Programação de *shaders* em Computação Gráfica

Caio Magno Aguiar de Carvalho[1], **Daybson Bertassonni Salles Paisante**[1]

[1]Universidade Federal do Maranhão - Campos Codó (UFMA)
Codó – MA – Brazil

daybson.paisante@discente.ufma.br, magno.caio@ufma.br

Abstract. *The objective of this work is to present the relevance of programming shaders in Computer Graphics both for three-dimensional lighting calculations and for computing non-graphic data through GPGPU. It presents the main concepts of Computer Graphics and the use of Linear Algebra to simulate three-dimensional space on the computer. The fundamentals of the use of vectors and operations involving matrices of spatial transformations, such as translation, rotation and scale, are exposed. Lambert, Phong and Blin-Phong shading models are discussed, exposing their mathematical formulation and programming using the Unity game engine.*

Resumo. *O objetivo deste trabalho é apresentar a relevância da programação de shaders em Computação Gráfica tanto para cálculos de iluminação tridimensionais quanto para computação de dados não gráficos através de GPGPU. Apresenta-se os principais conceitos de Computação Gráfica e a utilização da Álgebra Linear para simulação do espaço tridimensional no computador. São expostos os fundamentos do uso de vetores e operações envolvendo matrizes de transformações espaciais, tais como translação, rotação e escala. São abordados os modelos de shading de Lambert, Phong e Blin-Phong, expondo sua formulação matemática e sua programação usando o motor de jogos Unity.*
Palavras-chave*: shader, Computação gráfica, vetor.*

1. Introdução

A busca pela definição de uma área de estudo requer conhecer seus principais problemas e soluções, pois estes são a essência da mesma. Sendo assim, o problema elementar da Computação Gráfica (CG) pode ser resumido em: transformar dados em imagens. Tais dados estão armazenados na memória do computador, e por não haver restrições quanto a origem ou natureza destes dados, qualquer área de pesquisa que se faça uso do computador pode se valer do uso da CG para visualização destes dados para auxiliar na interpretação humana da informação [Velho and Gomes 2015].

De acordo com a natureza e familiaridades dos problemas da CG, pode-se ramificá-la em quatro sub-áreas, a saber: (1) a Modelagem Geométrica, que busca organizar dados geométricos no computador; (2) a Síntese de Imagens, que se encarrega de gerar uma imagem digital à partir de dados armazenados; (3) o Processamento de Imagens, que realiza operações sobre alguma imagem pré existente afim de alterar características visuais (gerando uma nova imagem como resultado); e por fim, (4) a Visão Computacional, que busca obter dados geométricos, físicos ou topológicos à partir de uma ou várias imagens existentes [Velho and Gomes 2015].

Exemplos do uso da CG podem ser percebidos desde uma simples interface gráfica para interação do usuário com o computador ou *smartphone* no cotidiano, exibição de gráficos científicos, projeto e visualização de obras arquitetônica de engenharia, jogos digitais, animações e efeitos especiais para o cinema [Stemkoski and Pascale 2022].

1.1. Exemplos de aplicações na medicina

Na medicina, de acordo com [Kordt et al. 2021], é comum realizar a segmentação (i.e., identificação de regiões) de imagens médicas, como as obtidas por ressonância magnética (RM), para se obter informações sobre o quadro de indivíduos com tumores cerebrais. Tal tarefa exige categorizar cada pixel da imagem manualmente, para identificar tais tumores e avaliar sua evolução e resposta do paciente ao tratamento. Entretanto, isso consome bastante tempo, uma vez que uma RM gera mais de cem secções de imagens bidimensionais (2D) do cérebro, e cada qual deve ser segmentada. A automação dessas tarefas por Aprendizado de Máquina Supervisionado exige grandes quantidades de imagens já categorizadas para gerar bases de treinamento consistentes. Ainda, [Kordt et al. 2021] aponta que diversas pesquisas sobre segmentação de imagens se baseiam no uso de GPUs (*Graphic Processor Unit*) através de abordagens GPGPU (*General Purporse GPU*) ou uso de *shaders*. Em seu trabalho, produziram um sistema semi-automatizado de segmentação de imagens que, ao mesmo tempo em que segmenta automaticamente o volume de crescimento do tumor, também permite ao usuário realizar ajustes sobre as dimensões da área para aprimorar ou corrigir as predições feitas pelo sistema. Tal sistema é executado em navegadores mobile usando a API (*Application Programming Interface*) gráfica WebGL, uma versão portada do OpenGL de desktop para navegadores WEB, capaz de processar dados geométricos e *shaders* via GPU.

No trabalho de [Myronenko 2019], é informado que tais tumores (gliomas), quando malignos, crescem rapidamente, requerem cirurgias e possuem baixa taxa de sobrevivência dos pacientes. A automação da segmentação de imagens poupa tempo, permite diagnósticos mais precisos e gera melhor planejamento das cirurgias. Assim, a aplicação de técnicas de Aprendizado Profundo e Redes Neurais Convulocionais (processadas por GPUs) foram aplicadas em sua pesquisa com grande êxito, tal como ilustrado na Figura 1, onde se vê a imagem segmentada humanamente na parte superior, e abaixo, a segmentação automática, demonstrando excelente taxa de correspondência entre ambas.

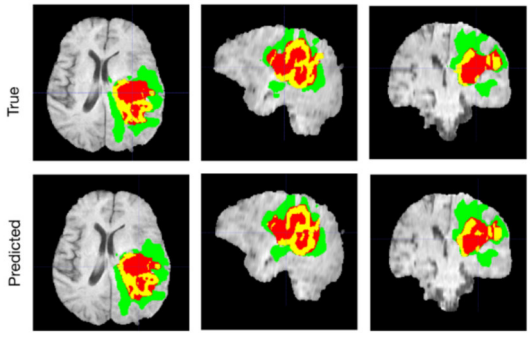


Figura 1. Segmentação real vs. automatizada. Fonte: [Myronenko 2019].

2. Álgebra Linear na Computação Gráfica

A computação gráfica se baseia em diversos ramos da matemática, desde a trigonometria até o Cálculo. Porém, o escopo deste trabalho se limita à apresentação dos fundamentos de Álgebra Linear aplicados na CG para determinação de eixos de sistemas de coordenadas, pontos e vetores e uso de matrizes como operadores para realizar diversas transformações no espaço tridimensional (3D), como translação, rotação e escalonamento [Gregory 2018].

2.1. Vetores

Um vetor pode ser entendido como uma entidade num espaço *n*-dimensional e que possui comprimento e direção, sendo comumente representado por uma seta onde sua cauda representa a origem do vetor no espaço, e a ponta que representa sua direção. De maneira geral, adota-se neste trabalho vetores no espaço $\mathbb{R}^3 = \{(x, y, z)|x, y, z \in \mathbb{R}\}$, exceto quando transformados por matrizes em coordenadas homogêneas (abordado em 2.2.1). Distingue-se também pontos no espaço de vetores: vetores representam uma quantidade de deslocamento a partir de uma origem numa direção, enquanto pontos representam apenas coordenadas neste espaço [Gregory 2018].

2.1.1. Operações com vetores

Um vetor é dito unitário quando seu comprimento possui valor de uma unidade, e um vetor pode ser transformado em um vetor unitário (i.e. normalizado) dividindo cada um de seus componentes por sua magnitude inicial. O vetor unitário é útil em operações onde se interessa apenas na direção do vetor, tais como determinação de direções de fontes de luz [Gregory 2018].

Um vetor normal é o nome dado a um vetor que é perpendicular em relação a alguma superfície, como um plano ou triângulo da malha geométrica (como visto na Figura 2, onde as linhas azuis projetadas no centro das faces da malha são a representação dos vetores normais daquelas faces), e geralmente também são unitários. Tais vetores são extremamente úteis no cálculo da luz incidente em uma malha ao serem comparados com o vetor de direção da luz [Gregory 2018].

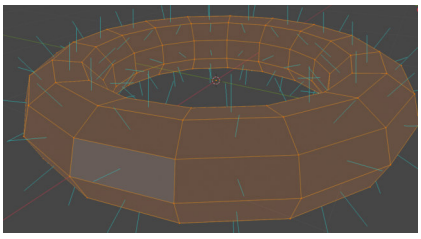


Figura 2. Vetor normal das faces de uma malha no Blender. Fonte: o autor.

Vetores são usados também para construção de uma base ortogonal (a origem do sistema de coordenadas) composta por vetores unitários, que permitem determinar qualquer ponto e vetor nessa base a partir de sua origem apenas multiplicando os valores escalares de cada dimensão pelos valores da base [Gregory 2018].

