Introdução ao OpenGL 2.1 e GLSL 1.2

Felipe Bessa Coelho

Universidade de São Paulo

25 de Setembro de 2012

Sumário

Objetivos

- Entender por que foi necessária a mudança para a nova estrutura;
- Entender as diferenças entre o modelo antigo e o atual;
- Perceber que, apesar de "mais código", a estrutura geral de uma aplicação com OpenGL fica muito mais simples, e os detalhes não são mais implícitos.

Objetivos

- Entender por que foi necessária a mudança para a nova estrutura;
- Entender as diferenças entre o modelo antigo e o atual;
- Perceber que, apesar de "mais código", a estrutura geral de uma aplicação com OpenGL fica muito mais simples, e os detalhes não são mais implícitos.

Objetivos

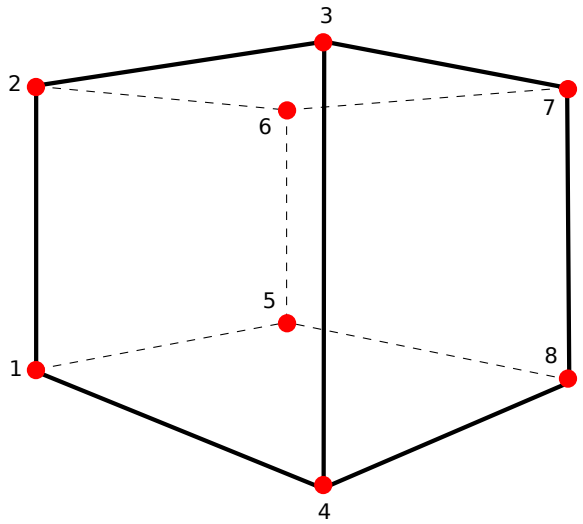
- Entender por que foi necessária a mudança para a nova estrutura;
- Entender as diferenças entre o modelo antigo e o atual;
- Perceber que, apesar de "mais código", a estrutura geral de uma aplicação com OpenGL fica muito mais simples, e os detalhes não são mais implícitos.

Motivação

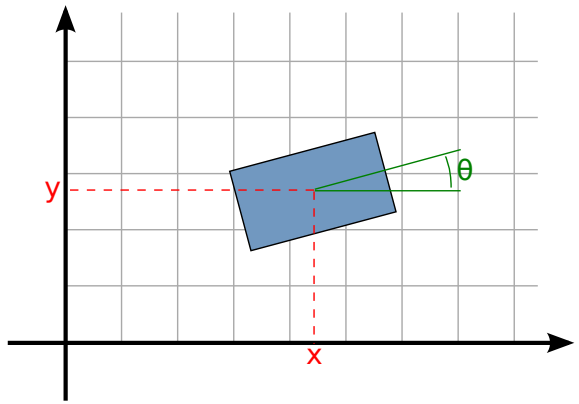
- Objetos podem estar em qualquer lugar;
- Precisamos de um meio para estabelecer a posição e a orientação de cada elemento;
- Precisamos alterar a posição de objetos para produzir o movimento em nossas cenas. Isso tem que ser feito de forma **determinística**.

Elementos

- Cada objeto é apenas um conjunto de pontos...



- E cada um destes pontos tem uma **posição**;
- Cada objeto, por outro lado, além da posição tem uma **orientação**.



Coordenadas homogêneas

- Para representar cada elemento no espaço, utilizamos o que se conhece por *coordenadas homogêneas*;
- Cada ponto tem sua posição representada por um vetor:

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- E cada objeto é representado utilizando-se uma matrix 4x4:

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

- Coordenadas homogêneas permitem que operações como **translação**, **rotação**, **escala** e a própria **projeção** sejam implementadas como operações matriciais;

Transformações

- Realizar transformações geométricas quando se tem coordenadas homogênas é tão simples quanto realizar uma multiplicação de matrizes!
- Dada uma transformação $\mathbf{T}$ e um ponto $\mathbf{p}$, para aplicar a transformação sobre o ponto basta fazer:

$$\mathbf{p}' = \mathbf{T}\mathbf{p} = \begin{bmatrix} t_0 & t_4 & t_8 & t_{12} \\ t_1 & t_5 & t_9 & t_{13} \\ t_2 & t_6 & t_{10} & t_{14} \\ t_3 & t_7 & t_{11} & t_{15} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Transformações - Translação

- A matriz que realiza uma translação tem a seguinte forma:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \mathrm{dx} \\ 0 & 1 & 0 & \mathrm{dy} \\ 0 & 0 & 1 & \mathrm{dz} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformações - Escala

