Universidade Federal de Santa Maria
Departamento de Computação Aplicada
Prof. Cesar Tadeu Pozzer

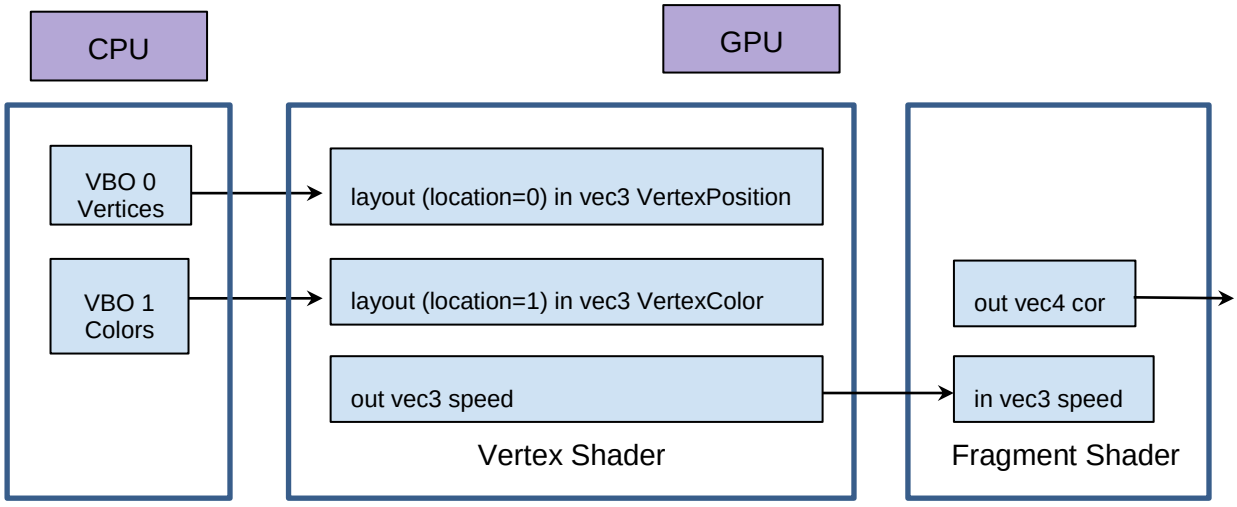
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
19/08/2017

Autores
Alex Frasson, Tiago Engel, Cesar Pozzer
Cícero Pahins e Guilherme Schardong

OpenGL 4.x

1. Overview

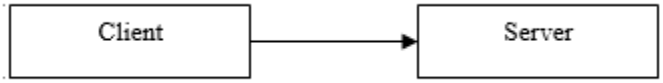
A programação em OpenGL 4.x consiste, basicamente, em definir buffers de dados na CPU que são passados para a GPU, onde executam programas de shader que processam esses dados conforme necessidade do programador. Existe um mapeamento dos dados entre CPU-GPU, da mesma forma como os dados são passados entre estágios na GPU, como mostrado na seguinte figura. Existem 5 tipos de shader, mas obrigatoriamente deve-se ter o vertex e o fragment shader.



2. Introdução

2.1. Arquitetura cliente-servidor

O OpenGL trabalha com a filosofia cliente-servidor. Isso significa que se pode trabalhar em um computador (cliente) e ver os resultados em outro (servidor). O cliente é responsável pela geração dos comandos OpenGL enquanto que o servidor é responsável pelo processamento (render) destes dados e exibição na tela. Caso não existirem computadores em rede, o mesmo computador exerce o papel de cliente e servidor (maior parte dos casos).



Arquitetura cliente-servidor.

Esse conceito é importante para melhor compreensão de conceitos avançados do OpenGL. A arquitetura cliente-servidor mais comum é onde os comandos e dados são gerenciados pelo cliente (CPU) e processados no servidor (GPU). O processamento gráfico é feito na placa de vídeo e é conveniente dado sua arquitetura ser especialmente desenhada para esse tipo de processamento.

2.2. Suporte do OpenGL 4.x

A primeira versão do OpenGL, versão 1.0, foi lançada em Janeiro de 1992. Desde então, OpenGL tem sido estendido pela liberação de novas versões da especificação. Tais lançamentos definem um conjunto básico de características que todas as placas gráficas conformantes devem suportar. O OpenGL 4.0 foi lançado juntamente com a versão 3.3 em Março de 2010. Ele foi projetado para hardware capaz de suportar DirectX 11.

OVERALL DISTRIBUTION OF CARDS	DEC	JAN	FEB	MAR	APR	
DirectX 12 GPUs	67.35%	67.78%	68.91%	70.22%	**70.45%**	+0.23%
DirectX 11 GPUs	14.85%	14.58%	13.98%	13.61%	**13.40%**	-0.21%
DirectX 10 GPUs	15.86%	15.73%	15.32%	14.50%	**14.47%**	-0.03%
DirectX 9 Shader Model 2b and 3.0 GPUs	0.75%	0.75%	0.71%	0.65%	**0.64%**	-0.01%
DirectX 9 Shader Model 2.0 GPUs	0.55%	0.56%	0.54%	0.52%	**0.52%**	0.00%
DirectX 8 GPUs and below	0.64%	0.60%	0.54%	0.50%	**0.52%**	+0.02%

Distribuição das GPUs utilizadas pelos usuários da Steam™ em Abril de 2016. Assumindo a equivalência do OpenGL 4.x ao DirectX 11, pode-se concluir que mais de 80% dos usuários utilizam placas com suporte ao OpenGL 4.x para o período. Fonte: (http://store.steampowered.com/hwsurvey).

A partir da série HD 5xxx (2009), todas as placas suportam ao menos OpenGL 4.4. A partir da série Radeon Rx 300 (2015), todas as placas suportam ao menos OpenGL 4.5.

A partir da série GeForce 400 (2010), todas as placas suportam ao menos OpenGL 4.5.

A partir da sétima geração, Ivy Bridge (2012), Silvermont e Haswell, todas as placas suportam ao menos OpenGL 4.0 (Intel i3 4ª Geração). A partir da oitava geração, Broadwell (2015 – Intel i3 5ª Geração) e Airmont, todas as placas suportam ao menos OpenGL 4.4.

3. Conceitos do OpenGL

OpenGL é definida como uma “máquina de estados”. As várias chamadas da API mudam o estado do OpenGL, consultam alguma parte desse estado, ou utilizam o estado atual para renderizar algo. Uma vez que o valor de uma propriedade é definido, ele persiste até que um novo valor é dado.

3.1. Contexto OpenGL

Embora não seja crucial para programadores saber como o contexto do OpenGL é implementado, é importante entender o que é a fim de saber como usá-lo.

Um contexto de renderização OpenGL é uma porta através da qual todos os comandos OpenGL passam. Contextos de renderização ligam o OpenGL ao sistema de janelas do sistema operacional. Um contexto do OpenGL também é a estrutura de dados que armazena todos os estados necessários para renderizar uma imagem. Ele contém referências de buffers, texturas, shaders, etc. Pense no contexto como um objeto que armazena todo OpenGL. Quando o contexto é destruído, o OpenGL é destruído.

Como o OpenGL não existe até que você crie um contexto, a criação de contextos OpenGL não é regida pela especificação da API OpenGL. Em vez isso é governado por APIs específicas de cada plataforma. Existem várias bibliotecas auxiliares que facilitam a criação de contextos do OpenGL. Uma delas é a GLFW (veja seção 5.2) que, ao criar uma janela, pode gerar um contexto do OpenGL independente da plataforma utilizada. Um exemplo de uso pode ser encontrado na função `initGLFW` dos arquivos `main.cpp` dos demos da disciplina.

Nenhum comando do OpenGL pode ser executado até que um contexto seja criado e vinculado. Cada thread que faz chamadas OpenGL deve possuir um contexto de renderização ativo. Se uma aplicação faz chamadas OpenGL de uma thread que não possui um contexto ativo, nada acontece; a chamada não tem efeito.

Uma aplicação normalmente cria um contexto de renderização, define-o como o contexto ativo de uma thread, e em seguida, chama funções OpenGL. Quando funções OpenGL não precisam ser chamadas, o aplicativo separa o contexto da thread, e exclui o contexto de renderização. Uma janela pode ter vários contextos de renderização desenhando de uma vez, mas uma thread pode ter apenas um contexto ativo.

Contextos são localizados dentro de um determinado processo de execução. Um processo pode criar vários contextos OpenGL. Cada contexto pode representar uma superfície visível separada, como uma janela em um aplicativo. Para que comandos do OpenGL funcionem, deve existir um contexto ativo. Todos os comandos do OpenGL afetam o estado do contexto ativo. O contexto atual é uma variável local de uma thread, portanto, como um único processo pode ter várias threads, cada uma pode ter seu próprio contexto ativo. No entanto, um único contexto não pode estar ativo em várias threads ao mesmo tempo.

3.2. OpenGL Object

Um OpenGL Object é um objeto que contém algum estado. Quando eles estão vinculados ao contexto, o estado que eles contêm é mapeado para o estado do contexto. Assim, as alterações do contexto serão armazenadas neste objeto, e as funções que atuam sobre este estado do contexto usará o estado armazenado no objeto.

Objetos são sempre recipientes de um estado. Cada tipo de objeto em particular é definido pelo estado particular que ele contém. Um objeto OpenGL é uma maneira de encapsular um grupo particular de estados e mudar todo o estado em uma única chamada de função.

OpenGL objects podem ser separados em duas diferentes categorias: objetos regulares e objetos recipientes.

