COMPARAÇÃO ENTRE OPENGL MULTIPLATAFORMA E METAL NATIVO DE UM SISTEMA DE RENDERIZAÇÃO 3D SIMPLES EM TEMPO REAL

Alex Juliano Nascimento (IC) e Pedro Henrique Cacique Braga (Orientador)

Apoio:PIVIC Mackenzie

RESUMO

Neste artigo é comparada a performance de um sistema de renderização 3D simples em tempo real, com uma instância em OpenGL e outra em Metal, a nova API gráfica da Apple, anunciada em 2014. Com a chegada dessa nova API, o suporte para OpenGL foi depreciado nas plataformas da Apple sob o pretexto da nova tecnologia ser mais performática, por ser uma camada mais fina de abstração em cima da GPU. Ao final, Metal apresentou um resultado de 40% a até 60% de melhor performance quando comparado com a aplicação em OpenGL nas cenas mais complexas, no entanto as cenas mais simples mostraram um resultado similar devido ao teto de 60 quadros por segundo.

Palavras-chave: OpenGL. Metal. Shaders.

ABSTRACT

This paper compares the performance of a simple real time 3D rendering software with an instance in OpenGL and another in Metal, Apple's new graphics API announced in 2014. With the arrival of the new API, OpenGL support was deprecated on Apple's platforms by the pretext of the new technology presenting better performance overall, by being a thinner layer of abstraction on top of the GPU. In the end, Metal presented an advantage on performance of 40% up until 60% in comparison to the OpenGL application on the more complex scenes, however on the simpler scenes there was a similar result given the roof of 60 frames per second.

Keywords: OpenGL. Metal. Shaders.

1. INTRODUÇÃO

A substituição do OpenGL pelo Metal nas plataformas da Apple fez com que os desenvolvedores tivessem que adaptar seus programas para a nova tecnologia.

Por um lado, perde-se a compatibilidade multiplataforma fornecida pelo OpenGL, passando a ser necessária uma versão do programa explicitamente para as plataformas da Apple, o que deixa mais difícil aos desenvolvedores disponibilizarem seus programas para o maior número de usuários possível. Além disso, OpenGL é tido como uma API gráfica de fácil acesso, com muitos recursos disponíveis em livros, blogs, vídeos, etc, e considerada mais simples para iniciantes do que outras APIs gráficas modernas, como Metal.

Por outro lado, Metal é mais performático que OpenGL (APPLE INC. WWDC 2014), o que fornece mais liberdade para os desenvolvedores utilizarem os recursos disponíveis de modo a criarem programas com gráficos melhores. Isso abre precedentes para reinserir as plataformas da Apple no mercado de jogos AAA, competindo de frente com o Direct3D da Microsoft e o Vulkan, a nova API gráfica do Khronos Group.

Com Metal, é possível tirar o máximo proveito da GPU, para atingir o ápice da qualidade gráfica de um dispositivo. Mas qual seria o ganho, em comparação com OpenGL, em um programa de renderização mais simples? A performance ainda seria muito melhor, mesmo não utilizando funções mais especializadas da API? E mesmo sendo melhor, seria algo notável pelo usuário? Essas são algumas das perguntas que esse estudo pretende abordar.

2. REFERENCIAL TEÓRICO

2.1 OpenGL Overview

OpenGL (Open Graphics Library), é uma API multiplataforma voltada para renderização de gráficos 2D e 3D. Foi criada pela Silicon Graphics Inc. em 1992 e hoje é mantida pelo Khronos Group, um consórcio composto por diversas empresas, com enfoque na criação e melhoria de tecnologias abertas voltadas principalmente para computação gráfica e aceleração.

Apesar de ser amplamente tratada como uma API, inclusive pelo próprio Khronos Group, OpenGL é somente uma especificação, que define exatamente qual deve ser o funcionamento e resultado de cada chamada de função. Isso significa que a implementação da API é feita pelos fabricantes das placas gráficas, nos drivers que as acompanham.

É verdade que várias linguagens diferentes podem utilizar OpenGL, mas a especificação foi inicialmente projetada para a linguagem C, o que justifica algumas decisões de sua arquitetura. As chamadas de função muitas vezes requerem um ponteiro para um *unsigned integer*, que então recebe identificador para localizar aquele objeto posteriormente. Isso é uma maneira de criar referências a objetos em uma linguagem que não possui o conceito de objeto, mas que também serve ao propósito de guardar uma referência de um endereço pertencente à VRAM, isto é, à memória da placa gráfica. A natureza procedural da API também se faz clara ao criar e manipular esses objetos.

Diferente dos padrões mais modernos de programação, OpenGL utiliza uma arquitetura de máquina de estado, em que todas as funções chamadas servem para alterar e configurar o estado dessa máquina maior, que é representada pelo *OpenGL context*.

No início, OpenGL era uma API que disponibilizava somente funções para renderizar primitivas e customizar o processo, na tal chamada *fixed function pipeline* (KHRONOS GROUP, Rendering Pipeline Overview). O controle direto dos processos de renderização só veio mais tarde com a versão 3.3, que forneceu a opção ao usuário[1] de escrever seus próprios *shaders*, em um modo chamado *core profile*, também chamado de OpenGL moderno e o que será utilizado neste artigo.