A Figura 3 exibe algumas das principais operações com vetores, descritas a seguir. A soma entre vetores permite gerar um novo vetor através da adição dos componentes dos vetores iniciais, e conforme visto em a), representa a sequência de deslocamento feita desde a origem do primeiro vetor até a seta do segundo vetor. A subtração entre vetores subtrai os componentes dos vetores, e gerando um novo vetor interpretado como a distância euclidiana entre ambos, apontando para o primeiro vetor da operação, tal qual visto no item b). Um vetor pode ser escalonado de forma uniforme ao ter seus componentes multiplicados por algum valor escalar, e quando multiplicado por -1, sua direção atual é invertida (também interpretado como um giro de 180°). O cálculo do comprimento do vetor pode ser realizado através do Teorema de Pitágoras para o cálculo da hipotenusa do triângulo retângulo. O Produto Interno entre vetores provê um valor escalar que representa uma medida da diferença de direção entre dois vetores, e quando tais vetores são unitários, o valor resultante $v \in \mathbb{R}, |0 \leq v \leq 1$, para $v = -1$: vetores opostos; $v = 0$: vetores perpendiculares; $v = 1$: vetores sobrepostos, tal como percebido em c) onde a linha tracejada divide em dois planos os vetores **p** e **q** de acordo com o sinal do Produto Interno. O comprimento da projeção de dois vetores pode ser obtida através da divisão do Produto Interno entre eles pela magnitude do vetor sobre o qual se deseja projetar, como ilustrado no item d) [Gregory 2018].

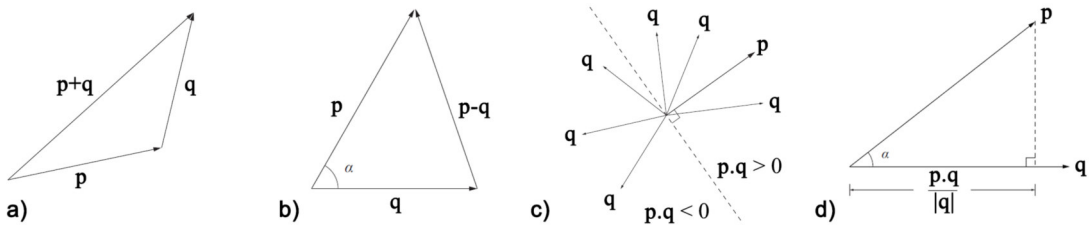


Figura 3. Operações com vetores. Fonte: [Gregory 2018] - adaptado.

Uma aplicação muito utilizada em jogos para economizar tempo de processamento é determinar se há intersecção entre duas circunferências ou esferas, que podem ser usadas para representar uma área aproximada de uma geometria mais complexa. Para isso, basta comparar se a magnitude do vetor de diferença entre ambas circunferências/esferas é menor que a soma do raio das mesmas [Gregory 2018].

2.2. Matrizes e Transformações

Matrizes podem ser entendidas como um *array* de valores numéricos organizados em linhas e colunas $m \times n$, representadas por M_{mn}. Matrizes de transformação são sempre quadradas e possuem valores em índices específicos que, ao serem multiplicadas por um vetor, permitem realizar transformações desse vetor no espaço *n*-dimensional, como rotações, translações ou escalonamento. A multiplicação de todas as matrizes de transformação a serem realizadas permite a sua combinação em uma única matriz final que engloba tais transformações, que pode ser então multiplicada por todos os vértices da geometria para transformá-la no espaço numa única operação [Gregory 2018].

As formas de algumas matrizes comuns são apresentadas a seguir, conforme [Gregory 2018]: para se rotacionar um vetor ao redor de algum eixo escolhido, faz-se uso das matrizes nas formas da equação 1; já a matriz na equação 2 é uma matriz de

escalonamento que permite alterar o comprimento do vetor. A matriz de translação é apresentada adiante usando-se de coordenadas homogêneas na Equação 4.

$$\mathrm{X}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}, \mathrm{Y}(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}, \mathrm{Z}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$\mathrm{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tag{2}$$

Um exemplo de rotação ao redor do eixo *y* é demonstrado na Figura 4, onde vemos um bule no *software* Blender: o bule verde se encontra na posição **p**=(3, 2, 0) e é rotacionado no eixo *y* por 30°, para a posição **p'**=(2.598, 2, -1.5) (ver painel "*Location*"no canto superior direito da imagem). Demonstra-se o mesmo resultado no desenvolvimento da operação 3, onde se multiplica o vetor **p** pela matriz Y de rotação.

$$\begin{bmatrix} 3 & 2 & 0 \end{bmatrix} \cdot \begin{bmatrix} \cos 30 & 0 & -\sin 30 \\ 0 & 1 & 0 \\ \sin 30 & 0 & \cos 30 \end{bmatrix} = \begin{bmatrix} 3\cos(30^\circ) + 2\cdot 0 + 0\cdot\sin(30^\circ) \\ 3\cdot 0 + 2\cdot 1 + 0\cdot 0 \\ 3(-\sin(30^\circ)) + 2\cdot 0 + 0\cdot\cos(30^\circ) \end{bmatrix} = \begin{bmatrix} 2.598 & 2 & -1.5 \end{bmatrix} \tag{3}$$

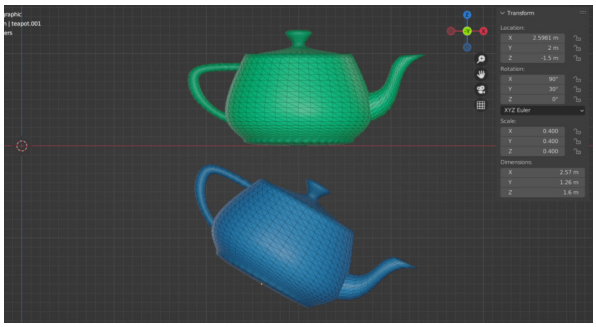


Figura 4. Rotação de um objeto ao redor do eixo *y* no Blender. Fonte: autor.

2.2.1. Coordenadas Homogêneas

As matrizes de transformação assumem a forma M_{33} no espaço tridimensional, porém, para que as transformações lineares possam ser combinadas com a operação de translação (que não é uma transformação linear) numa única matriz, é necessário o uso de Coordenadas Homogêneas. Tais coordenadas adicionam um novo vetor à última linha da matriz M_{33}, gerando uma nova matriz M_{44}, capaz de combinar a translação com as demais transformações lineares, tal como observado na Equação 4. Essa nova representação matricial permite que todas as matrizes sejam multiplicadas entre si, resultando numa única matriz de transformação chamada Matriz Afim, que pode ser multiplicada pelos vértices da geometria para gerar suas novas coordenadas [Gregory 2018].

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & 0 \\ M_{10} & M_{11} & M_{12} & 0 \\ M_{20} & M_{21} & M_{22} & 0 \\ \mathbf{t}_x & \mathbf{t}_y & \mathbf{t}_z & 1 \end{bmatrix} \tag{4}$$

Os vetores também recebem uma coordenada extra para garantir que possam ser multiplicados pelas matrizes afins 4x4, denominada pela letra *w*. Como um vetor pode representar tanto pontos no espaço quanto direções, a coordenada *w* assume valores diferentes para tais casos, sendo: $w = 1$ para pontos e $w = 0$ para direções. Dessa forma, ao se multiplicar por uma matriz, um ponto é afetado pela transformação matricial enquanto um vetor direcional não é, mantendo sua direção original [Gregory 2018]. Um exemplo

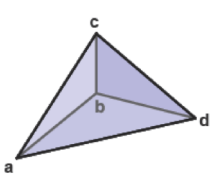
de multiplicação de vetor posicional **r** por uma matriz de translação (onde ocorre alteração de posição do novo vértice) é exemplificado na Equação 5:

$$\begin{bmatrix} \mathbf{r}_x & \mathbf{r}_y & \mathbf{r}_z & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathbf{t}_x & \mathbf{t}_y & \mathbf{t}_z & 1 \end{bmatrix} = \begin{bmatrix} (\mathbf{r}_x + \mathbf{t}_x) & (\mathbf{r}_y + \mathbf{t}_y) & (\mathbf{r}_z + \mathbf{t}_z) & 1 \end{bmatrix} \tag{5}$$

3. Principais conceitos de Computação Gráfica

3.1. Vértices e geometrias

O modelo mais elementar para armazenar informações de um objeto geométrico é o vértice, que contém as coordenadas *n*-dimensionais de algum ponto no espaço, geralmente descrito numa estrutura de dados de vetor. A superfície de um objeto 3D é composta por uma malha de triângulos que compartilham vértices de forma eficiente através de índices na memória, já que triângulos vizinhos reutilizam ao menos dois vértices em comum. Para se compor um triângulo, é necessária a existência de três vértices com posições diferentes no espaço. Vértices também podem armazenar outras estruturas de informação como coordenadas de texturas, materiais e vetor normal, por exemplo. A Figura 5 exemplifica uma malha de quatro vértices (a, b, c, d) que formam três triângulos, e o gráfico ao lado indica o índice (símbolo '#') do vértice e sua posição no espaço e o índice do triângulo e o índice de quais vértices que o compõem [Marschner and Shirley 2021].



shared vertices:

triangles	
#	*vertices*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

vertices	
#	*position*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Figura 5. Exemplo de malha geométrica. Fonte: [Marschner and Shirley 2021] - adaptado.