Objetos regulares:	Objetos recipientes:
• Buffer Objects (VBO, etc) • Texture Objects • Query Objects • Renderbuffer Objects • Sampler Objects	• Vertex Array Objects (VAO) • Framebuffer Objects • Program Pipeline Objects • Transform Feedback Objects

Existem também objetos que não seguem o padrão de OpenGL objects. Abaixo segue uma lista dos objetos não convencionais:

- Shader e Program Objects
- Sync Objects

3.2.1. Gerando e destruindo OpenGL Objects

Para criar um objeto você gera o nome do objeto (um inteiro). Isto cria uma referência para o objeto. As funções para gerar nomes de objetos são da forma `glGen*`, onde * é o tipo do objeto em forma plural. Todas as funções deste tipo têm o mesmo formato:

```
void glGen*(GLsizei n, GLuint *objects);
```

São alguns exemplos:

```
GLuint vaoID;
glGenVertexArrays(1, &vaoID);

GLuint handle[3];
glGenBuffers(3, handle);
```

Esta função gera `n` objetos do tipo dado, armazenando-os no array dado pelo parâmetro `objects`. Isto permite-lhe criar vários objetos com uma chamada.

Uma vez que você terminou de utilizar um objeto, você deve excluí-lo. As funções para isso são a forma `glDelete*`, usando o mesmo tipo de objeto, como antes. Essas funções têm o seguinte formato:

```
void glDelete*(GLsizei n, const GLuint *objects);
```

Isso funciona como a função `glGen*`, só que exclui os objetos em vez de criá-los. Quando os objetos OpenGL são excluídos, os seus nomes não são mais considerados válidos.

3.2.2. Usando OpenGL Objects

Como os objetos em OpenGL são definidos como uma coleção de estados, para modificar objetos, primeiro você deve vinculá-los ao contexto OpenGL. Vincular objetos ao contexto faz com que o estado contido neles seja copiado para o estado do contexto atual.

Objetos diferentes têm funções para vincular diferentes. Ainda assim, eles compartilham uma convenção de nomenclatura e parâmetros gerais:

```
void glBind*(GLenum target, GLuint object);
```

O * é o tipo de objeto, e `object` é o objeto a ser vinculado. O `target` é onde diferentes tipos de objetos diferem. Se um objeto está vinculado a um local onde outro objeto já está vinculado, o objeto anteriormente vinculado será desvinculado.

3.2.3. Objeto 0

O valor `GLuint 0` é tratado especialmente por objetos do OpenGL. `0` nunca será retornado por uma função `glGen*`. Para a maioria dos tipos de objetos, objeto `0` é muito parecido com o ponteiro NULL: não é um objeto. Se `0` é vinculado por esses tipos de objetos, tentativas de usar esse objeto para fins de renderização falharão. Para alguns objetos, objeto `0` representa uma espécie de "objeto padrão".

Recomendação: Com exceção de FrameBuffer Objects, você deve tratar objeto `0` como um objeto não-funcional. Mesmo se um tipo de objeto tem um objeto `0` válido, você deve tratá-lo como se ele não o fosse. Trate-o como se fosse o ponteiro NULL em C/C++; você pode armazená-lo em um ponteiro, mas você não pode usar esse ponteiro até que um valor real seja atribuído.

3.3. Buffer Objects

Buffer objects são OpenGL Objects que armazenam um array de memória não formatada alocados pelo contexto do OpenGL. Estes podem ser utilizados para armazenar dados de vértice, dados de pixel recuperados a partir de imagens ou o FrameBuffer, dentre outros.

Abaixo segue uma lista de todos os buffer objects:

- Vertex Buffer Objects
- Buffer Object Streaming
- Pixel Buffer Objects
- Shader Storage Buffer Objects
- Uniform Buffer Objects
- Atomic Counters

Buffer Objects são objetos do OpenGL. Eles, portanto, seguem todas as regras de objetos OpenGL regulares. Para criar um objeto de buffer, você chama `glGenBuffers`, para excluí-los, `glDeleteBuffers`.

3.3.1. Usando Buffer Objects

Tal como acontece com o paradigma OpenGL objeto padrão, isso só cria o nome do objeto, a referência para o objeto. Para configurar o seu estado interno, você deve vinculá-lo ao contexto. Você pode fazer isso usando a seguinte função:

```
void glBindBuffer(enum target, uint bufferName);
```

O `target` define como você pretende usar essa ligação do Buffer Object. Quando você está apenas criando e/ou preenchendo o Buffer Object com dados, o `target` que você usa tecnicamente não faz diferença.

3.4. Vertex Buffer Object

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, hajam 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 (os VAOs). Abaixo iremos construir um exemplo utilizando VBO. Note que o código está adequado ao OpenGL 4.x. Normalmente são utilizados VBOs buffers específicos para:

- Vértices: os vértices são enviados como pontos 3D.
- Normais: são enviadas como vetores 3D.
- Índices: são os índices inteiros dos vértices que formam primitivas (normalmente triângulos ou quadrados), formando um vetor contínuo onde cada três (ou quatro) valores quando indexados no vetor de vértices formam as primitivas.
- Coordenadas de textura: pode estar associada a cada vértice.
- Cores: podem ser enviadas através de buffers ou definidas diretamente no fragment shader. Além disso, quando a cor for definida por uma textura, deve ser feito o mapeamento da textura no fragment shader.
- Dados criados pelo usuário, como velocidade, temperatura, deformação, etc.

Vertex Array Object

Um Vertex Array Object (VAO) é um OpenGL Object que armazena todos os estados do contexto necessários para fornecer dados dos vértices. Ele armazena o formato de dados do vértice, bem como os Buffer Objects que fornecem os arrays de vértices.

Uma vez os VBOs configurados, eles são agrupados em uma estrutura única chamada Vertex Array Objects (VAO), o qual é responsável por armazenar as informações e conexões entre os atributos dos VBOs. Ele é criado chamando a função glGenVertexArrays, e as informações passadas ao VAO são guardadas em índices e habilitadas pela função glEnableVertexAttribArray. O VAO permite que os atributos por vértice sejam configurados uma vez e salvos para uso durante a renderização, na qual basta o identificador do VAO para que seja possível invocar o desenho dos VBOs associados.

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, 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 Objects, 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) necessários para a “configuração do VBO”.

4. Pipeline Gráfico

O pipeline gráfico descreve o conjunto de passos que são percorridos para tornar dados brutos em imagens, sendo cada etapa responsável por uma atividade (que serão detalhadas a seguir). O pipeline vêm evoluindo constantemente e a tendência é que se torne cada vez mais flexível, a fim de proporcionar funcionalidades específicas e atender as necessidades dos desenvolvedores já que os requisitos de qualidade estão cada vez mais altos.

A Figura 3.1 mostra o fluxo de dados do pipeline gráfico. A aplicação, que roda na CPU, envia dados para GPU através de texturas, dados de vértices e variáveis Uniform (variáveis que mudam com pouca frequência, como por exemplo, a matriz de ModelView). O primeiro estágio do pipeline, o Vertex Shader, é alimentado pelos dados de vértices. Texturas e variáveis Uniform podem ser acessadas de qualquer estágio. Além disso, variáveis de saída de um estágio, com o qualificador `out`, alimentam as variáveis de entrada do estágio seguinte, com o qualificador `in`.

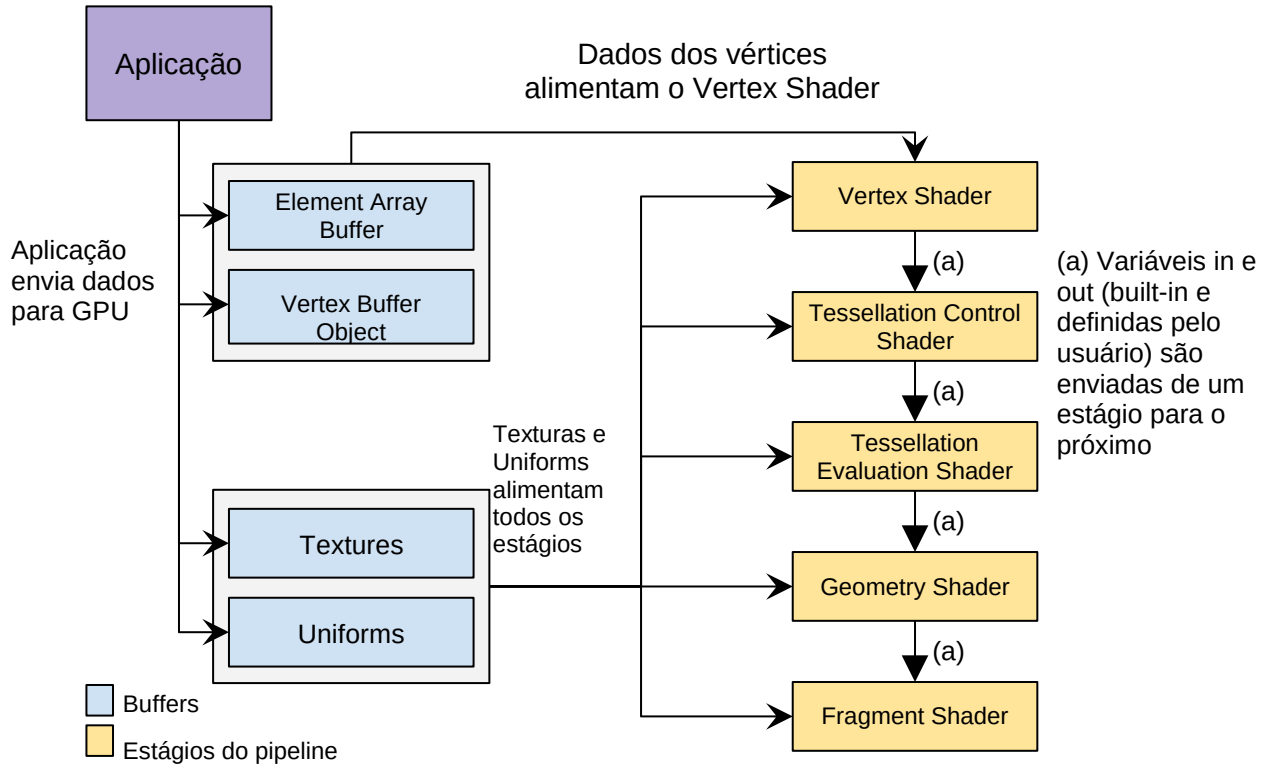


Figura 3.1. Fluxo de dados do pipeline gráfico do OpenGL 4.x. Vertex array buffer object armazena os vértices e element array buffer armazena os índices que descrevem em que ordem os vértices são desenhados. Uma explicação mais aprofundada pode ser encontrada em 6.1.

