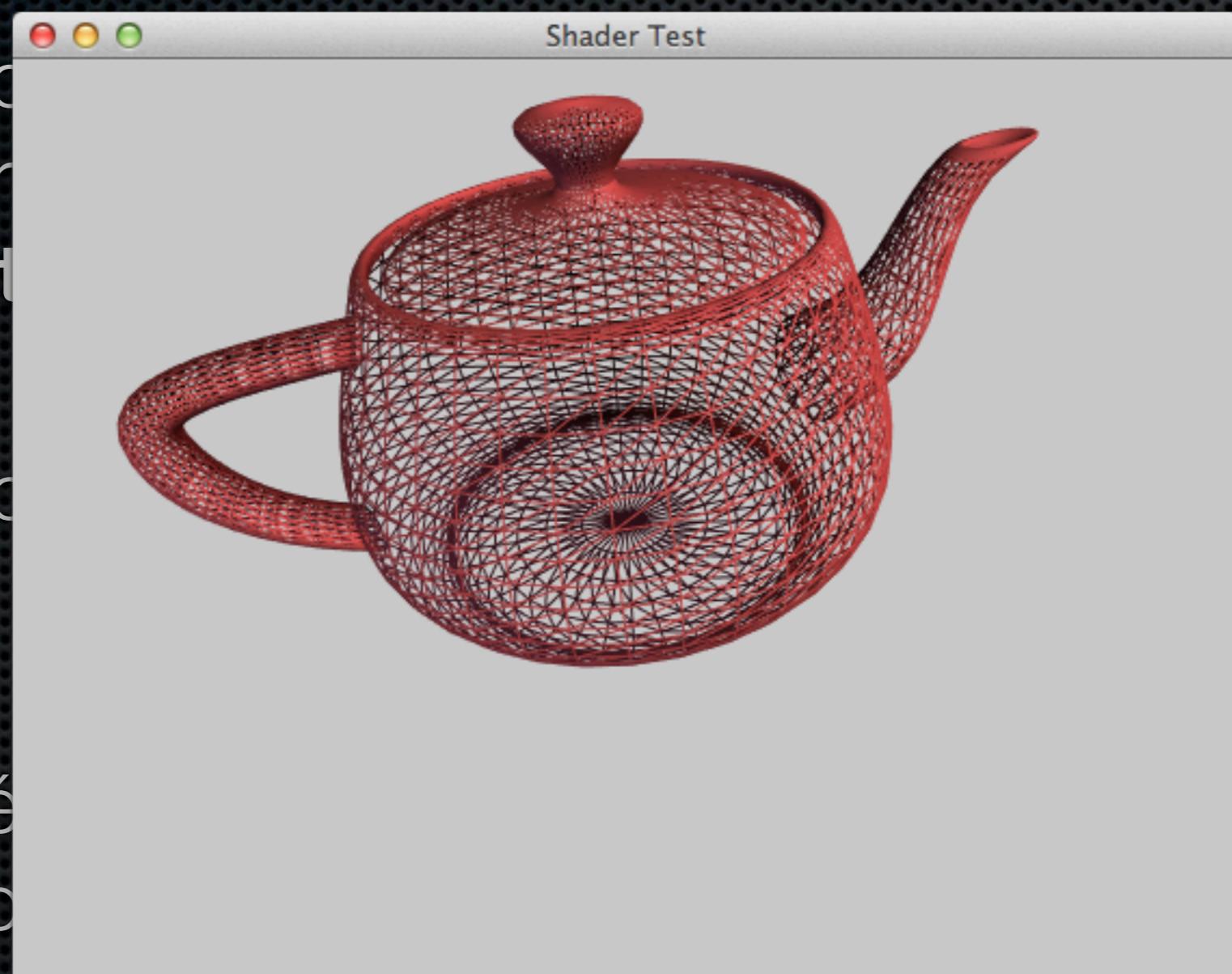
Geometry Program (wireframe)

- A função **emitVertex()** emite um novo vértice na geometria com os valores atuais dos **out** e **gl_position**.
- A função **endPrimitive()** finaliza uma **primitiva**. Permite **emitter** varias **primitivas diferentes**.
- **gl_in[]** é uma array contendo os valores de posicao passados pelo vertex program.

