

Introdução aos *shaders* (OpenGL moderna)

por Rossana B Queiroz

Sumário

1. Pipeline Programável
2. Shaders

OpenGL moderno

- O que é OpenGL “moderno”?
 - Core-profile mode => força uso das práticas modernas
- A partir da versão OpenGL 3.3: modo imediato está descontinuado
 - Versões subsequentes utilizam a mesma abordagem do 3.3
 - Adicionam-se features ou maneiras mais inteligentes de realizar certas tarefas
 - A versão atual do OpenGL é a 4.6
 - Futuro = nova API, chamada Vulkan
- Usar a versão mais atual nem sempre é a melhor opção: apenas as GPUs mais modernas têm suporte

Pipeline programável do OpenGL

- Estágio de **Vertex** shading
 - Processa cada vértice separadamente
- Estágio de **Tessellation** shading
 - Gera geometria dentro do pipeline (com código)
- Estágio de **Geometry** shading
 - Processa cada primitiva separadamente
- Estágio de **Fragment** shading
 - Processa cada fragmento separadamente

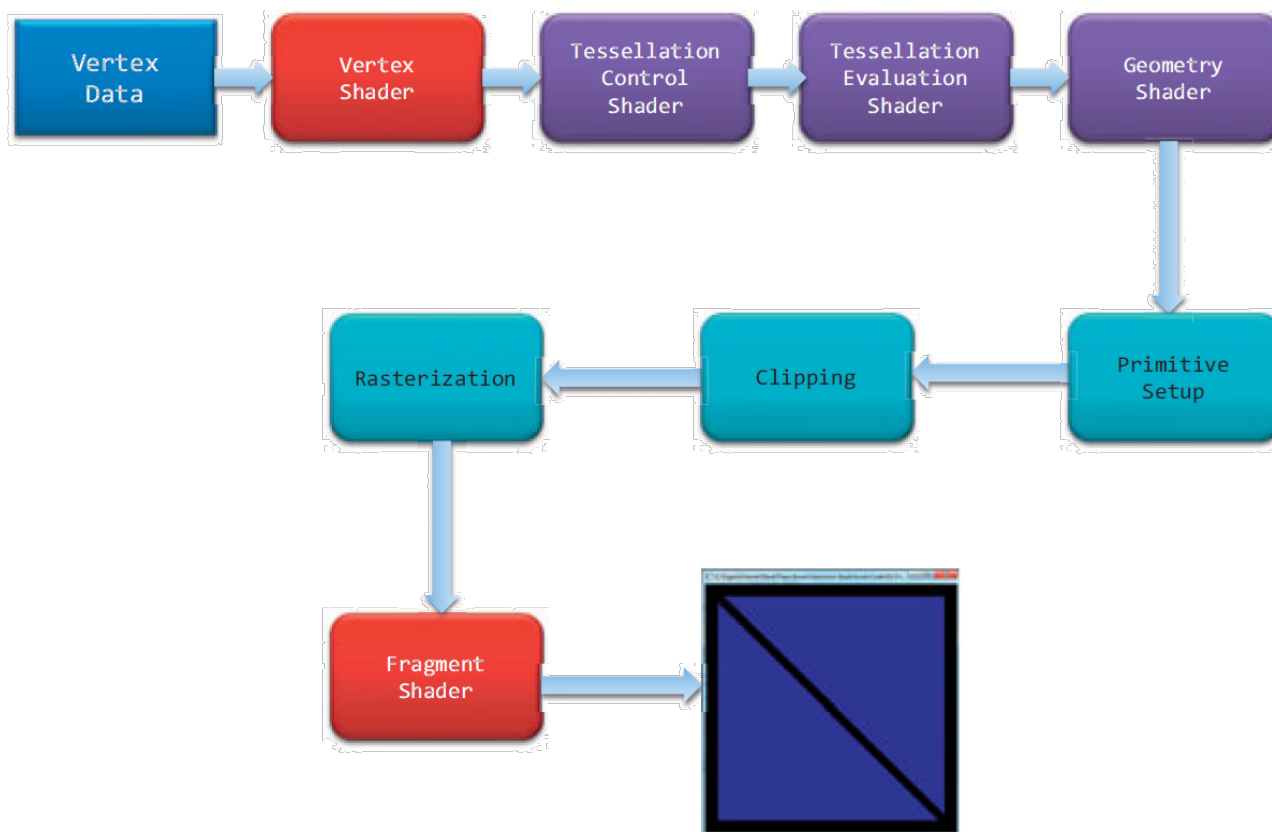