A Figura 3.2 mostra todas as etapas do pipeline, porém algumas delas não são necessariamente programáveis (etapas em cor verde). De fato, algumas delas são executadas pela API gráfica e são transparentes ao desenvolvedor, o qual é dado alguma interface para configuração das mesmas na CPU. Por exemplo, a etapa de rasterização permite o desenvolvedor configurar se um triângulo será preenchido (GL_FILL) ou wireframe (GL_LINE).

Por outro lado, as etapas em amarelo permitem que o desenvolvedor escreva seu próprio código para executar na GPU, esse programa é chamado de shader. De acordo com seu propósito, cada shader deve estar associado a uma etapa do pipeline e será chamado de acordo. Estamos particularmente interessados nas etapas programáveis, já que assim podemos customizar a execução de acordo com a necessidade. As próximas seções mostram as funções de cada etapa do pipeline do hardware, ou seja, assume-se que os dados (vértices, texturas, etc) já estão na GPU.

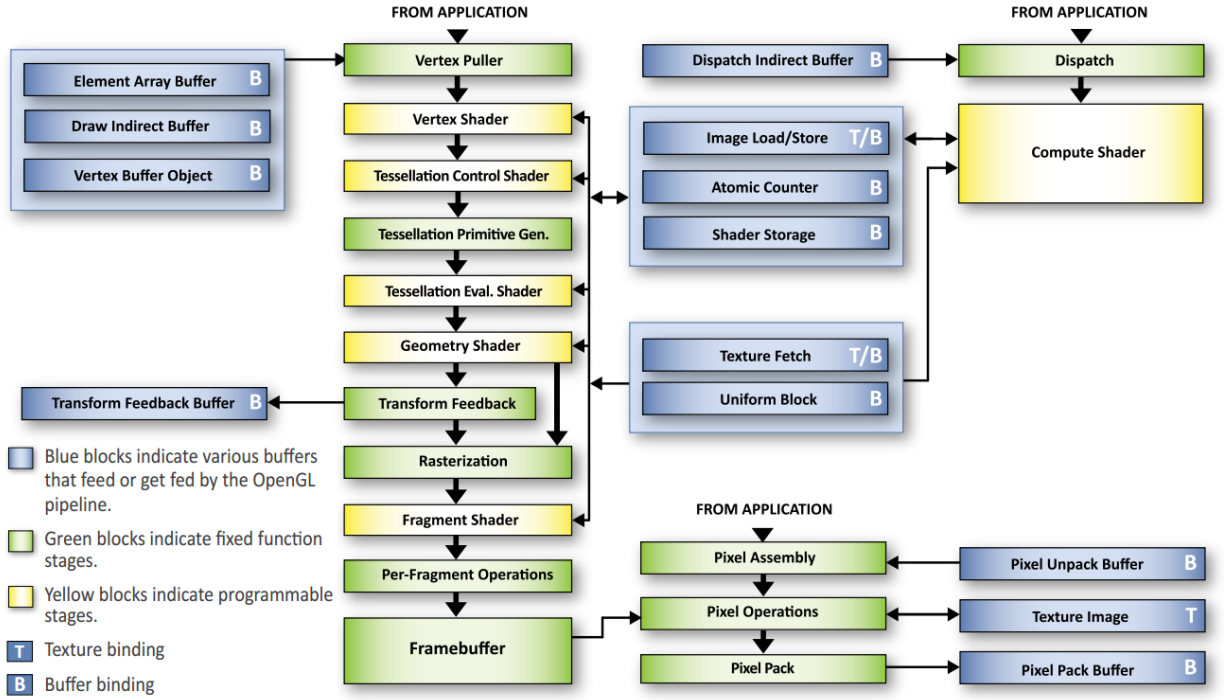


Figura 3.2. Pipeline gráfico do OpenGL 4.x.

4.1. Vertex Shader

O Vertex Shader é o estágio programável do pipeline de renderização que faz o processamento de vértices individuais. O Vertex Shader é alimentado por atributos de vértices. Um Vertex Shader recebe um único vértice do fluxo de entrada e gera um único vértice para o fluxo de saída de vértices. Deve haver um mapeamento 1:1 de vértices de entrada para saída. Vertex Shaders normalmente executam transformações para o espaço pós-projeção. Eles também podem ser usados para fazer a iluminação por vértice, ou para preparar os dados para os próximos estágios do pipeline gráfico.

4.2. Tessellation Shaders

O Tessellation foi introduzido no OpenGL 4 a fim de proporcionar a geração massiva de vértices diretamente na GPU. Embora o Geometry Shader também permita a geração de vértices, o tessellation é especialmente desenhado para tal, oferecendo alto desempenho. Esse estágio é opcional e é executado depois do vertex shader e antes do Geometry Shader (se utilizado).

Quando tessellation é utilizado, existe apenas um tipo de primitiva que pode ser utilizada: o patch (`GL_PATCHES`). Ele é um conjunto arbitrário de geometria (ou informações diversas) que são completamente definidos pelo programador, como por exemplo, os pontos de controle de

uma curva. O tessellation foi divido em duas etapas, a primeira sendo o Tessellation control shader (TCS) e a segunda o Tessellation evaluator shader (TES).

O TCS é responsável por definir o nível de tessellation utilizado em cada patch (atribuindo as variáveis `gl_TessLevelInner` e `gl_TessLevelOuter`) e qual algoritmo de geração utilizar. Essas informações são passadas para o Tessellation Primitive Generator (TPG), que é uma etapa configurável do pipeline, sendo essa uma unidade específica para a geração de primitivas. A saída dela são os vértices em coordenadas normalizadas (u,v,w), que serão alimentadas para a entrada do TES.

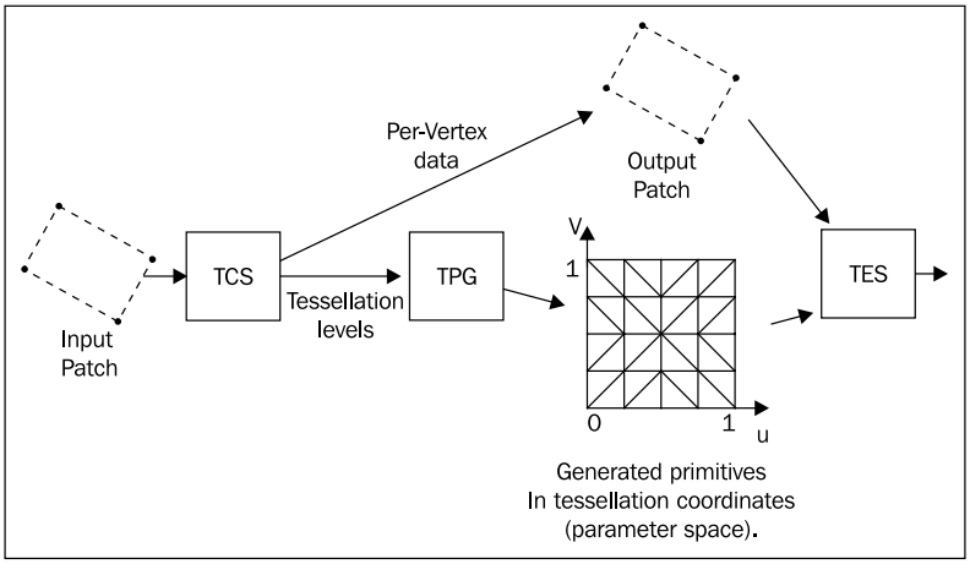


Figura 1.2.2.1. Fluxograma das três etapas do tessellation. Fonte: glslcookbook.

O TES recebe os vértices originais do patch e as coordenadas normalizadas geradas pelo TPG (além de dados arbitrários que podem ser enviados pelo TCS). A fim de posicionar os vértices no mundo, podem ser feitas diferentes interpolações sobre os valores, como, por exemplo, uma interpolação bicúbica, sendo que a variável de saída built-in é a gl_Position.

O tessellation possui diversas aplicações, entre elas:

- Refinamento de LOD em terreno;
- Geração e animação de cabelo na GPU;
- Detalhamento em personagens;
- Displacement mapping.

4.3. Geometry Shader

O Geometry Shader é um estágio opcional do pipeline gráfico que é executado depois do tessellation (se utilizado) ou do Vertex Shader. Ele passa uma vez por primitiva, as quais podem ser pontos, linhas, triângulos ou quadrados. Cada chamada tem acesso à todos os vértices da primitiva.

Invocações do geometry shader recebem uma única primitiva como entrada e podem produzir zero ou mais primitivas. Geometry shader são escritos para aceitar um tipo específico de primitiva como entrada e outro, ou o mesmo, como saída. Enquanto o geometry shader pode ser utilizado para amplificar a geometria, desse modo aplicando uma forma bruta de tessellation, isto geralmente não é um bom caso uso de um geometry shader.

O geometry shader pode ser utilizado para diversas finalidades, entre elas:

- Sistema de partículas;
- Cálculo de normais na GPU;
- Efeitos que operam sobre primitivas em geral (ex: animar triângulo);
- Culling de triângulos fora do campo de visão.

4.4. Fragment Shader

O fragment shader é um dos estágios mais importantes do pipeline. Ele é executado após a rasterização e opera por pixel, possibilitando diversas aplicações tais como:

- Iluminação por pixel;
- Aplicação de textura;
- Geração procedural de textura.

4.5. Compute Shader

O compute shader é um tipo especial de programa que roda na GPU, mas não faz parte do pipeline gráfico. Porém, ele pode ser utilizado para desempenhar alguma tarefa e alimentar o pipeline com dados. Por exemplo, ele pode ser utilizado para gerar texturas de ruídos enquanto o tessellation as utiliza para fazer o *displacement* do terreno.

