Computação Gráfica I

Professor:

Anselmo Montenegro
www.ic.uff.br/~anselmo

Conteúdo:

- Introdução à OpenGL

OpenGL: introdução

- OpenGL é uma *biblioteca gráfica*.
- É uma interface em software para os dispositivos gráficos.
- A interface consiste de comandos para especificar objetos e operações que compõem aplicações gráficas 2D e 3D.

OpenGL: características

- Possui primitivas e operações para a geração e manipulação de dados vetoriais e matriciais.
- Capaz de gerar imagens de alta qualidade.
- Comumente implementado de forma a tirar partido da aceleração gráfica (se disponível).
- Independente de plataforma.

OpenGL: características

- Possui primitivas e operações para a geração e manipulação de dados vetoriais e matriciais.
- Capaz de gerar imagens de alta qualidade.
- Comumente implementado de forma a tirar partido da aceleração gráfica (se disponível).
- Independente de plataforma.

OpenGL: o que é capaz de fazer

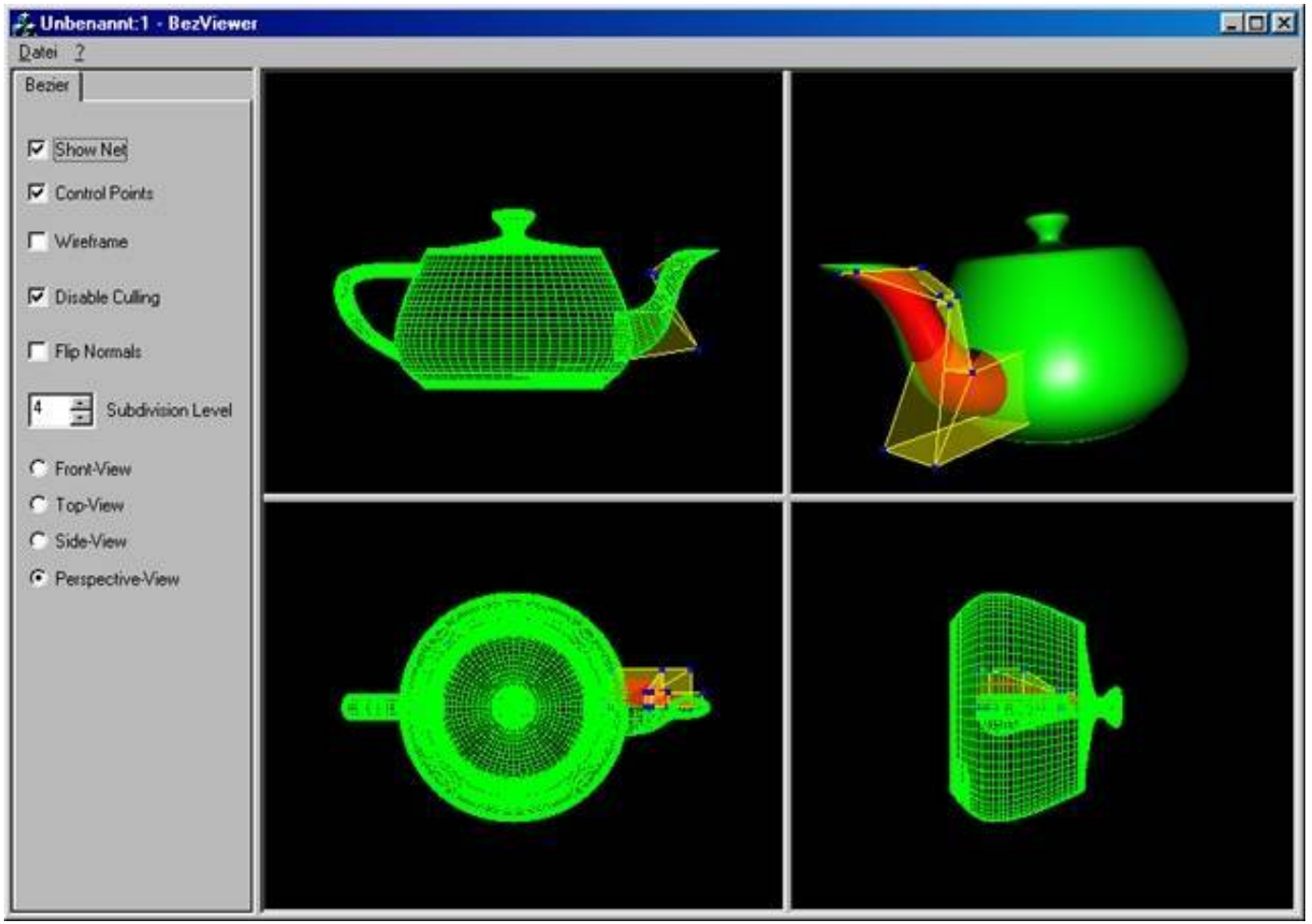
- Não gerencia janelas nem trata eventos produzidos por dispositivos de interação.
- Não possui comandos de alto nível para especificação de objetos 3D complexos.
- Objetos complexos devem ser construídos a partir de primitivas geométricas simples.

OpenGL: o que não é capaz de fazer

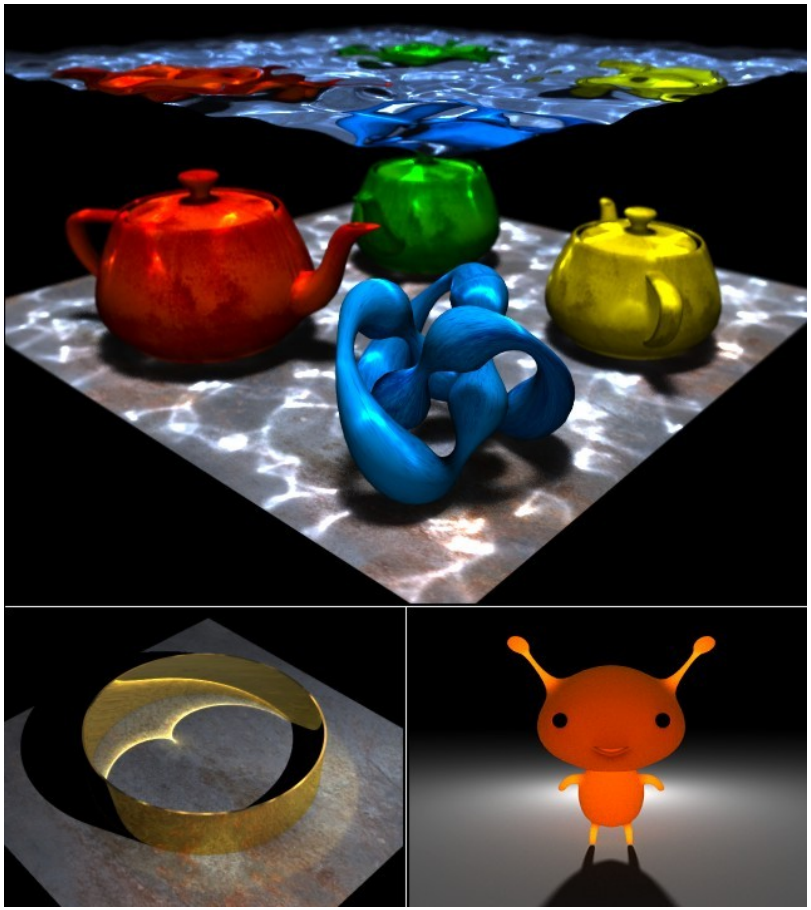
- Cria descrições matemáticas de objetos a partir de primitivas geométricas (pontos, linhas e polígonos) e imagens/bitmaps.

- Organiza os objetos no espaço 3D e seleciona o ponto de vista adequado para a cena.

