



FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA - UNIFOR
CENTRO DE CIÊNCIAS TECNOLÓGICAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

ANDERSON ARAUJO MACEDO

**ESTUDO COMPARATIVO DE FERRAMENTAS DE SHADERS EM DIFERENTES
GAME ENGINES**

**FORTALEZA – CEARÁ
2021**

LISTA DE ILUSTRAÇÕES

Figura 1 – A esquerda Doom fazia uso de 3D real enquanto a direita Wolfenstein posicionava imagens 2D em diferentes camadas para simular a profundidade.	12
Figura 2 – Hardware da placa gráfica da NVIDIA.	13
Figura 3 – O buffer de profundidade é mostrado em tons de cinza sendo que objetos próximos ficam com tonalidade mais escura enquanto objetos distantes assumem uma tonalidade mais clara.	15
Figura 4 – O stencil buffer permite a customização da forma como objetos 3D são renderizados.	15
Figura 5 – Pipeline gráfico do OpenGL.	16
Figura 6 – Processo de transformação entre os sistemas de coordenadas e seus espaços.	19
Figura 7 – Conteúdo das quatro colunas da matriz <i>modelView</i>.	20
Figura 8 – Modos de projeção de câmera. As letras correspondem a <i>left</i>, <i>right</i>, <i>bottom</i>, <i>top</i>, <i>near</i> e <i>far</i>.	20
Figura 9 – Demonstração de um shader simples de linha contorno.	22
Figura 10 – O mesmo resultado de contorno obtido com HLSL.	24
Figura 11 – Maiores níveis de tesselação produzem aumento no número de vértices.	25
Figura 12 – Regra da mão direita.	26
Figura 13 – Demonstração de como é possível criar visuais únicos utilizando shaders.	27
Figura 14 – Como os estágios de shaders se relacionam.	29
Figura 15 – A luz que penetra na superfície de um objeto translúcido é espalhada pela interação com o material e sai da superfície em um ponto diferente.	33
Figura 16 – Estrutura hierárquica da cena na Godot Engine.	34
Figura 17 – A programação da lógica do jogo pode ser feita utilizando o sistema visual de <i>BluePrint</i>.	37
Figura 18 – Uso de uma textura como base para definir a cor de um material.	38
Figura 19 – Fluxograma do processo	45
Figura 20 – A ferrovia fica mais estreita e se cruza no horizonte.	58

LISTA DE TABELAS

**Tabela 2 – Exemplo do número de instruções necessárias para operações específicas
no OpenGL**

41

LISTA DE QUADROS

Quadro 1 – Principais componentes das game engines	32
Quadro 2 – Funcionalidades gráficas presentes nas game engines	32

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Transcrição do shader de contorno de GLSL para HLSL	61
Código-fonte 2 – Shader GLSL 3D simples para efeito de contorno	64

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Process Unit
CRT	Cathode Ray Tube
FPS	Frames Por Segundo
GLSL	OpenGL Shading Language
GPGPU	GPU de propósito geral
GPU	Graphics Processing Unit
HDR	High Dynamic Range
HDRP	High Definition Render Pipeline
HLSL	High-Level Shader Language
IBM	International Business Machines
IDE	Ambiente de Desenvolvimento Integrado
IRIS GL	Integrated Raster Imaging System Graphical Library
LWRP	Lightweight Render Pipeline
MIT	Massachusetts Institute of Technology
NES	Nintendo Entertainment System
OpenGL	Open Graphics Library
SDK	Software Development Kit
SGI	Silicon Graphics International
SO	Sistema Operacional
T&L	Transform & Lighting