- A matriz que realiza uma escala tem a seguinte forma:

$$\mathbf{T} = \begin{bmatrix} \mathrm{x} & 0 & 0 & 0 \\ 0 & \mathrm{y} & 0 & 0 \\ 0 & 0 & \mathrm{z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformações - Rotação I

- Para a rotação, temos três casos básicos: rotação sobre o eixo x, y e z. As matrizes estão representadas abaixo:

$$\mathbf{R_x}(\psi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi & 0 \\ 0 & \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_y}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_z}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

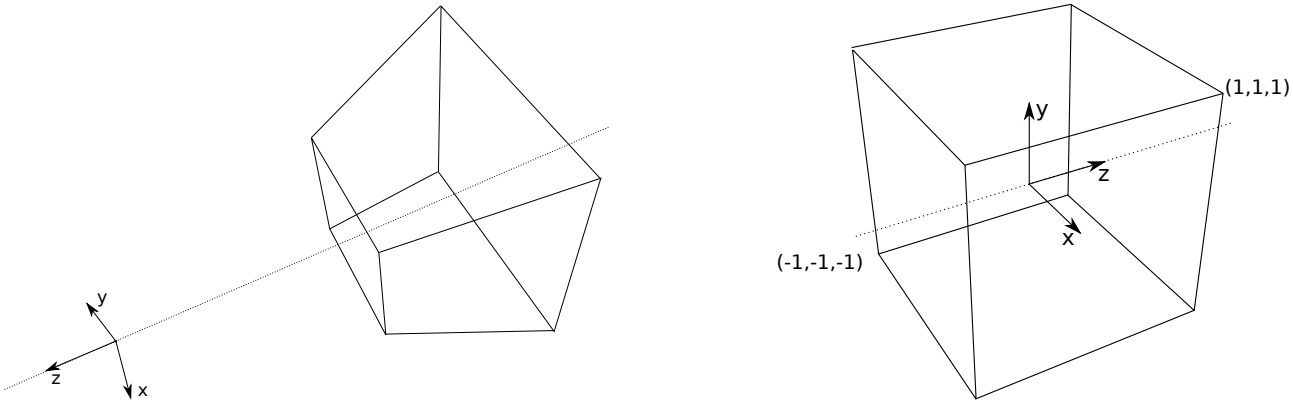
Transformações - Rotação II

- E além destes, um caso "um pouco"mais complicado: a rotação sobre um eixo arbitrário $\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T$:

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z \sin\theta & u_x u_z(1-\cos\theta) + u_y \sin\theta & 0 \\ u_y u_x(1-\cos\theta) + u_z \sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x \sin\theta & 0 \\ u_z u_x(1-\cos\theta) - u_y \sin\theta & u_z u_y(1-\cos\theta) + u_x \sin\theta & \cos\theta u_z^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformações - Projeção perspectiva

- Um outro caso muito importante é o da projeção perspectiva, i.e., transformar o volume de visualização 1a no volume de visualização 1b:



(a) Volume de visualização em perspectiva

(b) Volume de visualização ortográfico

Figura 1: Volumes de visualização

Transformações - Composição

- Para compor duas ou mais transformações, basta multiplicar as matrizes correspondentes;
- Por exemplo, para realizar uma translação $\mathbf{T}$ e depois uma rotação $\mathbf{R}$ sobre o ponto $\mathbf{p}$, pode-se fazer:

$$\mathbf{p_1} = \mathbf{Tp}$$
$$\mathbf{p_2} = \mathbf{Rp_1}$$

- Ou então, em uma "única"operação:

$$\mathbf{p_2} = \mathbf{Rp_1} = \mathbf{RTp}$$

Sistemas de coordenadas I

- Matrizes são usadas principalmente para representar mudanças entre sistemas de coordenadas;
- Constrói-se uma matriz que "traduz"posições relativas a um determinado sistema de coordenadas para outro.