3.2. Renderização

Renderização consiste na geração de imagens 2D à partir de cenas 3D, que podem conter vários objetos com malhas de vértices simples como esferas e cubos, ou mais sofisticadas, como personagens, carros e florestas, criados em *softwares* de modelagem 3D artística como Blender, Maya e ZBrush. Tais objetos podem possuir desde cores sólidas até texturas e materiais que ditam como seu aspecto visual será simulado através dos *shaders* associados a eles. Sombras também podem ser projetadas de acordo com as fontes de luzes [Stemkoski and Pascale 2022].

3.3. A imagem digital

A renderização de uma cena 3D produz um *array* de pixels (contração de *picture element*) chamado *raster*, que será exibido em algum tipo de matriz 2D, como um monitor, ou gravado em um arquivo de imagem. Os pixels são modelos computacionais da cor natural, geralmente processados em valores ponto flutuante durante o processo de renderização, podendo ser descritos como $p \in \mathbb{R}, |0 \leq p \leq 1$, sendo 0 ausência de cor e 1 a intensidade total da cor. Cada pixel é descrito como um *array* de três valores usados para representar as três frequências luminosas percebidas pelas células cones presentes no olho humano:

vermelho, verde e azul (*red, green, blue* - RGB), respectivamente. A variação de intensidade em cada canal RGB é responsável pela percepção das demais cores, tais como amarelo (1, 1, 0) e cinza (0.5, 0.5, 0.5 [Stemkoski and Pascale 2022].

É possível existir ainda um canal adicional *alpha* para dar suporte a imagens com transparência, geralmente gravadas em arquivos com extensão '.png'. Existem modelos de cores que se baseiam em outros parâmetros para a formulação da cor, como o modelo CMYK, que ao contrário do RGB (que é um modelo aditivo de cores para dispositivos que emitem luz), trata-se de um modelo subtrativo de cores baseado na absorção da luz, utilizado para impressão de imagens em papel [Breckon and Solomon 2013].

Uma imagem digital possui dois atributos que influenciam sua qualidade, a saber: (1) a resolução espacial, que é a dimensão $m \times n$ da matriz 2D da imagem, determinando assim a sua quantidade total de pixels; (2) a quantização de cor, que é a quantidade N de bits utilizados por cada canal de cor para representar as variações de cores possíveis, dada por 2^N [Breckon and Solomon 2013]. Ou seja, em um *display* de 24 bits (8 bits para cada canal RGB, gerando $2^8 = 256$ valores distintos por canal) é possível representar $2^{24} = 16.777.216$ milhões de cores. O tamanho de arquivo sem compressão que uma imagem RGB de 24 bits ocupa é dado pela $largura * altura * N$, logo, uma imagem em resolução *full hd* ocupa $1920 * 1080 * 24 = 49.766.400 bits$, ou aproximadamente $6.22 MB$ (*mega bytes*), o que ressalta a importância de técnicas de compressão de imagens para armazenamento e transmissão em rede. Na Figura 6 em I) pode-se observar a imagem em a) sofrendo redução da quantização de bits, perdendo tons de cinza intermediários em b) até se tornar completamente preto e branco em c); e em II) a mesma imagem sofre agora redução de resolução espacial, possuindo menor quantidade de pixels em b), embora mantenha sua quantização original.

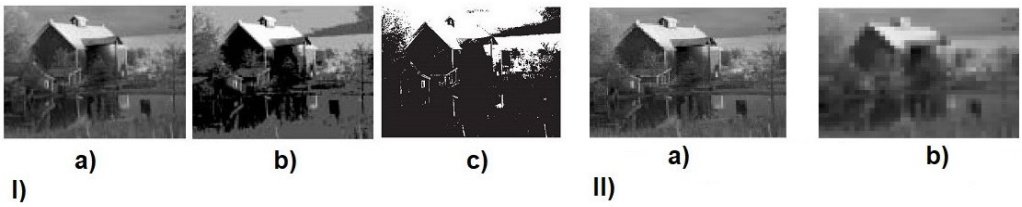


Figura 6. I) Exemplos de quantização e resolução espacial de imagem digital. Fonte: [Velho and Gomes 2015]

3.4. *Frustum* da câmera

A renderização de uma cena imita alguns conceitos do mundo real, como a existência de uma câmera virtual que determina a posição do observador na cena, seu ângulo de visão, em qual direção está observando a cena e seu campo de profundidade, isto é, qual a distância mínima (*near plane*) e máxima (*far plane*) dos planos de visão em que ela consegue enxergar algum objeto [Stemkoski and Pascale 2022]. Todos esses parâmetros são utilizados para a geração de uma entidade chamada *frustum*, que se assemelha a uma pirâmide com topo achatado, conforme ilustrado na Figura 7, onde em a) se vê de forma lateral o *frustum* da câmera realçado pela área acinzentada. Os vetores **n** e **v** realizam a orientação da câmera; observa-se os planos anterior (*near -n*) e posterior(*far -f*), o ângulo α de visão da câmera, sua posição **c** no espaço, e eixos y e -z do sistema de coordenadas. A importância do *frustum* é determinar o que será renderizado, conforme visto na parte

b) Figura 7: objetos dentro de sua área são visíveis no *render* final; objetos parcialmente dentro de sua área são recortados para descartar a geometria fora do *frustum* na etapa de *clipping* do *pipeline* gráfico, e por fim, objetos totalmente fora do *frustum* não serão exibidos na imagem final [Velho and Gomes 2015].

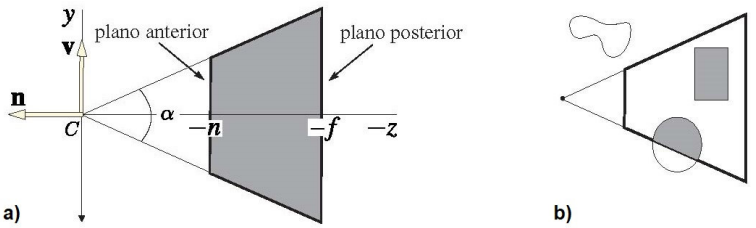


Figura 7. Frustum da câmera. Fonte: [Velho and Gomes 2015] - adaptado.

.

4. Modelos de iluminação e sombreamento

A iluminação é o processo de determinar como a cor e intensidade da luz incidem sobre uma superfície, enquanto o termo sombreamento é usado para descrever a cor e intensidade de luz que é refletida na direção do observador em cada ponto dessa superfície. Um modelo que descreva a interação entre a luz e uma superfície de forma fisicamente fidedigna se torna custoso computacionalmente e inviável para gráficos de renderização em tempo real [Lengyel 2011].

4.1. Fontes de luz

Uma cena 3D pode conter diversas fontes de luz, cada qual com seus aspectos e parâmetros individuais que controlam como a luz é emitida na cena. A cor final calculada sobre a superfície da malha do objeto é o somatório de todas as fontes de luz que a iluminam. Uma cor denominada C é tratada como um *array* das primitivas RGB, permitindo que cores possam ser somadas ou multiplicadas entre si ou por algum valor escalar [Lengyel 2011]. A seguir são descritos os tipos de iluminação mais comuns.

4.1.1. Luz ambiente

É uma luz de baixa intensidade que surge em um ambiente a partir de todas as reflexões da luz nas superfícies próximas. Essa luz é simulada de maneira a aparentar vir de todas as direções com intensidade luminosa constante e consequentemente ilumina toda superfície de uma malha de maneira uniforme, e elimina a necessidade de calcular todas as reflexões de luz entre os objetos da cena [Lengyel 2011].

4.1.2. Luz direcional

Trata-se de uma fonte de luz criada para simular o Sol, emitindo luz em uma única direção com raios paralelos, sem perder intensidade luminosa de acordo com sua distância e não levando em consideração sua posição no espaço, pois simula uma origem no infinito [Lengyel 2011]. A Figura 8 demonstra o conceito dos raios paralelos de luz e uma aplicação deste modelo de luz em uma cena com objetos 3D na Unity, onde o ícone do sol é a fonte de luz e o eixo azul representa sua direção de incidência.