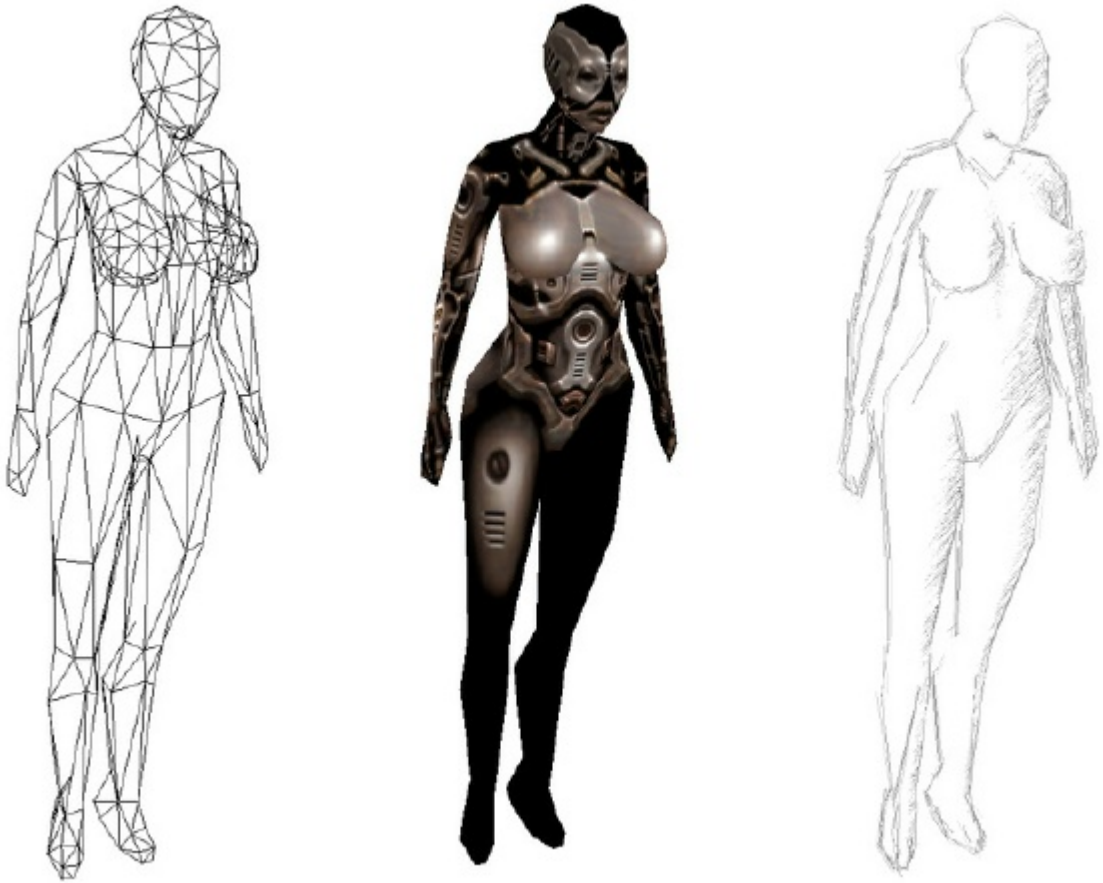
OpenGL: exemplos de aplicações



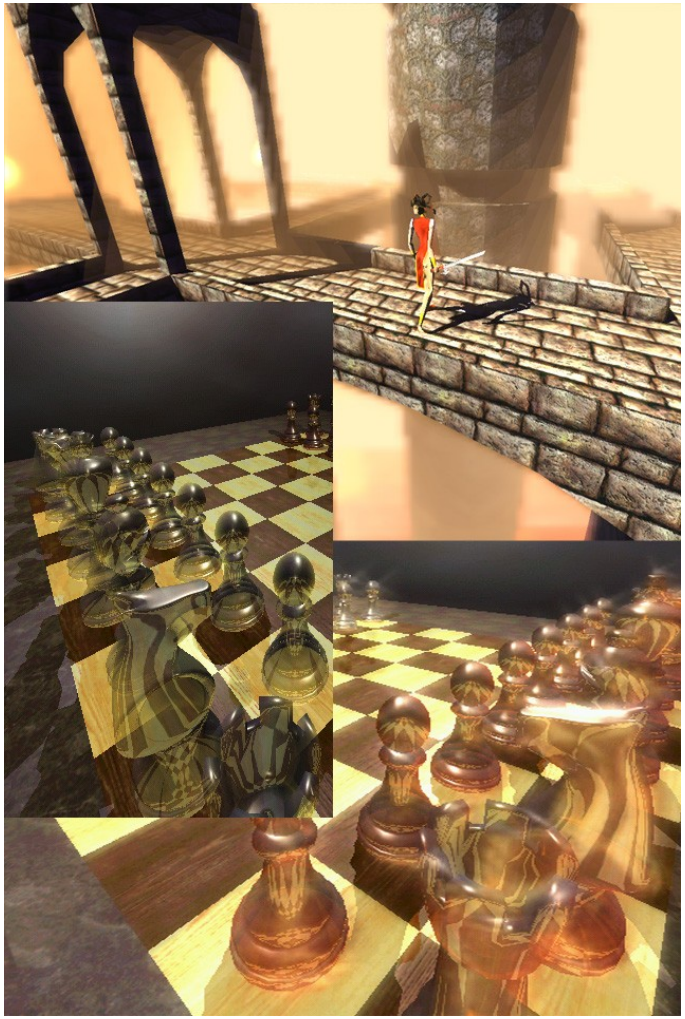
OpenGL: exemplos de aplicações



OpenGL: exemplos de aplicações



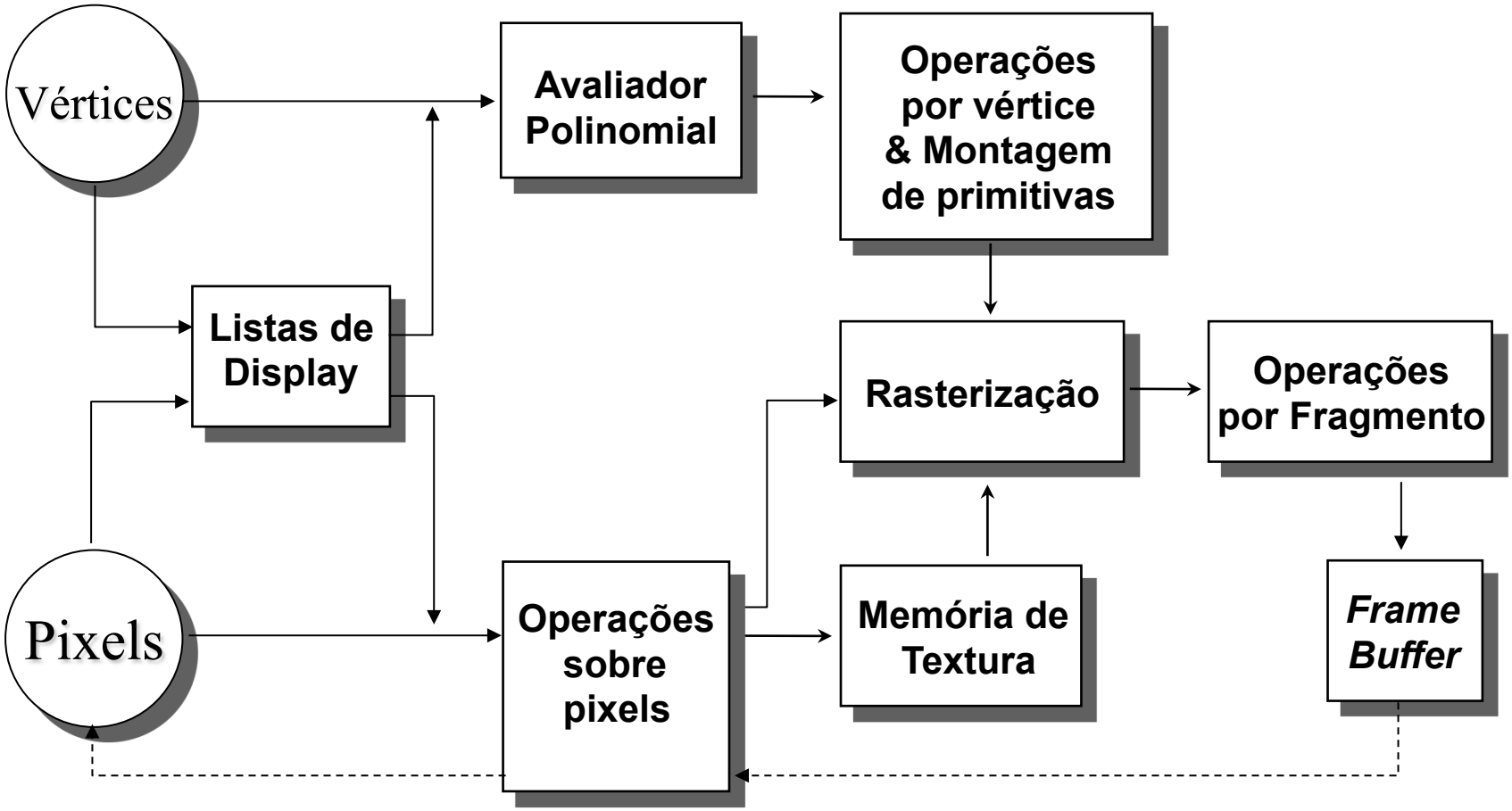
OpenGL: exemplos de aplicações



OpenGL: exemplos de aplicações

- Rendering – processo através do qual um computador gera imagens a partir de um modelo.
- Pixel – menor elemento visível que um dispositivo gráfico pode apresentar. Uma imagem é formada por vários pixels.
- Plano de bits – área de memória que armazena um bit de informação para cada pixel.
- Framebuffer – estrutura que armazena todas as informações necessárias para que o display gráfico possa controlar a intensidade da cor em cada pixel.

OpenGL: arquitetura (pipeline gráfico)

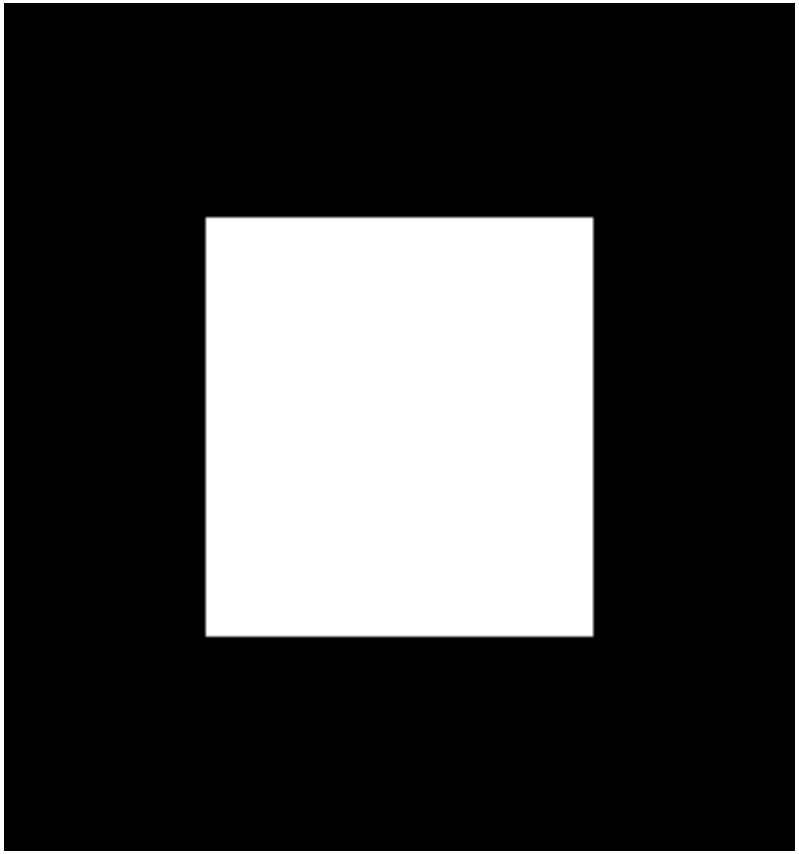


OpenGL: exemplos de aplicações

```
#include <whateverYouNeed.h>
main()
 { InitializeAWindowPlease();                 /* Chama rotinas para criação e gerenciamento da janela
                                                 de interface (não faz parte da OpenGL) */
   glClearColor(0.0, 0.0, 0.0, 0.0);          /* Seleciona a cor negra para limpar a janela */
   glClear(GL_COLOR_BUFFER_BIT);              /* Limpa o buffer de cor com a cor estabelecida */
   glColor3f(1.0, 1.0, 1.0);                  /* Escolhe a cor branca para desenhar as primitivas */
   glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);  /* Estabelece o sistema de coordenadas e a forma como a
                                                 imagem é mapeada na janela */
   glBegin(GL_POLYGON);                       /* Inicia o desenho da primitiva poligono */
      glVertex2f(-0.5, -0.5);                 /* Primeiro vertice */
      glVertex2f(-0.5, 0.5);
      glVertex2f(0.5, 0.5);
      glVertex2f(0.5, -0.5);                  /* Ultimo vertice */
   glEnd();                                   /* Encerra desenho da primitiva */
   glFlush();

   UpdateTheWindowAndCheckForEvents();  /* gerencia o conteúdo da janela e processa eventos */
 }
```