SUMÁRIO

1	INTRODUÇÃO	8
1.1	Justificativa	9
1.2	Objetivos	9
1.2.1	Objetivo Geral	9
1.2.2	Objetivos Específicos	10
1.3	Estrutura do trabalho	10
2	REFERENCIAL TEÓRICO	11
2.1	Evolução da Programação de Shaders	11
2.1.1	Como o OpenGL funciona	14
2.1.1.1	Pipeline gráfica do OpenGL	16
2.1.1.2	Matrizes de transformação de coordenadas	18
2.1.2	OpenGL Shading Language	21
2.1.3	Direct3D (HLSL) <i>versus</i> OpenGL (GLSL)	23
2.1.4	High-Level Shader Language	24
2.2	Conceitos Técnicos de Shaders	26
2.2.1	Vertex Shader	29
2.2.2	Fragment Shader	30
2.3	Motores de jogo e suas ferramentas	31
2.3.1	Godot	34
2.3.2	Unity	35
2.3.3	Unreal	37
2.3.3.1	Nós de material	38
2.4	Otimização e performance	39
3	METODOLOGIA	45
3.1	Fluxo de Desenvolvimento	45
3.2	Sobre a pesquisa	45
3.3	Escolha das game engines	46
3.4	Sobre os dados	46
4	RESULTADOS	48
4.1	Resultados do Experimento A	48
4.2	Resultados do Experimento B	48

5	CONCLUSÕES E TRABALHOS FUTUROS	49
5.1	Contribuições do Trabalho	49
5.2	Limitações	49
5.3	Trabalhos Futuros	50
	REFERÊNCIAS	51
	GLOSSÁRIO	56
	APÊNDICES	57
	APÊNDICE A – Coordenadas Homogêneas	58
	APÊNDICE B – Código-fonte do shader de contorno em HLSL	61
	ANEXOS	63
	ANEXO A – Código-fonte do shader de contorno em GLSL	64
	Índice	65

1 INTRODUÇÃO

Shader é um tipo de programa de computador utilizado para simular como a luz interage com os objetos ou as superfícies (ZUCCONI; LAMMERS, 2016). Por meio de seu uso é possível criar aspectos visuais nas superfícies de objetos 3D, para que com o uso de texturas, seja possível obter uma aparência de metal ou de madeira, por exemplo.

Por demandar recursos computacionais da GPU em tempo real, a performance de execução desses programas é um assunto que requer atenção, ainda mais levando em consideração o avanço da tecnologia de computação gráfica, que exige a renderização em um curto intervalo de tempo de gráficos cada vez mais realistas. Quanto maior a frequência de realização de cálculos e processamentos durante esse processo, maior será o impacto na performance de um jogo. Ao fazer uso de shaders custosos e não otimizados, podem ocorrer alguns problemas como surgimento de artefatos, incompatibilidade com hardwares de gerações passadas e o superaquecimento da GPU devido a cargas muito altas de trabalho.

Para realizar o desenvolvimento, a execução e o estudo de performance dos shaders, três dos mais populares motores de jogo foram escolhidos. O primeiro foi o Godot, um software para produção de jogos 2D e 3D criado no ano de 2007, quando seus desenvolvedores perceberam duas importantes mudanças no cenário de desenvolvimento de games: uma foi a melhoria de hardware disponível que permitiu que dispositivos portáteis ganhassem mais poder de processamento, a outra mudança foi na forma que as CPUs passaram a ser divididas em múltiplos núcleos, o que permitiu o advento do processamento paralelo (MANZUR; MARQUES, 2018).

O segundo motor de jogo, Unity, é a escolha mais comum entre desenvolvedores de jogos profissionais e amadores por sua capacidade de prototipação rápida e pela ampla gama de plataformas-alvo de compilação. Ela foi criada com os objetivos de fornecer uma engine de custo acessível com ferramentas profissionais e democratizar o acesso à indústria de desenvolvimento de games (HAAS, 2014).

O terceiro motor de jogo escolhido foi a Unreal Engine, produzida pela Epic Games para desenvolvimento de jogos e aplicações, seja de grandes orçamentos e níveis de promoção, seja de editoras ou produtoras independentes e com baixo orçamento. É o mais robusto e também é muito utilizado tanto por desenvolvedores profissionais quanto iniciantes (COOKSON; DOWLING SOKA; CRUMPLER, 2016).

1.1 JUSTIFICATIVA

O processo de criação de shaders pode vir a apresentar-se, dependendo do nível de complexidade exigido pela tarefa, como uma atividade custosa e que exige elevados recursos computacionais. Sendo assim, o estudo das ferramentas de criação de shaders é importante para definir processos de otimização de performance para que empresas, indivíduos ou entusiastas possam economizar tempo e recursos ao utilizar essas ferramentas.

