Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
19/08/2017

OpenGL 3.0

1. Evolução do OpenGL

Versão 1.0 (1992)
Versão 1.1 (1997) – Vertex Array
Versão 1.2 (1998) – Textura 3D
Versão 1.3 (2001) – Multi-textura, etc.
Versão 1.4 (2002) – Mipmap automático, etc.
Versão 1.5 (2003) – Vertex buffer, etc.
Versão 2.0 (2004) – Inclusão da Linguagem de shading
Versão 3.0 (2008) – **112 funções tornaram-se *deprecated*, restando 126 funções.** A programação por shader torna-se obrigatória (http://www.khronos.org/opengl/).
Versão 3.1 (2009) – **As funções deprecadas foram removidas da API.**
Versão 3.2 (2009) – Introduz os contextos *core* e de compatibilidade. Este último para incluir as funções deprecadas, enquanto o contexto *core* trabalha apenas com as novas funcionalidades.
Versão 3.3 (2010) – Lançado simultaneamente com a versão 4.0. Traz recursos avançados para as GPUs antigas. Mas não inclui os novos estágios do *pipeline* programável.
Versão 4.0 (2010) – Novos estágios de pipeline programável. *Tesselation Control* e *Tesselation Evaluation Shader*.
Versão 4.3 (2012) – Inclui o *compute shader*.
Versão 4.5 (2014) – Inclui Direct State Access.

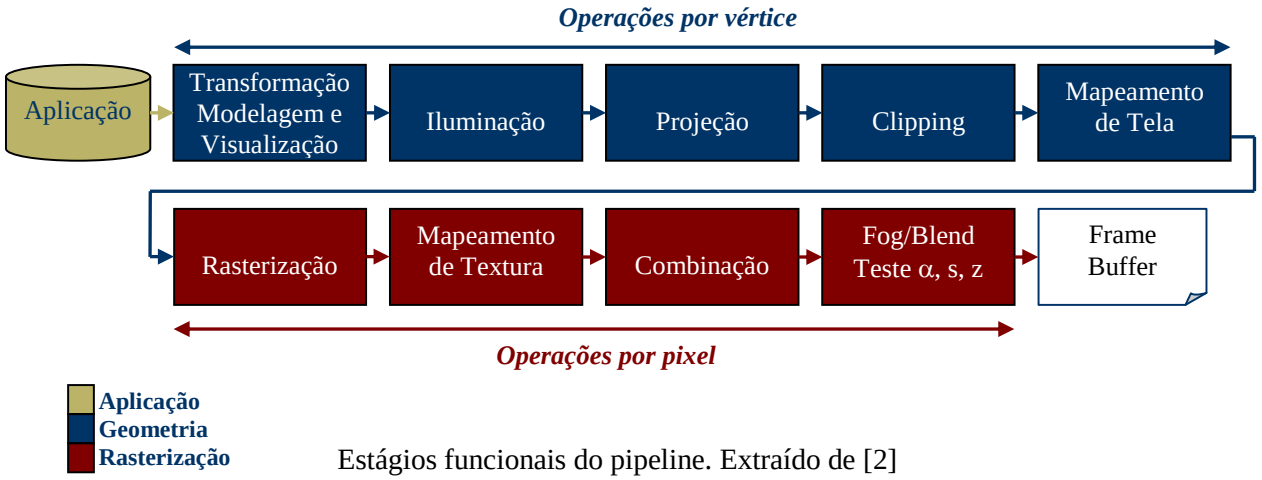
Todas as versões do OpenGL são *backward compatible, até a versão 3.0*.

Uma evolução das placas de vídeo pode ser vista em:
http://www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective

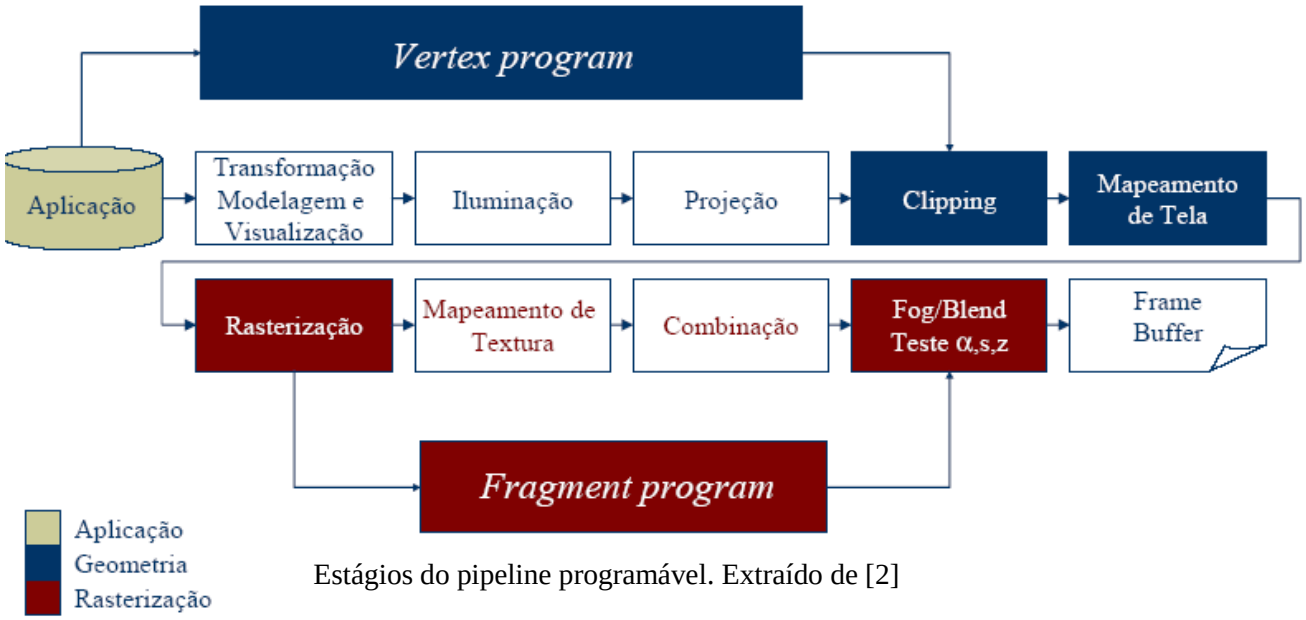
Comparativo

DirectX	**OpenGL**
Windows	Multi-plataforma
Controle centralizado – Microsoft	Consórcio formado por diversas empresas (Khronos)
Evolução refletida em versões	Evolução refletida em extensões

2. Pipeline Programável x Convencional



Estágios funcionais do pipeline. Extraído de [2]



Estágios do pipeline programável. Extraído de [2]

Operações substituídas (não executadas)	**Operações não substituídas**
Transformação de vértices (*modelview matrix*) Projeção (*projetction matrix*) Textura (*texture matrix*) Transformação de normais Transformação, re-escala, normalização Iluminação por vértice Cor do material Geração de coordenadas de textura Texturização Fog	Clipping contra frustum Divisão da perspectiva Mapeamento de tela Transformação de viewport Transformação de Z Clamping de cores [0,1]

3. Vertex Array (Deprecated)

Usado para:

- Reduzir o número de chamada de funções, que podem causar redução de desempenho da aplicação
  - Desenhar um polígono com 20 lados requer 22 chamadas de funções: uma para cada vértice + `glBegin()` + `glEnd()`. Passando-se as normais, pode-se duplicar o número de chamadas.
- Evitar a replicação de vértices, resultando em economia de memória.
  - Para se especificar um cubo, gasta-se 24 vértices, sendo cada vértice replicado 3x. O mesmo cubo poderia ser desenhado com apenas 8 vértices.

Com o uso de *vertex array* (OpenGL 1.1), os 20 vértices do polígono poderiam ser colocados em um vetor e passados em uma única chamada para o OpenGL. O mesmo poderia ocorrer com os vetores normais. Em relação ao número de vértices, pode-se facilmente definir vértices compartilhados por várias faces.

Passos para se utilizar *vertex array*:

- Ativar o vetor de dados que se deseja utilizar. Os mais comuns são de vértices, normais e textura;
- Colocar os dados nos arrays. Estes dados, no modelo **cliente-servidor**, são armazenados no **cliente**;
- Desenhar o objeto com os dados dos arrays. Neste momento, os dados são transferidos para o **servidor**.