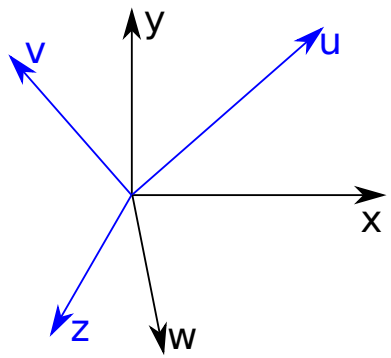


Figura 2: Sistemas de coordenadas diferentes

Sistemas de coordenadas II

- A matriz que faz a transformação do sistema de coordenadas $\mathbf{UVW}$ para o sistema $\mathbf{UVW}$ é:

$$\mathbf{M} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Caso os dois sistemas de coordenadas não tenham a mesma origem, pode-se adicionar uma componente de translação:

$$\mathbf{M_{uvw \rightarrow xyz}} = \begin{bmatrix} 1 & 0 & 0 & O_x \\ 0 & 1 & 0 & O_y \\ 0 & 0 & 1 & O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{M}$$

Sistemas de coordenadas "padrões"

- Em OpenGL e dentro da computação gráfica como um todo, são descritos alguns sistemas de coordenadas que são muito comuns, e fazem parte do *pipeline*. São eles:
  - Object space;
  - World space;
  - View space;
  - Projection space;
  - Normalized device coordinates.

Object space

- Este é o sistema de coordenadas que está associado a cada objeto;
- Os pontos que compõem o objeto são descritos em relação à origem desse sistema de coordenadas;

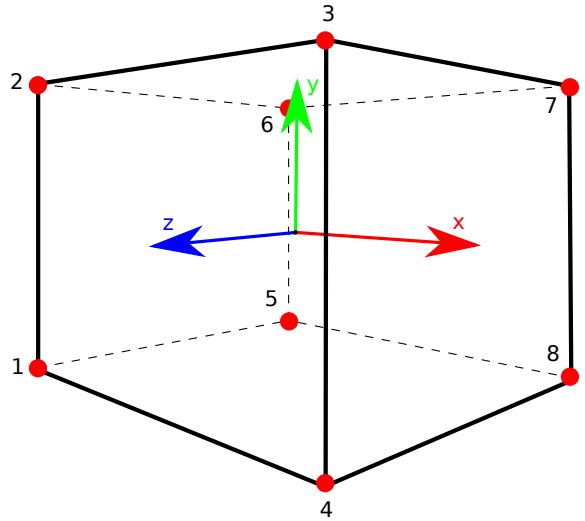


Figura 3: Sistema de coordenadas do objeto

World space

- Este é o sistema de coordenadas global, um sistema de coordenadas "fixo", em relação ao qual são posicionados todos os objetos;
- Geralmente, somente as posições dos objetos são descritas em relação a esse sistema de coordenadas;

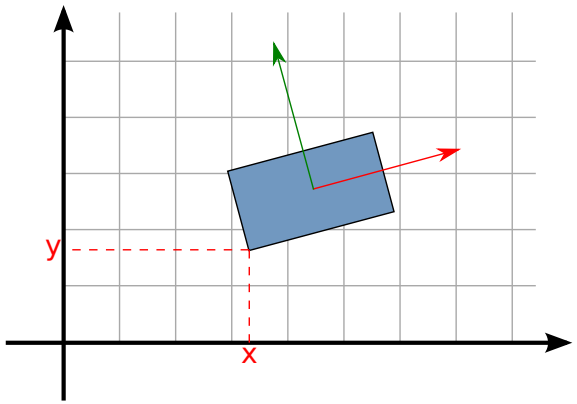


Figura 4: Sistema de coordenadas global

View space

- Esse sistema de coordenadas está acoplado à "câmera";
- Os objetos ainda não estão como vistos pela câmera!

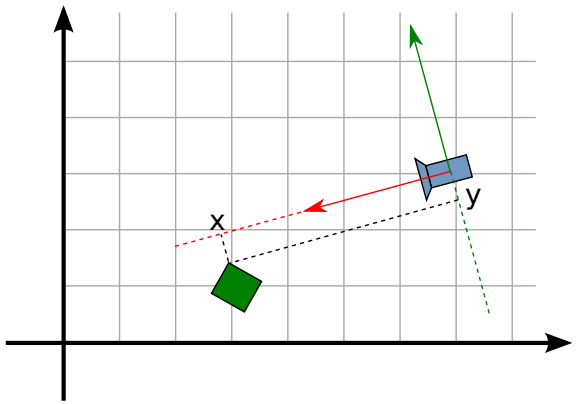


Figura 5: Sistema de coordenadas da câmera

Projection space

- Nesse espaço, já se pode saber quais pontos da cena fazem parte da vista final;
- Cada ponto $\mathbf{p} = \begin{bmatrix} x & y & z & w \end{bmatrix}^T$ faz parte do volume de visualização se $-w \leq x, y, z \leq w$.