OpenGL: resultado



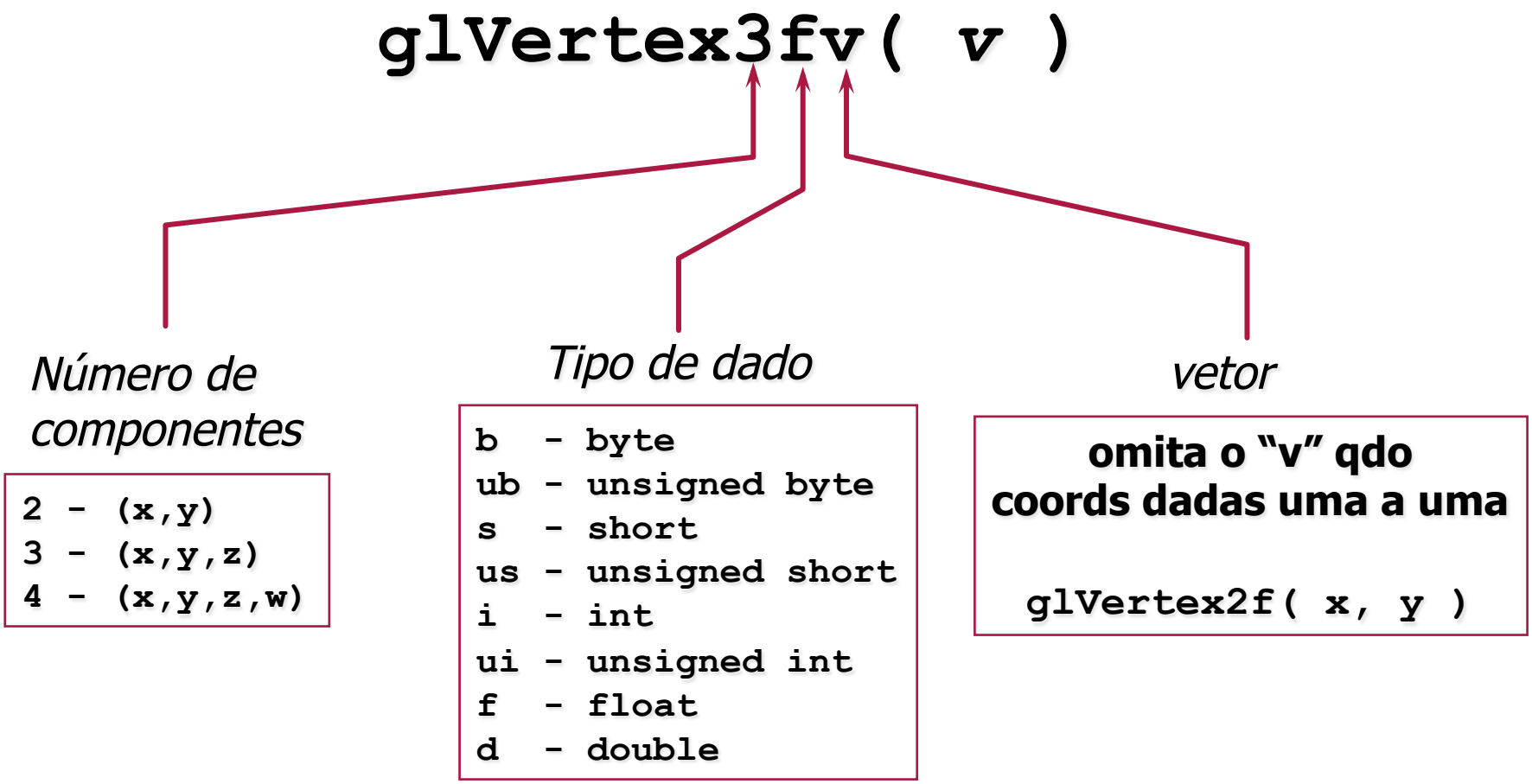
OpenGL: sintaxe dos comandos

- Todos os comandos começam com o prefixo gl (Ex.: glClearColor).
- As palavras nos nome dos comandos começam com letras maiúsculas (Ex.: glColor()).
- O sufixo indica o número e o tipo dos argumentos (Ex.: glVertex2i(1,3)).
- As constantes começam com GL_ (Ex.: GL_COLOR_BUFFER_BIT).

OpenGL: sintaxe dos comandos

- Todos os comandos começam com o sufixo gl (Ex.: glClearColor).
- As palavras nos nome dos comandos começam com letras maiúsculas (Ex.: glColor()).
- O sufixo indica o número e o tipo dos argumentos (Ex.: glVertex2i(1,3)).
- As constantes começam com GL_ (Ex.: GL_COLOR_BUFFER_BIT).

OpenGL: sintaxe dos comandos



OpenGL: sufixo e tipo dos argumentos

Sufixo	Tipo	C	OpenGL
b	Inteiro 8-bits	signed char	GLbyte
s	Inteiro 16-bits	short	GLshort
i	Inteiro 32-bits	long	GLint, GLsizei
f	Ponto-flutuante 32-bit	float	GLfloat, GLclampf
d	Ponto-flutuante 64-bit	double	GLdouble, GLclampd
ub	Caractere s/ sinal 8-bit	unsigned char	GLubyte, GLboolean
us	Caractere s/ sinal 16-bit	unsigned short	GLushort
ui	Caractere s /sinal 32-bit	unsigned long	GLuint, GLenum, GLbitfield

OpenGL: máquina de estados

- A OpenGL funciona como uma máquina de estados.
- Os estados correntes permanecem ativos até que sejam modificados.
- Exemplo: a cor de desenho corrente é aplicada a qualquer primitiva geométrica até que seja modificada.

OpenGL: máquina de estados

- Existem vários estados:
  - Cor de desenho corrente.
  - Transformações de visualização e projeção.
  - Padrões de linhas e polígonos.
  - Modo de desenho dos polígonos.
  - Posição e característica das fontes de luz.
  - Propriedades dos materiais associados aos objetos.
  - etc.

OpenGL: máquina de estados

- Vários estados se referem a modos que estão habilitados ou desabilitados.

- Estes estados são modificados através dos comandos:

  – glEnable() e glDisable().
  – Exemplo: glEnable(GL_LIGHTINING).

OpenGL: máquina de estados

- Alguns comandos para ler um estado:
  - glGetBooleanv(), glGetDoublev(), glGetFloatv(), glGetIntegerv(), glPointerv() ou glIsEnabled().
- Comandos para salvar um estado:
  - glPushAttrib() e glPushClientAttrib().
- Comandos para restaurar um estado:
  - glPopAttrib() e glPopClientAttrib().

OpenGL: API's relacionadas

- GLU (OpenGL Utility Library)
  - Parte do padrão OpenGL.
  - NURBS, trianguladores, quádricas, etc.
- AGL, GLX, WGL
  - Camadas entre o OpenGL os diversos sistemas de janelas.
- GLUT (OpenGL Utility Toolkit)
  - API portátil de acesso aos sistemas de janelas.
  - Encapsula e esconde as camadas proprietárias.
  - Não é parte official do OpenGL.

OpenGL: GLUT

- Biblioteca para criação de interfaces gráficas simples para programas gráficos baseados em OpenGL.

- Fornece um conjunto de primitivas para desenho de objetos mais complexos como quádricas e etc.

OpenGL: como utilizar a OpenGL e GLUT em código C/C++

- #include <GL/glut.h>
  - Já inclui automaticamente os headers do OpenGL:
    - #include <GL/gl.h>
    - #include <GL/glu.h>

- Se GLUT não for usado, os headers OpenGL têm que ser incluídos explicitamente, junto com os de outra camada de interface.

- Há APIs para construção de interfaces gráficas (GUI) construídas sobre o GLUT cujos headers incluem os do GLUT.
  - Por exemplo, o pacote GLUI requer:
    - #include <GL/glui.h>
    - (Já inclui glut.h)

OpenGL: como utilizar a OpenGL e GLUT em código C/C++

- #include <GL/glut.h>
  - Já inclui automaticamente os headers do OpenGL:
    - #include <GL/gl.h>
    - #include <GL/glu.h>

- Se GLUT não for usado, os headers OpenGL têm que ser incluídos explicitamente, junto com os de outra camada de interface.

- Há APIs para construção de interfaces gráficas (GUI) construídas sobre o GLUT cujos headers incluem os do GLUT.
  - Por exemplo, o pacote GLUI requer:
    - #include <GL/glui.h>
    - (Já inclui glut.h)

OpenGL: programação por eventos - callbacks

- Callbacks são rotinas que serão chamadas para tratar eventos.
- Para uma rotina callback ser efetivamente chamada ela precisa ser registrada através da função.
  - glutXxxFunc (*callback*)
  - Onde Xxx designa uma classe de eventos e *callback* é o nome da rotina.
- Por exemplo, para registrar uma callback de desenho chamada Desenho, usa-se
  - glutDisplayFunc (Desenho);

OpenGL: programação por eventos – callback de redesenho

- É a rotina chamada automaticamente sempre que a janela ou parte dela precisa ser redesenhada (ex.: janela estava obscurecida por outra que foi fechada)
- Todo programa GLUT precisa ter uma! Exemplo:

```
void display ( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
    glVertex3fv( v[0] );
    glVertex3fv( v[1] );
    glVertex3fv( v[2] );
    glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers(); /* Usamos double-buffering! */
}
```

OpenGL: programação por eventos – callback de redimensionamento

  - glutReshapeFunc (Reshape);
- Chamada sempre que a janela é redimensionada, isto é, teve seu tamanho alterado.
- Tem a forma
  - void reshape (int width, int height){...}
  - width/height são a nova largura/altura da janela (em pixels)

- Obs: Se uma rotina de redimensionamento não for especificada, o GLUT usa uma rotina de redimensionamento "default" que simplesmente ajusta o *viewport* para usar toda a área da janela.