Shaders são nada mais do que programas que são executado pela placa gráfica (DOPPIOSLASH, 2018). Em OpenGL, os shaders devem ser escritos numa linguagem chamada GLSL (OpenGL Shading Language) e compilados e ligados pelo próprio OpenGL, geralmente em tempo de execução.

2.2 Pipeline de Renderização

Como regra geral, todas as APIs gráficas possuem um sistema de renderização, que parte desde a definição dos vértices e suas propriedades até a exibição final da cor de cada pixel. Esse sistema chama-se *rendering pipeline* (dutos de renderização) que é composto por diversas funções cujas saídas são as entradas da próxima função na *pipeline*.

2.2.1 Vertex Specification

A entrada inicial que fornecemos à pipeline são os vértices que irão compor a cena que desejamos renderizar. No entanto, consultar a memória RAM do computador é um processo muito lento para os propósitos de um renderizador, por conta distância física dos

[1] Entenda “usuário” como o programador que utiliza a API.

dispositivos. Por conta disso, as placas gráficas modernas possuem uma memória própria, de acesso muito mais rápido, chamada de VRAM (*video RAM*).

Trecho de Código 1 - Contrução de um VBO

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

Fonte: Autor

O Trecho de Código 1 gera um buffer na VRAM e armazena um identificador na variável VBO (Vertex Buffer Object). Para fins de demonstração, será representado um simples triângulo colorido. Podemos representar e armazenar os vértices do triângulo da seguinte forma:

Trecho de Código 2 - Dados de um triângulo em OpenGL

```
float vertices[] = {
      //   x,     y,    z,       r,    g,    b
        0.0f,  0.5f, 0.0f,    1.0f, 0.0f, 0.0f,
        0.5f, -0.5f, 0.0f,    0.0f, 1.0f, 0.0f,
       -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f
};
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Fonte: Autor

Os vértices são armazenados em um *array* de *floats*, com suas respectivas posições x, y, z e cores r, g, b. Nota-se que se trata de um simples *array*, sem nada que defina como devem ser interpretados os seus elementos, o que será feito mais tarde. Em seguida chamamos a função `glBindBuffer` passando como primeiro argumento o target `GL_ARRAY_BUFFER` e como segundo argumento o identificador do *buffer* criado. Aqui é demonstrada a natureza de máquina de estados do OpenGL, sendo que agora o contexto possui no target `GL_ARRAY_BUFFER` uma referência ao VBO criado. Essa ligação só será desfeita quando outro buffer for ligado ao target ou caso for chamada função `glBindBuffer(GL_ARRAY_BUFFER, 0)`, que desfaz qualquer ligação anterior.

Em seguida são armazenados os dados dos vértices no VBO, mas repare que a chamada de função possui somente o target como argumento e não o identificador do objeto, já que este é referenciado por meio do target. Isso evidencia a natureza de máquina de estados do OpenGL.

Finalmente, é preciso instruir o OpenGL sobre como acessar os dados do buffer:

Trecho de Código 3 - Especificação de atributos de vértice