Normalized device coordinates

- Para construir este espaço, cada ponto $\mathbf{p}$ do espaço de projeção é transformado para $\mathbf{p} = \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$;
- É um cubo que ocupa o intervalo $[-1; 1]$ em todos os eixos.

Janela

- Após o último passo, o OpenGL transforma as coordenadas do espaço com $x, y, z \in [-1; 1]$ para a janela. Pode-se controlar a parte da janela que está sendo usada com `glViewport`.

Sequência de operações

- As principais matrizes envolvidas no processo são:
  - $\mathbf{M}$, a matriz do objeto, conhecida por *model matrix*;
  - $\mathbf{V}$, a matriz da câmera, conhecida por *view matrix*;
  - $\mathbf{P}$, a matriz de projeção, conhecida por *projection matrix*;
- A união das matrizes $\mathbf{M}$ e $\mathbf{V}$ forma a chamada matriz *modelview*;
- O processo completo é descrito na figura 6.

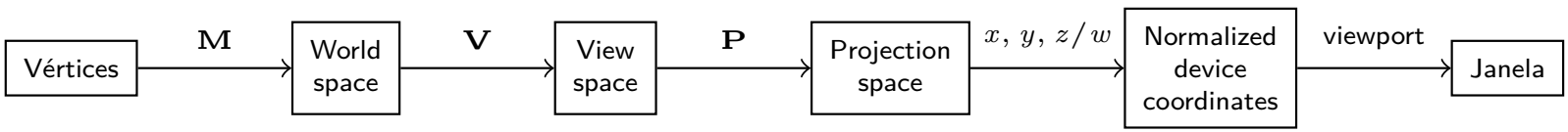


Figura 6: Sequência de operações desde os vértices de origem até a posição na janela

Importância

- O principal motivo de saber em que espaço cada elemento está é pra garantir que todas as contas serão feitas com elementos que estão *no mesmo espaço*!
- Por exemplo, para calcular a iluminação pelo modelo de *Phong*, não obteremos o resultado correto se a normal da superfície estiver descrita no sistema de coordenadas do objeto e a direção da luz estiver no sistema de coordenadas global. Se necessário, será preciso transformar todos os elementos para o mesmo espaço a fim de obter resultados corretos!

O que é OpenGL I

What is the OpenGL Graphics System?

OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.

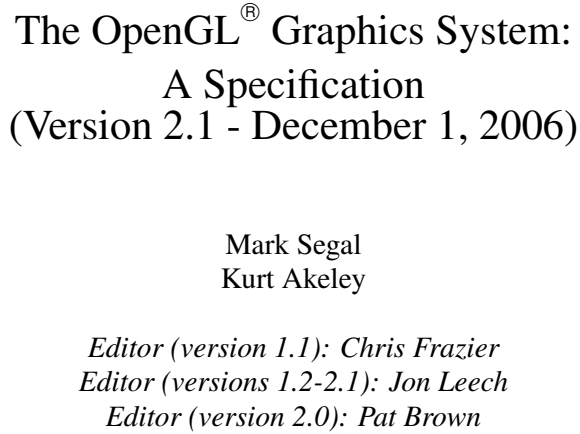
- É um padrão "aberto": associados ao Khronos Group podem votar e decidir o que vai ser implementado.

O que é OpenGL II

Figura 7: Promotores do Khronos Group

Especificação

- Tudo que é possível ser feito com OpenGL está escrito na espeficifação:



The OpenGL® Graphics System:
A Specification
(Version 2.1 - December 1, 2006)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-2.1): Jon Leech
Editor (version 2.0): Pat Brown

Figura 8: Primeira página da especificação para OpenGL 2.1

- É um documento que contém a descrição de todos os procedimentos disponíveis, seus argumentos, efeitos colaterais, equações que são usadas e mais muita coisa...

História

- 1980 - 1990: IrisGL
- 1992: OpenGL 1.0
- 1997: OpenGL 1.2
- 1999-2000: Primeiras GPUs com hardware pra T&L
- 2000: Direct3D 8.0 + Shaders
- 2003: Direct3D 9.0 + HLSL
- 2004: OpenGL 2.0 + GLSL
- 2005: Xbox 360 + Direct3D 9.0
- 2006: OpenGL 2.1 + GLSL 1.2
- 2007: Direct3D 10 + Geometry shaders
- 2008: OpenGL 3
- 2009: OpenGL 3.1, coisas antigas removidas
- 2009: Direct3D 11
- 2010: OpenGL 4
- ...
- Agosto 2012: D3D 11.1, OpenGL 4.3

OpenGL 2.1

- Shaders (2.0)
- VBO (1.5)
- FBO com extensões (3.0)

Implementação

- Cada implementação de OpenGL fornece alguns recursos diferentes;
- Dependendo do sistema operacional, uma versão base diferente de OpenGL pode estar disponível;
- Assume-se que, no Linux, existem disponíveis no começo apenas as funções até OpenGL 1.2;
- No Windows, encontra-se apenas até OpenGL 1.1;
- Para ter acesso a **qualquer coisa** além destas versões, é quase sempre necessário carregar as funções como *extensões*.