OpenGL: programação por eventos – outras callbacks

- Outras callbacks comumente usadas
  - Eventos de teclado
    - void keyboard(unsigned char key, int x, int y)
  - Eventos de mouse
    - void mouse(int button,int state,int x,int y)
    - void motion(int x, int y)
    - void passiveMotion(int x, int y)
  - Chamada continuamente quando nenhum outro evento ocorre
    - void idle(void)

OpenGL: programação por eventos – outras callbacks

- Inicialização do GLUT
  - glutInit (int* argc, char** argv)
    - Estabelece contato com sistema de janelas.
    - Em X, opções de linha de comando são processadas e removidas.

OpenGL: programação por eventos – inicialização

- Inicialização da(s) janela(s)
  - glutInitDisplayMode (int *modo*)
    - Estabelece o tipo de recursos necessários para as janelas que serão criadas. *Modo* é um "ou" bit-a-bit de constantes:
      - GLUT_RGB cores dos pixels serão expressos em RGB.
      - GLUT_DOUBLE bufferização dupla (ao invés de simples).
      - GLUT_DEPTH buffer de profundidade (z-buffer).
      - GLUT_ACCUM buffer de acumulação.
      - GLUT_ALPHA buffer de cores terá componente alfa.

OpenGL: programação por eventos – inicialização

- Inicialização da(s) janela(s)
  - glutInitDisplayMode (int *modo*)
    - Estabelece o tipo de recursos necessários para as janelas que serão criadas. *Modo* é um "ou" bit-a-bit de constantes:
      - GLUT_RGB cores dos pixels serão expressos em RGB.
      - GLUT_DOUBLE bufferização dupla (ao invés de simples).
      - GLUT_DEPTH buffer de profundidade (z-buffer).
      - GLUT_ACCUM buffer de acumulação.
      - GLUT_ALPHA buffer de cores terá componente alfa.

OpenGL: programação por eventos – inicialização

- glutInitWindowPosition (int x, int y)
  - Estabelece a posição inicial do canto superior esquerdo da janela a ser criada.

- glutInitWindowSize (int width, height)
  - Estabelece o tamanho (em pixels) da janela a ser criada.

OpenGL: programação por eventos – inicialização

- Criação da(s) janela(s)
  - int glutCreateWindow (char* *nome*)
    - Cria uma nova janela primária (*top-level*)
    - Nome é tipicamente usado para rotular a janela
    - O número inteiro retornado é usado pelo GLUT para identificar a janela

OpenGL: programação por eventos – inicialização

- Outras inicializações
  - Após a criação da janela é costumeiro configurar variáveis de estado do OpenGL que não mudarão no decorrer do programa. Por exemplo:
    - Cor do fundo
    - Tipo de sombreamento desejado

OpenGL: programação por eventos – inicialização

- Depois de registradas as callbacks, o controle é entregue ao sistema de janelas:
  - glutMainDisplayLoop (void)

- Esta rotina na verdade é o "despachante" de eventos.

- Ela nunca retorna.

OpenGL: exemplo do livro vermelho

```
void init (void)
{
  /* selecionar cor de fundo (preto) */
  glClearColor (0.0, 0.0, 0.0, 0.0);

  /* inicializar sistema de viz. */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
```

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE
      | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow ("hello");
   init ();
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

OpenGL: primitivas de desenho

- glBegin (PRIMITIVA);
  - *especificação de vértices, cores, coordenadas de textura, propriedades de material*
- glEnd ();

- Entre glBegin() e glEnd() apenas alguns comandos podem ser usados. Ex.:
  - glMaterial
  - glNormal
  - glTexCoord

OpenGL: primitivas de desenho

- glBegin (PRIMITIVA);
  - *especificação de vértices, cores, coordenadas de textura, propriedades de material*
- glEnd ();

- Entre glBegin() e glEnd() apenas alguns comandos podem ser usados. Ex.:
  - glMaterial
  - glNormal
  - glTexCoord

OpenGL: primitivas de desenho

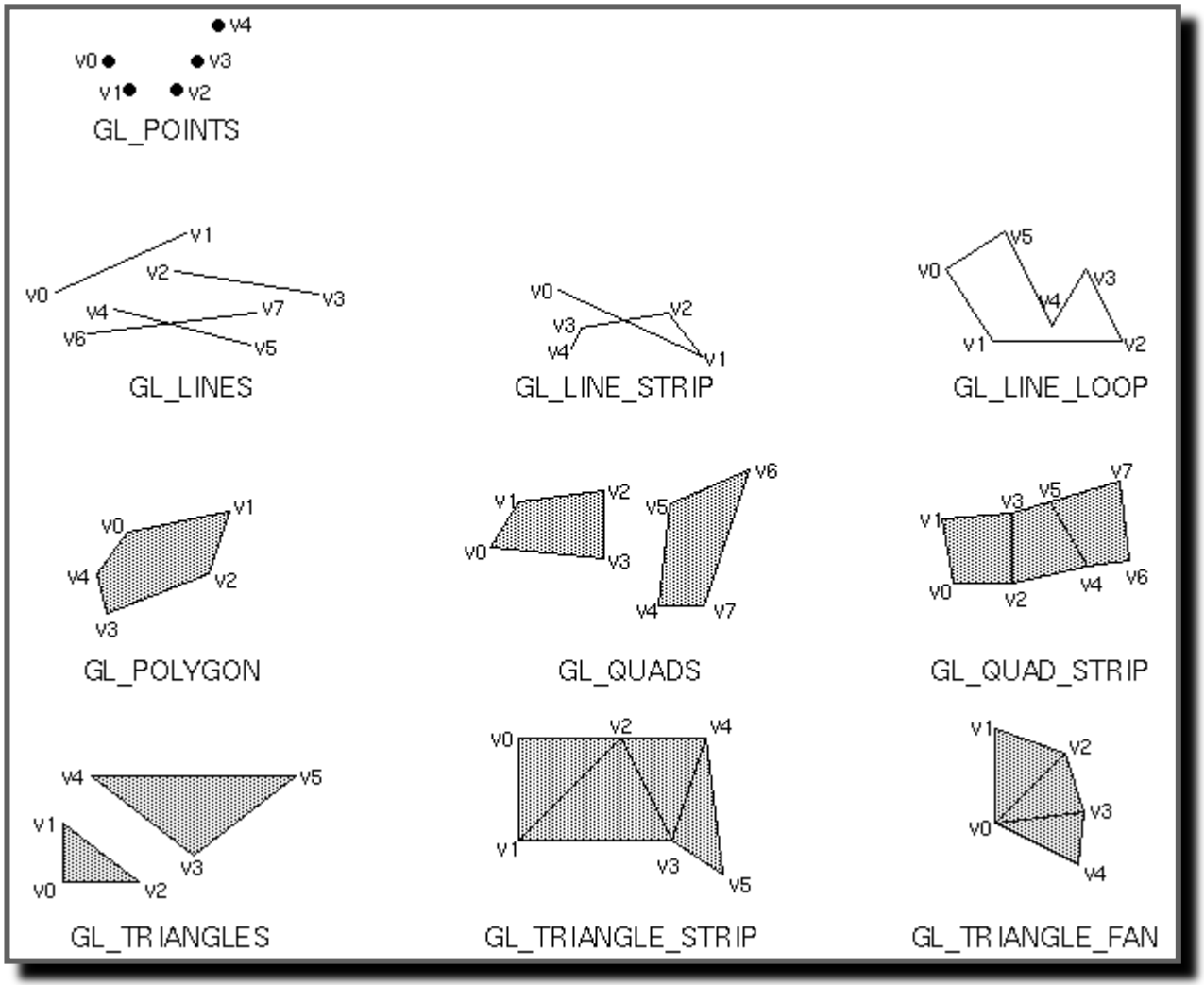
- Uma vez emitido um vértice (glVertex), este é desenhado com as propriedades (cor, material, normal, coordenadas de textura etc) registradas nas variáveis de estado correspondentes.

- Conclusão: Antes de emitir um vértice, assegurar-se que cor, material, normal e etc, têm o valor certo.

OpenGL: primitivas de desenho

Valor	Significado
GL_POINTS	Pontos individuais
GL_LINES	Pares de vértices interpretados como segmentos de reta individuais.
GL_LINE_STRIP	Serie de segmentos de reta conectados.
GL_LINE_LOOP	Igual ao anterior. Ultimo vertice conectado a primeiro
GL_TRIANGLES	Triplas de vértices interpretados como triângulos.
GL_TRIANGLE_STRIP	Cadeia triângulos conectados.
GL_TRIANGLE_FAN	Leque de triângulos conectados.
GL_QUADS	Quadrupla de vértices interpretados como quadriláteros.
GL_QUAD_STRIP	Cadeia de quadriláteros conectados.
GL_POLYGON	Borda de um polígono convexo simples.

OpenGL: primitivas de desenho



OpenGL: programação de hardware gráfico

- Uma característica fundamental do hardware gráfico moderno é sua programabilidade.

- Através da programação customizada é possível produzir efeitos mais sofisticados de iluminação e texturas.

- Com efeito, é possível obter resultados impossíveis de serem atingidos por meio da programação convencional.

OpenGL: programação de hardware gráfico

- As placas gráficas já são programáveis a alguns anos.

- Entretanto, tal processo só tornou-se acessível de fato com as linguagens de programção de tonalização em alto nível ou *shading languages*.

OpenGL: linguagens de tonalização ou shading languages

- Linguagens de tonalização servem para se determinar o aparência em uma superfície ou objeto.

- Podem ser considerados os seguintes fatores:

  - Absorção/difusão da luz
  - Geração de sombras
  - Mapeamento de texturas
  - Efeitos ópticos: reflexão e refração
  - Perturbações na superfície: *bump mapping*
  - Efeitos de pós-processamento

- Programas escritos nestas linguagens são denominados *shaders*.