Para ativar os buffers de dados, deve-se utilizar o comando `glEnableClientState()`, que recebe como parâmetro constantes associadas a cada vetor:

- `GL_VERTEX_ARRAY`,
- `GL_COLOR_ARRAY`,
- `GL_INDEX_ARRAY`,
- `GL_NORMAL_ARRAY`,
- `GL_TEXTURE_COORD_ARRAY`, e
- `GL_EDGE_FLAG_ARRAY`

```
void glEnableClientState(GLenum array);
void glDisableClientState(GLenum array);
```

```
glEnableClientState (GL_COLOR_ARRAY);
glEnableClientState (GL_VERTEX_ARRAY);
```

Para a definição dos dados, existem comandos específicos para cada um dos 6 tipos de dado. O comando `glVertexPointer()` é usado para indicar o vetor de vértices. O comando `glNormalPointer()` para especificação de normais.

```
void glVertexPointer(GLint size,
                     GLenum type,
                     GLsizei stride,
                     const GLvoid *pointer);

void glNormalPointer(GLenum type,
                     GLsizei stride,
                     const GLvoid *pointer);

void glTextCoordPointer(GLenum type,
                        GLsizei stride,
                        const GLvoid *pointer);

void glColorPointer(GLint size,
                    GLenum type,
                    GLsizei stride,
                    const GLvoid *pointer);
```

No comando `glVertexPointer()`:

- *`pointer`* especifica onde os dados das coordenadas podem ser acessados.

- *type* especifica o tipo de dados: GL_SHORT, GL_INT, GL_FLOAT, ou GL_DOUBLE de cada coordenada do array.
- *size* é o número de coordenadas por vértice (2, 3, ou 4). Não necessária para especificação de normais (sempre é 3). Para cor pode ser 3 ou 4.
- *stride* é o *offset* em bytes entre vértices consecutivos. Deve ser 0 se não existir espaço vago entre dois vértices.

Para enviar os dados para o processamento (enviar para o servidor na arquitetura cliente-servidor do OpenGL), existem vários comandos.

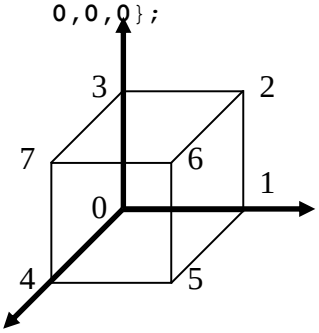
```
void glDrawElements(GLenum  mode,
                    GLsizei count,
                    GLenum  type,
                    void   *indices);
```

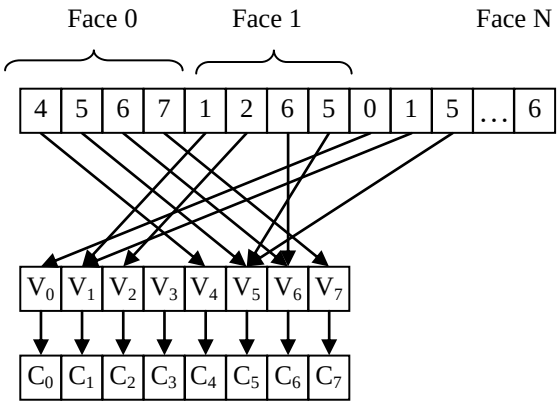
O comando glDrawElements() define uma seqüência de primitivas geométricas onde os parâmetros são:

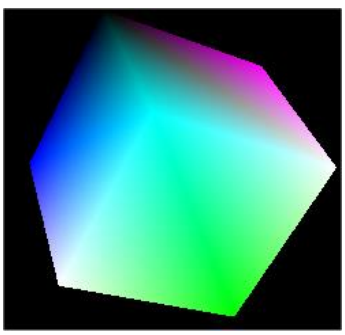
1. *count*: É o úmero de elementos do vetor
2. *indices*: Vetor de índices dos elementos
3. *type*: Indica o tipo de dados do array *indices*. Deve ser GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT ou GL_UNSIGNED_INT.
4. *mode*: indica o tipo de primitivas a serem geradas, como GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, etc.

Exemplo de uso:

```
GLint allVertex[] = { 0,0,0,   1,0,0,   1,1,0,   0,1,0,
                      0,0,1,   1,0,1,   1,1,1,   0,1,1};

//GLint allNormals[8] = {Nesta configuração, define-se uma normal por vértice};

GLfloat allColors[] = { 1,0,0,   0,1,0,   1,1,1,   1,0,1,
                        0,0,1,   1,1,1,   0,1,1,   0,0,0};

GLubyte allIndices[] = {4, 5, 6, 7,
                        1, 2, 6, 5,
                        0, 1, 5, 4,
                        0, 3, 2, 1,
                        0, 4, 7, 3,
                        2, 3, 7, 6};

glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
glVertexPointer(3, GL_INT,   0, allVertex);
glColorPointer( 3, GL_FLOAT, 0, allColors);
//glNormalPointer(GL_FLOAT, 0, allNormals);
glDrawElements(GL_QUADS, 6*4, GL_UNSIGNED_BYTE, allIndices);
```







Para a especificação de um único vértice por vez de um *vertex array* pode-se utilizar o comando `glArrayElement(GLint)`. O seguinte trecho de código tem o mesmo efeito que o comando `glDrawElements()` apresentado anteriormente.

```
glBegin (GL_QUADS);
   for (i = 0; i < 24; i++)
      glArrayElement(allIndices[i]);
glEnd();
```

Observações:

- Existe um **mapeamento direto de 1 para 1** entre o vetor de vértices com os vetores de normais, cores, etc.
- O vetor de índices é `GLubyte` pois como existem apenas 24 = 6*4 índices, usar um tipo maior como `GLuint` seria um desperdício de memória.
- Se um vértice compartilhado tiver **normais diferentes** em cada face, deve-se replicar os vértices e as normais (o mesmo vale para as cores). Neste caso, perde-se a otimização de memória mas continua-se ganhando com a redução de chamadas em relação a definição do objeto utilizando-se os comandos convencionais como `glVertex*()` e `glNormal*()`. Para o caso do cubo, seriam necessários 24 vértices (4 por face x 6 faces), 24 normais e 24 cores. [http://www.songho.ca/opengl/gl_vertexarray.html]
- Este recurso do GL, quando usado com normais, é melhor aplicado em superfícies planares, como terrenos. Para objetos volumétricos fechados como cubos, não faz sentido ter-se normais compartilhadas entre as 3 faces adjacentes, o que exige a replicação de vértices. O mesmo se aplica para cilindros.

4. Display List (Deprecated)

É um grupo de comandos OpenGL que são armazenados para execução futura. Quando um *display list* é chamado, os comandos são executados na ordem que foram definidos. Nem todos os comandos OpenGL podem ser armazenados em um *display list*. Não podem ser utilizados especialmente comandos relacionados com *vertex array*, comandos como `glFlush()`, `glFinish()`, `glGenLists()`, dentre outros.

Deve ser utilizada sempre que for necessário desenhar o mesmo objeto mais de uma vez. Neste caso, deve-se guardar os comandos que definem o objeto e então exibir a *display list* toda vez que o objeto precisar ser desenhado. No caso de um carro que possui 4 rodas, pode-se gerar um *display list* com a geometria de uma roda e invocar a renderização, uma vez para cada roda, lembrando de aplicar as devidas transformações para posicionar cada roda no devido lugar.

Os dados de um *display list* residem no **servidor**, reduzindo deste modo o trafego de informações. Dependendo da arquitetura do servidor, o *display list* pode ser armazenado em memória especial para agilizar o processamento.

Para criar um *display list* deve-se utilizar os seguintes comandos: `glGenLists()`, `glNewList()` e `glEndList()` como mostrado no seguinte exemplo:

```
GLuint id_objeto;

void init()
{
   id_objeto = glGenLists(1);
   glNewList(id_objeto, GL_COMPILE);
   //comandos de desenho do objeto
   //vértices, transformações, etc.
   glEndList();
}

void display()
{
   ...
```

```
    glRotated(10, 1, 0, 0);
    glCallList(id_objeto);

    glTranslated(10, 31, 0);
    glCallList(id_objeto);
    ...
    glSwapBuffers();
}
```

- **GLuint glGenLists**(**GLsizei** range): Aloca um intervalo contínuo de índices. Se o parâmetro *range* for igual a 1, gera um único índice que é o valor retornado pela função. O índice retornado é usado pelas funções `glNewList()` e `glCallList()`. O retorno zero significa erro.
- **void glNewList** (**GLuint** list, **GLenum** mode): Especifica o início do display list. Todos os comandos válidos especificados antes do comando `glEndList()` são guardados no *display list*.
  - **List**: identificador do *display list*.
  - **Mode**: aceita dois argumentos: `GL_COMPILE_AND_EXECUTE` e `GL_COMPILE` (indica que os comandos **não** devem ser executados à medida que são inseridos no *display list*).
- **void glEndList** (**void**): indica o fim do *display list*.
- **void glCallList** (**GLuint** list): Executa o *display list* referenciado por *list*. Os comandos são executados na ordem que foram especificados, do mesmo modo caso não fosse utilizado este recurso.