Linguagem GLSL

Geometry Program (wireframe)

- A função `gl_Position` é executada na geometria.
- A função `gl_in[]` é executada na geometria.
- `gl_in[]` é uma estrutura que passado



Resumo

- **História**
 - Origem
 - Shaders em GPUs
 - Shaders em APIs
- **Pipeline**
 - Fixo
 - Programável
 - Vantagens do pipeline programável
- **Shaders em OpenGL**
 - Versões
 - Core Profile vs Compatibility Profile
 - Requisitos
- **Linguagem GLSL**
 - Definições
 - Exemplo Simples
 - Exemplo menos Simples
 - Iluminação básica
 - Phong
 - Texturas
 - Geometry Shader
- **Usando GLSL no OpenGL**
 - Compilando os programas
 - Enviando dados à GPU
- **Futuro**
- **Bibliografia**

Usando GLSL no OpenGL

Compilando programa

- Cada shader é compilado em tempo de execução, fornecendo o código fonte em string simples.
- Os shaders são “linkados” e são ativados.
- A API fornece uma referência para futuramente atribuir as entradas e uniforms.

Usando GLSL no OpenGL

Compilando programa

Usando GLSL no OpenGL

Compilando programa

```
char* vertexSource = "...";
GLuint compiled, linked;
GLuint vertexProgram = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexProgram, 1, (const char**) vertexSource, 0);
glCompileShader(vertexProgram);
glGetShaderiv(vertexProgram, GL_COMPILE_STATUS, &compiled);
if (!compiled){...}

/* Mesma coisa para fragment e geometry */

GLuint program = glCreateProgram();
glAttachShader(program, vertexProgram);
glAttachShader(program, geometryProgram);
glAttachShader(program, fragmentProgram);

glLinkProgram(program);
glGetProgramiv(program, GL_LINK_STATUS, &linked);

if (!linked){...}

glUseProgram(program);
```

Usando GLSL no OpenGL

Enviando dados

- **Uniforms** são passados pelo comando **glUniform()**

```
GLfloat projectionMatrix[16] = {...};  
g_projectionMatrixLocation = glGetUniformLocation(program, "u_projectionMatrix");  
glUniformMatrix4fv(g_projectionMatrixLocation, 1, GL_FALSE, projectionMatrix);
```

- **Atributos** são passados por **VBO** e **VAO**.
 - **Vertex Buffer Objects** (VBO) são buffers de bytes contendo dados brutos a serem enviados para a GPU.
 - **Vertex Array Objects** (VAO) são as definições dos tipos de dados e posições(offsets) que estão no VBO.

Usando GLSL no OpenGL

Enviando dados

- **Uniforms** são passados pelo comando **glUniform()**

```
GLfloat projectionMatrix[16] = {...};  
g_projectionMatrixLocation = glGetUniformLocation(program, "u_projectionMatrix");  
glUniformMatrix4fv(g_projectionMatrixLocation, 1, GL_FALSE, projectionMatrix);
```

- **Atributos** são passados por **VBO** e **VAO**.
 - **Vertex Buffer Objects** (VBO) são buffers de bytes contendo dados brutos a serem enviados para a GPU.
 - **Vertex Array Objects** (VAO) são as definições dos tipos de dados e posições(offsets) que estão no VBO.