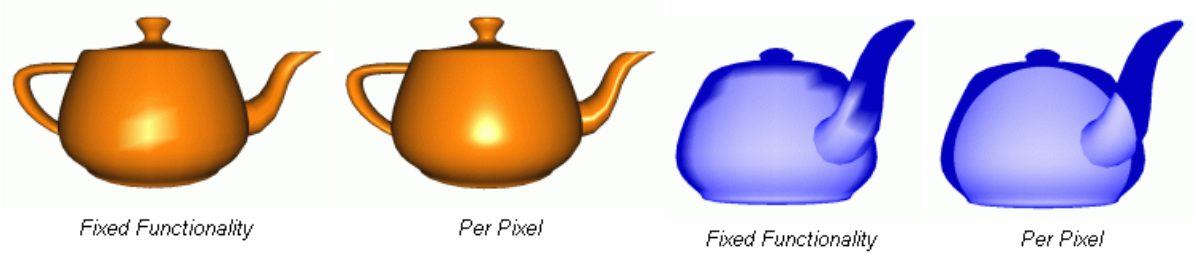
5. OpenGL Shading Language

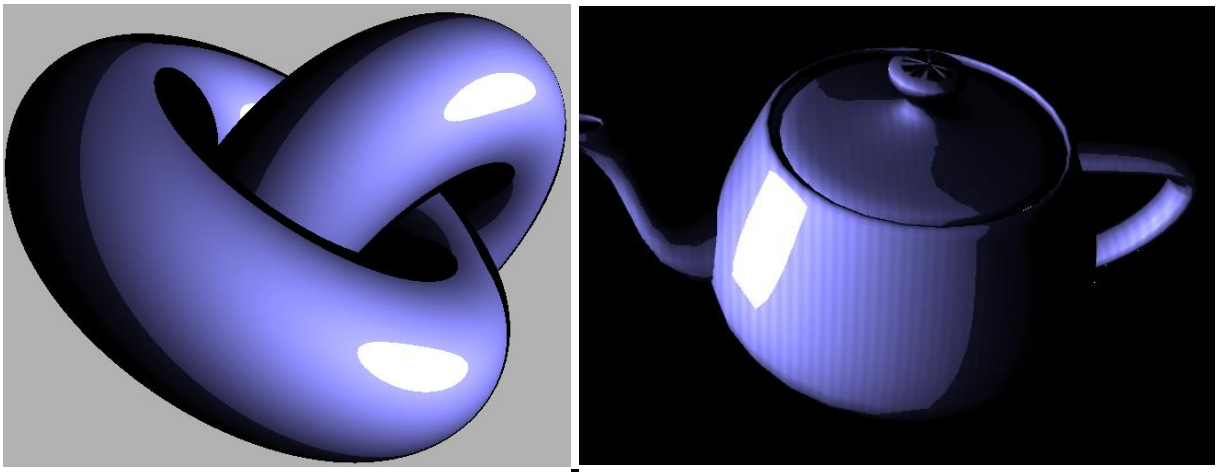
OpenGL Shading Language (GLSL), é uma shading language de alto nível baseada na sintaxe das linguagens de programação C/C++. Ela foi criada pelo OpenGL ARB (OpenGL Architecture Review Board) para dar aos desenvolvedores um maior controle sobre o pipeline gráfico sem ter que usar linguagens específicas de hardware. Com GLSL o programador pode codificar programas curtos, chamados shaders, que são executados na GPU.

Alguns benefícios da utilização de GLSL são:

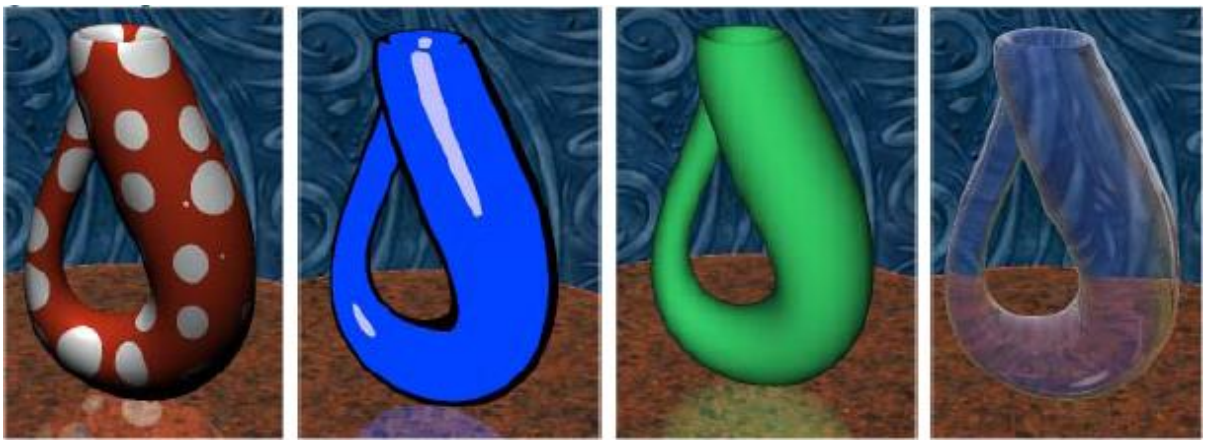
- Compatibilidade entre plataformas em vários sistemas operacionais, incluindo GNU / Linux, Mac OS X e Windows.
- A capacidade de escrever shaders que podem ser usados na placa gráfica de qualquer fornecedor de hardware que suporte a OpenGL Shading Language.
- Cada fornecedor de hardware inclui o compilador GLSL em seu driver, permitindo assim que cada fornecedor gere código otimizado para a arquitetura da sua placa gráfica.

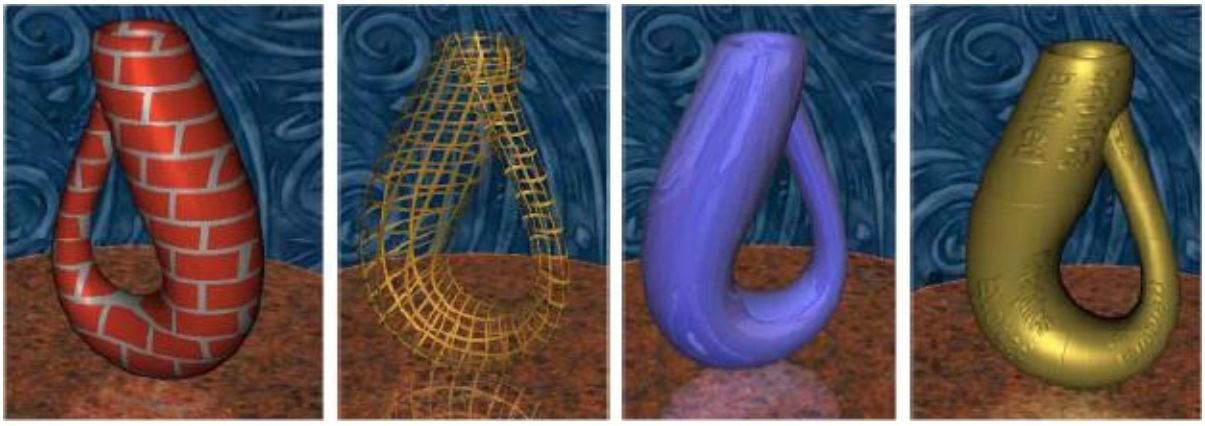
Ao contrário das versões antigas do OpenGL baseadas em pipeline fixo, GLSL permite fazer iluminação por pixel, o que permite criar imagens muito mais realistas.





Shader desenvolvido por Victor Chitolina Schetinger – 2009





5.1. GLSL e sua similaridade com C

A linguagem C foi estendida com tipos vetoriais e matriciais para facilitar operações típicas realizadas em gráficos 3D. Alguns mecanismos de C++ também foram herdados, tais como sobrecarga de funções com base nos tipos dos argumentos, e capacidade de declarar variáveis onde primeiro são necessárias, em vez de no início de blocos. A GLSL fornece muitos operadores familiares para aqueles com proficientes em C. Ela contém os operadores em C e C++, com a exceção de ponteiros. Semelhante à linguagem de programação C, GLSL suporta loops e branchs, incluindo: if-else, for, do-while, break, continue, etc. A linguagem não permite recursão.

5.2. Linguagem

O GLSL define uma série de funções padrão. Algumas funções padrão são específicas para determinados estágios do pipeline, enquanto a maioria está disponível em qualquer estágio. Há uma documentação de referência para estas funções disponíveis em [5] e [6].

Os diferentes estágios do pipeline têm um número de variáveis predefinidas para eles. Estes são fornecidos pelo sistema para vários usos específicos do sistema e também encontram-se documentadas em [6].

5.2.1. Variáveis Uniform

Uma uniform é uma variável GLSL global, definida pelo usuário, declarada com o qualificador de armazenamento "`uniform`". Estes agem como parâmetros que o usuário pode passar para um shader. Uniformes são assim chamadas porque elas não mudam de uma execução de um shader para outra dentro de uma chamada de renderização. Variáveis uniform devem ser definidas em GLSL no escopo global. Uniformes pode ser de qualquer tipo, ou tipos de agregação. A seguir está um exemplo de uniforms em GLSL:

```
struct TheStruct
{
  vec3 first;
  vec4 second;
  mat4 third;
};

uniform vec3 oneUniform;
uniform TheStruct aUniformOfArrayType;
uniform mat4 matrixArrayUniform[25];
uniform TheStruct uniformArrayOfStructs[10];
```

Uniforms são implicitamente constantes dentro do shader. Elas são constantes, mas não em tempo de compilação (por isso não são equivalentes ao qualificador `const`). A tentativa de alterá-las dentro do shader irá resultar em um erro de compilação.