```
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,6 * sizeof(float),(void*) 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,6  *  sizeof(float),(void*)  (3  *
sizeof(float)));
glEnableVertexAttribArray(1);
```

Fonte: Autor

O trecho de Código 3 especifica que o atributo de índice 0 possui 3 *floats* e um *stride*, em *bytes*, de 6 vezes o tamanho de um *float*. Já o segundo atributo possui índice 1, também possui 3 *floats* e o mesmo *stride*, mas está localizado com um *offset* de 3 floats a partir do início do trecho do *buffer* que corresponde àquele vértice.

A partir desse ponto, o OpenGL já possui as informações necessárias para renderizar o triângulo, mas o *core profile* só permite desenhar vértices com um outro tipo de objeto, chamado VAO (*Vertex Attribute Object*).

O VAO funciona em conjunto com um VBO, mas além de possuir uma referência ao *buffer*, também possui as informações de descrição dos atributos. Dessa forma, não é preciso re-declarar o *layout* dos atributos dos vértices sempre que desenhar um novo objeto.

Trecho de Código 4 - Construção de VAO
```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
```
Fonte: Autor

Qualquer VBO que for ligado com `glBindBuffer` nas linhas seguintes do Trecho de Código 4 será associado a esse VAO, até que seja ligado um outro VAO ou seja chamada a função `glBindVertexArray` com parâmetro 0.

2.2.2 Vertex Shader

Após receber os vértices, a rendering pipeline os encaminha para a próxima etapa, que é o *vertex shader*, um programa que recebe como input cada vértice passado e pode modificá-los, alterando sua posição, cor, ou demais atributos. Para o exemplo de renderização do triângulo, o *vertex shader* somente passará adiante os valores recebidos:

Trecho de Código 5 - Vertex shader passthrough
```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out vec3 Color;
void main()
{
    Color = aColor;
    gl_Position = vec4(aPos, 1.0);
}
```
Fonte : Autor

O trecho de Código 5 é escrito em GLSL, que é baseada em C e possui uma ampla biblioteca de funções matemáticas e funcionalidades específicas para facilitar o trabalho com vetores e matrizes. Nota-se que a penúltima linha cria um `vec4` (vetor de 4 *floats*) com

apenas dois argumentos, um vec3 e um outro float. O GLSL entende que os primeiros três valores do vetor devem corresponder ao primeiro vetor passado e o último valor, ao float no segundo argumento.

Na primeira linha, é preciso especificada a versão do OpenGL e o modo. Em seguida, são declarados os atributos dos vértices, com o seus devidos índices no *layout*. Nesse caso, é recebida a posição e a cor do vértice. Os índices devem ser os mesmos que foram passados na especificação dos atributos do trecho de código 3.

Em seguida, a *rendering pipeline* encaminha o resultado do *vertex shader* a algumas etapas opcionais, como *tesselation* e *geometry shader*, e finalmente ao rasterizador.

2.2.3 Rasterização

A rasterização é o processo de converter uma imagem vetorial para *pixels*, ou seja, para um *bitmap,* geralmente com o intuito de ser exibida em um display. O processo de rasterização em OpenGL é automático e não programável pelo usuário.

2.2.4 Fragment Shader

Após a rasterização, a cena é transformada em fragmentos, que são nada mais do que *pixels* em potencial. Isto é, diferença entre fragmentos e *pixels* é que os primeiros ainda podem ser descartados durante a fase de *depth testing* ou ser modificados durante a fase de *blending*. Também é importante destacar que o *pixel* é um conceito de hardware, uma única unidade que emite luz em um *display* físico, e *fragment* é uma abstração para o uso em *software*.

É no *fragment shader* que são feitos cálculos de iluminação e outros efeitos especiais mais avançados. Já que esse processo ocorre com todos os fragmentos de uma determinada cena, é o que mais exige otimização quando comparado com o *vertex shader*, pois geralmente uma cena terá muito mais fragmentos do que vértices.

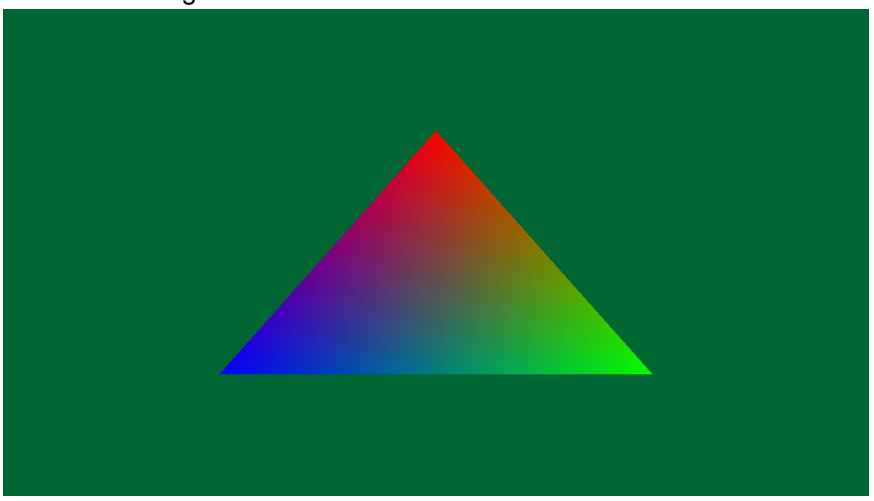
Trecho de Código 6 - Fragment shader passthrough