Pipeline programável do OpenGL

- Estágio de **Vertex** shading
 - Processa cada vértice separadamente
- Estágio de **Tessellation** shading
 - Gera geometria dentro do pipeline (com código)
- Estágio de **Geometry** shading
 - Processa cada primitiva separadamente
- Estágio de **Fragment** shading
 - Processa cada fragmento separadamente

OPCIONAL

Pipeline programável do OpenGL

- Conforme o OpenGL Programming Guide 4.3:



Shaders

- Shaders dizem ao OpenGL **como** desenhar algo
- São **mini-programas** que definem o estilo de renderização
- Compilados para rodar na GPU
 - Grande quantidade de processadores



Shaders

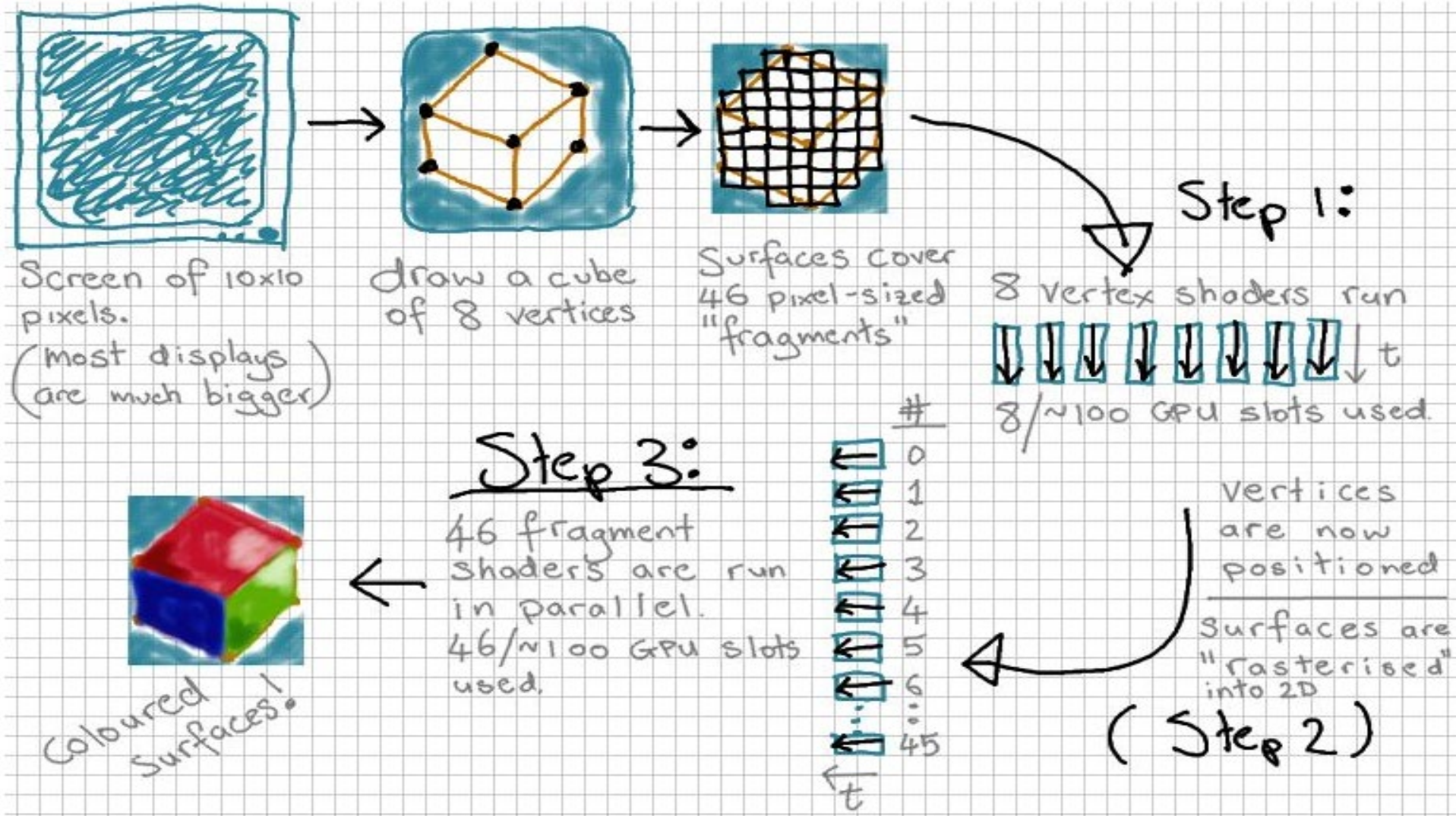
- Vertex Shaders: descrevem como tratar um vértice
 - Posição (3D → 2D)
 - Coordenadas de Textura
 - Cor
- Fragment Shaders: descrevem como tratar uma área (pixel-size)
 - Cor
 - Z-depth
 - Alpha value
- Processados paralelamente
 - Um para cada vértice de um modelo
 - Um para cada fragmento