Como OpenGL Objects, objetos do programa encapsulam um certo estado. Neste caso, este estado é um conjunto de uniforms que serão usados quando o programa é usado para renderização. Todos os estágios de um programa usam o mesmo conjunto de uniformes. Assim, se tivermos uma uniform chamada "projectionMatrix" definida como mat4 tanto no Vertex quanto no Fragment Shader, haverá apenas uma uniform com esse nome exposta pelo programa. Alterar esta uniform vai afetar tanto o Vertex quanto o Fragment Shader.

5.2.2. Variáveis de entrada e saída de um estágio

As variáveis globais declaradas com o qualificador `in` são variáveis de entrada do shader. Essas variáveis são valores dados pelo estágio anterior. Estas variáveis não são constantes (no sentido de `const`), mas elas não podem ser alteradas pelo código do shader.

As variáveis globais declaradas com o qualificador `out` são variáveis de saída do shader. Estes valores são passados para o próximo estágio do pipeline. O shader deve atribuir todas as variáveis de saída em algum momento de sua execução.

Vertex Shader

```
layout (location=0) in vec3 VertexPos;

out vec3 cor;

uniform mat4 MVP;

void main()
{
    cor = vec3(1.0, 0.0, 0.0);
    gl_Position = MVP * vec4(VertexPos, 1.0);
}
```

Fragment Shader

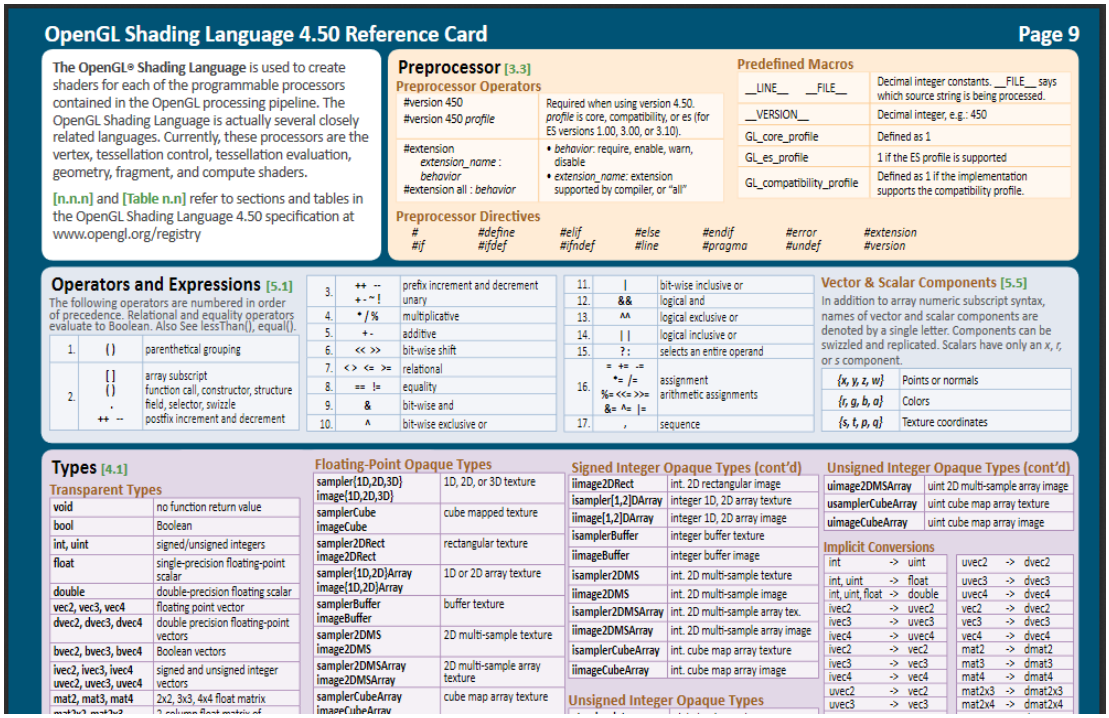
```
in vec3 cor;

                    out vec4 out1;
layout (location=0) out vec4 out2;

void main()
{
   gl_FragColor = vec4(cor, 1.0); //GL4.0
   out1         = vec4(cor, 1.0); //GL4.6
   out2         = vec4(cor, 1.0); //GL4.6
}
```

No exemplo acima, a variável `cor`, no estágio Vertex Shader, possui o qualificador `out` e é atribuída durante a execução do shader. O valor atribuído estará então disponível no Fragment Shader através da variável `cor` com o qualificador `in`. Para gerenciar a saída do Fragment Shader, existem algumas possiblidades, dependendo da versão do OpenGL. Pode-se, por exemplo, determinar em qual buffer a imagem será salva.

É importante notar que, além das variáveis de entrada e saída definidas pelo usuário, também existem variáveis pré-definidas (built-in) pela linguagem GLSL (no exemplo acima as variáveis **gl_Position** e **gl_FragColor**). Uma lista completa dessas variáveis pode ser encontrada no capítulo Built-In Variables do Quick Reference Card do OpenGL [6]. Acesse https://www.khronos.org/files/opengl46-quick-reference-card.pdf para uma lista completa das variáveis, funções e tipos da linguagem.



OpenGL Shading Language 4.50 Reference Card — Page 9

The OpenGL® Shading Language is used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline. The OpenGL Shading Language is actually several closely related languages. Currently, these processors are the vertex, tessellation control, tessellation evaluation, geometry, fragment, and compute shaders.

[n.n.n] and [Table n.n] refer to sections and tables in the OpenGL Shading Language 4.50 specification at www.opengl.org/registry

Preprocessor [3.3]

Preprocessor Operators

#version 450 #version 450 *profile*	Required when using version 4.50. *profile* is core, compatibility, or es (for ES versions 1.00, 3.00, or 3.10).
#extension *extension_name* : *behavior* #extension all : *behavior*	• *behavior*: require, enable, warn, disable • *extension_name*: extension supported by compiler, or "all"

Predefined Macros

__LINE__ __FILE__	Decimal integer constants. __FILE__ says which source string is being processed.
__VERSION__	Decimal integer, e.g.: 450
GL_core_profile	Defined as 1
GL_es_profile	1 if the ES profile is supported
GL_compatibility_profile	Defined as 1 if the implementation supports the compatibility profile.

Preprocessor Directives

#	#define	#elif	#else	#endif	#error	#extension	
#if	#ifdef	#ifndef	#line	#pragma	#undef	#version	

Operators and Expressions [5.1]

The following operators are numbered in order of precedence. Relational and equality operators evaluate to Boolean. Also See lessThan(), equal().

1.	( )	parenthetical grouping
2.	[ ] ( ) . ++ --	array subscript function call, constructor, structure field, selector, swizzle postfix increment and decrement
3.	++ -- + - ~ !	prefix increment and decrement unary
4.	* / %	multiplicative
5.	+ -	additive
6.	<< >>	bit-wise shift
7.	< > <= >=	relational
8.	== !=	equality
9.	&	bit-wise and
10.	^	bit-wise exclusive or
11.	|	bit-wise inclusive or
12.	&&	logical and
13.	^^	logical exclusive or
14.	||	logical inclusive or
15.	? :	selects an entire operand
16.	= += -= *= /= %= <<= >>= &= ^= |=	assignment arithmetic assignments
17.	,	sequence

Vector & Scalar Components [5.5]

In addition to array numeric subscript syntax, names of vector and scalar components are denoted by a single letter. Components can be swizzled and replicated. Scalars have only an *x*, *r*, or *s* component.

{x, y, z, w}	Points or normals
{r, g, b, a}	Colors
{s, t, p, q}	Texture coordinates

Types [4.1]

Transparent Types

void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar
vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 uvec2, uvec3, uvec4	signed and unsigned integer vectors
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix

Floating-Point Opaque Types

sampler{1D,2D,3D} image{1D,2D,3D}	1D, 2D, or 3D texture
samplerCube imageCube	cube mapped texture
sampler2DRect image2DRect	rectangular texture
sampler{1D,2D}Array image{1D,2D}Array	1D or 2D array texture
samplerBuffer imageBuffer	buffer texture
sampler2DMS image2DMS	2D multi-sample texture
sampler2DMSArray image2DMSArray	2D multi-sample array texture
samplerCubeArray imageCubeArray	cube map array texture

Signed Integer Opaque Types (cont'd)

iimage2DRect	int. 2D rectangular image
isampler{1,2}DArray	integer 1D, 2D array texture
iimage{1,2}DArray	integer 1D, 2D array image
isamplerBuffer	integer buffer texture
iimageBuffer	integer buffer image
isampler2DMS	int. 2D multi-sample texture
iimage2DMS	int. 2D multi-sample image
isampler2DMSArray	int. 2D multi-sample array tex.
iimage2DMSArray	int. 2D multi-sample array image
isamplerCubeArray	int. cube map array texture
iimageCubeArray	int. cube map array image

Unsigned Integer Opaque Types

Unsigned Integer Opaque Types (cont'd)

uimage2DMSArray	uint 2D multi-sample array image
usamplerCubeArray	uint cube map array texture
uimageCubeArray	uint cube map array image

Implicit Conversions

int	->	uint	uvec2	->	dvec2
int, uint	->	float	uvec3	->	dvec3
int, uint, float	->	double	uvec4	->	dvec4
ivec2	->	uvec2	vec2	->	dvec2
ivec3	->	uvec3	vec3	->	dvec3
ivec4	->	uvec4	vec4	->	dvec4
ivec2	->	vec2	mat2	->	dmat2
ivec3	->	vec3	mat3	->	dmat3
ivec4	->	vec4	mat4	->	dmat4
uvec2	->	vec2	mat2x3	->	dmat2x3
uvec3	->	vec3	mat2x4	->	dmat2x4

5.2.3. Layout

Um número de variáveis e definições do GLSL podem ter qualificadores `layout` associados a eles. Qualificadores `layout` afetam de onde o armazenamento de uma variável vem, assim como outras propriedades que podem ser definidas pelo usuário. Usa-se layout para receber dados de um VBO, como no seguinte exemplo:

```
//criacao dos VBOs (Codigo C)
glEnableVertexAttribArray(0);  // Vertex position -> layout 0 in the VS
glEnableVertexAttribArray(1);  // Vertex color    -> layout 1 in the VS
glEnableVertexAttribArray(2);  // Vertex Normal   -> layout 2 in the VS
glEnableVertexAttribArray(3);  // Vertex Temper   -> layout 3 in the VS
glEnableVertexAttribArray(4);  // Vertex Speed    -> layout 4 in the VS
...
```

```
//Recebimento dos parametros no programa de vertices (código GLSL)
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColor;
layout (location=2) in vec3 VertexNormal;
layout (location=3) in vec3 VertexTemperature;
layout (location=4) in vec3 VertexSpeed;
```

5.2.4. Tipos primitivos

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. Deve-se usar apenas o nome.