OpenGL: linguagens de tonalização ou shading languages

- Exemplo mais conhecido de shading language: Renderman (Pixar 1988).
- A Renderman era voltada para geração de imagens com altíssimo nível de realismo.
- Ainda é amplamente utilizada pela indústria do cinema.
- *Não é uma linguagem para geração de imagens em tempo real.*

OpenGL: linguagens de tonalização ou shading languages

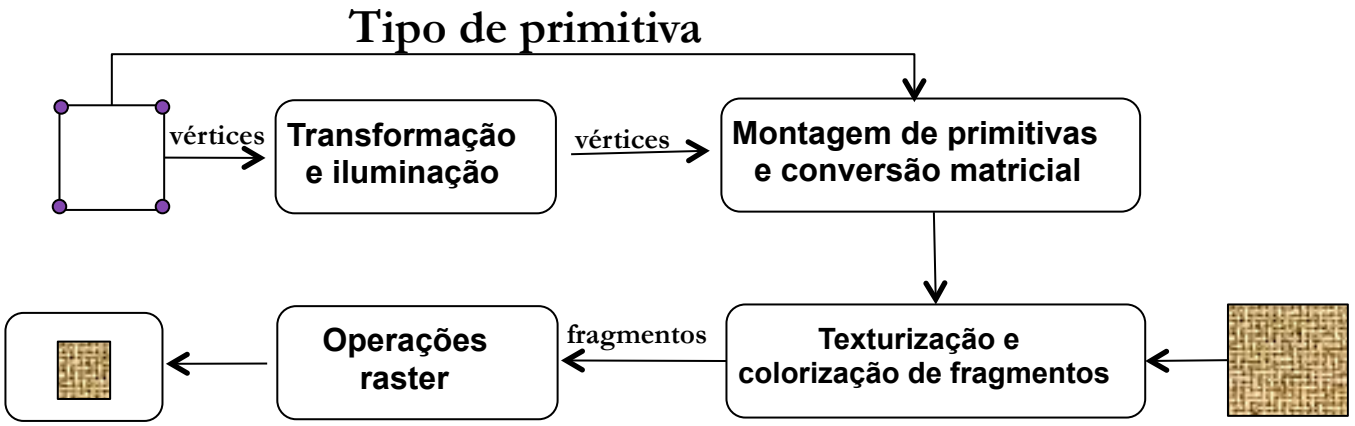
- *Programas escritos em linguagens de tonalização em tempo real precisam executar no hardware gráfico.*

- *Exemplos de shading languages:*
  - Cg – Fernando, 2003 – requer a instalação e configuração de um kit.
  - HLSL – Engel , 2003 – presente somente no ambiente Windows, pois é integrada ao DirectX.
  - OpenGL Shading Language – integrada na API versão 2.0 da biblioteca OpenGL.

OpenGL: pipeline convencional vs pipeline programável

- Para entender o funcionamento dos shaders é necessário primeiramente ter uma visão clara do funcionamento do pipeline de rendering

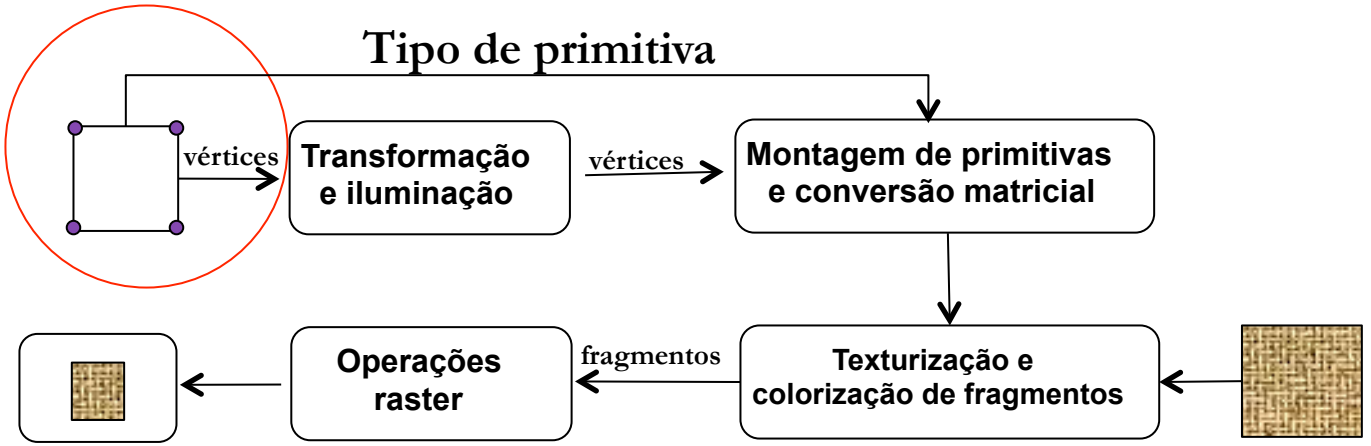
- Abaixo mostramos um versão simplificada do pipeline convencional



OpenGL: pipeline convencional vs pipeline programável

- O diagrama representa o caminho de um único polígono.

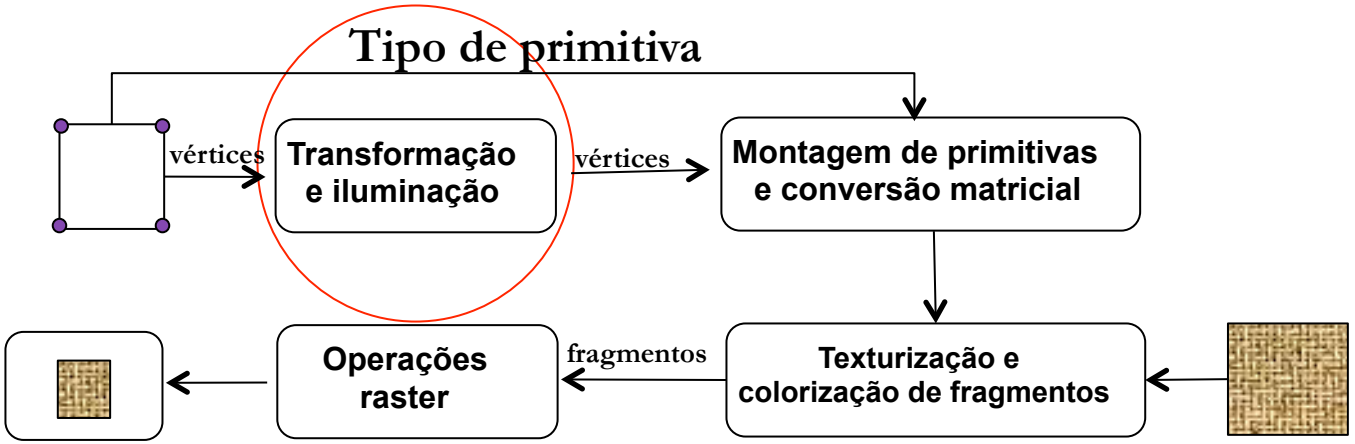
- O programa faz chamadas às funções da OpenGL e com isso um conjunto de informações para os vértices são geradas: coordenadas, coordenadas de textura, vetor normal.



OpenGL: pipeline convencional vs pipeline programável

- Na primeira etapa são aplicadas as transformações geométricas que convertem as coordenadas do objeto para coordenadas da câmera e aplica-se a projeção especificada.

- Isto corresponde a transformação do vértice por meio de sua multiplicação pelas matrizes modelview e projection.

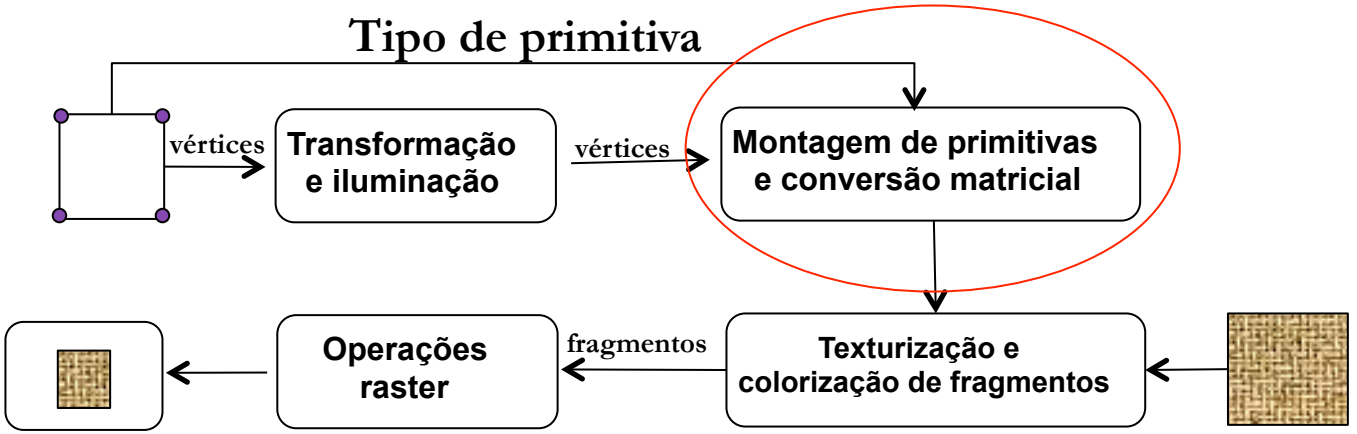


OpenGL: pipeline convencional vs pipeline programável

- Na segunda etapa é montada a primitiva conforme especificada pelo usuário. A etapa anterior não dispões de tal informação.

- Em seguida é feita a conversão matricial, onde são determinadas as coordenadas dos fragmentos que compõem a primitiva.