```
#version 330 core
out vec4 FragColor;
in vec3 Color;
void main()
{
    FragColor = vec4(Color, 1.0);
}
```

Fonte: Autor

O trecho de Código 6 completa a *rendering pipeline*, fazendo o *fragment shader* somente passar a frente a cor recebida. Então é possível renderizar o triângulo na tela:

Figura 1 - Render de um triângulo colorido



Fonte: Autor

A mistura das cores dos vértices, observada na Figura 1, ocorre devido a uma característica do processo de rasterização chamada **interpolação de fragmentos**. Quando o rasterizador transforma as *shapes* em fragmentos, os valores de output dos vértices também são passados aos fragmentos, sendo que os vértices mais próximos do fragmento possuem uma influência maior sobre esses valores do que vértices mais distantes, o que forma o efeito de mistura de cores observado. Demais possíveis valores também são interpolados, como coordenadas UV, vetores de normais, entre outros muitos possíveis atributos que podem estar associados aos vértices.

Após a passagem pelo *fragment shader* ainda há algumas possíveis etapas, como *alpha blending*, *depth testing* e *stencil testing*. Essas etapas tratam os fragmentos em uma mesma posição de modo a determinar qual deles será exibido ou ainda, no caso do *blending*, se uma mistura entre eles será exibida e em qual proporção. Ao final dessas etapas, finalmente é preenchido o *frame buffer* com a imagem final, que deve ser exibido na tela.

2.3 Espaços de coordenadas

Para que o OpenGL, ou qualquer outra API gráfica, possa desenhar um objeto na tela, é preciso que os vértices desse objeto estejam em um espaço chamado *Normalized Device Coordinates* (NDC). Nesse espaço de coordenadas, todos os valores devem estar

entre -1 e 1, mapeados para as extremidades do "dispositivo"[2]. Em OpenGL, isso significa que as coordenadas de um vértice devem estar entre -1 e 1 em todos os eixos, caso contrário ele será descartado antes da rasterização. No caso de um triângulo parcialmente dentro da NDC, ele é automaticamente subdividido em triângulos menores e os triângulos fora da NDC são descartados por completo.

O triângulo renderizado anteriormente se encontra por completo dentro da NDC. No entanto, é muito mais comum especificar um espaço de coordenadas definido pelo próprio usuário e, no *vertex shader*, transformá-lo para NDC. Essa operação é feita com uma série de espaços de coordenadas intermediários, cada um com um propósito específico, e que são transformados no espaço seguinte por meio de uma multiplicação de matrizes. Ao final de todas as transformações, todos os vértices da cena terão sido transformados e aqueles dentro da NDC serão renderizados.

2.3.1 Local Space

O espaço de coordenadas mais simples e o primeiro da série de transformações, é aquele próprio ao objeto. Possui esse nome por estar associado ao próprio modelo 3D e a posição local de seus vértices.

2.3.2 World Space

World Space é o espaço de coordenadas comum a todos os objetos da cena. Nesse espaço, os objetos, que possuem suas próprias origens locais (0,0,0), ficam sujeitos a uma outra origem (0,0,0) comum a todos. Esse espaço é utilizado para definir a composição da cena, efetuando transformações como de translação, rotação e escala, em cada um dos objetos. A matriz utilizada para transformar o *Local Space* em *World Space* é chamada de *Model Matrix*.

2.3.3 Eye Space

Eye Space, também chamado de *Camera Space*, é o espaço de coordenadas representado do ponto de vista da câmera virtual. Esse espaço pode ser representado como uma câmera situada dentro do *World Space*, com uma posição e rotação específicas. No entanto, a transformação que essa câmera imaginária deve efetuar no espaço deve ser inversa àquela que a representa nesse mesmo espaço. Por exemplo, no caso de uma câmera posicionada na posição -5 do eixo z, a transformação que deve ser aplicada ao espaço (*World Space*) deve ser de +5 no eixo z. O mesmo se aplica para rotação. Escala

[2] A NDC não mapeia necessariamente ao limite físico do dispositivo, mas sim à viewport em que o conteúdo será renderizado, que pode ou não ter as mesmas dimensões que a tela do dispositivo.

não é uma operação comum desse tipo de espaço. A matriz utilizada para transformar o *World Space* em *Eye Space* é chamada de *View Matrix*.