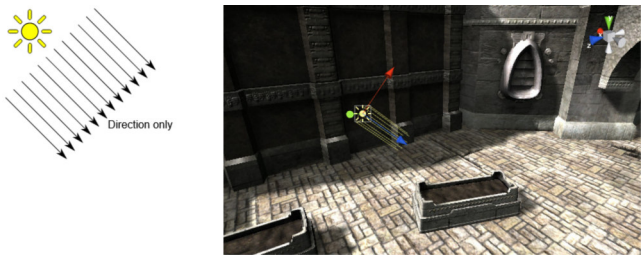


Figura 8. Luz direcional na Unity. Fonte: [Unity Technologies 2022a] - adaptado.

4.1.3. Luz pontual

Simula uma luz emitida em todas as direções a partir de um único ponto no espaço, cuja intensidade luminosa diminui à medida inversa do quadrado da distância. Seja uma fonte de luz pontual originada no ponto **p**, a intensidade de luz C que atinge um ponto **q** no espaço é dada por

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0 \tag{6}$$

onde C_0 é a cor da luz emitida, **d** é a distância da origem da luz até o ponto **q**, e k_c, k_l, k_q são os valores de atenuação constante, linear e quadrático da luz, respectivamente. Otimizações existentes em OpenGL permitem que um limiar máximo de distância torne a intensidade *C* anulada, afim de evitar cálculos para luzes muito distantes do objeto que não influenciarão significativamente em sua superfície [Lengyel 2011]. Na Figura 9 vemos o esquema gráfico do centro do ponto luminoso e a atenuação da luz de acordo com o raio de alcance, bem como um exemplo de luz pontual em determinada posição no espaço 3D e sua incidência sobre um plano na Unity.

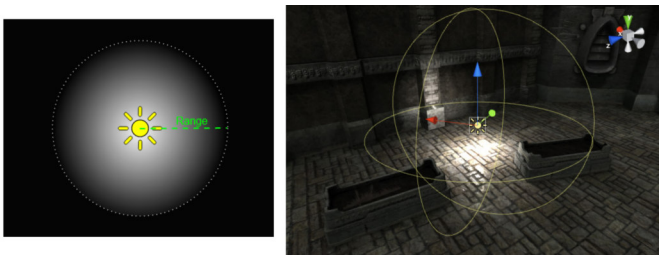


Figura 9. Luz pontual na Unity. Fonte: [Unity Technologies 2022a] - adaptado.

4.1.4. Luz focal

Uma luz focal ou holofote é uma fonte de luz originada num ponto do espaço que emite luz numa direção num formato similar a um cone, mas possui as mesmas características de atenuação da luz pontual acrescido de um outro fator, chamado efeito holofote [Lengyel 2011].

Seja uma luz focal na posição **p** e direção **r**, a intensidade de cor C que atinge um ponto **q** no espaço é dada por:

$$C = \frac{max(-\mathbf{r} \cdot \mathbf{l})^p}{k_c + k_l d + k_q d^2} C_0 \tag{7}$$

onde **l** é o vetor unitário na direção de $\mathbf{p}-\mathbf{q}$, ou seja, a distância entre o ponto da superfície e a origem da luz; já o expoente p é uma constante usada para determinar o grau de foco da luz (a abertura do cone sobre a superfície), e quanto maior o valor de p, menor é a abertura do cone, ou seja, mais focada será a iluminação, tal como visto na Figura 10 onde se faz vê variações da luz para p=2, p=10, p=50 e p=100.

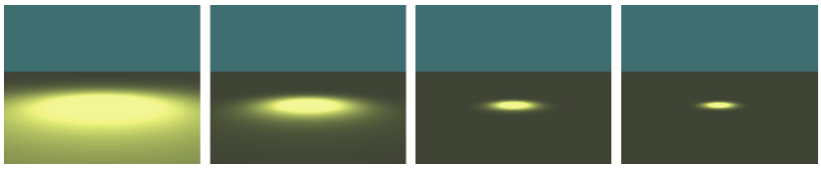


Figura 10. Variação de luz focal. Fonte: [Lengyel 2011]

A Figura 11 mostra um gráfico com as propriedades de ângulo e raio de alcance da luz focal na Unity, com um exemplo de sua aplicação em uma cena 3D, onde percebe-se (para fins de interpretação visual) o contorno do cone de distribuição dessa luz.

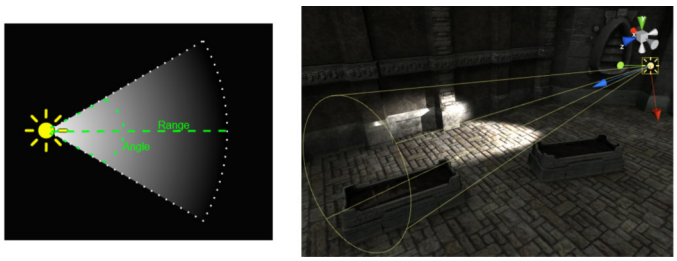


Figura 11. Luz focal na Unity. Fonte: [Unity Technologies 2022a] - adaptado.

4.2. Reflexão difusa de Lambertian

Uma superfície é dita difusa quando parte da luz que incide sobre algum ponto sobre ela é espalhada em direções aleatórias, ou seja, a cor da luz que incide e a cor de reflexão difusa da superfície são refletidas igualmente em todas as direções, independente da posição do observador [Lengyel 2011].

Um feixe de luz que possua uma área A obtida por um corte transversal ilumina a mesma área A sobre uma superfície apenas se a normal dessa superfície for perpendicular a direção do feixe de luz. Isso implica que a medida que o ângulo θ entre a direção do feixe **l** e a normal da superfície **n** aumenta, a área iluminada também aumenta. Entretanto, a superfície da área iluminada será dada pela relação $A \div \cos\theta$, que implica numa redução da intensidade da luz a medida que a área aumenta, tal como visto na Figura 12 [Lengyel 2011].

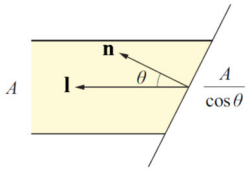


Figura 12. Área da superfície com vetor normal n iluminada por um feixe de luz de direção l, reduz luminosidade na proporção $A \div \cos\theta$. Fonte: [Lengyel 2011]

Com ambos vetores de direção do feixe de luz **l** e normal da superfície **n** são unitários, o cosseno de θ é obtido através do produto interno $\mathbf{n} \cdot \mathbf{l}$, onde um resultado negativo indica que a superfície está oposta à luz, não devendo ser iluminada.

Uma fórmula que calcula a cor da luz K refletida para o observador num ponto **q** da superfície, que leva em consideração todas as cores C_i das n luzes que iluminam o ponto **q**, que é multiplicada pela cor de difusão D da superfície e da luz ambiente A pode ser expressa como

$$K_{diffuse} = DA + D\sum_{i=1}^{n} C_i max(\mathbf{n} \cdot \mathbf{l}_i, 0) \tag{8}$$

onde $\mathbf{l}_i$ é um vetor unitário que aponta de **q** para a $i - èsima$ fonte de luz.

4.3. Reflexão especular

Além da reflexão difusa, uma superfície também reflete luz de forma mais intensa de acordo com a direção da luz incidente **l** e a normal da superfície **n**, o que resulta num efeito de ponto brilhante denominado especularidade, que é mais intenso à medida que o ângulo formado entre a direção da visão do observador **v** e o vetor refletido da luz incidente **r** é menor, ou seja, quanto mais alinhados **v** e **r** forem, maior a especularidade, tal como visto na Figura 13: a).

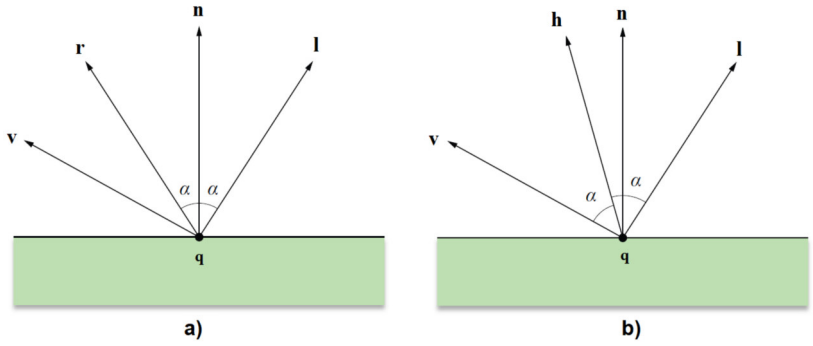


Figura 13. a) Reflexão especular r do raio l de acordo com a normal n e observador v. b) Reflexão especular que usa o vetor intermediário h. Fonte: [Lengyel 2011], adaptado.