Além de reduzir o número de chamada de funções, bem como a transferência de dados para o servidor, reduz-se também processamento para geração dos objetos. Supondo que fosse necessário gerar um círculo (ou uma esfera) por coordenadas polares, diversos cálculos de ângulos podem ser realizados uma única vez, visto que apenas comandos OpenGL são armazenados na *display list* para processamento futuro.

- Entretanto, uma vez gerada o *display list*, os valores das coordenadas do círculo não podem mais ser alterados.
- Comandos GLUT também pode ser inseridos, uma vez que são desmembrados em comandos básicos do OpenGL.
- Pode-se criar *display list* hierárquicos, onde um *display list* invoca outros.

```
    ...
    id_objeto = glGenLists(1);
    glNewList(id_objeto, GL_COMPILE);
    glBegin(GL_POLYGON);
    for(int i=0; i<1000; i++)
    {
       x = cos(ang)*radius;
       y = sin(ang)*radius;
       ang+=(360.0/1000);
       glVertex2f(x,y);
    }
    glEnd();
    glEndList();
    ...
```

Para maiores detalhes dos benefícios do uso de *display list* e recursos adicionais, consulte [1].

5. Vertex Buffer Object e Vertex Array Object

Por Cícero Pahins e Guilherme Schardong, 2013

Buffer Objects são espaços de memória alocados no contexto do **OpenGL** que servem para armazenar dados. Vertex Buffer Objects (VBO) são usados para armazenar dados de vértices (coordenadas, cores, normais e outros dados relevantes), os quais são necessários para desenhar geometria na tela. A maior vantagem de usar VBOs é que os dados ficam armazenados na GPU, o que evita que o programador tenha que enviá-los para a GPU a cada ciclo de renderização.

Os VBOs foram concebidos para aliar as funcionalidades dos Vertex Arrays e Display Lists, enquanto evitam suas desvantagens:

- Os Vertex Arrays reduzem o número de chamadas de funções do OpenGL e proporciona o uso redundante de vértices compartilhados. No entanto, a desvantagem dessa técnica é que as funções do Vertex Array estão no cliente e os dados devem ser reenviados para o servidor a cada vez que o array for referenciado, diminuindo o desempenho da aplicação como um todo.

- A Display List é uma função do servidor, logo os dados não precisam ser reenviados a cada renderização. O problema surge quando há a necessidade de alterar os dados já enviados, uma vez que uma Display List compilada não admite alterações, causando a necessidade de recriá-la para cada alteração a ser executada.

Os VBOs permitem que o usuário defina como os dados serão armazenados na memória da GPU, ou seja, permite que, por exemplo, haja vários buffers, cada um contendo um dado relevante para a renderização, ou que tudo seja armazenado em um buffer apenas. Deve-se notar que a GPU não sabe como estes dados estão organizados, para isso há a necessidade de uma estrutura que descreva como interpretá-los para a correta renderização da geometria.

Abaixo iremos construir um exemplo utilizando VBO. Note que o código está adequado ao OpenGL 3.2+.

A partir do OpenGL 3.2 foi inserido o conceito de ***profile*** na criação de contextos, sendo eles o ***core*** e o ***compatibility***. Enquanto que o primeiro expõem as últimas e mais modernas funcionalidades da API, tornando a utilização de *shaders* uma exigência, o segundo inclui as funções relativas ao pipeline fixo e depreciadas, ou seja, aquelas que possuem seu uso desaconselhado.

A escolha do *profile* a ser utilizado é feita no momento da criação do contexto OpenGL, desta maneira, depende diretamente do sistema operacional ou APIs utilizadas para este fim.

O código abaixo é utilizado na API GLFW para que o contexto OpenGL suporte ao mínimo a versão 3.2 da API e exclusivamente as funcionalidades definidas no *core (*a criação de um contexto não suportado pelo driver de vídeo resultará em erro*)*:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

O uso das funcionalidades (extensões) mais recentes do OpenGL é facilitada pelo uso da API GLEW. Desta maneira, procure sempre fazer a linkagem dela em seu projeto. Veja mais em [1].

Primeiramente precisamos definir a lógica da estrutura de dados que será armazenada na GPU. Como dito mais acima, há duas opções: utilizar vários buffers, cada um contendo um tipo de dado, ou, somente um buffer para todos os dados. A escolha depende diretamente de como o programador armazena, cria ou importa os seus dados, uma vez que diferentes API's para importação de modelos 3D oferecem diferentes interfaces. Outro fator a ser considerado é o tamanho do VBO resultante [2]:

- Pequeno
  - Mais facilmente gerenciado, uma vez que cada objeto da cena pode ser representado por um único VBO.
  - Se o VBO tiver poucos dados o ganho de performance em relação a outras técnicas pode não ser aparente. Muito tempo irá ser perdido na troca de buffers.
- Grande
  - A renderização de vários objetos pode utilizar somente uma chamada de função, melhorado a performance.
  - Caso o gerenciamento separado de objetos seja muito complicado, isto irá gerar um overhead desnecessário.
  - Se os dados do VBO forem muito grandes é possível que não possam ser movidos para a VRAM (Video RAM)

A formatação dos dados (ordem de armazenamento) também pode influenciar no desempenho final, a wiki oficial do OpenGL fornece informações das melhores práticas em [3].

Para este exemplo iremos armazenar três informações a respeito do vértice: posição (3 floats), normal (3 floats) e coordenada de textura (2 floats), como demonstra a Figura 1.

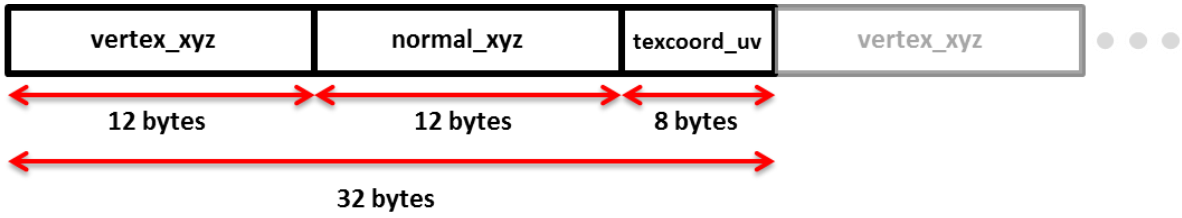


Figura 1. Formatação de dados utilizada no exemplo de VBO.

Note que o tamanho total de cada informação (posição + normal + coordenada de textura) é de **32 bytes**, ou seja, se alocarmos **n** informações a respeito do vértice (n * sizeof(informação)) encontraremos a primeira tupla nos bytes 0 à 31, a segunda nos bytes 32 à 63, e assim por diante. Este modo de armazenamento é dito *intercalado*, e pode ser representado pela seguinte sequência de caracteres:

(VVVNNNTT) (VVVNNNTT) (VVVNNNTT)...,

onde V = vertex, N = normal e T = texture coordinate.

Observe no Código 1 a etapa de declaração de variáveis e definição da estrutura de dados e ordem de armazenamento.

```
GLuint id_vbuffer = 0;
GLuint id_ibuffer = 0;

struct Vertex {
  GLfloat vertex_xyz[3];
  GLfloat normal_xyz[3];
  GLfloat texcoord_uv[2];
};
```

```
Vertex vertexdata[NUM_VERTS] = { ... };
GLuint indexdata[NUM_INDICES] = { 0, 1, 2, ... };
```

Código 1. Exemplo VBO. Etapa de declaração de variáveis e definição da estrutra de dados.

Semelhante a utilização de Vertex Array, é criado um vetor de índices (*indexdata*) que irá coordenar a ordem de acesso ao vetor de dados (*vertexdata*).
As variáveis *id_vbuffer* e *id_ibuffer* serão utilizadas para armazenar os ids dos VBOs a serem criados, neste exemplo um para dados e outro para índices, e posteriormente utilizadas para referenciá-los. O OpenGL não permite a mistura de dados de vértices aos de índices, por isso a necessidade da criação de dois buffers.