Extensões

- Extensões definem novas funções, constantes, estruturas dentro do pipeline, construções para GLSL etc.;
- Para utilizar uma extensão dentro do programa que está sendo desenvolvido, é necessário pedir a extensão de interesse ao *driver* instalado no computador;
- Para carregar uma determinada função pertencente a uma extensão, realiza-se o seguinte procedimento:

```
typedef GLuint (*PFNGLCREATESHADERPROC)(GLenum type);
PFNGLCREATESHADERPROC glCreateShader;
...
glCreateShader = glxGetProcAddress("glCreateShader");
if (glCreateShader)
    printf("Uhu, temos glCreateShader!");
```

Extensões

- Carregar 10, 20, 30 funções diferentes deste jeito é um processo bastante tedioso..
- Pra isso existe GLEW! (e outras também, mas vamos usar essa)

```
GLenum err = glewInit();
if (GLEW_OK != err)
    printf("Nao consegui carregar as extensoes");
if (GLEW_VERSION_2_1)
    printf("Extensoes pra OpenGL 2.1 carregadas!");
```

Pipeline fixo

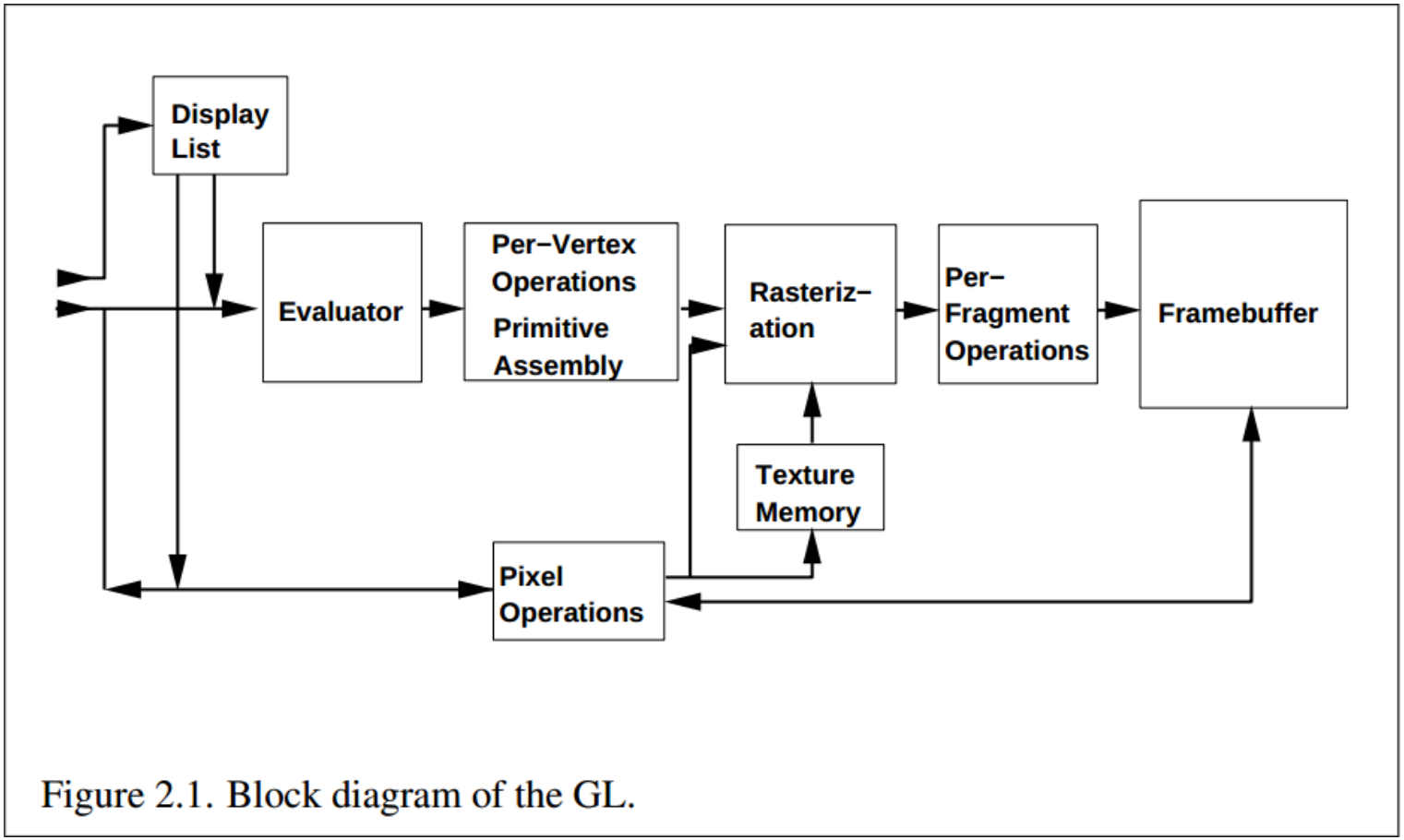


Figura 9: Pipeline fixo

Pipeline fixo