Usando GLSL no OpenGL

Enviando dados

Usando GLSL no OpenGL

Enviando dados

```
GLfloat* vertexValues = calloc(sizeof(GLfloat), MAX_VERTICES*4);
GLuint vertexVB0, vertexVA0, a_vertexLocation;

... Preenche vertexValues com geometria

a_vertexLocation = glGetAttribLocation(program, "a_vertex");

//Carrega no VBO
glGenBuffers(1, &vertexVB0);
glBindBuffer(GL_ARRAY_BUFFER, vertexVB0);
glBufferData(GL_ARRAY_BUFFER, MAX_VERTICES * 4 * sizeof(GLfloat), vertexValues, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

free(vertexValues);

//Define um VAO
glGenVertexArrays(1, & vertexVA0);
glBindVertexArray(vertexVA0);

//Identifica que a_vertex esta no vertexVB0 e é do tipo vec4 (float[4])
glBindBuffer(GL_ARRAY_BUFFER, vertexVB0);
glVertexAttribPointer(a_vertexLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(a_vertexLocation);

//Outros Buffers (Normais por exemplo)
glBindBuffer(GL_ARRAY_BUFFER, normalVB0);
glVertexAttribPointer(a_normalLocation, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glEnableVertexAttribArray(a_normalLocation);

glDrawArrays(GL_TRIANGLES, 0, MAX_VERTICES);
```

Futuro

OpenGL 4.2 -> OpenGL 5.0

- Qual será o futuro do compatibility profile?
- O que falta deixar programável no pipeline?
 - Blending.
 - Depth Test.
- OpenGL ES 3.0 chegando.
- WebGL ficará popular?

Bibliografia

e links diversos.

- Exemplos em OpenGL 3.2 e 4.0 incluindo shaders
<http://nopper.tv/opengl.html>
- Especificação OpenGL 3.2 Core Profile
<http://www.opengl.org/registry/doc/glspec32.core.20090803.pdf>
- OpenGL Wiki
<http://www.opengl.org/wiki/>
- Curso de Shaders da Oregon University
<http://web.engr.oregonstate.edu/~mjb/cs519/>
- Tutoriais Lighthouse 3D
<http://www.lighthouse3d.com/tutorials/>
- David Wolff, OpenGL 4.0 Shading Language Cookbook
http://www.packtpub.com/opengl-4-0-shading-language-cookbook/book?utm_source=rk&utm_campaign=opengl4-abr1&utm_medium=0811

Bibliografia

e links diversos.

- Migrating to OpenGL 3.2 Core Profile
<http://athile.net/library/blog/?p=582>
- Pacote de exemplos de shaders diversos
<http://www.g-truc.net/project-0026.html>
- WebGL sandboxes
[http://mrdoob.com/139/GLSL Sandbox](http://mrdoob.com/139/GLSL_Sandbox)
- Shader Editor escrito em WebGL
http://www.kickjs.org/example/shader_editor/shader_editor.html

**Todos os Links desta apresentação
foram conferidos dia 28/05/2012.**

Exercícios

para divertir.

- Faça um fragment shader que receba:
 - \mathbf{N}_e (*v_normal*) - vetor normal no espaço da câmera.
 - \mathbf{V}_e (*v_eye*) - vetor com a direção da câmera.
 - ϵ (err) - Uma constante do tipo float.
 - Se $(\mathbf{V}_e \cdot \mathbf{N}_e) < \epsilon$ - pinta o fragmento de preto.
 - Senão pinta o fragmento de branco.