A **criação** de uma VBO requer 3 passos:

1. Geração de um novo buffer com *glGenBuffers()*

```
void glGenBuffers(GLsizei n,  GLuint * buffers);
```

Parâmetros:

- *n*: número de buffers a serem criados.
- *buffers*: endereço de uma variável ou array do tipo GLuint para armazenamento do id de cada buffer.

2. Vinculação do buffer criado ao OpenGL com *glBindBuffer()*

```
void glBindBuffer(GLenum target,  GLuint buffer);
```

Parâmetros:

- *target*: especifica o tipo de dado a ser armazenado, pode ser:
  - *dado de vértice*: GL_ARRAY_BUFFER
  - *dado de índice*: GL_ELEMENT_ARRAY_BUFFER
- *buffer*: id do buffer.

3. Cópia dos dados com *glBufferData()*

```
void glBufferData(GLenum target,  GLsizeiptr size,  const
GLvoid * data,  GLenum usage);
```

Parâmetros:

- *target*: deve ser o mesmo utilizado em *glBindBuffer()*.
- *size*: tamanho em bytes do buffer a ser criado (corresponde ao tamanho total em bytes dos dados).
- *data*: ponteiro para o array de dados a ser copiado. Caso NULL, o VBO irá somente reservar o espaço de memória.
- *usage*: especifica o padrão de uso esperado do armazenamento de dados, pode ser qualquer combinação entre os grupos abaixo (9 possibilidades):

  - Grupo A (“o conteúdo do armazenamento será:”)
    - STATIC: especificado uma vez e utilizado muitas vezes
    - STREAM: especificado e utilizado repetidamente

- DYNAMIC: especificado uma vez e utilizado uma vez

- Grupo B (“os dados serão movidos do(a):”)
  - DRAW: aplicação para OpenGL
  - READ: OpenGL para aplicação
  - COPY: OpenGL para OpenGL (DRAW + READ)

A fim de descobrir o melhor local para armazenar os dados dos VBOs criados a implementação do OpenGL (driver de vídeo) utilizará o parâmetro *usage* da função *glBufferData()* como “dica” e, por isso, ele é muito importante na busca por maior desempenho.
Observe atentamente o Código 2 e faça a correlação dos parâmetros de função e a estrutura de dados utilizados.

```
// Criação do VBO de dados de vértices
glGenBuffers(1, &id_vbuffer);
glBindBuffer(GL_ARRAY_BUFFER, id_vbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * NUM_VERTS, vertexdata,
GL_STATIC_DRAW); // transferência do dados de vértices

// Criação do VBO de dados de índices
glGenBuffers(1, &id_ibuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, id_ibuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * NUM_INDICES, indexdata,
GL_STATIC_DRAW); // transferência do dados de índices
```

Código 2. Exemplo VBO. Etapa de transferência de dados para GPU.

Criados os VBOs e transferidos os dados, resta a etapa de **renderização**, a qual requer 3 passos (semelhante ao utilizado em Vertex Array):

1. Vinculação do buffer a ser utilizado pelo OpenGL com g*lBindBuffer().*
2. Ativação e definição dos atributos dos vértices

```
void glEnableVertexAttribArray(GLuint index);
```

```
void glVertexAttribPointer(GLuint index,  GLint size,  GLenum type,
GLboolean normalized,  GLsizei stride,  const GLvoid * pointer);
```

Parâmetros:

- *index*: índice do atributo do vértice (correspondente ao vertex shader)
- *size*: especifica o número de componentes do atributo, podendo ser 1, 2, 3 ou 4
- *type*: especifica o tipo de dado de cada componente do array
- *stride*: especifica o *offset* (intercalação) entre atributos consecutivos
- *pointer*: especifica o ponto de início (**definido como um *offset*, e não como um ponteiro**) para o primeiro atributo do tipo definido no array

3. Chamada da função de desenho com *glDrawElements()* ou outra.

Um atributo do vértice (posição, índice, normal, coordenada de textura, etc.) é habilitado e definido com as funções *glEnableVertexAttribArray()* e *glVertexAttribPointer()*, respectivamente. Através da função *glVertexAttribPointer()* a formatação dos dados é enviada ao OpenGL. Note que no Código 3 existem três chamadas a essa função, cada uma definindo a estrutura de dados utilizada neste exemplo. **Observe estas funções e compare-as com a Figura 1.**

```
// Bind dos VBOs a serem utilizados
glBindBuffer(GL_ARRAY_BUFFER, id_vbuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, id_ibuffer);
```

```
// Ativação e definição dos atributos dos vértices
glEnableVertexAttribArray(0); // texcoord_uv
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, (void*)(sizeof(GLfloat) * 6));

glEnableVertexAttribArray(1); // normal_xyz
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, (void*)(sizeof(GLfloat) * 3));

glEnableVertexAttribArray(2); // vertex_xyz
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, (void*)(0));

// Chamada da função de desenho
glDrawElements(GL_TRIANGLES, NUM_INDICES, GL_UNSIGNED_INT, (void*)0);
```

Código 3. Exemplo VBO. Etapa de renderização. Informações referentes à Figura 1 estão sublinhadas.

Note que o índice utilizado para a habilitação e definição de um determinado atributo através das funções *glEnableVertexAttribArray()* e *glVertexAttribPointer()* deve ser o mesmo, no exemplo é utilizado o *índice 0* para as coordenadas de textura, o *índice 1* para as normais e o *índice 2* para os vértices. (Quando utilizado *shader* os índices devem ser correspondentes).

Uma boa prática de programação a ser seguida é sempre retornar ao estado do OpenGL antes da ativação de atributos, buffers, flags ou outros, procurando desabilitar recursos utilizados pontualmente, como no caso da criação ou renderização de VBOs.

O Código 4 ilustra a como desabilitar as funcionalidades utilizadas na etapa de renderização do VBO (Código 3) e deve sempre ser seguida para evitar problemas desnecessários em aplicações maiores ou mais complexas.

```
glDisableVertexAttribArray(2);
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

Código 4. Boa prática de programação em OpenGL.

A **modificação** de um VBO pode ser dada utilizando a função *glBufferSubData()*, caso em que parte dos dados são reenviados para o buffer (note que é preciso ter cópias válidas dos dados no cliente e no servidor), ou utilizando a função *glMapBuffer()*, a qual mapeia o buffer para o espaço de memória do cliente. O uso da segunda opção é preferível por não exigir cópia dos dados no cliente e servidor, mas há situações que o uso da primeira opção pode trazer maior desempenho [4].

No Código 5 é exibido o uso da função *glMapBuffer()*, a qual retorna um ponteiro que permite o programador modificar os dados do VBO. São parâmetros o tipo de buffer e de acesso, nesta ordem.

```
glBindBuffer(GL_ARRAY_BUFFER, id_vbuffer);
Vertex* ptr = (Vextex*)glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);

if (ptr)
{
    updateMyVBO(ptr, ...);
    glUnmapBuffer(GL_ARRAY_BUFFER);
}
```

Código 5. Exemplo VBO. Mapeamendo do VBO para o espaço de memória do cliente.

Vertex Array Objects (VAOs) são objetos OpenGL que **encapsulam** todos os estados necessários para a definição de dados de vértices. Eles definem o formato desses dados bem como seus arrays fonte [5], ou seja, descrevem como estão armazenados os atributos de vértices.

Note que o VAO não contém os dados dos vértices, estes estão armazenados em **Vertex Buffer Objets**, mas somente referencia aqueles já existentes no buffer.

Para usar um VAO ele deve ser criado e definido como ativo (semelhante ao VBO). Feito isso, os atributos para os VBOs devem ser habilitados e passados para que o VAO saiba onde estão os dados e como eles estão armazenados. De maneira simplificada, um VAO registra os estados (flags) necessárias para a "configuração do VBO".

No Código 6 é exibido o *uso de um VAO em conjunto com o exemplo de VBO anterior*. Note que toda a etapa de renderização do VBO (Código 3) é registrada na etapa de configuração do VAO, com exceção

da chamada a função de desenho (*glDrawElements()*). No Código 7 a etapa de renderização é realizada. **Compare com a mesma etapa do VBO**.