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 não pode ser alterada
mat2 m = mat2(1);   //valor replicado para cada elemento da diagonal principal
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 v1 = vec4(1, 2, 3, 4);  //construtor
vec4 v2 = vec4(1, 2, 3, 4);
v2 = v1.xxyy;
v2 = v1.wxyz;
v2.wx = vec2(5.0, 6.0); //mantem as demais componentes inalteradas

vec4 tmp = v1 + v2;
tmp = v2 + f; //f é escalar. Faz soma com cada componente.
f = tmp[2];   //indexação em matriz e vetor
tmp--; //decrementa cada componente

mat2 m = mat2(1); //somente a diagonal principal recebe 1. O resto recebe 0
  //1 2
  //3 4
m[0][0] = 1;  //comandos para montar a matriz acima
m[1][0] = 2;
m[0][1] = 3;
m[1][1] = 4;
vec2 v = m[0];  //a indexação é por coluna, ou seja, v.x = 1 e v.y = 3
```

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.

5.2.5. Funções built-in

Pode haver sobrecarga de funções, desde que os parâmetros de entrada sejam diferentes. Quase todas 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:

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 ruído 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; //não pega o alpha
cor = clamp(cor, 0.0, 1.0); //os limites 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);
```

6. Bibliotecas auxiliares

O OpenGL cresceu com diversas bibliotecas que estenderam suas funcionalidades, tornando-o uma das APIs gráficas mais utilizadas hoje em dia. Abaixo são listadas as mais populares, que também são utilizadas nos códigos exemplo.

6.1. OpenGL Extension Wrangler (GLEW)

A GLEW (http://glew.sourceforge.net/) oferece acesso às funções do OpenGL, sendo ela responsável pela ligação em tempo de execução dos ponteiros de funções das versões mais novas do OpenGL e suas extensões, uma vez que o Windows é congelado na versão 1.1. Sem ela, os usuários teriam de registrar manualmente as funções, o que é um trabalho longo e passível de erros.

6.2. GLFW

O OpenGL não oferece sistema de janelas nem eventos, precisando de bibliotecas auxiliares para tal. Dessa maneira, o código OpenGL em si se mantém indiferente ao Sistema Operacional, já que o sistema de janelas é abstraído para uma biblioteca auxiliar. A GLFW (http://www.glfw.org/) é uma biblioteca multiplataforma que oferece sistema de janelas e eventos (teclado, mouse, etc) para o OpenGL, oferecendo uma interface simples de funções.

6.3. OpenGL Mathematics (GLM)

A GLM (http://glm.g-truc.net/) é uma biblioteca que oferece a implementação de estruturas de dados de vetores e matrizes bem como operações entre eles. Com a evolução do OpenGL, foi passado para o programador o gerenciamento de matrizes de transformação (GL_MODELVIEW e GL_PROJECTION). A biblioteca foi pensada para ser utilizada com shaders, já que utiliza uma sintaxe similar aos programas escritos em GLSL, sendo uma das mais utilizadas atualmente.

6.4. FreeImage

A FreeImage (http://freeimage.sourceforge.net/) é uma biblioteca auxiliar para a leitura e importação de imagens. Ela oferece uma interface simplificada que permite o carregamento de diversos formatos de imagens, se tornando útil no dia-a-dia e eliminando a necessidade de loaders específicos para cada formato.

6.5. A classe GLSLProgram

Nos códigos exemplo dessa apostila, a utilização dos programas shader foi simplificada através da utilização da classe GLSLProgram. Ela fornece uma camada que de abstração que permite a rápida criação, compilação e ligação de shaders. A criação do shader é feita utilizando a função `compileShader` que recebe o nome do arquivo e o tipo do shader desejado, o código pode ser visto no exemplo abaixo:

```
GLSLProgram compileAndLinkShader()
    GLSLProgram prog;
    prog.compileShader("shader/file_name.vs", GLSLShader::VERTEX);
    prog.compileShader("shader/file_name.fs", GLSLShader::FRAGMENT);
    prog.link();
    prog.use();
    return prog;
}
```

A função `link()` prepara o objeto shader compilado para a execução. Enquanto o método `use()` o configura como programa atual, sendo que é permitido somente um programa executando por vez. A alternância entre programas shader pode ser feita usando esse método, porém essa troca é custosa e não deve ser feita levianamente. Engines modernas ordenam os objetos por shader e assim minimizam a troca (ex. renderizar todas as árvores, depois todos os personagens, etc.).

Uma vez o programa shader ativo, tipicamente é necessário o envio de variáveis para a GPU (além dos buffers de dados), esses valores são **uniformes** em todos os estágios do pipeline, e são enviados através do método `setUniform` da classe GLSLProgram. Os usos mais comuns de variáveis uniformes são:

- Matrizes de transformação (model, view, projection);
- Tempo;
- Posição e configuração da luz;
- Material do objeto;
- Variáveis de controle em geral;

Um exemplo de chamada é a de envio da matriz de mode-view-projection para a GPU, assumindo que ela foi calculada previamente. Antes de renderizar é preciso enviá-la para o pipeline. A tabela abaixo mostra um exemplo de envio. Note que a string passada como parâmetro no setUniform deve ser exatamente o mesmo nome da variável utilizada no shader.

Na CPU	Na GPU
`glm::mat4 modelViewProjectionMatrix;` `shader.setUniform("MVP", modelViewProjectionMatrix);`	`uniform mat4 MVP;`

O método setUniform recebe o nome da variável que está sendo atribuída e o valor. O método possui várias sobrecargas, permitindo envio de matrizes e vetores (ex: `vec3`) de diversos tipos e tamanhos. Para mais detalhes sobre a compilação, ligação do programa e envio de variáveis uniformes, consulte a classe `GLSLProgram` nos programas exemplo da apostila.

7. Exemplos

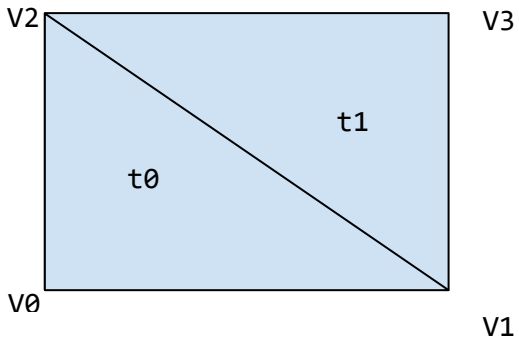
Este capítulo descreve alguns dos exemplos encontrados nos demos da cadeira de computação gráfica avançada. O projeto e todo o código fonte podem ser encontrados em [7].

7.1. Plano

Este simples exemplo demonstra como desenhar um quadrado utilizando o OpenGL 4.0. Na classe Plane podemos encontrar o ponto de partida. O método `Plane::init()` faz toda a inicialização necessária. Ela pode ser dividida em três passos:

- Geração do plano
- Envio dos dados para GPU
- Compilação dos shaders

Nós iremos definir um plano como um conjunto de dois triângulos, `t0` e `t1`. Como os dois triângulos compartilham dois vértices, precisamos de apenas quatro vértices, `v0`, `v1`, `v2` e `v3`.



Primeiramente precisamos gerar os vértices que formam o plano. Estes serão armazenados no vector `vertices`. Para cada vértice, também será gerado um atributo cor que será armazenado em `colors`. Esses valores serão enviados para GPU que no estágio Fragment Shader irá calcular a cor de cada fragmento baseado nas cores dos vértices.

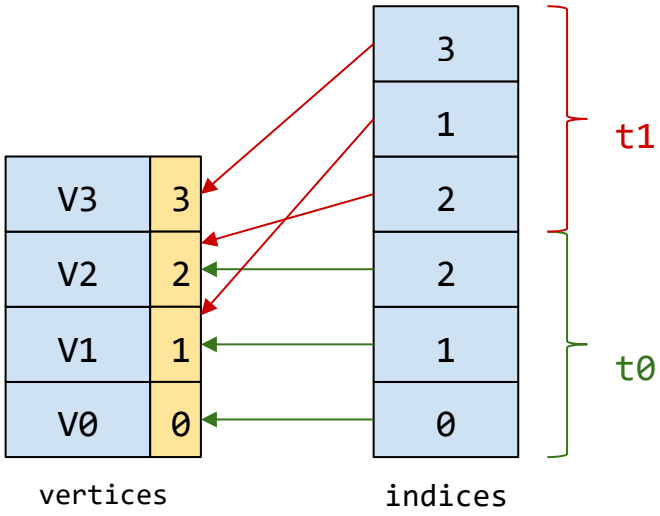
```
void Plane::genPlane(){
    // v0 -- bottom left
    vertices.push_back(vec3(-size, -size, 0.0f));
    colors.push_back(vec3(1.0f, 0.0f, 0.0f));
    //v1 -- bottom right
    vertices.push_back(vec3(size, -size, 0.0f));
    colors.push_back(vec3(0.0f, 1.0f, 0.0f));
    //v2 -- top left
    vertices.push_back(vec3(-size, size, 0.0f));
    colors.push_back(vec3(0.0f, 0.0f, 1.0f));
    //v3 -- top right
    vertices.push_back(vec3(size, size, 0.0f));
    colors.push_back(vec3(1.0f, 1.0f, 0.0f));
```

No momento dispomos de todos os vértices necessários, mas não existe nenhuma ligação entre eles. Para que possamos desenhar o plano, precisamos ligar os vértices para gerar os triângulos `t0` e `t1`.

```
    // triangle 0
    indices.push_back(0);
    indices.push_back(1);
    indices.push_back(2);

    // triangle 1
    indices.push_back(2);
    indices.push_back(1);
    indices.push_back(3);
}
```

Para isso vamos gerar índices que irão dizer para GPU quais vértices contidos em `vertices` formam os dois triângulos e armazená-los em `indices`.