Shaders

- Cada estágio de renderização pode ser dividido em processos separados (*pipeline* gráfico)
- Cada processo pode ser feito em um dos processadores disponíveis na GPU
 - Transformar cada vértice separadamente
 - Colorir cada fragmento separadamente
- Isso significa que podemos processar grande parte da renderização em paralelo!

Pipeline Gráfico

[Anton's 2014]



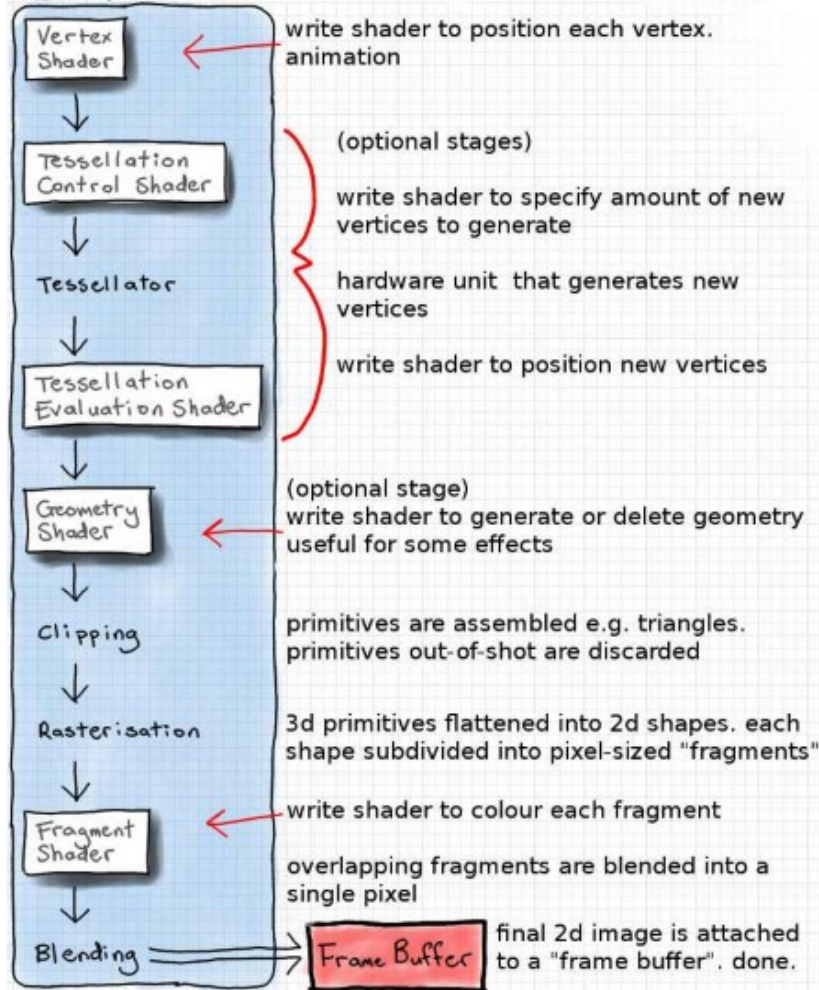
Shaders

- Shaders são uma forma de **reprogramar** o pipeline gráfico
- Se quisermos usar uma **cor** diferente ou **animar** e **girar** um objeto, podemos dizer ao OpenGL para usar um programa de shader diferente.
- Todos os estágios do pipeline gráfico que ocorrem na **GPU** são chamados de **pipeline do hardware**

C.P.U.
glDrawArrays()
↓ go!

OpenGL 4 Hardware Pipeline

G.P.U.



Um **Programa de Shader** compreende um conjunto de shaders (mini-programas) que controlam cada estágio

Todo o conjunto é compilado para gerar um único programa de shader

No mínimo **1 vertex shader e 1 fragment shader.**

[Anton's 2014]

Paralelismo

- Programas de shader rodam na GPU e são altamente **paralelizados**
- Cada vertex shader atua apenas em **1** vértice
- Malha de **2000** vértices?
 - 2000 vertex shaders serão executados ao desenhar
- Dependendo do nº de processadores da GPU, é possível rodar **toda malha em paralelo!**

Comparação de GPUs

GeForce 605	48 shader cores
Radeon HD 7350	80 shader cores
GeForce GTX 580	512 shader cores
Radeon HD 8750	768 shader cores
GeForce GTX 690	1536 shader cores
Radeon HD 8990	2304 shader cores

[Anton's 2014]

Paralelismo - Otimização

- Só podemos mandar desenhar um conjunto de primitivas por vez (buffer VAO)