```
// Bind do VAO
GLuint id_vao;
glGenVertexArrays(1, &id_vao);
glBindVertexArray(id_vao); // início do registro dos estados pelo VAO

// Bind dos VBOs a serem utilizados
glBindBuffer(GL_ARRAY_BUFFER, id_vbuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, id_ibuffer);

// Ativação e definição dos atributos dos vértices
glEnableVertexAttribArray(0); // TexCoordPointer
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8,
(void*)(sizeof(GLfloat) * 6));

glEnableVertexAttribArray(1); // NormalPointer
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8,
(void*)(sizeof(GLfloat) * 3));

glEnableVertexAttribArray(2); // VertexPointer
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 8, (void*)(0));

// Chamada da função de desenho NÃO DEVE SER CHAMADA
glDrawElements(GL_TRIANGLES, NUM_INDICES, GL_UNSIGNED_INT, (void*)0);

glBindVertexArray(0); // fim do registro dos estados pelo VAO
```

Código 6. Exemplo VAO. Etapa de configuração.

```
glBindVertexArray(id_vao);
glDrawElements(GL_TRIANGLES, NUM_INDICES, GL_UNSIGNED_INT, (void*)0);
glBindVertexArray(0);
```

Código 7. Exemplo VAO. Etapa de renderização.

Um VAO possui um determinado número de atributos que devem ser associados e habilitados para que possam ser usados pelo OpenGL. Alguns exemplos de atributos são: dados de vértices, índices, vetores normais e cores. A tentativa de renderização de um VAO "vazio" irá resultar em erro.
Para consultar os protótipos de função do OpenGL veja [6].

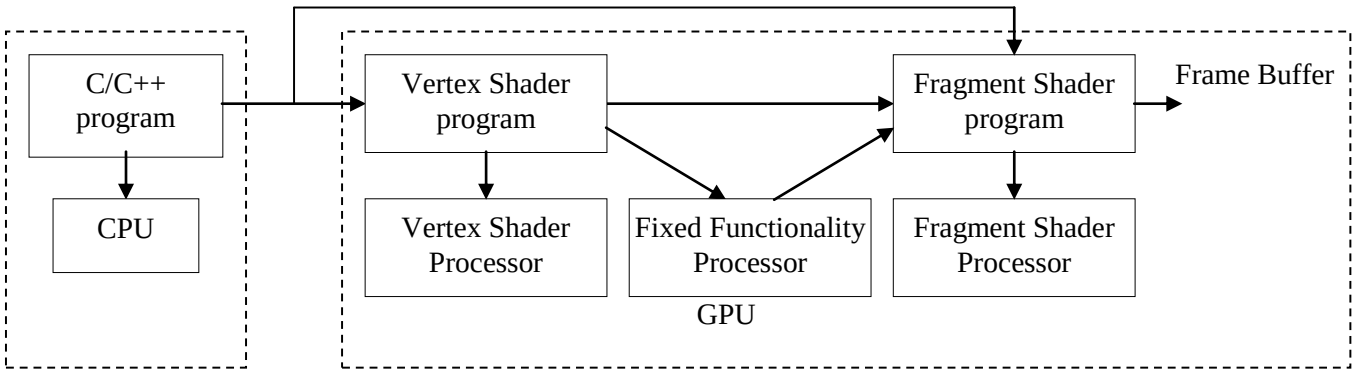
6. Arquitetura do OpenGL Shading Language

6.1. Vertex Processor

- Transformações de vértices
- Transformações de normais
- Geração de coordenadas de textura
- Transformação de coordenadas de textura
- Iluminação

6.2. Fragment Processor

- Definição da cor dos fragmentos
- Definição do z dos fragmentos



Obs:

- Atualmente, o Vertex Processor e o Fragment Processor das placas gráficas estão sendo substituídos por **Stream Processor**, que são processadores genéricos na placa de vídeo que podem desempenhar qualquer das duas tarefas. Cada Stream Processor opera paralelamente aos demais (**SIMD** – *Single Instruction Multiple Data*), dando a placa gráfica características de processamento vetorial.
- Além de Vertex e Fragment (pixel) shaders, com o surgimento do Directx 10 surgiu o **Geometry Shader**. Ele permite criar primitivas gráficas na placa de vídeo. Executa após o Vertex Shader. Atualmente o Geometry Shader é suportando no OpenGL por meio de extensões.

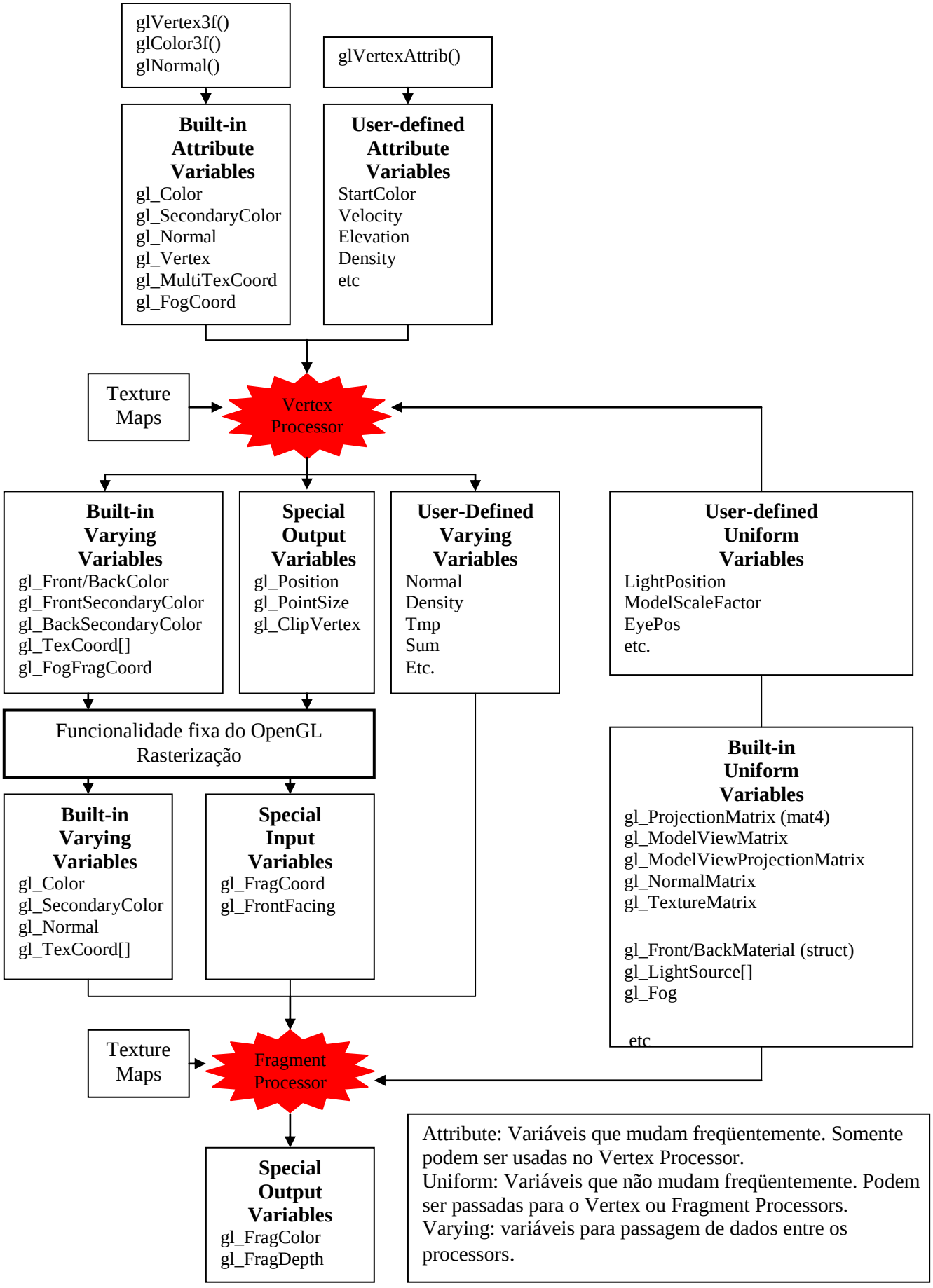
As seguintes tabelas apresentam um quadro comparativo entre o número de unidades vertex/pixel/stream em alguns modelos de placa de vídeo da ATI/Nvidia. Deve-se observar que somente o número de unidades não é fator determinante de desempenho. Deve-se também observar o clock da GPU e velocidade e 13ransferência da memória de vídeo.

ATI Radeon	Vertex Shaders Processors	Pixel Shaders Processors	Stream Processing units
X1050/1300	2	4	
X1600	5	12	
X1950	8	48	-
HD 2400	-	-	40
HD 2600	-	-	120
HD 2900	-	-	320
HD 3870 X2	-	-	320 x 2
HD 4870 X2	-	-	800 x 2
HD 5870			1600
HD 5970			3200
HD 7970			2048

Tabela comparativa dos processadores Nvidia.

nVidia GeForce	Vertex Shaders Processors	Pixel Shaders Processors	Stream Processing units	Cuda Cores
6200	3	4		
7600	5	12		
7950 GX2	8 x 2	24 x 2		
8500	-	-	16	
8600	-	-	32	
8800 GTS	-	-	96	
8800 Ultra	-	-	128	
9800 GX2			128 x 2	
285 GTX	-	-	240	
295 GTX	-	-	240 x 2	
480 GTX	-	-	-	480
580 GTX	-	-	-	512
680 GTX	-	-	-	1536
690 GTX	-	-	-	1536 x 2

O seguinte diagrama exemplifica em detalhes o relacionamento entre os processadores e a aplicação. Também ilustra todas as variáveis que podem ser usadas ao se escrever um shader.



Entrada e saída do Fragment e Vertex Processors. Adaptada de [1].

O quadro anterior ilustra as variáveis das versões antigas do OpenGL (pré OpenGL 3.3). O quadro a seguir serve pra ilustrar o novo estado das variáveis nas versões 3.3 e mais novas do OpenGL.

Vale notar que foram alterados os qualificadores das variáveis (*Attribute*, *Uniform* e *Varying*) nas versões novas do OpenGL (>=3.3). Os qualificadores *Attribute* e *Varying* foram removidos da linguagem GLSL, restando apenas o qualificador *Uniform*. Os qualificadores *in* e *out* foram incluídos para indicarem variáveis de entrada e saída dos shaders. As informações do quadro a seguir podem ser encontradas em [8].

7. A Linguagem GLSL (OpenGL Shading Language)

7.1. Tipos

A linguagem GLSL é muito semelhante a linguagem C/C++, com algumas características removidas e outras adicionadas. Além do tipo void, a linguagem oferece 6 tipos de variáveis, que podem ser declaradas em qualquer parte do código. São elas:

Escalares: int, float, bool.

Vector: Podem ser int, float, bool e podem conter 2, 3 ou 4 coordenadas. Para acessar as componentes de cada vetor pode-se utilizar os argumentos (x, y, z, w), (r, g, b, a) ou (s, t, p, q). Ex: position.x, cor.r, etc.

vec2 – vetor de 2 floats
ivec3 – vetor de 3 inteiros
bvec4 – vetor de 4 booleanos.

Matrix: Todas as matrizes são do tipo float, e pode assumir dimensão 2, 3 ou 4. (mat2, mat3, mat4). São geralmente utilizadas para guardar transformações lineares. Se transform é uma mat4, então transform[2] é um tipo vec4 (e representa a terceira coluna). Pode-se também acessar cada componente com transform[1][2].

Samplers: Usados para acessar texturas.

Estruturas: semelhante a sintaxe da linguagem C, exceto que para definir variáveis, deve-se omitir a palavra struct.

Arrays: Podem ser de qualquer tipo. Como a linguagem não dispõe de ponteiros, deve-se utilizar o operador [] para acessar os elementos. Ex: vec4 points[10];

Exemplos de declaração, inicialização e acesso:

```
float a, b=3.0, c;
int a;
const int size = 4; //constante que nao pode ser alterada
attribute float t; //nao pode ser inicializado (attribuite, uniform ou varying)
varying int color;
mat2 m = mat2(1); //valor é replicado para cada componente da diagonal principal
int a = 10;
float f = float(a); //conversão de tipo
float f = 10;       //gera warning

struct Ponto{
   int x, y;
};
Ponto p;
p.x = 10;

vec3 cor = vec3(1);  //este valor é replicado para cada componente.
vec4 dup = vec4(1,2,3,4);
vec4 v = vec4(1, 2, 3, 4);  //construtor
dup = v.xxyy;
```

```
dup = v.wxyz;
dup.wx = vec2(5.0, 6.0); //mantem as demais componentes inalteradas

vec4 tmp = dup + v;
tmp = dup + f; //f é escalar. Faz soma com cada componente.
f = tmp[2];   //indixacao em matriz e vetor
tmp--; //decrementa cada componente



   //1 1
   //2 2
   m[0][0] = 1;  //comandos para montar a matriz acima
   m[1][0] = 1;
   m[0][1] = 2;
   m[1][1] = 2;
   vec2 v = m[0];  //a indexacao eh por coluna. v.x = 1 e v.y = 2
```

Controle de fluxo: o ponto de entrada é a função main, que deve ser única para cada shader. Para permitir controle de execução, pode-se utilizar if-else, while, do-while e for.

7.2. Funções built-in

Pode haver sobrecarga de funções, desde que os parâmetros de entrada sejam diferentes. Quase toda as funções tem suporte a tipos `float`, `vec2`, `vec3` e `vec4`, como no seguinte exemplo.

```
float sin(float radians)
vec2 sin(vec2 radians)
vec3 sin(vec3 radians)
vec4 sin(vec4 radians)
```

Quando o argumento for um vetor, por exemplo, as operações são realizadas para cada componente separadamente, ou seja, se for passado um `vec4`, são realizadas 4 operações trigonométricas, uma para cada componente. Os resultados são armazenados na variável de retorno.

As seguintes tabelas apresentam **algumas das funções** disponibilizadas na linguagem GLSL. Para maiores detalhes, consulte [1]:

Trigonométricas:

`X radians(X)`	Conversão de graus para radianos
`X degrees(X)`	Conversão de radianos para graus
`X sin, cos, tan, asin, acos, atan(X)`	Funções trigonométricas

Comuns:

`X abs(X)`	Retorna o valor absoluto
`X sign(X)`	Retorna 1 ou -1
`X floor, ceil(X)`	Arredondamento
`X min, max(X,X)`	
`X clamp(X, float, float), step, smoothstep`	Funções para geração texturas procedurais

Geométricas:

`float length(X)`	Comprimento de um vetor
`float distance(X,X)`	Distância entre 2 pontos
`float dot(X)`	Produto escalar
`vec3 cross(vec3)`	Produto vetorial.
`X normalize(X)`	Vetor com tamanho unitário

Matrizes (X = mat2, mat3 ou mat4):

`X matrixcompmult(X,X)`	Multiplica componente a componente, diferentemente de m1 * m2.

Relacionais para vetores (X = vec2, vec3, vec4, ivec2, ivec3 ou ivec4):

`bX lessThan(X,X)`	Compara componentes dos vetores
`bX lessThanEqual(X,X)`	Compara componentes dos vetores
`bX greaterThan(X,X)`	Compara componentes dos vetores
`bX greaterThanEqual(X,X)`	Compara componentes dos vetores
`bX Equal, notEqual(X,X)`	Compara componentes dos vetores
`bool any(X)`	Retona 1 se qualquer componente for true. X deve ser bool
`bool all(X)`	Retona 1 se todas componentes são true. X deve ser bool

Ruído (Para síntese procedural):

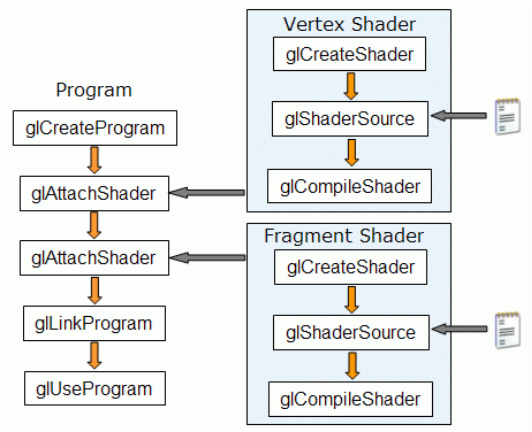
`DIM noise1, noise2, noise3, noise4(X)`	Retorna um ruido n-dimensional em função de X. Se dimensão for 1, retorna um float. Se for 2, um vec2, etc.

Exemplos de funções built-in