2.3.4 Clip Space

Clip Space é o espaço de coordenadas que mais se adequa ao NDC. É nesse espaço que é determinado quais vértices serão **descartados** por estarem além do intervalo de -1 e 1, o que justifica o nome. A transformação para *clip space* faz o mapeamento de um frustum que representa a visão da câmera, para o NDC (AHN, SONG HO. OpenGL Projection Matrix).

É nessa etapa que é efetuado o cálculo de perspectiva, ou de uma projeção ortográfica, fazendo também a compensação de *aspect ratio*. A matriz utilizada para transformar o *Eys Space* em *Clip Space* é chamada de *projection matrix*.

Ao final de todas essas transformações, os vértices que se encontram dentro do NDC serão renderizados na *viewport*. As matrizes de *Model*, *View* e *Projection* podem ser multiplicadas e armazenadas em uma só matriz, chamada *ModelViewProjection Matrix*, ou MVP.

2.4 Iluminação

Os cálculos de iluminação são, historicamente, a razão da existência dos *shaders*. O próprio nome “shader” pode ser traduzido como “sombreamento”, isto é, a diferença de tonalidade de um objeto tridimensional, produzida pela incidência da luz. O sombreamento é, inclusive, o que fornece a ilusão de tridimensionalidade em desenhos 2D, o que também se aplica à computação gráfica, pois, a imagem é projetada em uma tela plana.

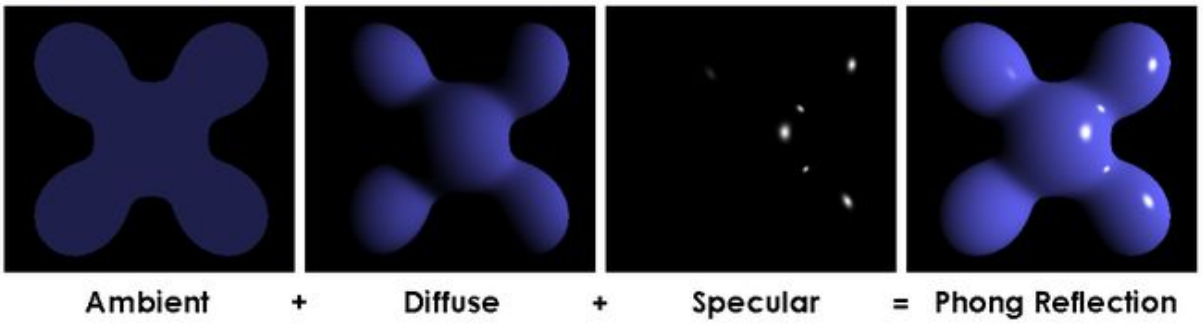
Há diversos modelos de iluminação para computação gráfica mas um dos mais simples, e que produz bons resultados, é chamado de **Phong**, ou ainda, **Phong-Gouraud**.

O modelo Phong é composto com três tipos de iluminação em mente:

- Luz ambiente
- Luz difusa
- Luz especular

O resultado da soma desses três tipos de iluminação é a iluminação completa do objeto, como mostra a figura a seguir:

Figura 2 - Modelo de iluminação Phong



Fonte: Wikipedia, Phong Shading

A luz ambiente possui a função de simular o efeito de *global illumination*, isto é, da luz indireta, proveniente dos raios de luz rebatidos em outras geometrias da cena. No entanto, *global illumination* exige cálculos computacionalmente muito custosos, portanto a luz ambiente Phong é somente uma cor fixa, um vetor RGB que é aplicado igualmente a todo o objeto, para que as partes escuras do objeto não fiquem totalmente pretas.

A luz difusa é a mais responsável pelo efeito de tridimensionalidade no modelo Phong. O cálculo da luz difusa é baseado no ângulo formado pelo vetor de incidência da luz e o vetor normal da superfície. Para obter esse valor, geralmente é feito o cálculo do produto escalar dos dois vetores no *fragment shader*. A esse valor é multiplicada uma cor, que produz o resultado do componente difuso da iluminação Phong do objeto. O produto escalar está relacionado ao ângulo α entre os vetores P e Q pela seguinte equação (LENGYEL, 2012, p.16):

$$P \cdot Q = ||P||\,||Q||\,cos\,\alpha$$

O cálculo final da luz difusa, com uma cor D, de *n* fontes luminosas com intensidades Ci, vetor normal do fragmento N e vetores de incidência Li pode ser feito pela seguinte equação (LENGYEL, 2012, p.162):

$$\kappa diffuse = D \sum_{i=1}^{n} Ci\ max(N \cdot Li,\ 0)$$

A luz especular é usada para representar os focos de luz refletidos pela superfície. O cálculo da luz especular é baseado no ângulo formado pelo vetor de reflexão da luz na superfície, R, e o vetor de direção da câmera (ou olho), V. É importante notar que somente a luz especular é influenciada pela posição do observador, sendo essa informação irrelevante no cálculo da luz difusa. O tal ângulo costuma ser calculado, assim como a luz difusa, com o produto escalar dos dois vetores. O resultado é então elevado à potência de um parâmetro m, responsável pela especularidade (quanto mais alto esse valor, menores

os pontos de luz, maior a percepção do material como liso em oposição a rugoso) e o resultado é multiplicado por uma cor S e a intensidade luminosa C para produzir o resultado final do componente de iluminação especular (LENGYEL, 2012, p.163).

$$SC\ max(R \cdot V,\ 0)^{m}\ (N \cdot L > 0)^{3}$$

O modelo de iluminação Phong pode ser aplicado em um *fragment shader* em GLSL da seguinte maneira:

Trecho de código 7 - Phong em GLSL