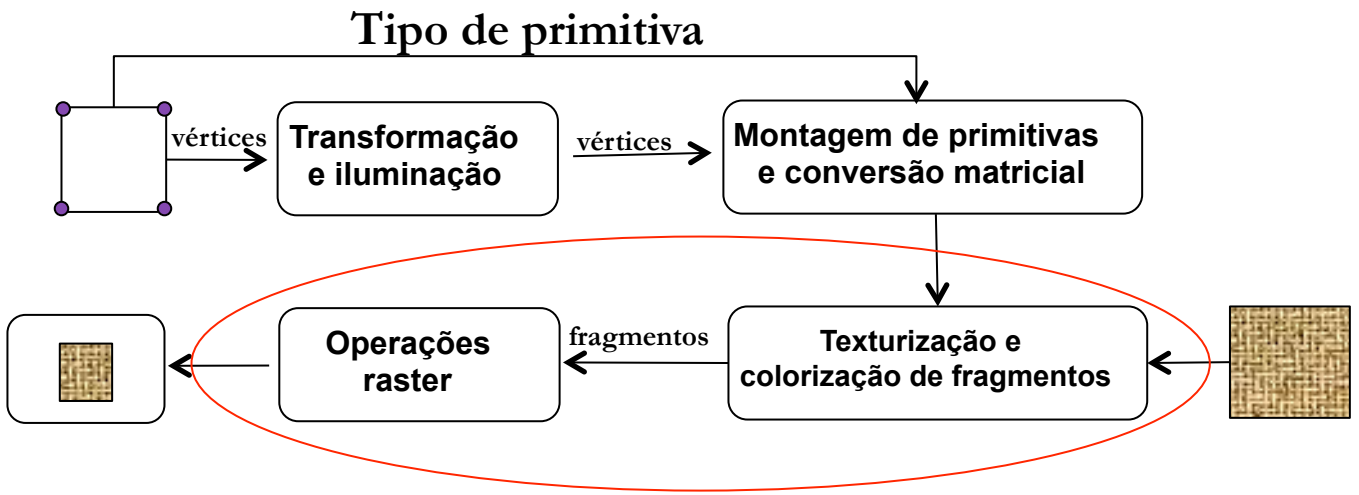
- Nesta etapa são determinados os valores (cor, coordenada de textura, profundidade de cada fragmento através de interpolação).



OpenGL: pipeline convencional vs pipeline programável

- Na terceira etapa as informações calculadas para cada fragmento podem ser modificadas em função do mapeamento de textura e aplicação de neblina (fog).

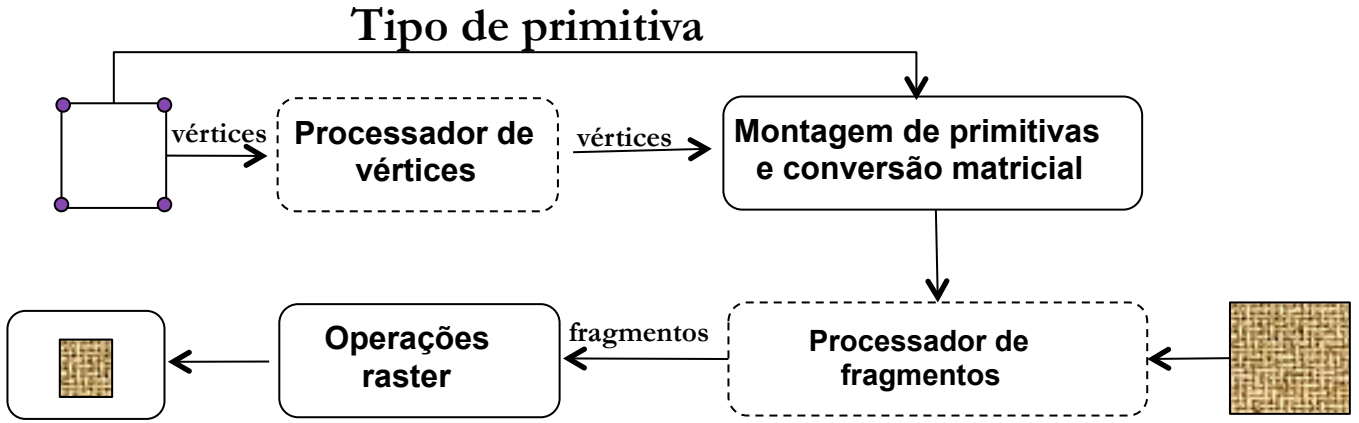
- Finalmente as operações raster determinam se um fragmento efetivamente chegará à tela ou não sendo aplicados testes de stencil, alpha e profundidade.



OpenGL: pipeline convencional vs pipeline programável

- As etapas de transformação + iluminação são substituidas por um processador de vértices que executa programas de vértice (vertex shader).

- As etapas de texturização + colorização de fragmentos são substituidas por um processador de fragmentos que executa programas de fragmento (fragment shader).



OpenGL: pipeline convencional vs pipeline programável

- Os programas de vértice e fragmentos são compostos de instruções de máquina que o processador gráfico consegue executar diretamente.
- Na prática existem várias unidades de hardware para vértices e fragmentos capazes de executar instruções em paralelo eficientemente.
- Por este motiva é comum considerar os modernos processadores gráficos atuais como um certo tipo de máquina SIMD.
- É possível, entretanto implementar tais programas em linguagens de alto nível o que requer a utilização de um compilador capaz de transformar o código fonte em código objeto.

OpenGL: OpenGL Shading Language

- Apresentada primeiramente como uma extensão ARB (Architecture Review Board 2003).
- Tornou-se parte integrante da biblioteca OpenGL a partir da versão 2.0.
- Muito semelhante a linguagem C.
- Apesar de possuir sintaxe semelhante, possui diferenças e restrições consideráveis em relação à linguagem C.
- Tais restrições se devem ao tipo de tarefa a que foi destinada e a limitação dos dispositivos gráficos.

OpenGL: OpenGL Shading Language

- A arquitetura dos dispositivos mais recentes eliminou parte destas restrições juntamente com o surgimento de linguagens apropriadas para tais arquiteturas.

- Exemplos são as arquiteturas/plataformas propostas pela NVidia CUDA – Compute Unified Device Architecture) e ATI-AMD (CAL – Compute Abstraction Layer)

- Nestas arquiteturas é possível desenvolver aplicações de propósito geral sem as idiossincrasias das linguagens de tonalização.

- Entretanto tanto CUDA quanto CAL adotam um modelo de programação completamente distinto do utilizado convencionalmente.

OpenGL: Tipos de dados

- A GLSL suporta tipos de dados vetoriais já que foi criada para geração de efeitos de tonalização e cálculo de illuminação.

- A vantagem em se utilizar tipos vetoriais é a de que o hardware é capaz de realizar operações sobre eles com muito mais eficiência do que sobre tipos escalares.

OpenGL: Tipos de dados

- Tipos fundamentais:

Tipos	Descrição
void	Tipo nulo empregado em funções que não retornam valores
vec2, vec3, vec4	Vetores *float* de 2, 3 e 4 componentes
mat2, mat3, mat4	Matrizes *float* 2x2, 3x3 e 4x4
bool, int, float	Tipos escalares comuns
sampler1D, sampler2D, sampler3D	Utilizados para acessar texturas 1D, 2D, 3D e cubemaps
sampler1Dshadow, sampler2Dshadow	Utilizados para acessar texturas 1D e 2D contendo valores de profundidade para cálculo de sombras.

OpenGL: Tipos de dados

- Os tipos mais comuns são baseados em valores reais.
- Por outro lado, existem variações como inteiros (ivec2, ivec3, ivec4) e booleano (bvec2, bvec3 e bvec4).
- O usuário pode criar vetores de qualquer tamanho como em C com a restrição de que eles sejam unidimensionais.

OpenGL: Tipos de dados

- Não existe tipo char nem strings.
- Não existem ponteiros.
- Não é possível alocar vetores dinâmicamente.

OpenGL: Tipos de dados

- A linguagem suporta estruturas pré-definidas ou definidas pelo usuário
- Exemplo:

```
struct gl_LightSourceParameters {
  vec4 ambient; // ambiente
  vec4 diffuse; // difuso
  vec4 specular; // especular
  vec4 position; // posicao
  vec4   halfVector; // vetor médio
  vec3 spotDirection; // direção se for spot
  float spotExponent; // expoente se for spot
  float spotCutoff; // ângulo de cutoff  se for spot
  float spotCosCutoff; // cosseno do ângulo de cutoff
  float constantAttenuation; // fator de atenuação constante
  float linearAttenuation; // fator de atenuação linear
  float quadraticAttenuation;// fator de atenuação quadrático
};
uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

OpenGL: Tipos de dados

- Em GLSL cada variável possui um qualificador associado que indica como a variável deve ser interpretada:

Qualificador	Descrição
nenhum	Variável local ou parâmetro de entrada para função
const	Constante ou parâmetro de função somente de leitura
attribute	Define uma conexão entre o programa de vértice e a OpenGL para valores que variam por vértice
uniform	Define um valor que não é alterado durante o desenho de uma primitiva
varying	Define uma ligação entre a saída de um program de vértices e o resultado interpolado que serve de entrada para um programa de fragmentos
in	Parâmetro de entrada para uma função
out	Parâmetro de saída não inicializado
inout	Parâmetro de entrada e saída

OpenGL: Tipos de dados

- Variáveis globais (definidas fora de uma função) podem ter qualificadores const, attribute, uniform ou varying.

- Variáveis locais só podem utilizar o quantificador const

- Parâmetros para funções podem usar in, out ou inout.

OpenGL: qualificador attribute

- É empregado somente em programas de vértices.
- Serve para que este possa receber valores que variam a cada vértice.
- Tais valores são enviados diretamente do programa do usuário ou providos pela biblioteca.
- Exemplos:

```
attribute vec4 gl_Color;
attribute vec3 gl_Normal;
attribute vec4 gl_Vertex;
```

OpenGL: qualificador uniform

- Empregado tanto nos programas de vértice quanto nos programas de fragmento.

- Representam variáveis recebidas do programa do usuário ou internas à biblioteca cujo conteúdo não varia durante o desenho de uma primitiva.

- Por exemplo: não é possível realizar transformações geométricas entre um par glBegin ... glEnd e por esse motivo, a variável gl_ModelviewMatrix (que contém a matriz de visualização) é definida como uniform.