```
   vec3 cor = texture2D(texture, position).rgb; //nao pega o alpha
   cor = clamp(cor, 0.0, 1.0); //passando (cor,0,1) não funciona. Tem que ser float.
   gl_FragColor = vec4( cor, 1);

   vec3 v1 = vec3(0.0, 1.0, 2.0), res;
   res = cos(v1);  //aplica o cosseno para cada componente do vetor
   if(res.x != res.y)
      gl_FragColor = vec4( 1,0,0, 1);

   vec2  v1 = vec2(2, 3);
   vec2  v2 = vec2(1, 4);
   bvec2 v3 = lessThan(v1, v2); //v3 contem (false, true) pois 2>1 e 1<4
   if(v3.y == true)
      gl_FragColor = vec4( 1,0,0, 1);
```

8. Estrutura de um programa GLSL

Um programa que faça uso de GLSL é escrito em duas partes: uma que contém o código C/C++/GL e outra que contém o código GLSL. O código em GLSL é compilado e anexado ao código de um programa, como ilustrado no diagrama da seguinte figura.



http://www.lighthouse3d.com/opengl/glsl/index.php?ogloverview

Cada programa pode ter 1 ou mais shaders. Para anexar um shader a um programa utiliza-se o comando `glAttachShader()`. Um programa GLSL pode ter semente um shader de vértice (`GL_VERTEX_SHADER`), somente um shader de fragmento (`GL_FRAGMENT_SHADER`), ambos ou vários de cada, porém pode haver uma única função main em cada shader.

Um programa OpenGL pode ter 1 ou mais programas GLSL, porém no máximo um único ativo a cada momento. Para tornar um programa GLSL ativo deve-se utilizar o comando `glUseProgram()` [5], passando como argumento o programa a ser usado (que neste caso é um identificador inteiro) ou 0 caso se desejar desabilitar esta funcionalidade. Neste caso será usada a funcionalidade fixa do OpenGL par vértice e fragmento.

```
GLuint p = glCreateProgram();
...
glLinkProgram(p);
glUseProgram(p);

int v[1];
glGetIntegerv(GL_CURRENT_PROGRAM, v);
printf("\rPrograma ativo: %d", v[0]);
...
```

Se um programa GLSL tiver somente o shader de fragmento, por exemplo, será usada a funcionalidade fixa do OpenGL para fazer o processamento dos vértices. O contrário ocorre se existir somente o shader de vértice. Ao se usar a um programa de vértice,

http://nehe.gamedev.net/data/articles/article.asp?article=21
http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_3.pdf

8.1. Funções de Interface C-GLSL

A linguagem GLSL oferece um conjunto de funções para realizar o envio de parâmetros da aplicacão escrita em C/C++ para a GPU. As mais comuns são [4]:

```
GLint glGetUniformLocation(GLuint p,
                           const GLchar *name);
```

Retorna a localização de uma variável `Uniform`, dada por `name`, em um programa `p`. Tendo-se a localização da variável dentro do código do shader, pode-se então usar o comado `glUniform` para passar valores a esta variável.

```
GLint glGetAttribLocation(GLuint p,
                          const GLchar *name);
```

Retorna a localização de uma variável `attribute`, dada por `name`, em um programa `p`. Tendo-se a localização da variável dentro do código do shader, pode-se então usar o comado `glVertexAttrib` para passar valores a esta variável.

```
void glUniform{1234}{if}(Glint location,
                         TYPE v);
```

Usado para passar valores à variáveis `Uniform` localizadas dentro do código do shader. Pode-se passar vários parâmetros, dependendo do tipo da variável dentro do código do shader. O parâmetro `location` é obtido com o comando `glGetUniformLocation`.

```
void glVertexAttrib{1234}{sfd}(Glint location,
                               TYPE v);
```

Usado para passar valores à variáveis `Attribute` localizadas dentro do código do shader. Pode-se passar vários parâmetros, dependendo do tipo da variável dentro do código do shader. O parâmetro `location` é obtido com o comando `glGetAttribLocation`.

Exemplos de uso

```
...
GLint loc_u_time;
GLint loc_u_light;
Glint loc_a_color;

float u_lightPos[3] = {1,2,3};

void main()
{
   ...
   GLuint p = glCreateProgram();
   ...
   glLinkProgram(p);
   glUseProgram(p);

   loc_u_light = glGetUniformLocation(p, "lightPosition");
   loc_u_time  = glGetUniformLocation(p, "time");
   loc_a_color = glGetAttribLocation(p,  "color");
   ...
   glutMainLoop();
}

void display()
{
   ...
   glUniform1f(loc_u_time, u_time++);
   glUniform3f(loc_u_light, u_lightPos[0], u_lightPos[1], u_lightPos[2]);
   glBegin(GL_TRIANGLES);
      glVertexAttrib3f(loc_a_cor, 0.0, 1.0, 0.0);
      glVertex();
      ...
   glEnd();
}
```

9. A Biblioteca GLEW

A biblioteca GLEW (*OpenGL Extension Wrangler* - http://glew.sourceforge.net) é uma biblioteca *open source* multi-plataforma em C/C++ que simplifica o uso de extensões e novas versões OpenGL. Ela provê um mecanismo eficiente para determinação da versão OpenGL e extensões que são suportadas na plataforma diretamente do driver da placa gráfica. Para instalar esta biblioteca deve-se copiar os seguintes arquivos nos seguintes diretórios:

Arquivo	**Local**
`bin/glew32.dll`	`%SystemRoot%/system32`
`lib/glew32.lib`	`{VC Root}/Lib`
`include/GL/glew.h`	`{VC Root}/Include/GL`
`include/GL/wglew.h`	`{VC Root}/Include/GL`

onde `{VC Root}` para o compilador Microsoft .Net é …`/Vc7/PlatformSDK`.

No pacote existem dois programas utilitários (glewinfo e visualinfo) que exibem informações do sistema como recurso disponíveis de cada versão OpenGL bem como extensões suportadas, como pode ser visto nas seguintes figuras. Ambos os programas exibem a versão do OpenGL suportado.

```
--------------------------
    GLEW Extension Info
--------------------------

GLEW version 1.5.0
Reporting capabilities of pixelformat 1
Running on a GeForce FX 5200/AGP/SSE2 from NVIDIA Corporation
OpenGL version 2.1.1 is supported

GL_VERSION_1_1:                                          OK
---------------

GL_VERSION_1_2:                                          OK
---------------
  glCopyTexSubImage3D:                                   OK
  glDrawRangeElements:                                   OK
  glTexImage3D:                                          OK
  glTexSubImage3D:                                       OK

GL_VERSION_1_3:                                          OK
---------------
  glActiveTexture:                                       OK
  glClientActiveTexture:                                 OK
  glCompressedTexImage1D:                                OK
  glCompressedTexImage2D:                                OK
  glCompressedTexImage3D:                                OK
  ...
```

```
OpenGL vendor string: NVIDIA Corporation
OpenGL renderer string: GeForce FX 5200/AGP/SSE2
OpenGL version string: 2.1.1
OpenGL extensions (GL_):
    GL_ARB_depth_texture, GL_ARB_fragment_program,
    GL_ARB_fragment_program_shadow, GL_ARB_fragment_shader,
    GL_ARB_half_float_pixel, GL_ARB_imaging, GL_ARB_multisample,
    GL_ARB_multitexture, GL_ARB_occlusion_query, GL_ARB_pixel_buffer_object,
    GL_ARB_point_parameters, GL_ARB_point_sprite, GL_ARB_shadow,
    ...
```

10. Programando em GLSL

Na função main da aplicação, além das inicializações tradicionais do GL, deve-se tambem inicializar o GLSL. Para isso chama-se a função `init_glsl()`. Esta função verifica o suporte à linguagem GLSL (por meio da GLEW) e chama a função `setShaders()` que inicializa os shaders, os compila e então cria o programa, informando por possíveis erros de codificação. O código dos shaders é lido pela função `textFileRead()`. Costuma-se utilizar 2 arquivos fontes, um para o vertex e outro para o fragment shader, com extensões .vert e .frag.