```
#version 330 core
in vec3 Normal;
in vec3 WorldPos;
out vec4 FragColor;
uniform vec3 cameraPos;
uniform vec3 lightColor;
uniform vec3 lightDirection;
struct Material
{
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
	float shininess;
};
uniform Material material;

void main()
{
	vec3 ambient = material.ambient;
	vec3 lightDir = normalize(lightDirection);
	float nDotL = max(dot(Normal, -lightDir), 0.0f);
	vec3 diffuse = nDotL * material.diffuse;
	vec3 r = reflect(lightDir, Normal);
	vec3 camDir = normalize(cameraPos - WorldPos);
	float vDotR = max(dot(camDir, r), 0.0f);
	vec3 specular =pow(vDotR,material.shininess)*material.specular;
	vec3 color = (ambient + diffuse + specular) * lightColor;
```

[3] A expressão (NL>0) é somente uma booleana para impedir que luz especular seja visível em superfícies de costas para a fonte luminosa.

```
    FragColor = vec4(color, 1.0f);
}
```

Fonte: Autor

2.5 Metal

Metal é uma API gráfica de baixo nível e alto desempenho com baixo *overhead*, desenvolvida pela Apple e revelada na conferência anual da Apple de 2014 (APPLE INC. WWDC 2014). Como o próprio nome sugere, é uma fina camada de abstração sobre o *hardware*, fornecendo ao usuário um grande controle sobre as funcionalidades da GPU, o "metal".

Diferente do OpenGL, Metal utiliza uma arquitetura mais orientada a objetos, podendo ser utilizada somente em duas linguagens principais: Objective-C e Swift; e também uma terceira: Objective-C++; que é um *superset* que inclui ambos Objective-C e C++, fornecendo uma opção mais confortável para aqueles acostumados com C++, que é o caso de uma boa parte dos programadores de gráficos e de games.

Muitos dos objetos presentes na API são definidos e controlados por meio de protocolos. O `MTLDevice` é o protocolo que representa a placa gráfica do dispositivo e quase todos os outros objetos são criados a partir, ou de objetos provenientes, dele. É muito comum utilizar um tipo específico de objeto, chamado *descriptor*, para customizar a construção de um outro objeto. Após a utilização de um *descriptor* ele pode ser descartado ou guardado para uso posterior em uma outra construção.

Metal utiliza um protocolo para renderização chamado `MTLCommandEncoder`, que serve para inserir comandos em um `MTLCommandBuffer`. Diferente do OpenGL, para renderizar um objeto é preciso enviar os comandos de desenho através desse protocolo e então enviar o buffer de comandos à GPU, com o método `commit()`. Somente após a chamada desse método é que a GPU recebe os comandos e passa a executá-los. Isso significa que a renderização só ocorrerá após a chamada do método `commit()` e não dos métodos de desenho, como `drawPrimitives(type:vertexStart: vertexCount:)`, que simplesmente armazenam a instrução no *command buffer* por meio do *command encoder*.

A linguagem de shaders do Metal é a MSL (Metal Shading Language), que é baseada em C++14. O processo de escrever *shaders* é muito parecido com OpenGL, tendo algumas diferenças somente na passagem de atributos e *uniforms*[4] e na criação das

[4] *Uniforms* são dados passados para os shaders de maneira uniforme, ou seja, de modo que todas as threads daquele *shader* recebam os mesmo valores. Em MSL eles são chamados de *constants*

pipelines de renderização, que no caso de Metal não precisam ser compiladas e ligadas explicitamente pois isso é feito internamente por um objeto chamado `MTLLibrary`.

3. METODOLOGIA

Para estudar as diferenças entre ambas as APIs gráficas em um contexto de renderização 3D simples, foram desenvolvidos dois programas: um utiliza OpenGL e outro, Metal. Ambos renderizam uma mesma cena, com um único modelo em várias posições diferentes e iluminação Phong, com uma única fonte de luz direcional e sem projeção de sombras. O modelo escolhido foi o Utah Teapot, icônico modelo da computação gráfica, que inclusive incorporado como forma primitiva em alguns softwares de modelagem 3D, como o 3D Studio Max.

Já que os resultados podem variar conforme a quantidade de elementos a serem renderizados, serão utilizadas 5 cenas com quantidades crescentes de objetos. Os vetores de posição (x,y,z) dos objetos e o vetor de direção de luz directional dessas cenas, podem ser observados na Tabela 1.

Tabela 1 - Descrição das cenas

	Cena 1	**Cena 2**	**Cena 3**	**Cena 4***	**Cena 5***
Luz Direcional	(-1,-1,-1)				
Teapot 1	(0,-1,0)				
Teapot 2		(8,0,-7)			
Teapot 3		(-3,1,10)			
Teapot 4		(2,-6,1)			
Teapot 5		(-10,-7,-20)			
Teapot 6			(8,-9,-4)		
Teapot 7			(5,4,-3)		
Teapot 8			(17,-8,-24)		
Teapot 9			(4,3,-43)		
Teapot 10			(5,-5,-14)		
Teapot 11			(18,9,-19)		
Teapot 12			(-3,5,-27)		
Teapot 13			(14,6,-33)		
Teapot 14			(-9,-3,-6)		

Teapot 15			(12,11,-13)		
Teapot 16			(-9,-11,-9)		
Teapot 17			(-7,-6,1)		
Teapot 18			(-9,-3,-30)		
Teapot 19			(13,3,-9)		
Teapot 20			(0,8,-17)		
Objetos	1	5	20	1000	10000
Vértices Totais:	3245	16225	64900	3245000	32450000
Polígonos Totais:	6320	31600	126400	6320000	63200000

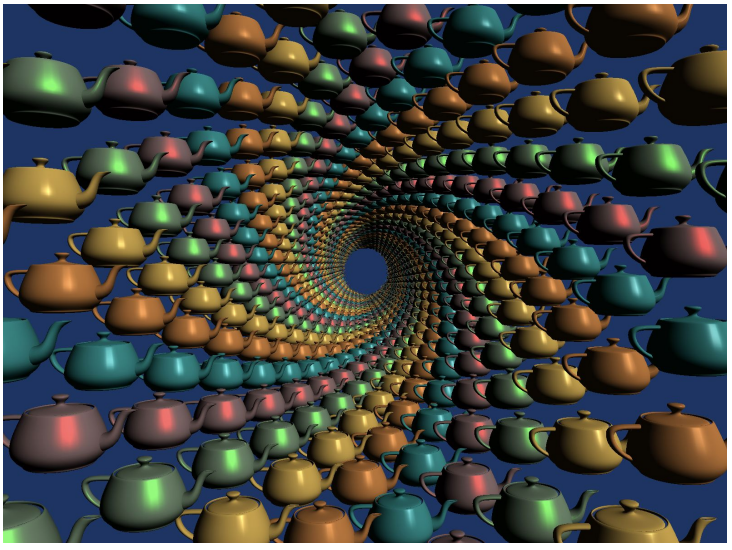
Fonte: Autor

*As cenas 4 e 5 servirão possuem 1.000 (um mil) e 10.000 (dez mil) objetos em cena, posicionados segundo o algoritmo:

$$x = sin(i\alpha)R,\ y = cos(i\alpha)R,\ z = -id$$

O resultado é um cilindro de raio R, com objetos distanciados por um ângulo α nos eixos x e y, e uma distância $-d$ no eixo z.

Figura 3 - Cena 4, cilindro de teapots



Fonte: Autor

Os modelos serão coloridos alternando sequencialmente entre uma lista de materiais. Isso cria uma chamada a mais por objeto por frame, que a CPU deve fazer à GPU.

A renderização dos objetos será feita em um laço simples dentro do *render loop*. Não serão utilizadas técnicas mais avançadas como Instanced Rendering, pois não se enquadraria na proposta de criar um sistema simples de renderização 3D. Apesar disso, as implementações de ambos os sistemas foram feitas seguindo as recomendações de boas práticas e baseadas em exemplos reais das respectivas APIs.

A aplicação em OpenGL foi baseada nos tutoriais do site Learn OpenGL (VRIES, Joey de), um dos mais conhecidos e bem avaliados recursos para iniciantes aprenderem a tecnologia. O site é estruturado de maneira a fornecer tanto a teoria da computação gráfica quanto sua implementação em OpenGL e os porquês das decisões, assim como eventuais dicas para melhorar performance e boas práticas. O resultado é uma aplicação em C++, que utiliza bibliotecas comuns de aplicações OpenGL, como **glm** para matemática de vetores e matrizes, e **GLFW** para criação de janelas e inputs de usuários multiplataforma.

Já a aplicação em Metal foi baseada no livro Metal by Example (MOORE, Warren. 2015), e adaptada de acordo com a documentação oficial da Apple para seguir as boas práticas e comparar com exemplos de implementação fornecidos pela empresa. A linguagem escolhida foi o Swift, já que é a nova linguagem oficial utilizada pela Apple, e também de preferência do autor. No entanto, a grande maioria dos recursos disponíveis sobre Metal estão em Objective-C, inclusive grande parte da documentação oficial.

O dispositivo que será utilizado para fazer a comparação é um MacBook Pro (13-inch, 2018, Four Thunderbolt 3 Ports) e a IDE, o Xcode, com or *targets* em modo *Release* para evitar qualquer tipo de lentidão que o modo *Debug* poderia apresentar.

Para efetuar os testes de comparação, será feita uma contagem de 600 *frames* e o tempo de renderização dos frames será armazenado em uma variável e o resultado da média do tempo de renderização, será impresso ao final.

Os resultados são computados ao longo do processo mas somente são impressos no *standard output* ao final dos 600 frames para evitar qualquer tipo de lentidão em que poderia acarretar imprimir a cada frame renderizado. Lentidão essa que seria multiplicada pelo programa com maior FPS (*frames per second*), possivelmente criando uma comparação não tão confiável entre as aplicações.

O tempo médio de renderização é uma informação mais importante de se observar do que o FPS pois em Metal, o *frame rate* é travado em um teto 60fps e ajustado em intervalos de 10 ou 5 a partir disso. Isso significa que a aplicação OpenGL poderia apresentar valores mais altos de FPS, mas não indica que estaria renderizando aqueles *frames* mais rapidamente. O tempo total para renderizar os 600 *frames* também não será levado em consideração pelo mesmo motivo.

Para uma experiência agradável ao usuário, espera-se um *frame rate* de 60fps, que configura um tempo médio de renderização de aproximadamente 0.016ms. Por conta disso, valores acima de 0.017ms serão considerados lentos, e acima de 0.034ms, muito lentos, pois aí o *frame rate* estaria abaixo dos 30fps, o mínimo para uma experiência estável de renderização em tempo real.

As diferentes cenas no código em OpenGL se encontram no arquivo Scenes.hpp e são separadas por diretivas de pré-processamento, de modo a escolher somente uma delas e evitar desperdiçar espaço na memória. Para rodar a cena escolhida, basta trocar a definição (`#define`) no topo do arquivo `main.cpp`.

As diferentes cenas no código em Metal foram declaradas globalmente, pois variáveis globais em Swift são *lazy* por default (THE SWIFT PROGRAMMING LANGUAGE, Properties). Assim evita-se também o desperdício de memória das cenas que não estariam sendo utilizadas.

O código de carregamento do modelo 3D é compartilhado por ambas as aplicações, para não haver diferenças em performance a esse respeito. Esse código está em C++, sendo que o código em Swift utiliza um *bridging* por meio de Objective-C++, já que não consegue interagir nativamente com C++.

A aplicação em OpenGL utiliza uma técnica de *double buffering*, para apresentar os resultados na tela sem haver artefatos visuais. A aplicação em Metal também possui um mesmo funcionamento pela própria natureza do *command encoder*,:

"When a command queue schedules a command buffer for execution, the drawable tracks all render or write requests on itself in that command buffer. The operating system doesn't present the drawable onscreen until the commands have finished executing" (APPLE INC.)

As janelas de ambas as aplicações terão as mesmas dimensões de 800x600.