Cabe ressaltar que a execução de programas de shaders muito custosos pode acarretar em problemas como queda da taxa de quadros por segundo, travamentos durante a execução do programa e na pior hipótese danos permanentes ao hardware que acabam por prejudicar o utilizador final e que de maneira geral acarretam em uma má experiência de usuário.

No contexto específico dos jogos eletrônicos, o uso de shaders não otimizados pode fazer com que o jogo torne-se lento e apresente travamentos. Essas são características que tornam um jogo não atrativo e que geram sensações negativas no usuário. Elas fazem com que ele perca o interesse e se sinta frustrado, sendo levado à compartilhar feedback negativo, cujo acaba por prejudicar a imagem e as vendas do produto. Isso tem como consequência motivar outros possíveis usuários a não comprarem o jogo, principalmente aqueles que não possuem hardware compatível.

Nesse caso, a utilidade desse estudo consiste em descobrir qual game engine, utilizando critérios quantizados de performance, apresenta a melhor ferramenta para criação de Shaders. Além disso, shaders otimizados tornam-se favoráveis para serem aplicados para um público maior por ampliar a possibilidade de hardware compatível, ou seja, os jogos ou aplicações que fazem uso desse recurso conseguem ter um alcance maior e mais vendas.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Analisa e comparar as principais ferramentas de desenvolvimento de shaders dentre as game engines especificadas no escopo deste trabalho com foco na otimização de performance em cada uma, identificando os processos-chave característicos de construção e execução de shaders.

1.2.2 Objetivos Específicos

- a) Discriminar as ferramentas de criação de shader de cada game engine bem como suas características individuais.
- b) Determinar os indicadores que serão utilizados para mensurar os parâmetros que serão avaliados nos testes dos shaders.
- c) Desenvolver um “cenário” padrão que possa ser aplicado aos shaders a serem testados.
- d) Realizar testes de performance dos shaders para cada game engine.
- e) Avaliar os resultados obtidos após a conclusão dos testes.

1.3 ESTRUTURA DO TRABALHO

Este estudo está divido em cinco capítulos. O primeiro capítulo contém uma breve explanação do conteúdo introdutório, que detalha o problema de pesquisa, delimita o objetivo geral e enumera os objetivos específicos.

Já no segundo capítulo, há uma exposição da revisão bibliográfica associada ao estudo, explicando conceitos fundamentais para seu entendimento como o desenvolvimento dos shaders, o funcionamento da API de renderização OpenGL, as ferramentas de motores de jogos, os conceitos técnicos de shaders e as principais formas de otimização.

O terceiro capítulo descreve a metodologia utilizada para a realização do trabalho e contém as etapas para o delineamento da sequência lógica do estudo. O quarto capítulo apresenta os objetos desse estudo e os dados pertinentes.

O quinto capítulo abrange a conclusão do trabalho, expondo as considerações finais; sumarizando os resultados do estudo e as ponderações a respeito dos dados e informações demonstradas. Por último, nas referências bibliográficas, estão especificadas todas as fontes, além de seus devidos autores, aplicadas para a realização deste trabalho.

2 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados os assuntos fundamentais para o entendimento dos processos envolvidos no uso das tecnologias abordadas no decorrer do trabalho. No início será discutida a criação dos shaders e seu uso ao longo do tempo, em seguida serão expostos itens de ordem técnica sobre os shaders e os motores de jogo. Ao final será tratada a integração dessas tecnologias com os processos de otimização.

2.1 EVOLUÇÃO DA PROGRAMAÇÃO DE SHADERS

As representações visuais feitas através de imagens são até hoje uma característica importante da formação da humanidade. Através do sentido da visão conseguimos absorver informações rapidamente, fazer associações durante o aprendizado e o estudo, ou ainda distinguir se algo é visualmente agradável o suficiente ou não para prender nossa atenção (LUTEN, 2014).