```
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (600, 600);
   glutInitWindowPosition (100, 100);
   glutCreateWindow("Fragment Shader");
   glutDisplayFunc(display);
   glutIdleFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);

   init_gl();
```

```
void setShaders()
{
   char *vs = NULL,*fs = NULL,*fs2 = NULL;

   v = glCreateShader(GL_VERTEX_SHADER);
   f = glCreateShader(GL_FRAGMENT_SHADER);

   vs = textFileRead("hello.vert");
   fs = textFileRead("hello.frag");

   const char * ff = fs;
   const char * vv = vs;
```

```
    shader1 = new Glsl("tea_1.vert", "tea_1.frag");
    loc_u_time = shader1->getUniformLoc("time");

    printf("%d ", loc_u_time_1);

    glutMainLoop();
    return 0;
}
```

```
    glShaderSource(v, 1, &vv, NULL);
    glShaderSource(f, 1, &ff, NULL);

    free(vs);
    free(fs);

    glCompileShader(v);
    glCompileShader(f);

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

    glLinkProgram(p);
    //glUseProgram(p);
}
```

```
void init_glsl(char *vert, char *frag)
{
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
       fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    }
    fprintf(stdout, "Status: Using GLEW %s\n", glewGetString(GLEW_VERSION));

    if (GLEW_ARB_vertex_shader && GLEW_ARB_fragment_shader)
       printf("Ready for GLSL\n");
    else {
       printf("Not totally ready :( \n");
       exit(1);
    }
    if (glewIsSupported("GL_VERSION_2_0"))
       printf("Ready for OpenGL 2.0\n");
    else {
       printf("OpenGL 2.0 not supported\n");
       exit(1);
    }

    setShaders(vert, frag);

    if (glGetError() != GL_NO_ERROR )
    {
       printProgramInfoLog(p);
    }
}
```

```
display()
{
    ...
    shader1->setActive(true);
    ...
}
```

```
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, &infologLength);
```

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

11. Exemplos de uso

Para uma lista mais detalhada de shaders, consulte http://www.typhoonlabs.com [6].

11.1 Deformando geometrias

Para isso, no programa de vértice deve-se mudar as coordenadas de cada vértice antes de se aplicar a transformação `ftransform()`. Com isso muda-se as coordenadas geradas pela aplicação. O argumento `uniform time` é usado para simular uma animação controlada dos vértices de forma global. O programa de fragmento neste caso é opcional. Foi usado uma abordagem que muda a cor dos vértices em função de suas coordenadas, neste caso pelas variáveis `varying cor1` e `cor2`.

```
uniform int   flag;
uniform float time;
out float cor1, cor2;

void main()
{
   if( flag == 1 )
   {
      vec4 v = vec4(gl_Vertex);
      v.z = v.z + sin(v.z*10.0+time)/10.0;
      v.x = v.x + sin(v.x*10.0+time)/10.0;

      cor1 = (sin(v.z*10.0)+2.0)/2.0;
      cor2 = (sin(v.x*5.0)+2.0)/2.0;

      // the following three lines provide the same result
      //gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
      //gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      //gl_Position = ftransform();

      gl_Position = gl_ModelViewProjectionMatrix * v;
   }
   else //desenha o bule sem nenhuma transformacao
   {
      gl_Position = ftransform();
      cor1 = cor2 = 1;
   }
}

in float cor1, cor2;
void main()
{
   gl_FragColor = vec4(cor1, cor2, 0, 1.0);
}
```

11.2 Gerando texturas Procedurais

A geração de texturas na placa pode ser um processo bem simplificado. Pode-se utilizar funções procedurais como *noise* ou funções trigonométricas. Pode-se utilizar a coordenada do vértice como parâmetro ou algum atributo específico passado pela aplicação. A cor de cada pixel é então escrita diretamente no buffer de cor pelo programa de fragmento.

11.3 Acessando texturas

As texturas, geradas em formato RGB888 na aplicação, com tons entre [0, 255] quando lidas pelo programa de shader, possuem componentes que variam entre [0,1], a mesma variação de cor que deve ser enviada para o buffer de cor, como no seguinte exemplo.

```
uniform sampler2D texture_0
...
vec3 cor = texture2D(texture_0, position).rgb;
gl_FragColor = vec4(cor.x, cor.y, cor.z, 0);
```

Neste exemplo, `texture` é um sampler2D, que pode ser associado com cada unidade de textura da placa. Geralmente as placas tem no mínimo 4 unidades de textura. Para se vincular uma textura com uma unidade de textura, pode-se utilizar os seguintes comandos na aplicação em C++.

```
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture( GL_TEXTURE_2D, tex1 );

glActiveTexture(GL_TEXTURE1);
glEnable(GL_TEXTURE_2D);
glBindTexture( GL_TEXTURE_2D, tex2 );
```

No seguinte exemplo passa-se duas texturas para o programa de fragmento. Uma é aplicada na metade do quadrado e a outra na outra metade, segundo um fator de brilho que é comum para ambos.

```
out float x_coord;

void main()
{
   x_coord = gl_Vertex.x; //guarda a coordenada antes da transformação.
   gl_Position = ftransform();

   //gl_TexCoord[0]     --> built-in varying variable
   //gl_MultiTexCoord0  --> built-in attribute variable
   gl_TexCoord[0] = gl_MultiTexCoord0;
}


in sampler2D texture_0;
in sampler2D texture_1;
in float     brilho;
in float     x_coord;
void main()
{
   vec2 position = gl_TexCoord[0].st;
   vec3 cor;
   if(x_coord < 0)
      cor = texture2D(texture_0, position).rgb; //nao pega o alpha
   else
      cor = texture2D(texture_1, position).rgb; //nao pega o alpha
   cor*=brilho;
   cor = clamp(cor, 0.0, 1.0); //os parametros 0.0 e 1.0 devem ser float.
   gl_FragColor = vec4( cor, 1);
}
```

11.4 Renderizando para Textura

Este recurso é muito útil em diversos aspectos. Ele pode ser usado para gerar texturas que venham a ser usadas no próprio contexto, como por exemplo renderizar a imagem vista a partir de um espelho retrovisor, recurso este muito usado em jogos 3D, ou para realizar processamento genérico em GPU. Para este segundo caso pode-se por exemplo usar a GPU para fazer algum tipo de processamento sobre uma matriz de dados (geralmente enviados para a placa via uma textura), e pegar o resultado lendo-se a matriz de cor do framebuffer.

Para se ler o resultado do framebuffer, pode-se utilizar uma estratégia muito simples: Definir uma projeção ortográfica do tamanho da textura a ser processada, criar a textura, copiar a informação para a textura, ativar o programa de shader, enviar um quadrado do tamanho da textura para ser processado, recuperar a imagem do framebuffer com `glCopyTexSubImage2D()` ou `glCopyTexImage2D()` e usar esta informação da maneira adequada.

```
static unsigned char data[3*DIM *DIM];
glEnable (GL_TEXTURE_2D);
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, DIM, DIM, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   //renderiza para o buffer do OpenGL usando o programa de shader
   shader1->setActive(true);

   //envia um quadrado para ser processado com a dimensao da textura
   glBegin(GL_QUADS);
      glVertex2f(0, 0);
      glVertex2f(0, DIM);
      glVertex2f(DIM,DIM);
      glVertex2f(DIM, 0);
   glEnd();

   glFlush ();

   glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, DIM_JANELA, DIM_JANELA);
   //glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 0, 0, DIM_JANELA, DIM_JANELA, 0);

   //limpa a imagem gerada no framebuffer
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   //usa-se a textura lida
   shader1->setActive(false);

   // faz alguma coisa com a textura

   glutSwapBuffers();
}
```

Referências Bibliográficas

[1] Rost, Randi. **OpenGL Shading Language** (OpenGL Orange Book). Addison Wesley. 2004.

[2] Celes, W. Notas de Aula. PUC-Rio, 2006.

[3] LigthHouse3D - tutorials. Disponível em: http://www.lighthouse3d.com/opengl/

[4] OpenGL Shading Language API Man Pages. Disponível em: http://developer.3dlabs.com/documents/glslmanpage_index.htm

[5] OpenGL Org. http://www.opengl.org/sdk/docs/man/xhtml/glUseProgram.xml

[6] Jacobo Rodriguez Villar. TyphoonLabs' OpenGL Shading Language tutorials. Disponível em http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

[7] OpenGL Org. **Rendering Pipeline Overview**. http://www.opengl.org/wiki/Rendering_Pipeline_Overview

[8] OpenGL.org, **Built-in Variable (GLSL).** http://www.opengl.org/wiki/Built-in_Variable_(GLSL)

[9] Normal Matrix: http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/

[10] Transformação de Luz: http://stackoverflow.com/questions/8109239/glsl-gl2-1-lighting-transforming-to-eye-space

[11] Bump Mapping com shaders: http://www.fabiensanglard.net/bumpMapping/index.php