```
uniform mat3 gl_ModelviewMatrix;
```

OpenGL: qualificador varying

- Tem como função criar uma ligação entre uma variável no programa de vértice e a mesma no programa de fragmento.
- Variáveis declaradas como varying têm seu conteúdo interpolado pela etapa de conversão matricial.
- Desta forma, o programa de fragmento recebe o valor específico para o fragmento em questão e não a saída do programa de vértice.

OpenGL: operadores especiais

- GLSL suporta a maior parte dos operadores definidos para a linguagem C.

- Existem entretanto exceções e operadores adicionais tais como;

  - operadores de atribuição especiais;
  - operadores de swizzling;
  - operadores vetoriais.

OpenGL: operadores de atribuição

- Considere as seguintes declarações

```
float f;
int i;
vec2 v2;
vec3 v3;
vec4 v4;
mat2 m2
mat3 m3;
```

- São válidas as seguintes atribuições entre escalares

```
i=2;
f=1.5
i = int(f);
f = float(i);
```

OpenGL: operadores de atribuição

- A atribuição de valores a variáveis vetoriais admite diferentes formas:

```
v2 = vec2(1.0,2.0); // inicializa todo o vetor
v2 = vec2(1.0); // ambos componentes recebem o valor 1.0
v3 = vec3(0.0,0.0,0.0);
v3 = vec3(0.0);
v4 = vec4(v2,v2); // combina dois vetores para formar o terceiro
```

- Matrizes são construídas de forma semelhante

```
m3 = mat3(2.0,1.0,0.0,1.0,2.0,0.0,0.0,0.0,1.0);
m3 = mat3(2.0); // inicializa a diagonal com 2.0
m3[2][2] = 1.0;
m2 = mat2(m3); //canto superior esquerdo 2x2 de m3
v3 = vec3(m3);//primeira coluna da matriz
f = float(m3); //primeiro elemento da matriz
```

OpenGL: operadores de atribuição

- A atribuição de valores a variáveis vetoriais admite diferentes formas:

```
v2 = vec2(1.0,2.0); // inicializa todo o vetor
v2 = vec2(1.0); // ambos componentes recebem o valor 1.0
v3 = vec3(0.0,0.0,0.0);
v3 = vec3(0.0);
v4 = vec4(v2,v2); // combina dois vetores para formar o terceiro
```

- Matrizes são construídas de forma semelhante

```
m3 = mat3(2.0,1.0,0.0,1.0,2.0,0.0,0.0,0.0,1.0);
m3 = mat3(2.0); // inicializa a diagonal com 2.0
m3[2][2] = 1.0;
m2 = mat2(m3); //canto superior esquerdo 2x2 de m3
v3 = vec3(m3);//primeira coluna da matriz
f = float(m3); //primeiro elemento da matriz
```

OpenGL: operadores de swizzling

- Permite acessar partes de um vetor como se fosse uma estrutura, mas com a vantagem de ser possível reordenar ou repetir os componentes livremente.

- Pode-se usar os elementos xyzw (coordenadas espaciais), rgba (componentes de cor) e stpq (coordenadas de textura).

- As notações acima existem porque os vetores, em geral, são utilizados para representar posições, direções, cores e coordenadas de textura.

OpenGL: operadores de swizzling

- Exemplos:

```
v3.xy; // apenas x e y do vetor xyz
v3.xxx; // o componente x repetido 3 vezes
v3.rrr; // mesma coisa utilizando notação rgba
v3.sss; // mesma coisa utilizando notação stpq
v2 = v3.xy; // atribui a v2 os componentes xy de v3
v2 = v3.yz; // atribui a v2 os componentes yz de v3
v3 = v4.xxz // atribui (x,x,z) de v4 a v3
```

- Erros:

```
v2.z = 1.0; // v2 não contém o componente z
v3.rgz = vec3(1.0,2.0,3.0); // mistura de notações
v3.xy = 12.0; // deveria atribuir um tipo vec2
```

OpenGL: operadores aritméticas entre dados vetoriais

- Seja as declarações

```
vec2 a,b,c;
mat2 ma, mb, mc;
```

- São válidas as operações:

```
a = b+c; // adição vetorial
a.x = b.x + c.x;
a.y = b.y + c.y; // ineficiente

b = a * ma; // multiplicação de vetor por matriz resultando em vetor
mc = ma * mb // multiplicação matricial
```

OpenGL: exemplo de shader para iluminação por pixel

```
//Vertex shader

varying vec3 N;
varying vec3 v;

void main(void) {
        v = vec3(gl_ModelViewMatrix * gl_Vertex);
        N = normalize(gl_NormalMatrix * gl_Normal);
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
// Fragment Shader
varying vec3 N;
varying vec3 v;

void main(void) {
        vec3 L = normalize(gl_LightSource[0].position.xyz - v);
        vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(N,L), 0.0);
        Idiff = clamp(Idiff, 0.0, 1.0);
        gl_FragColor = Idiff;

}
```

OpenGL: funções para interface com GLSL

- Existem 4 atividades necessárias para utilização da GLSL com o programas do usuário:
  - Compilação e ligação dos shaders em um programa completo.
  - Envio dos valores a serem utilizados pelos shaders (uniform e attribute).
  - Inicialização da GLSL.
  - Utilização do programa.

OpenGL: compilação e ligação

- Em primeiro lugar é necessário criar os objetos que representarão os programas de vértice ou fragmento através da seguinte função:
  - GLuint glCreateShader(GLenum shaderType)
- shaderType pode ser GL_VERTEX_SHADER (programa de vértice) ou GL_FRAGMENT_SHADER (programa de fragmento)

OpenGL: compilação e ligação

- A segunda etapa é enviar o código-fonte GLSL que é armazenado em uma string ou vetor de caracteres para ser compilado:
  - void glShaderSource(GLuint shader, GLsizei count, const GLchar ** string, const GLint *length)
- shader é um identificador válido de programa de vértice ou fragmento;
- count indica quantas linhas de texto deverão ser interpretadas;
- string aponta para uma ou mais linhas de código do programa;
- length é um array que indica o tamanha de cada linha, podendo ser igual a null, o que indica a que as strings são terminadas com o caractere nulo do C.

OpenGL: compilação e ligação

- A terceira etapa consiste em compilar o código utilizando:

  void glCompileShader(GLuint shader)

- É bastante útil utilizar funções que indiquem erros de compilação

- Isto pode ser feito utilizando as funções

  void glGetShaderiv(GLuint shader, GLenum pname, GLint *params);

  void glGetShaderInfoLog(GLuint shader, GLsizei maxLenght,
  GLsizei *length, GLChar * infoLog);

OpenGL: compilação e ligação

- A função void glGetShaderiv(GLuint shader, GLenum pname, GLint *params) obtêm parâmetros relativos a um shader tais como:
  - GL_SHADER_TYPE – (tipo do shader, GL_VERTEX_SHADER ou GL_FRAGMENT_SHADER)
  - GL_DELETE_STATUS – (true se o shader tiver sido marcado para deleção)
  - GL_COMPILE_STATUS – (true se a última compilação tiver tido sucesso)
  - GL_INFO_LOG_LENGTH – (tamanho do log de compilação)
  - GL_SHADER_SOURCE_LENGTH – (tamanho do código fonte)

OpenGL: compilação e ligação

- A função void glGetShaderInfoLog(GLuint shader, GLsizei maxLength, GLsizei *length, GLChar * infoLog) obtém o log de compilação do shader indicado no parâmetro shader.
  - maxLength indica o tamanho máximo do log que pode ser armazenado na string infoLog;
  - length devolve a quantidade de caracteres efetivamente copiados
  - infoLog aponta para a string capaz de armazenar maxLength caracteres

OpenGL: compilação e ligação

- A segunda etapa é enviar o código-fonte GLSL que é armazenado em uma string ou vetor de caracteres para ser compilado:
  - void glShaderSource(GLuint shader, GLsizei count, const GLchar ** string, const GLint *length)
- shader é um identificador válido de programa de vértice ou fragmento;
- count indica quantas linhas de texto deverão ser interpretadas;
- string aponta para uma ou mais linhas de código do programa;
- length é um array que indica o tamanha de cada linha, podendo ser igual a null, o que indica a que as strings são terminadas com o caractere nulo do C.

OpenGL: compilação e ligação

- A última etapa consiste na criação de um programa GLSL que na verdade é formado por um conjunto de programas de vértice ou fragmento integrados.

- Um programa GLSL deve conter pelo menos um programa de vértice ou um programa de fragmento.

- Para criar um programa de vértice, associar shaders a ele e efetuar a ligação utiliza-se as seguintes funções:

```
GLuint glCreateProgram(void);
void glAttachshader(GLuint program, GLuint shader);
void glLinkProgram(GLuint program);
```

OpenGL: compilação e ligação

- É possível checar o status da ligação obtendo o valor do parâmetro GL_LINK_STATUS com a função glGetShaderiv conforme abaixo:

  glGetShaderiv(program, GL_LINK_STATUS, &ligacao_OK)

- ligacao_OK é do tipo GLint e retorna com valor 0 se houver algum problema.

OpenGL: compilação e ligação

- Existem funções para cancelar a associação de shaders a programas GLSL, assim como remover shaders ou programas:

- void glDetachShader(GLuint program, GLuint shader);

- void glDeleteShader(GLuint shader) - libera a area ocupada por um shader; o shader só será removido se não estiver associado a nenhum programa

- void glDeleteProgram(GLuint program) – libera a área de memória ocupada por um programa; se o programa tiver shaders associados, estes serão desconectados mas não removidos a não ser que tenha sido solicitada sua remoção através da função glDeleteShader

OpenGL: envio de valores a GLSL

- Os qualificadores uniform e attribute permitem o envio de valores do programa do usuário ao programa GLSL.
- Para isto, é necessário obter um identificador que relacione uma posição de memória a uma variável especifica.
- Tais identificadores podem ser recuperados pelas seguintes funções:
  - GLint glGetUniformLocation(GLuint program, const GLchar *name);
  - GLint glGetAttributeLocation(GLuint program, const GLchar *name)

OpenGL: envio de valores a GLSL

- GLint glGetUniformLocation(GLuint program, const GLchar *name);
  - Retorna a posição de uma variável uniform com nome indicado pelo parâmetro name no programa GLSL indicado pelo parâmetro program.
- GLint glGetAttributeLocation(GLuint program, const GLchar *name);
  - Retorna a posição de uma variável attribute com nome indicado pelo parâmetro name no programa GLSL indicado pelo parâmetro program.
- Ambas as funções só podem ser chamadas após a ligação do programa através de glLinkProgram.
- Retornam -1 se a variável não existir.

OpenGL: envio de valores a GLSL

- Para enviar os valores em si utiliza-se no caso de uma variável uniform, alguma das variações de glUniform{1234}f[v]:

```
void glUniform1f(GLint location, GLfloat v0);
void glUniform1fv(GLint location, GLsizei count, GLfloat *value);
```

- A primeira função atribui um único valor a uma variável uniform cuja posição é indicada por location obtido anteriormente por glGetUniformLocation

- A segunda função atribui um vetor cujo tamanho é indicado por count.

OpenGL: envio de valores a GLSL

- Para atribuição de valores a variáveis attribute utiliza-se processo similar:

```
void glVertexAttrib1f  (GLuint index, GLfloat v0);
void glVertexAttrib1s (GLuint index, GLshort v0);
void glVertexAttrib1d (GLuint index, GLdouble v0);
void glVertexAttrib1fv (GLuint index, GLsizei count, GLfloat *value);
```

OpenGL: inicialização e uso

- A inicialização é normalmente feita através de uma biblioteca auxiliar como a GLEW já que os headers da biblioteca OpenGL não foram atualizados para a versão 2.0

- Uma vez que o programa GLSL tenha sido carregado, pode-se utilizá-lo através do comando void glUseProgram(GLuint prog);

- Passando-se o valor especial 0 (zero) no parâmetro prog, desativa-se o pipeline programável e retorna-se ao pipeline convencional.

OpenGL: exemplo

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

#include <glew.h>
#include <glut.h>

#include "image.h"
#include "textfile.h"

float   a=0.0,inter=1,textMapping;
GLuint  loc,enableTexture;
GLuint  textureHnd,v,f,f2,p;
tpImage * grassImage;

GLfloat luzAmbiente[4]={0.0,0.0,0.0,1.0};
GLfloat luzDifusa[4]={0.5,0.5,0.5,1.0};
GLfloat luzEspecular[4]={0.8,0.8,0.8,1.0};
GLfloat posicaoLuz[4]={0.0,10.0,10.0,1.0};
GLfloat especularidade[4]={0.5,0.5,0.5,1.0};
GLfloat materialdifuso[4]={0.5,0.5,0.5,1.0};
GLfloat materialespecular[4]={0.8,0.8,0.8,1.0};
GLfloat spotDirection[4] = {0.0,-1.0,-1.0,1.0};

#define printOpenGLError() printOglError(__FILE__, __LINE__)
```

OpenGL: exemplo

```
void printShaderInfoLog(GLuint obj)
{
   int infologLength = 0;
   int charsWritten  = 0;
   char *infoLog;

  glGetShaderiv(obj,
      GL_INFO_LOG_LENGTH,&infologLength);

   if (infologLength > 0)
   {
      infoLog = (char *)malloc(infologLength);
      glGetShaderInfoLog(obj, infologLength,
      &charsWritten, infoLog);
      printf("%s\n",infoLog);
      free(infoLog);
   }
}
```

```
void printProgramInfoLog(GLuint obj)
{
   int infologLength = 0;
   int charsWritten  = 0;
   char *infoLog;

  glGetProgramiv(obj,
      GL_INFO_LOG_LENGTH,&infologLen
      gth);

   if (infologLength > 0)
   {
      infoLog = (char *)malloc(infologLength);
      glGetProgramInfoLog(obj, infologLength,
      &charsWritten, infoLog);
      printf("%s\n",infoLog);
      free(infoLog);
   }
}
```

OpenGL: exemplo

```
void setShaders() {

   char *vs = NULL,*fs = NULL,*fs2 = NULL;
   v = glCreateShader(GL_VERTEX_SHADER);
   f  = glCreateShader(GL_FRAGMENT_SHADER);

   vs = textFileRead("shader1.vert");
   fs = textFileRead("shader1.frag");
   const char * vv = vs;
   const char * ff  = fs;

   glShaderSource(v, 1, &vv,NULL);
   glShaderSource(f, 1, &ff,NULL);
   free(vs);free(fs);

   glCompileShader(v);
   glCompileShader(f);
   printShaderInfoLog(v);
   printShaderInfoLog(f);

   p = glCreateProgram();
   glAttachShader(p,v);
   glAttachShader(p,f);

   glLinkProgram(p);
   printProgramInfoLog(p);
   glUseProgram(p);

   loc = glGetUniformLocation(p,"time");
   enableTexture = glGetUniformLocation(p,"enableTexture");
   glUniform1fARB(glGetUniformLocation(p,"internal"),inter);
   glUniform1iARB(enableTexture,0);
}
```

OpenGL: exemplo

```
void InitShaders() {
  glewInit();

  if (glewIsSupported("GL_VERSION_2_0"))
       printf("Ready for OpenGL 2.0\n");
 else {
      printf("OpenGL 2.0 not supported\n");
      exit(1);
  }
 setShaders();
}

void Init() {
 glClearColor(0.0,0.0,0.0,0.0);
 glEnable(GL_COLOR_MATERIAL);

 glShadeModel(GL_SMOOTH);
 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, luzAmbiente);

 glEnable(GL_DEPTH_TEST);
 glEnable(GL_CULL_FACE);
 glCullFace(GL_BACK);
 glEnable(GL_NORMALIZE);

 grassImage = ReadPpmImage2RGBA("grass.ppm");
 textureHnd = CreateTexture(grassImage,GL_LINEAR,GL_LINEAR);
 DestroyImage(grassImage);
}
```

OpenGL: exemplo

```
void Display(void)
{
 glClear(GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();

 gluLookAt(50.0,50.0,50.0,
 0.0,0.0,0.0,
 0.0f,1.0f,0.0f);

 //glUniform1fARB(loc, a);
 //glColor4f(1.0,1.0,1.0,1.0);

 glBindTexture(GL_TEXTURE_2D,-1);
 glFrontFace(GL_CW);
 glutSolidTeapot(8.0);
 glFrontFace(GL_CCW);

 glPushMatrix();
   glTranslatef(0.0,-20.0,0.0);
   glRotatef(90,1.0,0.0,0.0);
   glutSolidTorus(4.0,16.0,40,80);
 glPopMatrix();

 glUniform1iARB(enableTexture,1);
 glBindTexture(GL_TEXTURE_2D,textureHnd);
 glEnable(GL_TEXTURE_2D);
 glEnable(GL_BLEND);
 glBlendFunc
         (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
 glNormal3f(0.0,1.0,0.0);
         glTexCoord2d(1.0,0.0);
         glVertex3f(200.0,-50.0,200.0);
         glNormal3f(0.0,1.0,0.0);
         glTexCoord2d(1.0,1.0);
    glVertex3f(200.0,-50.0,-200.0);
         glNormal3f(0.0,1.0,0.0);
         glTexCoord2d(0.0,1.0);
         glVertex3f(-200.0,-50.0,-200.0);
         glNormal3f(0.0,1.0,0.0);
         glTexCoord2d(0.0,0.0);
         glVertex3f(-200.0,-50.0,200.0);
 glEnd();
 glDisable(GL_BLEND);
 glDisable(GL_TEXTURE_2D);
 glUniform1iARB(enableTexture,0);

 glEnable(GL_LIGHT0);
 glEnable(GL_LIGHTING);
 glLightfv(GL_LIGHT0,GL_AMBIENT,luzAmbiente);
 glLightfv(GL_LIGHT0,GL_DIFFUSE,luzDifusa);
 glLightfv(GL_LIGHT0,GL_SPECULAR,luzEspecular);
 glLightfv(GL_LIGHT0,GL_POSITION,posicaoLuz);
 glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spotDirection);
 glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,45.0f);
 glLightf(GL_LIGHT0,GL_SPOT_EXPONENT,1.5f);
 glMaterialfv(GL_FRONT, GL_DIFFUSE,materialdifuso);
 glMaterialfv(GL_FRONT, GL_SPECULAR,materialespecular);
 glMaterialf(GL_FRONT, GL_SHININESS,5.0f);


 a += 0.01;
 glutSwapBuffers();
}
```

OpenGL: exemplo

```
void Reshape(int w, int h) {
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(50,1.0,1.0,500.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

int main(int argc, char ** argv)  {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
   glutInitWindowSize(512,512);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Shader 1");
   glutDisplayFunc(Display);
   glutIdleFunc(Display);
   glutReshapeFunc(Reshape);
   Init();
   InitShaders();
  glutMainLoop();
  return 0;
}
```

OpenGL: exemplo

```
//vertex shader
uniform float time;
varying vec3 normal;
varying vec3 sup;
varying float pattern;

void main()
{
    vec4 pos = vec4(gl_Vertex);
    //pos.z = sin(5.0*pos.x+time)*0.25;
    pattern = fract(4.0*(pos.y+pos.x+pos.z));
    sup = gl_ModelViewMatrix * pos;
    normal = gl_NormalMatrix*gl_Normal;
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position  = gl_ModelViewProjectionMatrix * pos;


}
```

OpenGL: exemplo

```
varying vec4 color;
varying vec3 normal;
varying vec3 sup;
varying float pattern;
uniform float interna;
uniform sampler2D tex;
uniform bool enableTexture;

void main(void) {
  vec3 luz   = normalize(gl_LightSource[0].position.xyz-sup);
  vec3 obs   = normalize(-sup);
  vec3 medio = normalize(obs+luz);
  normal     = normalize(normal);
  float spot = max(dot(gl_LightSource[0].spotDirection,-luz),0.0);
  float borda = smoothstep(gl_LightSource[0].spotCosCutoff,1.0,spot);
  spot=pow(spot,gl_LightSource[0].spotExponent)*borda;
  float difuso    = max(dot(normal,luz),0.0);
  float especular = pow(max(dot(normal,medio),0.0),gl_FrontMaterial.shininess);
  float especular2= especular*pattern;
  vec4 color   = gl_FrontLightModelProduct.sceneColor+gl_FrontLightProduct[0].ambient+
              spot*(gl_LightSource[0].diffuse*difuso+gl_LightSource[0].specular*(especular+especular2));
  vec4 texColor = texture2D(tex,gl_TexCoord[0].st);

  if (enableTexture)
    gl_FragColor = vec4(color.rgb*texColor.rgb,color.a*texColor.a);
  else
    gl_FragColor = color;
}
```

OpenGL: exemplo