Exercícios

Exercícios

Fragment Shader (*Silhueta*)

```
#version 150
uniform float err;

in vec3 v_normal;
in vec3 v_eye;

out vec4 fragColor;

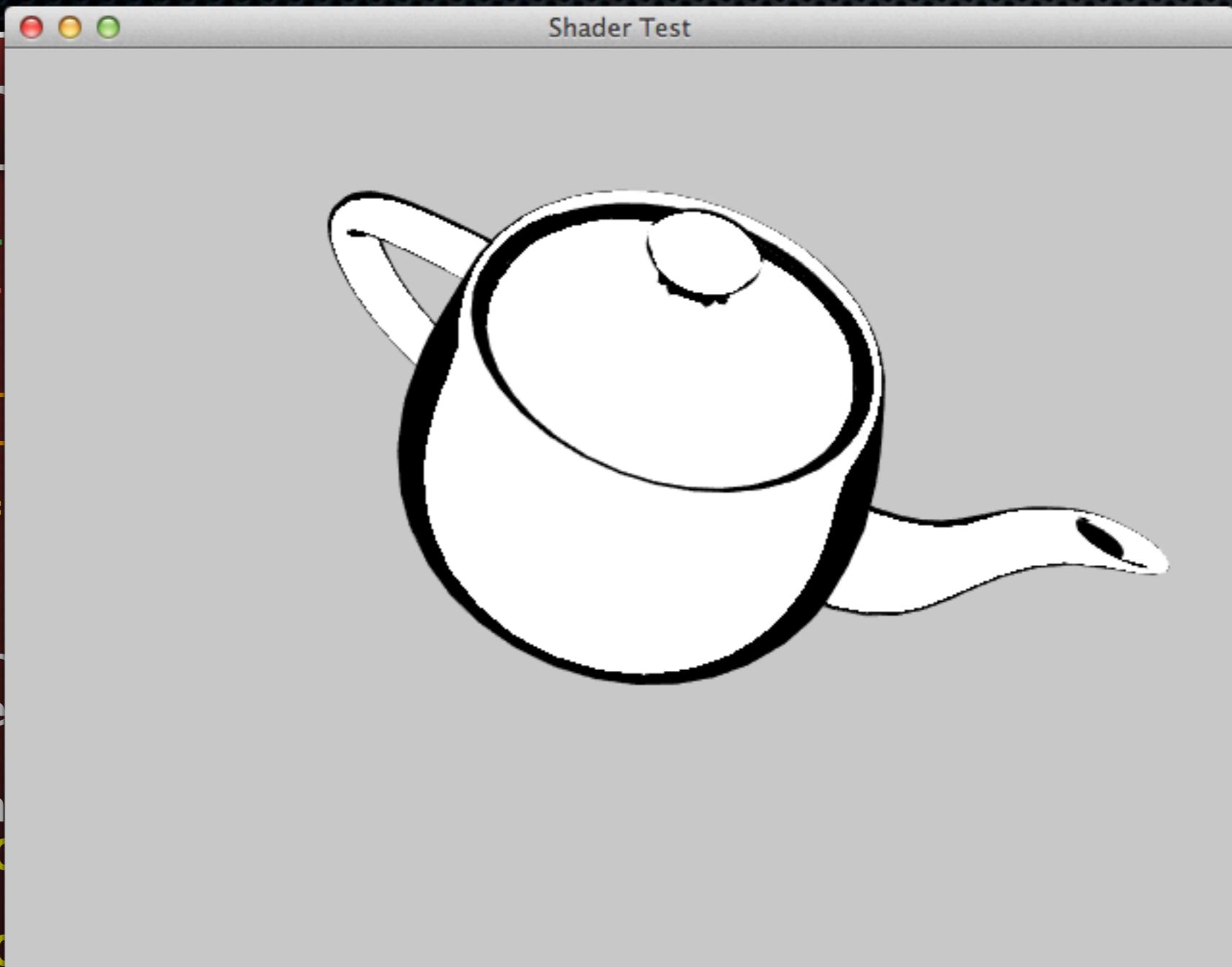
void main(){
    vec3 normal = normalize(v_normal);
    vec3 eye = normalize(v_eye);

    if(dot(normal,eye) < err){
        fragColor = vec4(0.0,0.0,0.0,1.0);
    }else{
        fragColor = vec4(1.0,1.0,1.0,1.0);
    }
}
```

Exercícios

Fragm

```
#version 1  
uniform fl  
  
in vec3 v_  
in vec3 v_  
  
out vec4 f  
  
void main(  
    vec3 nor  
    vec3 eye  
  
    if(dot(n  
        fragC  
    }else{  
        fragC  
    }  
}
```



Exercícios

para divertir.

- Faça um **vertex shader** que sempre apresente o **modelo 3D voltado para a camera (billboard)**.
- Por que devemos **renormalizar** a **normal** no **fragment shader** para ter a luz (per pixel) corretamente calculada?
- Em qual tipo de **shader** a técnica de **Displacement mapping** deve ser implementada? E quanto à técnica de **bumping mapping**? Discuta sobre a performance vs qualidade dessas técnicas no contexto de **shaders**.