 - Manter o número de malhas separadas **baixo**
 - Ou seja, tentar juntar o máximo de objetos em uma **única malha** (se possível)
- A ideia é usar o máximo de processadores em **paralelo**
- Cada chamada de desenho (***drawcall***) é uma chamada à execução de um dos programas de shader da aplicação, atuando sobre um conjunto de primitivas gráficas (buffer)

O segredo é minimizar o número de *drawcalls*

Programação de Shaders

- Na OpenGL 4 os shaders são programados em GLSL (OpenGL Shader Language)
- A primeira linha deve conter a **versão** simplificada do GLSL

- OpenGL 1.2 - no GLSL - no tag
- OpenGL 2.0 - GLSL 1.10.59 - #version 110
- OpenGL 2.1 - GLSL 1.20.8 - #version 120
- OpenGL 3.0 - GLSL 1.30.10 - #version 130
- OpenGL 3.1 - GLSL 1.40.08 - #version 140
- OpenGL 3.2 - GLSL 1.50.11 - #version 150
- OpenGL 3.3 - GLSL 3.30.6 - #version 330
- OpenGL 4.0 - GLSL 4.00.9 - #version 400
- OpenGL 4.1 - GLSL 4.10.6 - #version 410
- OpenGL 4.2 - GLSL 4.20.6 - #version 420
- OpenGL 4.3 - GLSL 4.30.6 - #version 430

Tipos de Dados em GLSL

- **void** – vazio, funções que não retornam valor
- **bool** – valor booleano
- **int** – valor inteiro com sinal
- **float** – valor de ponto flutuante
- **vec3** – ponto flutuante 3D (pontos e vetores de direção)
- **vec4** – ponto flutuante 4D (pontos, vetores de direção, cores)
- **mat3** – matrix 3x3 de ponto flutuante (transformações)
- **mat4** – matrix 4x4 de ponto flutuante (transformações)
- **sampler2D** – textura 2D carregada de uma imagem
- **samplerCube** – textura com 6 lados para sky-box
- **sampler2DShadow** – sombra projetada numa textura