Qualquer outra aplicação não necessária para efetuar os testes deve ser fechada, deixando somente o Xcode e a janela de execução das aplicações em observação.

O repositório desse projeto está público na plataforma Github através do link: https://github.com/anascim/Metal_vs_OpenGL.

4. RESULTADO E DISCUSSÃO

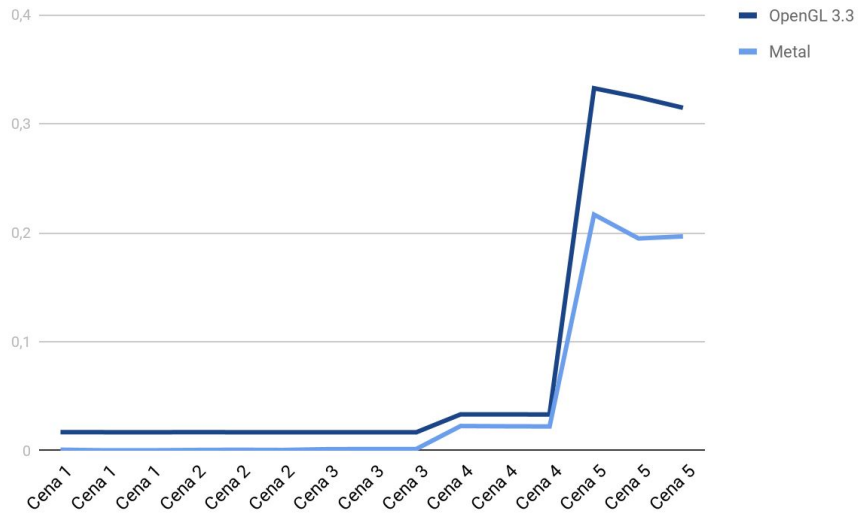
Como pode-se observar na Tabela 2, os tempos de renderização da aplicação em Metal são consistentemente mais rápidos que os da aplicação em OpenGL. Nas primeiras três cenas, o os valores da aplicação em OpenGL variam muito pouco, que pode ser por alguma configuração interna que deixa um teto de 60fps. Devido à implementação da captação desse valor, não é possível obter um valor mais acurado, como é no caso do Metal. No entanto, os valores das cenas 4 e 5 revelam uma diferença significativa em performance, de aproximadamente 45% na cena 4 e até 60% na cena 5, em favor a aplicação em Metal.

Tabela 2 - Tempo médio de renderização de 600 frames, em milisegundos(ms). 3 Tentativas

		OpenGL 3.3	Metal
Cena 1	Tentativa 1	0.0171277	0.0009745724801905454
	Tentativa 2	0.0170485	0.00043496834608959035
	Tentativa 3	0.0169877	0.0004357569560913059
Cena 2	Tentativa 1	0.0171113	0.0007407638839019152
	Tentativa 2	0.0170245	0.0008942987234331667
	Tentativa 3	0.0170339	0.0007366562212700956
Cena 3	Tentativa 1	0.0170219	0.0014704322230924543
	Tentativa 2	0.0170457	0.0016058113975062345
	Tentativa 3	0.0169631	0.0016126827463934508
Cena 4	Tentativa 1	0.0334337	0.022749403617635836
	Tentativa 2	0.0333825	0.022479312078406415
	Tentativa 3	0.0333132	0.022340470559041325
Cena 5	Tentativa 1	0.33255	0.21666659999308951
	Tentativa 2	0.324315	0.1947979013736282
	Tentativa 3	0.314649	0.19666336418430244

Fonte: Autor

Gráfico 1 - Visualização da Tabela 2



Fonte: Autor

Durante o teste de renderização da cena 5, pôde-se observar alguns *logs* informando falha na aquisição do *render pass descriptor*, o que indica uma perda de *frame* por conta de ainda não ter sido liberado pelo *render pass* anterior, que é devido ao alto tempo de renderização exigido por uma cena com muitos polígonos. O Metal então ajusta o *frame rate* e essas perdas de *frame* acabam sendo cada vez mais escassas.

Os testes em OpenGL foram feitos na versão 3.3, pois a diferença em performance comparada com outras versões não se mostrou significativa.

CONSIDERAÇÕES FINAIS

Nesse estudo foi elaborado um projeto para comparar a performance de um sistema de renderização simples entre uma aplicação com back-end gráfico em OpenGL e outra em Metal. Após testar ambas as aplicações com diferentes quantidades de objetos observou-se que a aplicação em Metal foi consistentemente superior em termos de performance, melhorando tempos de renderização em até 60%, comparado com OpenGL.

Algo que não foi abordado neste trabalho foi a nova API gráfica multiplataforma Vulkan, que em muitos aspectos é mais comparável ao Metal, devido a sua arquitetura moderna. No entanto, Vulkan não é compatível com as plataformas da Apple.

Existem bibliotecas, como MotenVK, que traduzem aplicações em Vulkan para Metal tornando possível utilizar a tecnologia nas plataformas Apple, e também a biblioteca MoltenGL faz o mesmo para OpenGL. A análise da eficiência dessas bibliotecas não faz parte do escopo deste estudo, mas poderia ser objeto de estudo de um trabalho futuro.

Também não foram analisadas técnicas mais avançadas de renderização 3D, como *instanced rendering* ou *deferred rendering*, que são muito comuns para otimização de sistemas 3D com grandes quantias de objetos e fontes luminosas, respectivamente.

Além disso, a área de *ray tracing* em tempo real tem se mostrado extremamente promissora, com a chegada das placas gráficas especializadas em *ray tracing*, como a linha GeForce RTX da NVidia, o que abre espaço para possíveis estudos de performance e análise de quando é benéfico, em comparação com técnicas de rasterização.

5. REFERÊNCIAS

DOPPIOSLASH, Claudia. Physically Based Shader Development for Unity 2017: Develop Custom Lighting Systems. Apress, 2018.

VRIES, Joey de. Hello Triangle. Disponível em: <learnopengl.com>. Acesso em:19/07/2020.

MOORE, WARREN. Metal by Example. 2016.

KHRONOS GROUP. Rendering Pipeline Overview. Disponível em: <www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview>. Acesso em: 19/07/2020.

APPLE INC. WWDC 2014 Keynote (Apresentação do Metal em 1:37:20). Disponível em: <www.youtube.com/watch?v=w87fOAG8fjk> . Acesso em: 19/07/2020.

AHN, SONG HO. OpenGL Projection Matrix. Disponível em: <www.songho.ca/opengl/gl_projectionmatrix.html>. Acesso em: 19/07/2020.

PHONG SHADING, In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2020. Disponível em: <en.wikipedia.org/wiki/Phong_shading>. Acesso em: 19/07/2020.

THE SWIFT PROGRAMMING LANGUAGE . Properties. Disponível em: <docs.swift.org/swift-book/LanguageGuide/Properties.html#ID263>. Acesso em: 19/07/2020.

LENGYEL, Eric. Mathematics for 3D Game Programming and Computer Graphics. 3. ed. Course Technology Cengage Learning, 2012.

Contatos: alexjnascim@gmail.com e phcacique@gmail.com