O acesso aos primeiros computadores era restrito devido aos custos elevados e a logística complexa. A representação visual dos pulsos elétricos gerados pelo seu processamento de dados era feita através de várias lâmpadas conectadas em placas ou de cartões de papel perfurados. Esse cenário começou a mudar depois da aplicação da tecnologia do tubo de raios catódicos (CRT), em 1951, pelo Massachusetts Institute of Technology (MIT) para visualizar a saída de um programa instantaneamente (LUTEN, 2014).

O estabelecimento da computação gráfica teve início 10 anos depois. A partir da criação de um programa de computador por Ivan Sutherland chamado Sketchpad, que permitia desenhar formas geométricas utilizando uma caneta óptica em um CRT com visualização em tempo real (LUTEN, 2014). Isso causou uma mudança de padrão na forma como as pessoas entendiam e utilizavam os computadores e foi o ponto de partida para desenvolvimento da computação gráfica em tempo real.

Com a criação dos circuitos integrados a indústria de microprocessadores sofreu um crescimento enorme. Os computadores deixaram de ser um monopólio das grandes companhias e tornaram-se mais acessíveis a pessoas simples. Isso abriu várias possibilidades para o mercado de computadores pessoais, entre elas destaca-se o surgimento das primeiras placas gráficas produzidas pela IBM (International Business Machines).

Com as melhorias de hardware disponíveis, a indústria de jogos eletrônicos tinha mais recursos para explorar. Os casos mais marcantes, mostrados na Figura 1, se deram pela empresa id Software na década de 90. O primeiro sendo Wolfenstein 3D — que na realidade

utilizava o modo 7 do Super NES (Nintendo Entertainment System) para emular a ambientação tridimensional — e o segundo sendo Doom que fazia uso de renderização com perspectiva 3D em tempo real por meio de software desenvolvido pela própria id Software.

Figura 1 – A esquerda Doom fazia uso de 3D real enquanto a direita Wolfenstein posicionava imagens 2D em diferentes camadas para simular a profundidade.



Fonte: Retro Refurbs (2021)

Paralelamente, a Silicon Graphics (SGI) — companhia especializada em computação gráfica 3D — trabalhava no lançamento da Open Graphics Library (OpenGL), uma API (Application Programming Interface) open source padronizada multiplataforma de processamento de gráficos de computador em tempo real derivada de outra biblioteca proprietária da mesma empresa, a IRIS GL (Integrated Raster Imaging System Graphical Library) e que rapidamente dominou o mercado (LUTEN, 2014).

A Microsoft, para competir, comprou a empresa RenderMorphics, criadora da API Reality Lab, que teve o nome alterado para Direct3D e foi distribuído como um SDK (Software Development Kit) conhecido como DirectX, que acabou sendo o concorrente direto da OpenGL (LUTEN, 2014). Essa rivalidade acabou sendo benéfica tanto para o mercado de jogos eletrônicos quanto para os seus consumidores, já que acelerou o desenvolvimento de tecnologias que exploravam ao máximo o potencial do hardware disponível.

Em 1999, a empresa NVIDIA lançou a placa gráfica GeForce 256 (Figura 2), que possuía a tecnologia T&L (Transform & Lighting) que movia os cálculos de transformação e iluminação de vértices da CPU (Central Process Unit) para a GPU (Graphics Processing Unit), aumentando a velocidade em operações matemáticas de ponto flutuante. Nos anos seguintes houve um crescimento exponencial de performance de GPU.

Figura 2 – Hardware da placa gráfica da NVIDIA.



(a) GeForce 256



(b) GPU da GeForce 256

Fonte: Wikimedia (2021)

Uma GPU é um circuito eletrônico projetado para realizar manipulações rápidas em memória para acelerar a criação de imagens em um buffer de quadros que envia a saída para uma tela. Em aplicações que exigem muitas operações vetoriais, o poder de computação paralelo da GPU consegue entregar maior performance que uma CPU convencional. Daí seu vasto uso em jogos eletrônicos, mas também em outras áreas, principalmente na ciência (SHEA; LIU, 2013).