Uma equação que permite simular uma aproximação dessa especularidade para uma fonte de luz é dada por

$$K_{specular} = SCmax(\mathbf{r} \cdot \mathbf{v}, 0)^m(\mathbf{n} \cdot \mathbf{l} > 0) \tag{9}$$

onde S é a cor de reflexão especular da superfície, C é a intensidade da luz que incide na superfície, a expressão $(\mathbf{n} \cdot \mathbf{l} > 0)$ realiza o produto interno de **n** e **l**, e retorna *true* (1) caso seja maior que zero, ou *false* (0) caso seja menor, para anular a especularidade em casos em que a normal da superfície forme um ângulo obtuso com a luz, e por fim, m é o expoente especular que permite controlar a nitidez da luz de destaque, onde quanto maior m, mais nítido será a especularidade e menor a distância de esmaecimento da luz, tal como visto na Figura 14 [Lengyel 2011].

Uma alternativa desse método faz uso de um vetor intermediário **h** de direção exatamente entre **l** e **v**, que permite que a luz especular seja mais intensa quando **h** se aproxima de **n**, simplesmente substituindo $\mathbf{r} \cdot \mathbf{v}$ por $\mathbf{n} \cdot \mathbf{h}$, sendo **h** a forma unitária do vetor resultante de $\mathbf{l} + \mathbf{v}$ conforme ilustrado na Figura 13: b) [Lengyel 2011].

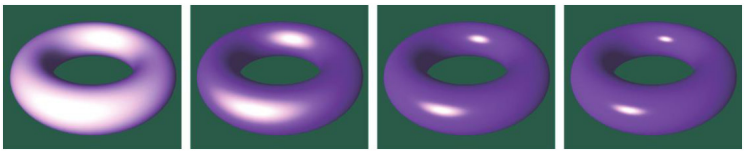


Figura 14. Reflexão especular de cor S branca, com expoentes m = 2, m = 10, m = 50 e m = 100, da esquerda para direita. Fonte: [Lengyel 2011]

Uma equação final que usa o vetor intermediário e leva em consideração todas as n fontes luminosas para compor a luz especular pode ser finalmente obtida por

$$K_{specular} = S \sum_{i=1}^{n} C_i max\left(\mathbf{n} \cdot \mathbf{h}_i, 0\right)^m \left(\mathbf{n} \cdot \mathbf{l}_i > 0\right) \tag{10}$$

4.4. *Shading*

Shading (sombreamento) é o processo de determinar a cor final a ser associada ao pixel contido no espaço do triângulo. Para cada pixel do triângulo é executado um *pixel shader* para calcular o brilho e cor ao pixel, e são considerados a quantidade de luz que incide sobre o triângulo e as propriedades do material para determinar como a luz se reflete naquele ponto [Lengyel 2011].

4.4.1. Sombreamento de Gouraud

Esse método realiza interpolação dos valores de cor e luz usando apenas os vértices do triângulo, e por muito tempo foi a forma mais comum de sombreamento antes das GPUs serem capazes de realizar cálculos de iluminação para cada pixel ao invés de apenas por vértice. O método calcula a cor primária através de

$$K_{primary} = E + DA + D \sum_{i=1}^{n} C_i max\left(\mathbf{n} \cdot \mathbf{l}_i, 0\right) \tag{11}$$

onde E é a cor que a superfície possa emitir caso seja desejado e a cor secundária através da Equação 10 (a mesma da cor especular), e a cor final do pixel é dada por

$$K = K_{primary} \circ T_1 \circ T_2 \circ T_k + K_{secondary} \tag{12}$$

onde T_i é a cor de uma dos k mapas de texturas aplicados na malha e $\circ$ a operação entre eles (como soma ou multiplicação, por exemplo) [Lengyel 2011].

4.4.2. Sombreamento de Blinn-Phong

O sombreamento primeiramente realizado por Phong e aprimorado por Jim Blinn, calcula a equação da luz para cada pixel e interpola os vetores normais $\mathbf{n}$, a direção da luz $\mathbf{l}$ e a direção do olhar $\mathbf{v}$ do observador sobre a superfície do triângulo. O estágio de fragment shading do pipeline é capaz de calcular a expressão 13 inteira para cada pixel que compõe a face do triângulo [Lengyel 2011]

$$\begin{gathered} K = K_{emission} + K_{diffuse} + K_{specular} = \\ EM + DTA + \sum_{i=1}^{n} C_i \left[DT(\mathbf{n} \cdot \mathbf{l_i}) + SG(\mathbf{n} \cdot \mathbf{h_i})^m (\mathbf{n} \cdot \mathbf{l_i} > 0)\right] \end{gathered} \tag{13}$$

Sua vantagem sobre o método de Gouraud é que com o produto interno $\mathbf{n} \cdot \mathbf{h}$ sendo calculado para cada pixel, a especularidade da luz é melhor representada, pois no método de Gouraud, o cálculo é feito apenas para os vértices do triângulo, levando a resultados visuais não fidedignos com relação a face do triângulo.

5. Exemplos de *shaders*

No presente trabalho, os algoritmos de *shaders* realizam a Síntese de Imagens 2D aplicando modelos de iluminação sobre os dados de objetos geométricos 3D, realizando uma série de etapas de processamento computacional chamadas de *pipeline* gráfico [Stemkoski and Pascale 2022].

Usou-se a *game engine* Unity na versão 2022.1.6 para criação dos *shaders*, por se tratar de um ambiente de criação de jogos digitais e aplicações gráficas interativas que já fornece todo arcabouço de ferramentas necessárias para acelerar o desenvolvimento do *software* desejado, tais como ambiente de composição de cenas 2D e 3D, simulação de corpos rígidos, ferramentas para animação e interface gráfica [Doran 2021].

Em Unity, todo objeto existente na cena é uma instância da classe GameObject, que por si só, é desprovida de funcionalidades, exceto sua integração com o próprio funcionamento interno da *game engine*. Entretanto, cada GameObject pode agregar vários objetos do tipo Component, que é uma classe que determina uma funcionalidade específica a ser executada, com suas propriedades pertinentes ao seu contexto de execução. Por exemplo, Transform é um Component que armazena a posição, rotação e escala de um objeto no espaço 3D, enquanto Light é um componente que indica o tipo de luz emitida (com propriedades de cor, sombras, etc.). O componente MeshFilter armazena a malha de vértices do objeto 3D, enquanto o componente MeshRenderer renderiza a malha do MeshFilter. Por fim, componentes do tipo Material indicam qual *shader* é utilizado para a renderização pelo MeshRenderer [Unity Technologies 2022a].

Foi utilizado o modo padrão de renderização da Unity (embora existam modos mais sofisticados com configurações mais avançadas, como *Physically Based Render* e suporte a *Ray Tracing*) com *shaders* escritos na linguagem Cg. A estrutura de um *shader* em Unity se organiza da seguinte maneira, conforme pode ser observado nos códigos do Apêndice 10 [Unity Technologies 2022b]:

1. Shader: escopo geral do *shader*, seguido de uma *string* que determina seu nome.
2. *Properties*: vinculam variáveis do *shader* com o editor da Unity.
3. *Subshader*: escopo do código do *shader* em Cg, localizado sempre entre as instruções CGPROGRAM e ENDCG.
4. *Tags*: conjuntos de chave-valor que determinam como e quando a Unity deverá usar o *shader*, como em *Holographic* que é renderizado na fila de *shaders* com suporte a transparência (10.1, linha 13).
5. O método "*void surf*"é sempre declarado para *shaders* de sombreamento de superfícies, pois é onde o cálculo de iluminação é realizado. O parâmetro *Input* é a *struct* que define as coordenadas de textura e outros parâmetros que o programador necessite manipular, e *SurfaceOutput* é outra estrutura preenchida com dados da superfície sendo sombreada, fornecida pela Unity.

6. Pode-se determinar como a luz é calculada através da criação de uma função cujo tenha o prefixo "*Lighting*", tal como escrito para os *shaders Simple Lambert* (10.2, linha 32), *Phong* (10.3, linha 39) e *Blin-Phong* (10.4, linha 42).

5.1. *Holographic Shader*

O efeito de renderização holográfico consiste em colorir apenas as bordas do objeto com cor, mantendo seu interior transparente. Entretanto, conforme o observador mude a direção em que observa tal objeto, seu contorno também deve mudar. Para tal, se usa o produto interno entre o vetor normal do triângulo e o vetor de visualização. Quando os vetores forem ortogonais (resultado zero), significa que alguma cor deve ser adicionada ao triângulo, do contrário, uma interpolação linear (linha 46 algoritmo, 10.1) é aplicada na transparência final do triângulo de acordo com quão alinhados os vetores estão [Doran 2021].