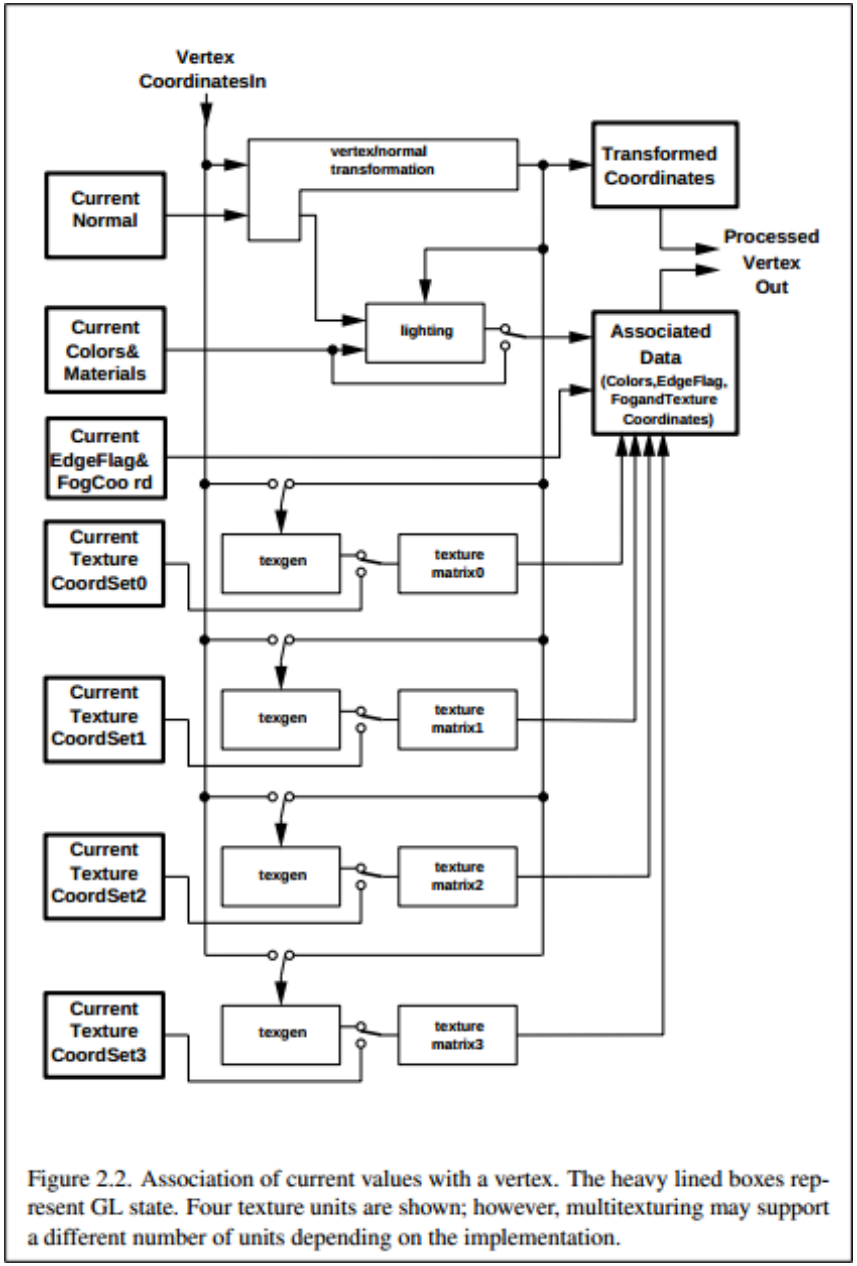


Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

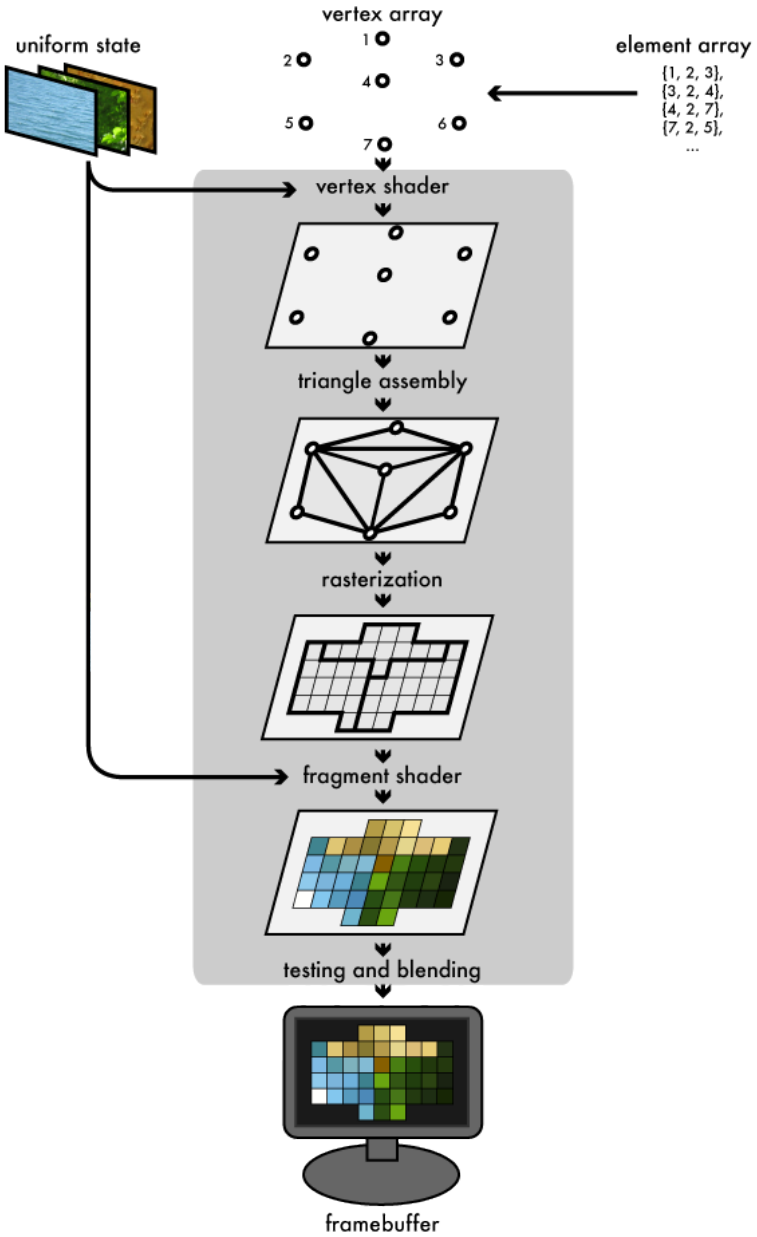
Pipeline fixo

- `glEnable`;
- `glDisable`;
- `glLight`
- ...
- *Não mais!*

Pipeline programável

- Permite sobreescrever partes do *pipeline* com a funcionalidade que *nós queremos*;
- Permite fazer coisas que antes eram, literalmente, impossíveis;
- Permite realizar processos da forma que desejarmos;
- Faz com que não sejam usados recursos desnecessários;
- .. É programável! Faça o que quiser!