Nomes de arquivo

- Os shaders podem ser escritos em arquivo texto ou armazenados como um array de caracteres
- É comum armazenar cada shader em um arquivo texto separadamente
- Exemplos de nomenclatura:
 - Cube_vs.glsl
 - Cube_fs.glsl
 - Map_vs.glsl
 - Map_fs.glsl

Vertex Shader

- Responsável por **posicionar** os vértices nas coordenadas finais
- Antes que o OpenGL **rasterize** (achate) a geometria para o 2D
- O shader abaixo recebe as posições dos vértices e joga na saída

```
#version 410

layout (location = 0) in vec3 vertex_position;

void main()
{
    gl_Position = vec4(vertex_position, 1.0);
}
```

Vertex Shader

- **in**: variável de entrada para o shader vinda do estágio anterior do pipeline
 - No caso, vinda dos Vertex Buffers
- **out**: envia a variável para o próximo estágio do pipeline
- **gl_Position**: define a posição final de determinado vértice

```
#version 410

layout (location = 0) in vec3 vertex_position;

void main()
{
    gl_Position = vec4(vertex_position, 1.0);
}
```

Será executada
uma instância para
cada vértice
contido no Vertex
Buffer

Fragmentos x Pixels

- Pixel (Picture Element)
 - Elementos que compõem a imagem 2D, que será exibida na matriz da tela
- Fragmento
 - Área de uma superfície que possui o tamanho de um pixel
 - “Candidatos a pixel”
- O fragment shader determina a cor de cada fragmento
 - Cor que efetivamente o pixel terá

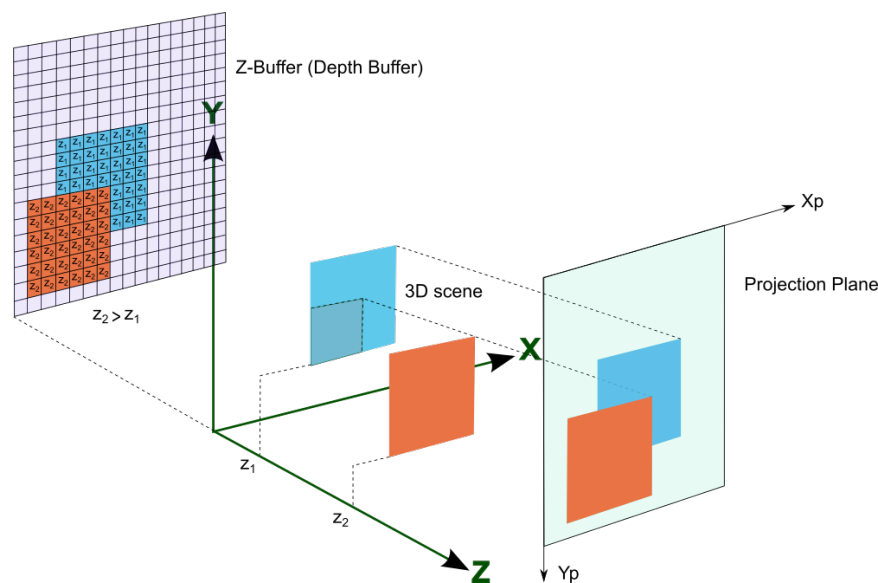
Fragmentos x Pixels

- Muitas vezes as superfícies se **sobrepõem**
 - Logo, temos mais de 1 fragmento por pixel.
- Por padrão, todos os fragmentos são desenhados, até mesmo os que estão ocultos.