O código fonte de um shader que simula um efeito holográfico pelo uso do produto interno entre vetores para determinar a borda de uma geometria está listado em 10.1 e uma imagem gerada com seu uso é visto na Figura 15.

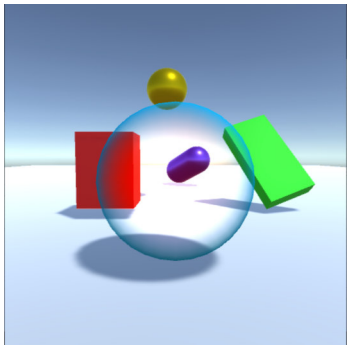


Figura 15. Shader holográfico aplicado em uma esfera. Fonte: o autor.

5.2. Lambert

Shaders baseados no modelo de Lambertian são considerados não foto-realísticos, embora se possam obter excelentes resultados em geometrias low-poly por produzir um contraste eficiente tanto visual quanto computacionalmente [Doran 2021].

A Figura 16 demonstra o resultado do modo de iluminação de Lambert (algoritmo 10.2), onde percebe-se na linha 35 o cálculo do produto interno para fins de iluminação tal como previsto na Equação 8.

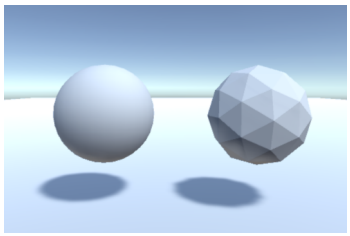


Figura 16. Shader de Lambert aplicado em uma esfera com muitos polígonos (esquerda) e low-poly (direita). Fonte: o autor.

5.3. Phong

Calcula a reflexão da luz na superfície de acordo com a direção do observador, sendo considerado não tão realístico quanto a forma de reflexão especular, porém, entrega bons resultados visuais e pode ser aplicado para objetos distantes na cena já que não serão renderizados com tantos detalhes [Doran 2021]. Como demonstrado na Equação 9, neste shader (10.3) é calculado o vetor de reflexão da luz na linha 44, e o parâmetro SpecPower é o expoente especular m, que controla a distribuição do brilho da luz. A Figura 17 a) exibe uma esfera com expoente especular I) m=2 e em II) m=20, onde se vê a diferença de especularidade causada pela mudança do expoente.

5.4. Blin-Phong

Sua característica de calcular o vetor intermediário entre observador e direção da luz, otimiza o processamento evitando calcular o vetor de reflexão, conforme observado no algoritmo 10.4 na linha 49, onde $halfVector$ é o vetor intermediário obtido pela normalização do vetor resultante de $lightDir + viewDir$. A Figura 17 b) exibe a mesma esfera com expoente especular de valor I) m=2 e em II) m=20, que pode ser comparada com shader de Phong da seção anterior. a

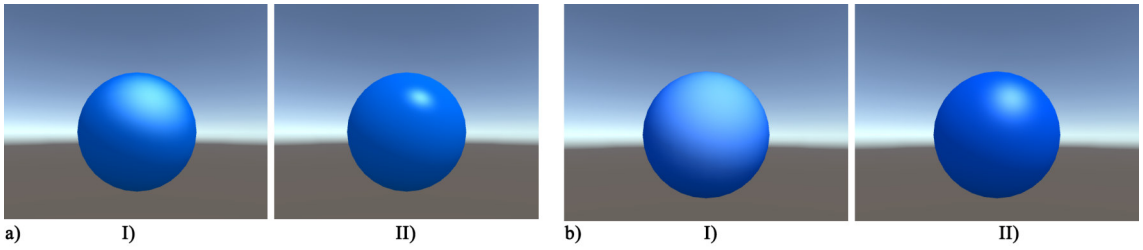


Figura 17. a) Phong e b) Blin-Phon, ambos com I) m=2 e II) m=20. Fonte: o autor.

6. OpenGL *Graphics Pipeline*

OpenGL é uma API para acesso de recursos em *hardware* de processamento gráfico, que foi projetada para funcionar como uma interface independente do hardware que a implementa. Foi lançada pela primeira vez pela *Silicon Graphics Computer Systems* em 1994 e desde então diversas outras bibliotecas de *software* a encorporam para simplificar o processo de desenvolvimento de aplicações para jogos, visualização científica e imagens médicas [Shreiner et al. 2017].

O *pipeline* de renderização de imagens em tempo real é a sequência de etapas necessárias para a geração de uma imagem 2D a partir de objetos 3D formados por dados de vértices, materiais e texturas que compõem uma cena virtual, onde são executados os diversos *shaders* que operam sobre as malhas dos objetos [Akenine-Moller et al. 2019].

O *pipeline* gráfico tem estágios que variam de acordo com cada API, embora todas possuam certas etapas em comum, como a rasterização, por exemplo. Certas etapas podem ser subdivididas em outras subetapas e processadas em paralelo. Existem etapas de execução fixa pelo *hardware* que não permitem nenhuma modificação de sua rotina, algumas são opcionais (podendo ou não serem executadas). Algumas etapas permitem

ao programador controlar totalmente o que deve ser realizado enquanto outras permitem apenas configurações de parâmetros, sem permitir alteração de seu funcionamento [Akenine-Moller et al. 2019]. A Figura 18 demonstra o fluxo de etapas do pipeline em OpenGL, cujas etapas são descritas a seguir.

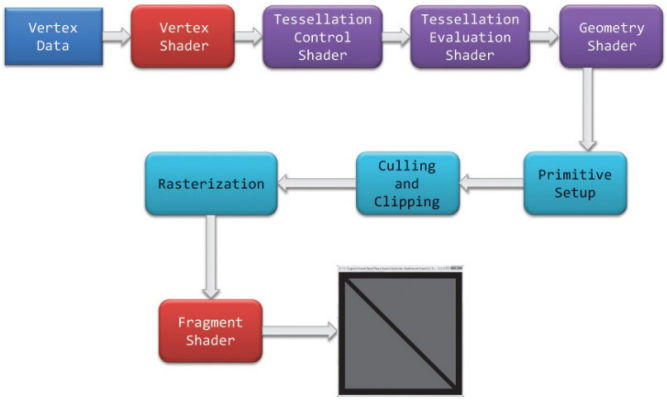


Figura 18. Visão geral do *pipeline* gráfico em OpenGL. Fonte: [Gordon and Clevenger 2021].

6.1. *Vertex Shader*

Esta etapa se encarrega de calcular qual a posição dos vértices da geometria no *display*, aplicando as diversas transformações espaciais e conversões entre os sistemas de coordenadas de objeto para coordenadas da câmera, sendo executada paralelamente para cada vértice. Também é nessa etapa que são aplicadas as luzes presentes na cena, materiais e texturas associadas aos objetos para determinar a cor de cada vértice [Akenine-Moller et al. 2019].

Atualmente, quase todas as operações de sombreamento ocorrem por pixel e não mais por vértice, tornando este estágio mais genérico e programável de acordo com o intuito do desenvolvedor, podendo inclusive, não computar nenhuma equação que envolva cálculos de luminosidade. Entretanto, nesta etapa nenhum vértice pode ser criado ou destruído, e o resultado do processamento de um vértice não pode ser utilizado por outro vértice [Akenine-Moller et al. 2019].

6.2. *Tesselation Shader*

A etapa de tesselação é uma etapa opcional que permite subdividir a malha de uma geometria para adicionar novos vértices, gerando uma malha geométrica com mais detalhes. Caso seja utilizada, a tesselação adiciona duas etapas no pipeline: o *tesselation control shader*, que determina a quantidade de vértices a serem gerados; e a *tesselation evaluation shader*, que se encarrega de posicionar tais vértices em coordenadas de tesselação e os envia para as outras etapas do pipeline[Gordon and Clevenger 2021].

A tesselação permite a existência de uma técnica chamada Nível de Detalhamento (*Level of Detail* – LOD) que renderiza geometrias próximas da câmera de forma mais detalhada com mais vértices e geometrias longe da câmera com menos vértices, pois serão menos perceptíveis ao observador. Assim, uma geometria simplificada pode ser armazenada ocupando menos memória e utilizada para os cálculos de física e animação de forma menos custosa (por possuir menos vértices), sendo no fim renderizada uma geometria mais detalhada [Luna 2016].

6.3. *Geometry Shader*

É uma etapa opcional que opera sobre primitivas inteiras formadas por triângulos e permite diversas operações, tais como: gerar novas primitivas a partir da entrada; destruí-la caso se enquadre com alguma condição programada; alterar seu formato e dimensões [Gordon and Clevenger 2021].