Após a geração dos dados que formam o plano, é necessário enviar essa informação para a GPU. O método `Plane::genBuffers()` se encarrega desta tarefa.

Sempre que possível iremos utilizar VAOs para armazenar o estado dos buffers e seus atributos, logo, a primeira tarefa a ser feita é gerar e vincular um VAO. Neste exemplo, `vaoID` é a única variável global que será usada no método render para ativar o VAO.

Com o VAO vinculado, agora podemos gerar os buffers objects que irão armazenar os dados na GPU. Como devemos enviar três arrays diferentes (vértices, cores e índices) para a memória da GPU iremos gerar três buffers.

Para enviarmos os dados de cada array precisamos primeiro vincular um buffer utilizando a função `glBindBuffer`. Em seguida a função glBufferData é utilizada para em fim enviar os dados do array para a memória da GPU. A função `glVertexAtribPointer` irá definir algumas características dos dados contidos no buffer, como por exemplo, o índice do vetor de atributos, o número de componentes de cada elemento do array (1, 2, 3 ou 4) e o tipo dos dados (float, int, boolean, etc) do array. A função `glEnableVertexAttribArray` é então utilizada para habilitar o índice de atributo que o buffer está utilizando. Se o buffer não for habilitado, os dados não serão processados.

O mesmo é feito para os arrays `colors` e `indices`. No entanto, por se tratar de um GL_ELEMENT_ARRAY_BUFFER, `indices` não será referenciado em um shader, mas sim utilizado pela GPU para definir quais vértices cada triângulo é formado. Por esse motivo não é necessária chamada das funções `glVertexAtribPointer` e `glEnableVertexAttribArray` para o buffer `indices`. Ao fim o VAO é desvinculado através da chamada `glBindVertexArray` com o valor `0`.

```
void Plane::genBuffers()
{
   glGenVertexArrays(1, &vaoID);
   glBindVertexArray(vaoID);

   unsigned int handle[3];
   glGenBuffers(3, handle);

   glBindBuffer(GL_ARRAY_BUFFER, handle[0]);
   glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(vec3), (GLvoid*)&vertices[0], GL_STATIC_DRAW);
   glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL);
   glEnableVertexAttribArray(0);  // Vertex position -> layout 0 in the VS

   glBindBuffer(GL_ARRAY_BUFFER, handle[1]);
   glBufferData(GL_ARRAY_BUFFER, colors.size() * sizeof(vec3), (GLvoid*)&colors[0], GL_STATIC_DRAW);
   glVertexAttribPointer((GLuint)1, 3, GL_FLOAT, GL_FALSE, 0, (GLubyte *)NULL);
   glEnableVertexAttribArray(1);  // Vertex color -> layout 1 in the VS

   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, handle[2]);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(int), (GLvoid*)&indices[0], GL_STATIC_DRAW);

   glBindVertexArray(0);
}
```

Antes de realizar a renderização é chamado o método `Plane::update`, responsável por definir e setar a `modelViewProjectionMatrix`, que é acessada no programa de vértice via a variável uniform MVP.

```
modelMatrix = glm::mat4(1.0f);
viewMatrix  = glm::lookAt(
                vec3(0.0f, 0.0f, -1.0f), //eye
                vec3(0.0f, 0.0f, 0.0f), //center
                vec3(0.0f, 1.0f, 0.0f)); //up
projectionMatrix = glm::perspective(glm::radians(60.0f), 1.0f, 0.1f, 100.0f);

void Plane::update(double deltaTime){

        // matrices setup
        modelMatrix = mat4(1.0f); // identity
        modelMatrix = glm::translate(modelMatrix, planePos); // translate back
        modelViewMatrix = viewMatrix * modelMatrix;
        modelViewProjectionMatrix = projectionMatrix * modelViewMatrix;

        // set var MVP on the shader
        shader.setUniform("MVP", modelViewProjectionMatrix); //ModelViewProjection
}
```

O código para renderização encontra-se no método `Plane::render`. A renderização é feita através de três chamadas de funções do OpenGL. O primeiro passo é vincular o VAO gerado em `Plane::genBuffers` através da função `glBindVertexArray`. Logo após, a função `glDrawElements` é chamada para em fim desenhar as primitivas que já encontram-se na memória da GPU. Ao fim o VAO é desvinculado através da chamada `glBindVertexArray` com o valor `0`.

```
void Plane::render(){
        glBindVertexArray(vaoID);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, (GLubyte *)NULL);
        glBindVertexArray(0);
}
```

Todo código visto até o momento nesta sessão é executado em CPU. Agora vamos analisar os shaders, programas que são executados em GPU, deste exemplo. Este exemplo possui shaders para dois, Vertex e Fragment shaders, dos seis estágios programáveis do pipeline gráfico.

Como visto na seção 'Envio dos dados para GPU', ambos os arrays `vertices` e `colors` são atribuídos um índice. Esse índice também encontra-se no código do shader no qualificador `location` das variáveis de entrada `VertexPosition` e `VertexColor`. Os dados de cada uma dessas variáveis de entrada vêm dos arrays que foram previamente enviados para GPU. Este Vertex Shader possui também uma variável de saída, `Color`, que terá um valor atribuído durante a execução e que estará disponível no próximo estágio do pipeline, no contexto deste exemplo, o Fragment Shader.

```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColor;

out vec3 Color;
uniform mat4 MVP;

void main()// Vertex Shader
{
    Color = VertexColor;
    gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

Este Vertex Shader possui duas funções: transformar o vértice através da matriz MVP (ModelViewProjection) e passar o valor da cor do vértice adiante para o Fragment Shader através da variável de saída `Color`. Deve-se atentar para a ordem que as matrizes são multiplicadas, tanto na função Update, como no **vertex program**. Isso depende se a matriz for definida como **column-major ou row-major** format. Dependendo da solução adotada, deve-se usar uma das seguintes soluções no vertex program:

```
gl_Position = Proj * View * Model * vPosition; // column-major
gl_Position = vPosition * Model * View * Proj; // row-major
```

Como esperado, o Fragment Shader possui uma variável de entrada `Color`. Esta variável possui o valor atribuído pelo Vertex Shader e é utilizada para determinar a cor do fragmento gerado pela GPU através da rasterização do triângulo.

```
in vec3 Color;

out vec4 saida; //variavel que contem cor do fragmento, salva no colorbuffer do framebuffer


void main()
{
   saida = vec4(Color, 1.0); //GL4.6
}
```

7.2. Textura

O exemplo de utilização de textura (glsl40_texture) demonstra uma aplicação simples de textura utilizando shaders. Cria-se um plano com quatro vértices (como no exemplo anterior), mas agora tem-se mais um atributo por vértice, que é a coordenada de textura. O vetor de índices permanece da mesma maneira. O buffer de textura é enviado para a GPU de maneira semelhante aos outros, sendo a única diferença que ele utiliza valores em 2D. A figura abaixo mostra como a textura é mapeada no plano.

Espaço UV utilizado para mapear texturas.

O processo de criação da textura envolve:

- **Leitura do arquivo de imagem**: para tal, utilizamos a biblioteca FreeImage, a qual é capaz de ler diversos formatos de imagem conhecidos;
- **Envio da textura para a GPU:** criamos uma classe auxiliar `TextureManager` para abstrair a interface com o OpenGL.

O processo de envio consiste em habilitar um slot de textura, criar uma textura, habilitá-la e enviar os dados da imagem para a GPU. O processo é executado nas seguintes funções, respectivamente:

```
glActiveTexture(GL_TEXTURE0); //usado para comunicacao com o Shader
glGenTextures(1, &gl_texID); // cria uma textura

glBindTexture(GL_TEXTURE_2D, gl_texID); // seta ela como atual
//send to GPU
glTexImage2D(GL_TEXTURE_2D,     // tipo da textura
             0,                 // Mipmap level (0 sendo o mais detalhado i.e. full size)
             GL_RGBA,           // Formato interno
             imageWidth,        // largura da textura
             imageHeight,       // altura da textura
             0,                 // borda em pixeis
```

```
            GL_BGRA,          // formato dos dados
            GL_UNSIGNED_BYTE, // tipo dos dados da textura
            textureData);     // dados da imagem (vetor de cores)
```

A função `glActiveTexture` recebe um enum, o qual pode ser utilizado para simplificar o processo de envio de identificador de textura. Nesse caso, a textura pode ser acessada no shader utilizando a seguinte declaração:

```
in vec2 TexCoord; // Coordenada de textura enviada pelo FS

layout (binding = 0) uniform sampler2D Texture0; // binding = 0 eh equivalente a GL_TEXTURE0.
layout (binding = 1) uniform sampler2D Texture1; // binding = 1 eh equivalente a GL_TEXTURE1.
layout (location = 0) out vec4 FragColor;


void main()
{
        //Amostra as texturas. TexCoord foi enviado pelo VS explicitamente
        // e interpolado para cada fragmento automaticamente.
        vec4 col1 = texture(Texture1, TexCoord);
        vec4 col2 = texture(Texture0, TexCoord);