Fragmentos x Pixels

- Para evitar a redundância: **Depth Testing**
 - Desenha apenas os fragmentos mais a frente, ignorando os que estão mais afastados



Fragment Shader

- Uma vez que todos os vertex shaders processaram a posição final de todos os vértices, os fragment shaders rodam para cada fragmento entre os vértices
- Responsável por definir a **cor** de cada fragmento

```
#version 410
uniform vec4 inputColor;
out vec4 color;

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


Fragment Shader

- **uniform**: shader recebe como entrada uma variável vinda da CPU
 - Esta variável é global a todos os shaders
 - Ou seja, seria possível acessá-la do vertex shader se quiséssemos
- Cores RGBA variam de 0.0 a 1.0 (não de 0 a 255) – são normalizadas

```
#version 410
uniform vec4 inputColor;
out vec4 color;

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

Enviando dados para a GPU

- Exemplo (enviando uma cor)
 - Selecionar o programa de shader ao qual quer enviar

```
glUseProgram(shaderProgram);
```

- Geração do identificador e envio

```
//Enviando a cor desejada (vec4) para o fragment shader  
GLint colorLoc = glGetUniformLocation(shaderProgram, "inputColor");  
assert(colorLoc > -1);  
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```

Enviando dados para a GPU

Código do shader (GLSL)

```
#version 410
uniform vec4 inputColor;
out vec4 color;

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

Este nome deve ser
o mesmo do código
em GLSL!!!!!!

Código em C++

```
//Enviando a cor desejada (vec4) para o fragment shader
GLint colorLoc = glGetUniformLocation(shaderProgram, "inputColor");
assert(colorLoc > -1);
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```

Enviando dados para a GPU

- O comando para envio varia de acordo com o tipo de dado