6.4. *Primitive Assembly*

Se encarrega de organizar todos os vértices processados até então em suas respectivas primitivas geométricas para que sejam entregues às etapas subsequentes de recorte e rasterização [Shreiner et al. 2017]

6.5. *Culling e Clipping*

Essa etapa se encarrega de determinar se uma geometria será exibida no *display* verificando se ela está dentro do *frustum* da câmera, ou caso ela esteja totalmente fora do *frustum*, não será exibida (*culled*) no *display*. Entretanto, se a geometria estiver parcialmente dentro do *frustum*, deverá então ser "recortada" (*clipping*) para remover os vértices fora do *frustum*. Isso permite que apenas as partes da geometria de fato visível sejam enviadas para as próximas etapas do *pipeline*, eliminando processamento em vértices desnecessários [Akenine-Moller et al. 2019].

As primitivas são comparadas contra os planos do *frustum* para determinar os pontos de intersecção entre eles, e no caso de haver recorte, é formada a nova primitiva a ser renderizada. O plano *near* permite que primitivas muito próximas da câmera sejam excluídas/recortadas da cena para evitar distorções de tamanho por estarem muito próximas do observador, enquanto o plano *far* exclui primitivas muito distantes que possam causar excesso de processamento ou imprecisões no cálculo de profundidade dos objetos em perspectiva [Shreiner et al. 2017].

6.6. *Rasterization*

A etapa de Rasterização recebe este nome por realizar a conversão de primitivas no espaço 3D para um *display* 2D, que é constituído de um *raster*: um *array* retangular de pixels [Gordon and Clevenger 2021]. Sua principal tarefa, conforme indicado por [Shreiner et al. 2017], é determinar quais pixels do monitor devem ser usados para desenhar cada primitiva que compõe a geometria.

É dito que a rasterização converte a primitiva em fragmentos, que são estruturas de dados que armazenam informações pertinentes a cada pixel a ser processado. Essas informações são construídas através da interpolação de todos os dados contidos nos vértices da primitiva usando as coordenadas dos pixels que ela ocupa na tela, para que tais informações sejam repassadas à próxima etapa do pipeline [Gordon and Clevenger 2021]. A Figura 19 mostra vértices de um triângulo com cores azul, vermelho e verde sendo interpoladas no interior dos pixels rasterizados no monitor.

Embora o resultado final seja um *raster* 2D, os valores de profundidade de cada primitiva permanecem associados aos seus pixels correspondentes, para que depois possam ser utilizadas para resolver casos de sobreposição de diferentes primitivas ocupando um mesmo pixel [Gordon and Clevenger 2021].

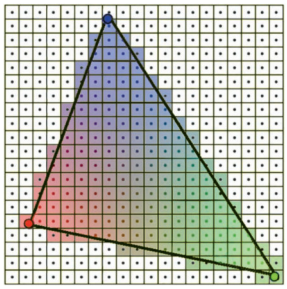


Figura 19. Exemplo de rasterização. Fonte: [Gordon and Clevenger 2021].

6.7. *Fragment Shading*

Nesse estágio, é dado total controle ao programador sobre a cor a ser atribuída ao pixel, permitindo uma liberdade criativa sobre como a cor deve ser calculada, ou mesmo descartá-lo por completo. Por exemplo, pode-se simplesmente determinar que um pixels com coordenadas fora de algum intervalo possuirão sempre uma cor azul, e os demais vermelho, permitindo efeitos fictícios de cores [Gordon and Clevenger 2021].

7. Paralelismo em GPU

Existem diferenças significativas entre um CPU e um GPU. Um CPU é otimizado para trabalhar com várias estruturas de dados e grandes bases de código geralmente processados em série, e se utiliza de técnicas para reduzir o tempo de espera para receber/acessar dados (latência) como memórias cache extremamente rápidas. Em contrapartida, GPUs seguem uma tendência de alto paralelismo de execução de um grupo muito específico de tarefas, dedicando estruturas de silício fixas em seu chip para sua execução, como tarefas de *z-buffer*, acesso a imagens de texturas e processamento de vértices [Akenine-Moller et al. 2019].

7.1. Arquiteturas de dados paralelos

Numa GPU, grande parte do chip é ocupada por um enorme conjunto de processadores chamados *shader cores*, que processam dados em um fluxo no qual dados similares (como um conjunto de vértices ou pixels) são processados no mesmo turno de execução de forma paralela e o mais independentemente possível, sem compartilhar memória ou necessitar dos resultados do processamento de outros cores [Akenine-Moller et al. 2019]. Exemplificando, se um conjunto de dois mil vértices tivessem que ser processados por um único core da GPU, o *shader* designado para essa execução seria invocado de forma sequencial uma vez para cada vértice (gerando dois mil turnos de execução), enquanto que uma GPU com dois mil *shader cores* executaria a mesma tarefa em apenas um turno de execução, com uma invocação do *shader* para cada core, de forma paralela.

As GPUs ainda implementam uma organização de execução das tarefas chamada SIMD (*single instruction, multiple data* – instrução única, vários dados) por separar a instrução lógica a ser executada dos dados a serem processados. Isso permite um único grupo de *shaders* executem a mesma instrução sobre uma grande quantidade de dados em apenas um ciclo de processamento [Akenine-Moller et al. 2019].

7.2. *Compute shader*

Um *shader* pode realizar tarefas não relacionadas a cálculos e algoritmos com as geometrias e imagens dentro do *pipeline*, tais como treinamento de redes neurais. O uso de

GPUs para propósitos não gráficos recebe o nome de “GPU *computing*” ou GPGPU, e para tal são utilizadas plataformas como CUDA e OpenCL para controlar a GPU como um grande processador paralelo. Uma grande vantagem de sua utilização é que *compute shaders* podem acessar e gravar dados na memória da GPU, evitando o atraso tradicional de se enviar os dados da GPU para a CPU [Akenine-Moller et al. 2019].

A principal restrição do uso de GPUs para esse propósito é que os algoritmos que serão executados pelo *compute shader* possam ser paralelizados, ou seja, devem realizar o mesmo tipo de operação sobre uma grande quantidade de dados para que haja um real ganho de desempenho em processamento paralelo ao contrário do processamento sequencial numa CPU [Luna 2016].

8. Protótipo interativo

Para fornecer uma interação direta do usuário com tais conceitos, uma aplicação foi desenvolvido na Unity e compilado para plataforma WEB, disponível no endereço `https://daybsonufma.github.io/shaders/`, para permitir aos leitores uma utilização direta via navegador, evitando a necessidade de instalações de programas. O código fonte do projeto da Unity pode ser obtido em `https://github.com/daybsonufma/shaders`.

A aplicação consiste numa cena com objetos 3D e controles visuais para permitir ao usuário alterar parâmetros dos *shaders* e características da cena, como: tipo de iluminação, intensidade de luz, alcance e cores base e especulares dos *shaders*, bem como ativar um *tour* em 360°ao redor do objeto, visualizar e ocultar sombras e elementos do cenário. O modelo 3D do carro utilizado foi obtido gratuitamente pela Asset Store da Unity, e demais elementos do cenário gerados na própria Unity. Um exemplo da aplicação pode ser visualizada na Figura 20.

Figura 20. Aplicação desenvolvida. Fonte: o autor.

9. Conclusão

Fica demonstrado a direta aplicação da álgebra linear na programação de shaders através dos modelos de iluminação demonstrados, com especial foco no uso de vetores e suas

relações para simular (ou aproximar) diversos comportamentos da luz no mundo real através do computador, ou mesmo aplicar efeitos impossíveis no mundo real, como o caso apresentado de holografia, que podem ser úteis para visualização de peças mecânicas e projetos de engenharia civil.

A evolução das GPUs impulsiona a qualidade e possibilidades de efeitos de renderização cada vez mais realísticos, tais como uso de *Ray Tracing* em tempo real. Ao mesmo tempo, permitem que cálculos de natureza não gráficos tenham sua velocidade de computação acelerada através dos *compute shaders*, que podem ser aplicados em diversas técnicas de Inteligência Artificial, tais como Aprendizado de Máquina. Isso traz benefícios significativos para diversas áreas como citado os casos de segmentação de imagens de tumores na Medicina, marcando assim, a relevância do assunto apresentado no trabalho e a importância da compreensão de seus fundamentos.

A aplicação desenvolvida permite uma interação visual e manipulação dos tipos de iluminação apresentados e de seus principais parâmetros. Isso fornece uma experiência mais palpável dos conceitos apresentados ao mesmo tempo em que demonstra a aplicação direta dos mesmos, facilitando o entendimento do leitor.

Referências

[Akenine-Moller et al. 2019] Akenine-Moller, T., Haines, E., and Hoffman, N. (2019). *Real-time rendering*. AK Peters/CRC Press, 4 edition.