Pipeline programável



GLSL

- Linguagem de programação baseada em C;
- Programas que vão ser executados **na GPU**;
- Um programa para cada etapa do pipeline:
  - Vertex shader (2.0)
  - Tesselation control (4.0)
  - Tesselation evaluation (4.0)
  - Geometry shader (3.2)
  - Fragment shader (2.0)

GLSL

- Linguagem de programação baseada em C;
- Programas que vão ser executados **na GPU**;
- Um programa para cada etapa do pipeline:
  - Vertex shader (2.0)
  - Tesselation control (4.0)
  - Tesselation evaluation (4.0)
  - Geometry shader (3.2)
  - Fragment shader (2.0)

GLSL

- Linguagem de programação baseada em C;
- Programas que vão ser executados **na GPU**;
- Um programa para cada etapa do pipeline:
  - Vertex shader (2.0)
  - Tesselation control (4.0)
  - Tesselation evaluation (4.0)
  - Geometry shader (3.2)
  - Fragment shader (2.0)

Pipeline

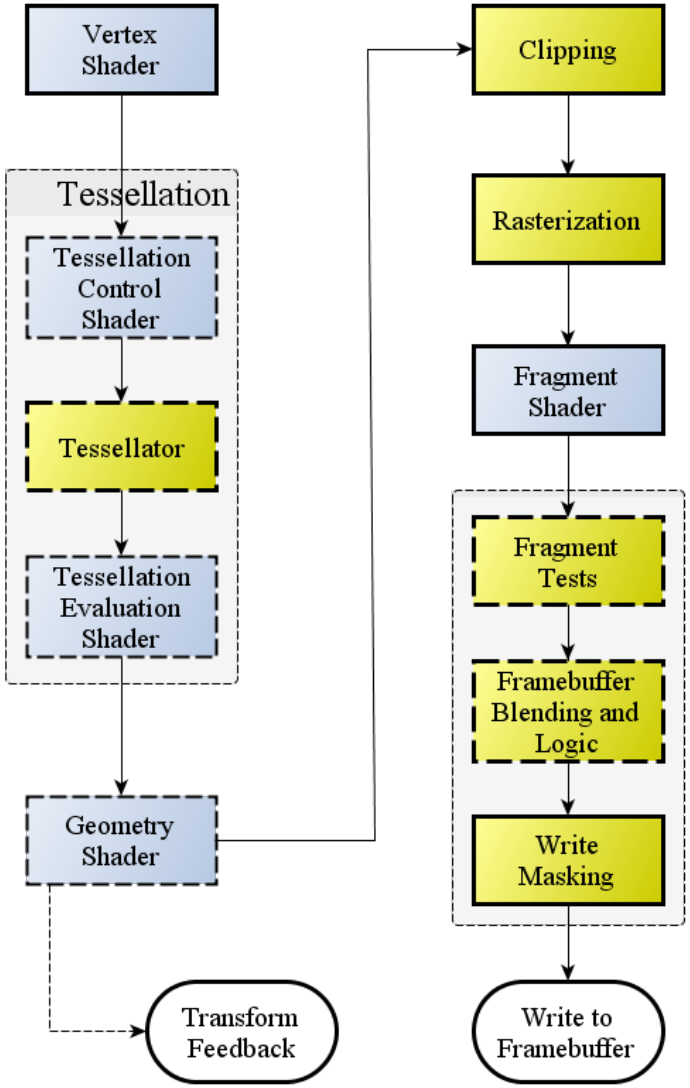


Figura 12: Pipeline atual

Shaders

- Recebem dados específicos pra cada estágio;
- Tem acesso a dados "globais"para todos os estágios;
- Dependendo do estágio, pode receber N entradas e produzir M saídas;

Elementos GLSL

Uniforms

Uniforms são variáveis que podem ser acessadas por todos os *shaders*

Varyings

São variáveis que são passadas ao estágio seguinte no *pipeline*

Attributes

São variáveis de entrada, que correspondem a dados dos vértices que estão sendo desenhados

Vertex shader

- Recebe os dados dos vértices:
  - posição
  - normal
  - coordenadas de textura
  - cor
  - quaisquer outros atributos de interesse
- Emitem necessariamente a posição do vértice no espaço de projeção (*Projection space* ou *clip coordinates*) na variável `gl_Position`;
- Podem emitir informações para os próximos estágios

Vertex shader

```
void main()
{
    gl_Position =
        gl_ProjectionMatrix
      * gl_ModelviewMatrix
      * gl_Vertex;
}
```

Fragment shader

- Não recebe nenhum dado obrigatório;
- Último estágio programável do *pipeline*;
- Se receber algum dado, vem geralmente do *vertex shader*;
- Emite necessariamente a cor final a ser armazenada no *buffer* (tela), geralmente armazenada na variável `gl_FragColor`. Caso esteja sendo utilizado MRT (*Multiple Render Targets*), a variável de saída será `gl_FragData`;
- Pode escrever a profundidade em `gl_FragDepth`, mas caso não seja feito, a variável recebe o valor correto automaticamente.

Fragment shader

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