```
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```

Nome do identificador

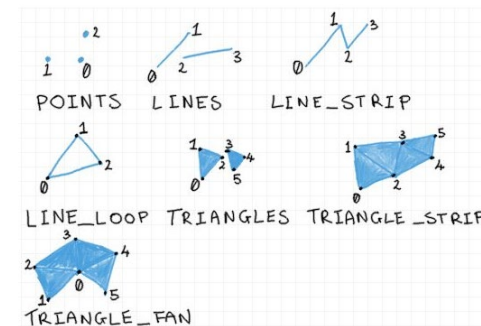
Tipo do dado a ser enviado

Número de dados a serem enviados

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml>

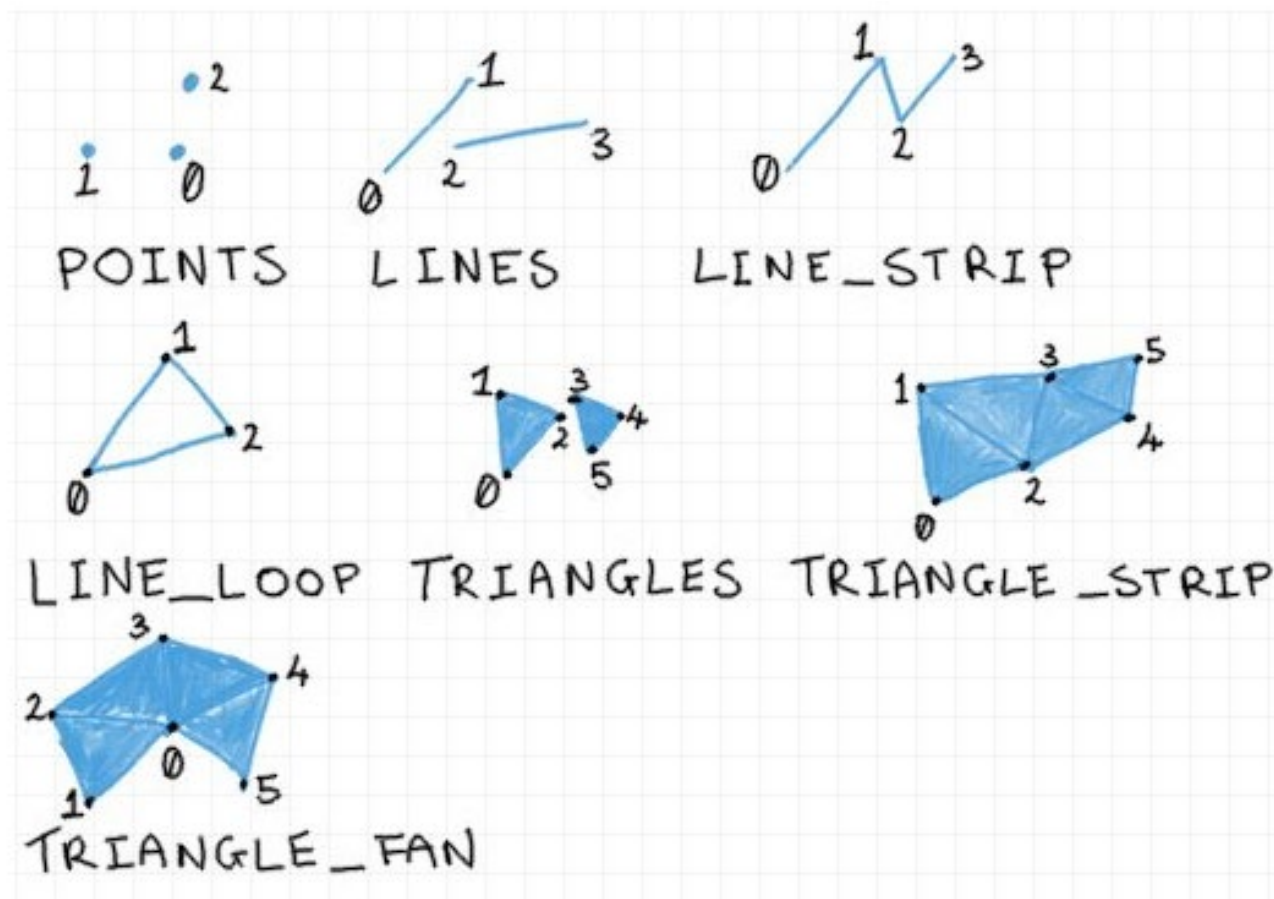
Desenhando primitivas

- **glDrawArrays** (<mode>, <first>, <count>)
 - **mode**: tipo de primitiva a desenhar
 - **GL_POINTS**, **GL_LINES**, **GL_TRIANGLES**...
 - **first**: índice inicial do VAO ativo
 - **count**: número de índices a serem renderizados



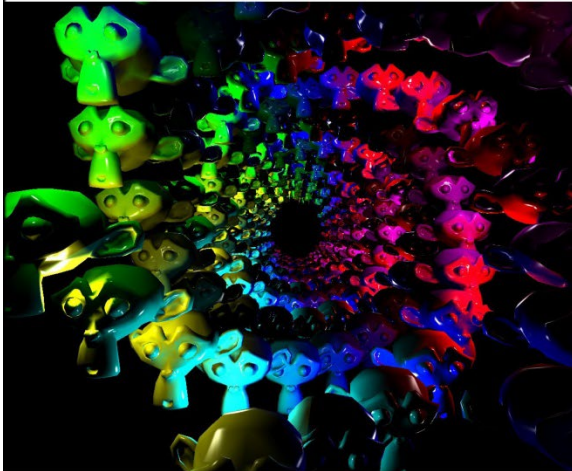
- Exemplo
 - `glDrawArrays (GL_TRIANGLES, 0, 3);`

Desenhando Primitivas



Referências bibliográficas

Anton's OpenGL 4 Tutorials



Anton Gerdelan

Ebook para Kindle

Muitos materiais online disponíveis em:

<http://antongerdelan.net/opengl/>

Referências bibliográficas

- Leituras recomendadas (LearnOpenGL e Anton's OpenGL 4 Tutorials)
- Slides sobre CG dos professores: Christian Hofsetz, Cristiano Franco, Marcelo Walter, Soraia Musse, Leandro Tonietto e Rafael Hocevar