[Breckon and Solomon 2013] Breckon, T. and Solomon, C. (2013). *Fundamentos de processamento digital de imagens: uma abordagem prática com exemplos em Matlab*. Grupo Gen - LTC.

[Doran 2021] Doran, J. P. (2021). *Unity 2021 Shaders and Effects Cookbook - Fourth Edition*. Packt Publishing, 4th edition.

[Gordon and Clevenger 2021] Gordon, V. S. and Clevenger, J. L. (2021). *Computer Graphics Programming in OpenGL with C++*. Mercury Learning & Information, 2 edition.

[Gregory 2018] Gregory, J. (2018). *Game engine architecture*. AK Peters/CRC Press, 4 edition.

[Kordt et al. 2021] Kordt, J., Brachmann, P., Limberger, D., and Lippert, C. (2021). Interactive volumetric region growing for brain tumor segmentation on mri using webgl. In *The 26th International Conference on 3D Web Technology*, Web3D '21, New York, NY, USA. Association for Computing Machinery.

[Lengyel 2011] Lengyel, E. (2011). *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. Course Technology Press, 3rd edition.

[Luna 2016] Luna, F. (2016). *Introduction to 3D Game Programming with DirectX 12*. Mercury Learning and Information.

[Marschner and Shirley 2021] Marschner, S. and Shirley, P. (2021). *Fundamentals of Computer Graphics*. AK Peters/CRC Press, 5 edition.

[Myronenko 2019] Myronenko, A. (2019). 3D MRI Brain Tumor Segmentation Using Autoencoder Regularization. In Crimi, A., Bakas, S., Kuijf, H., Keyvan, F., Reyes, M.,

and van Walsum, T., editors, *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*, pages 311–320, Cham. Springer International Publishing.

[Shreiner et al. 2017] Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2017). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. Pearson Education, 9 edition.

[Stemkoski and Pascale 2022] Stemkoski, L. and Pascale, M. (2022). *Developing Graphics Frameworks with Python and OpenGL*. CRC Press.

[Unity Technologies 2022a] Unity Technologies (2022a). Types of light. `https://docs.unity3d.com/Manual/Lighting.html`. Online; accessed 17 june 2022.

[Unity Technologies 2022b] Unity Technologies (2022b). Writing surface shaders. `https://docs.unity3d.com/Manual/SL-SubShader.html`. Online; accessed 17 june 2022.

[Velho and Gomes 2015] Velho, L. and Gomes, J. (2015). *Fundamentos da Computação Gráfica*. IMPA, 1st edition.

10. Apêndice: Lista de shaders

10.1. Holographic shader

```
Shader "UFMA/Holographic"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _DotProduct("Rim effect", Range(-1, 1)) = 0.25
    }
    SubShader
    {
        Tags
        {
            "Queue" = "Transparent"
            "IgnoreProjector" = "True"
            "RenderType" = "Transparent"
        }
        Cull front
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert alpha:fade
        #pragma target 3.0

        float _DotProduct;
        sampler2D _MainTex;
        fixed4 _Color;

        struct Input
        {
            float2 uv_MainTex;
            float3 worldNormal;
            float3 viewDir;
        };

        UNITY_INSTANCING_BUFFER_START(Props)
        UNITY_INSTANCING_BUFFER_END(Props)

        void surf (Input IN, inout SurfaceOutput o)
        {
            float4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;

            //quanto mais alinhado com a direção da vista do observador, menor será o valor de border (tornando-o mais transparente)
            float border = 1 - abs(dot(IN.viewDir, IN.worldNormal));
```

```
45
46              float alpha = (border * (1 - _DotProduct) + _DotProduct);
47              o.Alpha = c.a * alpha;
48          }
49          ENDCG
50      }
51      FallBack "Diffuse"
52  }
```

10.2. Simple Lambert

```
1   Shader "UFMA/SimpleLambert"
2   {
3       Properties
4       {
5           _MainTex ("Texture", 2D) = "white"
6       }
7       SubShader
8       {
9           Tags { "RenderType"="Opaque" }
10          LOD 200
11
12          CGPROGRAM
13          #pragma surface surf SimpleLambert
14          #pragma target 3.0
15
16          sampler2D _MainTex;
17
18          struct Input
19          {
20              float2 uv_MainTex;
21          };
22
23          UNITY_INSTANCING_BUFFER_START(Props)
24          UNITY_INSTANCING_BUFFER_END(Props)
25
26          void surf (Input IN, inout SurfaceOutput o)
27          {
28              o.Albedo = tex2D(_MainTex, IN.uv_MainTex).rgb;
29          }
30
31          // Cria o modo de iluminação SimpleLambert
32          half4 LightingSimpleLambert (SurfaceOutput s, half3 lightDir, half atten)
33          {
34              //Produto interno entre a direção da luz e a normal da superfície
35              half NDotL = dot(s.Normal, lightDir);
36
37              half4 color;
38              color.rgb = s.Albedo * _LightColor0.rgb * (NDotL * atten);
39              color.a = s.Alpha;
40
41              return color;
42          }
43          ENDCG
44      }
45      FallBack "Diffuse"
46  }
```

10.3. Phong

```
1   Shader "UFMA/Phong"
2   {
3       Properties
4       {
5           _MainTint ("Diffuse Tint", Color) = (1, 1, 1, 1)
6           _MainTex ("Base RGB", 2D) = "white" {}
7           _SpecularColor ("Specular Color", Color) = (1,1,1,1)
8           _SpecPower ("Specular Power", Range(0, 30)) = 1
9       }
10      SubShader
11      {
12          Tags { "RenderType"="Opaque" }
13          LOD 200
14
15          CGPROGRAM
16          #pragma surface surf Phong
17          #pragma target 3.0
18
19          float4 _SpecularColor;
20          sampler2D _MainTex;
```

```
        float4 _MainTint;
        float _SpecPower;

        struct Input
        {
            float2 uv_MainTex;
        };

        UNITY_INSTANCING_BUFFER_START(Props)
        UNITY_INSTANCING_BUFFER_END(Props)

        void surf (Input IN, inout SurfaceOutput o)
        {
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _MainTint;
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }

        fixed4 LightingPhong(SurfaceOutput s, fixed3 lightDir, half3 viewDir, fixed atten)
        {
            float NDotL = dot(s.Normal, lightDir);

            //Calcula o vetor de reflexão da luz
            float3 reflectionVector = normalize(2.0 * s.Normal * NDotL - lightDir);

            //Calcula o componente specular
            float spec = pow(max(0, dot(reflectionVector, viewDir)), _SpecPower);

            float3 finalSpec = _SpecularColor.rgb * spec;

            //Calcula a cor final
            fixed4 c;
            c.rgb = (s.Albedo * _LightColor0.rgb * max(0, NDotL) * atten) + (_LightColor0.rgb *
     finalSpec * atten);
            c.a = s.Alpha;
            return c;
        }

        ENDCG
    }
    FallBack "Diffuse"
}
```

10.4. Blin-Phong

```
Shader "UFMA/BlinPhong"
{
     Properties
    {
        _MainTint("Diffuse Tint", Color) = (1,1,1,1)
        _MainTex("Base (RGB)", 2D) = "white" {}
        _SpecularColor("Specular Color", Color) = (1,1,1,1)
        _SpecPower("Specular Power", Range(0.1,60)) = 3
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM

        #pragma surface surf CustomBlinnPhong
        #pragma target 3.0

        sampler2D _MainTex;
        float4 _MainTint;
        float4 _SpecularColor;
        float _SpecPower;

        struct Input
        {
            float2 uv_MainTex;
            float3 viewDir;
        };

        UNITY_INSTANCING_BUFFER_START(Props)
        UNITY_INSTANCING_BUFFER_END(Props)

        void surf(Input IN, inout SurfaceOutput o)
        {
            half4 c = tex2D(_MainTex, IN.uv_MainTex) * _MainTint;
            o.Albedo = c.rgb;
```

```
38              o.Alpha = c.a;
39              o.Normal = normalize(o.Normal);
40          }
41
42          fixed4 LightingCustomBlinnPhong(SurfaceOutput s,
43                                          fixed3 lightDir,
44                                          half3 viewDir,
45                                          fixed atten)
46          {
47              float NdotL = max(0, dot(s.Normal, lightDir));
48
49              float3 halfVector = normalize(lightDir + viewDir);
50              float NdotH = max(0, dot(s.Normal, halfVector));
51              float spec = pow(NdotH, _SpecPower) * _SpecularColor;
52
53              float4 color;
54              color.rgb = (s.Albedo * _LightColor0.rgb * NdotL) +
55                  (_LightColor0.rgb * _SpecularColor.rgb * spec) * atten;
56              color.a = s.Alpha;
57              return color;
58          }
59          ENDCG
60      }
61      FallBack "Diffuse"
62  }
```