        //Interpola linearmente as duas cores utilizando a coordenada X da textura
        //ou seja, linearmente da esquerda para a direita.
        FragColor = mix(col1, col2, TexCoord.x);
}
```

7.3. Tessellation Basic

Esse demo demonstra como o tessellation é feito utilizando quads. Para começar, temos que definir o número de vértices por PATCH (veja seção 4.2). Lembrando que o pacth pode ser qualquer tipo de informação que o desenvolvedor queira mandar para o tessellation, desde que nos estágios de TCS e TES sejam interpretados de alguma maneira. Por exemplo, podemos mandar 4 vértices e interpretá-los como os pontos de controle de uma curva de Bézier. Nesse exemplo, o patch tem 4 vértices, e a para desenhar a geometria tem-se uma chamada um pouco diferente:

```
glBindVertexArray(vaoID);
glPatchParameteri(GL_PATCH_VERTICES, 4);
glDrawElements(GL_PATCHES, indices.size(), GL_UNSIGNED_INT, (GLubyte *)NULL);
glBindVertexArray(0);
```

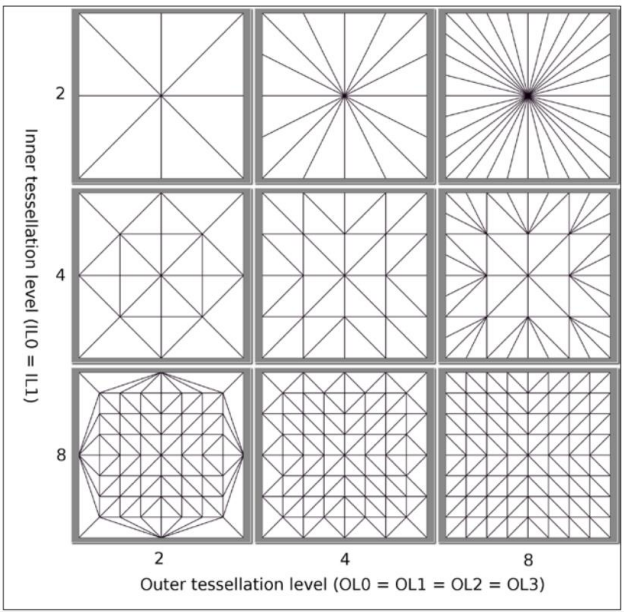
A segunda linha configura o número de vértices por patch, enquanto o terceiro efetua a chamada na GPU para o desenho. Note que até então na função `glDrawElements` tínhamos

triângulos como primitivas, no entanto o tessellation utiliza um tipo especial, que é o `GL_PATCHES`.

Uma vez feita achamada na CPU, vamos para a GPU. No vertex shader passa-se o vértice não modificado, já que queremos a posição do vértice no mundo para fazermos o refinamento. O estágio seguinte é o TCS que tem por atribuição definir os níveis de refinamento dos patches, sendo que os quads tem 2 níveis internos (`gl_TessLevelInner`) e 4 externos (`gl_TessLevelOuter`).

O demo permite o usuário controlar todos os níveis através das teclas (veja no demo). A figura X mostra como os diferentes níveis de tessellation influenciam na geração dos vértices. As GPUs atuais têm unidades especializadas em tessellation, que recebem esses parâmetros e produzem como saída um conjunto de vértices.

Após a geração dos vértices, é hora de posicioná-los no mundo. O estágio de TES é responsável por isso e recebe como entrada as coordenadas normalizadas (de 0-1) dos vértices gerados. Esse estágio tem acesso a todos os vértices do patch original também, permitindo que façamos uma interpolação entre os novos (que estão normalizados) e os originais para determinarmos a posição dos novos vértices no mundo. Finalmente, transformamos os vértices para coordenadas da câmera aplicando a transformação de model-view-projection.



Níveis de tessellation internos e externos em um quad

7.4. Displacement Mapping

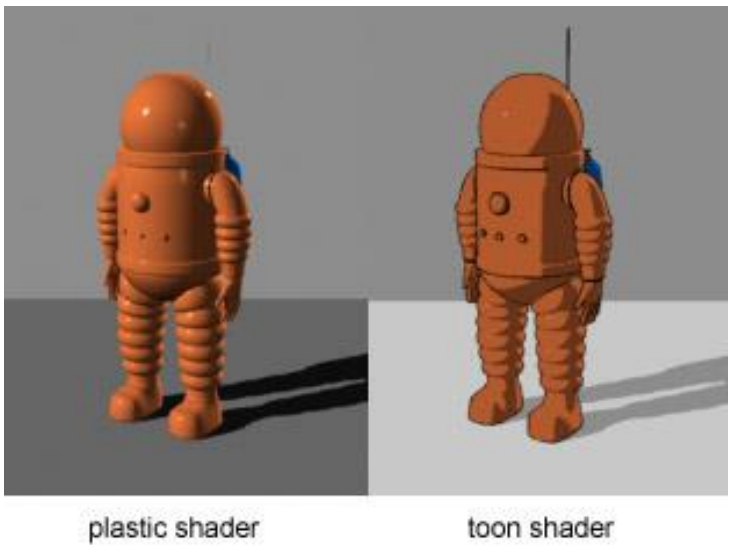
Displacement mapping é a abordagem moderna do bump mapping. Ao contrário do bump mapping onde o efeito é aplicado somente na iluminação, o displacement mapping modifica a própria geometria. Ela normalmente utiliza o tessellation e gera detalhes na malha de acordo com a textura de displacement. Utiliza-se uma textura de displacement para deslocar os vértices. A figura abaixo mostra um comparativo do resultado gerado.

Comparativo entre bump mapping e displacement mapping

O demo efetua o refinamento da malha no TCS - esse pode ser controlado pelas teclas W e S). O TES aplica o displacement amostrando a textura de displacement, e somando o valor do pixel na componente z do plano. As teclas Q e A podem ser utilizadas para a rotação do plano.

7.5. Toon

Toon shading é um modelo de iluminação não fotorealística utilizado para realçar as silhuetas e dar aspecto de cartoon ou de desenho manual para objetos 2D ou 3D. A figura abaixo mostra o efeito alcançado.



Efeito gerado pela toon shading

Para alcançar esse efeito deve-se no fragment shader aplicar diferentes intensidades de acordo com o ângulo de incidência da luz no objeto. Por exemplo, se o ângulo for entre determinadas faixas de valores, aplica-se uma cor no objeto. O exemplo pode ser visto no código abaixo.

```
intensity = max(dot(LightDir, Normal), 0.0);
if (intensity > 0.98)
	color = vec4(Color,1.0) * vec4(0.9,0.9,0.9,1.0);
else if (intensity > 0.5)
	color = vec4(Color,1.0) * vec4(0.4,0.4,0.4,1.0);
else if (intensity > 0.25)
	color = vec4(Color,1.0) * vec4(0.2,0.2,0.2,1.0);
else
	color = vec4(Color,1.0) * vec4(0.1,0.1,0.1,1.0);
```

Note que a normal de cada vértice é passada do vertex shader para o fragment shader, e no processo de rasterização seu valor é interpolado, assim temos uma normal por pixel.

7.6. Point Sprites

Esse demo ilustra o uso do geometry shader para a geração de sprites coloridos. O GS pode ser utilizado para gerar vértices, porém ele é muito mais lento que o tessellation, mas para a geração de quantidades pequenas de geometria, ele é muito útil.

Nesse demo, gera-se uma lista de pontos (`GL_POINTS`) na CPU e envia-os para a GPU, onde no geometry shader eles são recebidos ponto a ponto utilizando a diretiva `layout(points) in`. Para cada vértice de entrada, gera-se um quadrado colorido. Note que os quads são gerados em espaço de tela, e não no mundo. Além disso, passa-se a cor do quad para o fragment shader.

8. Referências

[1] https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units
[2] https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
[3] https://en.wikipedia.org/wiki/Intel_HD_and_Iris_Graphics
[4] https://www.opengl.org/wiki/
[5] https://www.opengl.org/sdk/docs/man4/
[6] https://www.khronos.org/files/opengl45-quick-reference-card.pdf
[7] http://www-usr.inf.ufsm.br/~pozzer/disciplinas/cga_demos.rar
[8] Cozzi, Patrick, and Christophe Riccio, eds. *OpenGL Insights*. CRC press, 2012.
[9] Wolff, David. *OpenGL 4.0 shading language cookbook*. Packt Publishing Ltd, 2011.

Exercícios Práticos em Laboratório

1. Estudar os demos de GL3.0
2. Fazer deformação da geometria via vertex shader, ao longo do tempo, das coordenadas do plano (demo plane)
3. Criar uma textura procedural qualquer e aplicar sobre um quadrado (alvos com círculos concêntricos e miolo preenchido)
4. Com texturas:
    a. Fazer o blending (*morphing*) de duas texturas com interpolação paramétrica entre ambas. Usar o parâmetro *t* para alternar suavemente entre uma imagem e outra;
    b. Dadas duas imagens, mostrar elas em tiras: 100 pixels da primeira, os 100 próximos da outra, e assim por diante. Fazer de duas formas:
        i. Usando coordenadas de textura
        ii. Usando coordenadas da geometria (ver gl_FragCoord).
    c. Simular uma linha que gira no centro da tela. De um lado da linha mostrar uma textura e do outro outra.
    d. Transformar a imagem em tons de cinza (luminância) e recurso para clarear e escurecer uma imagem, com controle pelo mouse; Outra opção é transformar de RGB em luminância de forma contínua.
    e. Borrar uma imagem, como em uma fotografia desfocada;
    f. Mostrar a textura somente nas proximidades do mouse (círculo), com um degrade. O resto da tela deve ser preto. Faça uma variante onde a imagem é mostrada em cores na proximidade do mouse e no resto em tons de cinza;
    g. Implementar uma lupa que segue o mouse sobre uma imagem. Defina um fator de escala de ampliação ou redução;
5. Simular toon shading sem usar fontes luminosas;
6. Simular bump mapping sob superfícies planas e orientadas em eixos com um *normal map*;
7. Simular bump mapping em qualquer superfície;
8. Simular uma lanterna sem usar os recursos de iluminação do OpenGL.
9. Modificar o demo de tesselation