Até então shaders eram bastante utilizados por melhorar a performance eliminando carga de trabalho excessiva da CPU, porém sua programação era difícil devido a sintaxe utilizada ser semelhante à programação em Assembly. A Microsoft então lançou a versão 9.0 do Direct3D que trazia High-Level Shader Language (HLSL) que permitia a programação de shaders em alto nível e possuía uma sintaxe bastante parecida com C. Enquanto isso, OpenGL também trouxe a sua própria linguagem de alto nível chamada GLSL (OpenGL Shading Language) (LUTEN, 2014).

2.1.1 Como o OpenGL funciona

A API do OpenGL desenha gráficos em uma memória especializada em quadros de imagem (frame buffer). Ela oferece suporte tanto a geometrias 3D quanto a imagens simples. O modelo de funcionamento dessa API pode ser descrito como cliente-servidor, pois a aplicação (cliente) faz solicitações por meio de comandos que são interpretados e processados pela implementação OpenGL (servidor) (ROST, 2006). É importante destacar que a sincronia entre cliente e servidor e suas informações/dados não ocorre quando um comando é executado mas sim quando ele é emitido.

Os comandos são sempre processados na ordem em que são recebidos pelo servidor (execução fora de ordem não é permitida). Os dados passados para um comando OpenGL são interpretados e copiados em memória caso seja necessário e as modificações subsequentes feitas pela aplicação não surtem efeito nos dados que estão armazenados internamente pelo OpenGL. Esses procedimentos são uma forma de garantir que um primitivo — segundo Abdala (2019), uma representação discreta em grade de um elemento geométrico fundamental, e.g. ponto, linha, círculo, etc — seja desenhado apenas se o primitivo anterior houver sido completamente desenhado (ROST, 2006).

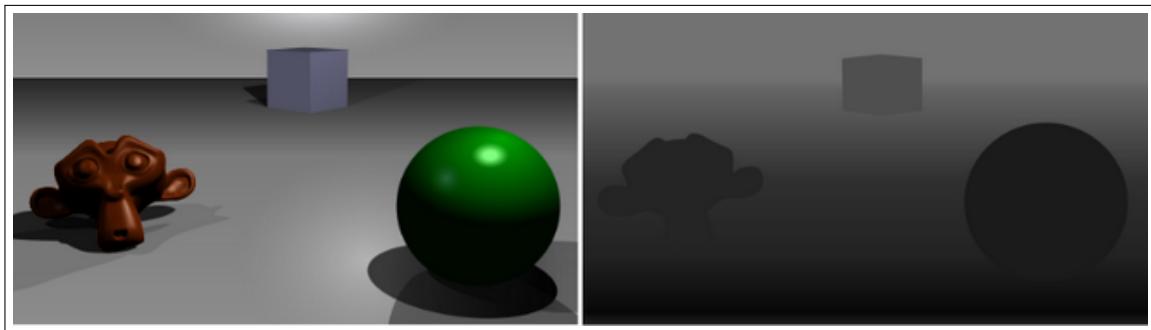
OpenGL foi projetada para atuar como uma máquina de estados composta de parametros que definem o comportamento da pipeline de renderização e da forma que as primitivas são transformadas em pixels na tela. O estado é composto por uma estrutura de dados chamada contexto gráfico que é gerenciada pelo sistema de janelas do SO (Sistema Operacional) (ROST, 2006).

O principio de funcionamento dessa API é transformar dados vindos de uma aplicação em algo visível na tela, esse processo é chamado de renderização e normalmente é acelerado por hardware com design específico chamado de acelerador gráfico, porém suas operações podem ser parcial ou totalmente implementadas por software executado pela CPU. Em um sistema de janelas, a janela que corresponde a região da memória gráfica que é modificada durante a renderização é chamada de frame buffer. Já em um cenário sem janelas (i.e. tela cheia) o frame buffer corresponde a toda a tela (ROST, 2006).

Para que uma janela suporte a renderização ela precisa de alguns elementos: até quatro buffers para as cores, um buffer de profundidade (Figura 3), um *stencil buffer* (Figura 4), um buffer de acumulação, um *multisample buffer* e um ou mais buffers auxiliares. A maioria dos hardwares suporta o carregamento duplo, técnica que faz uso de um buffer frontal e um buffer

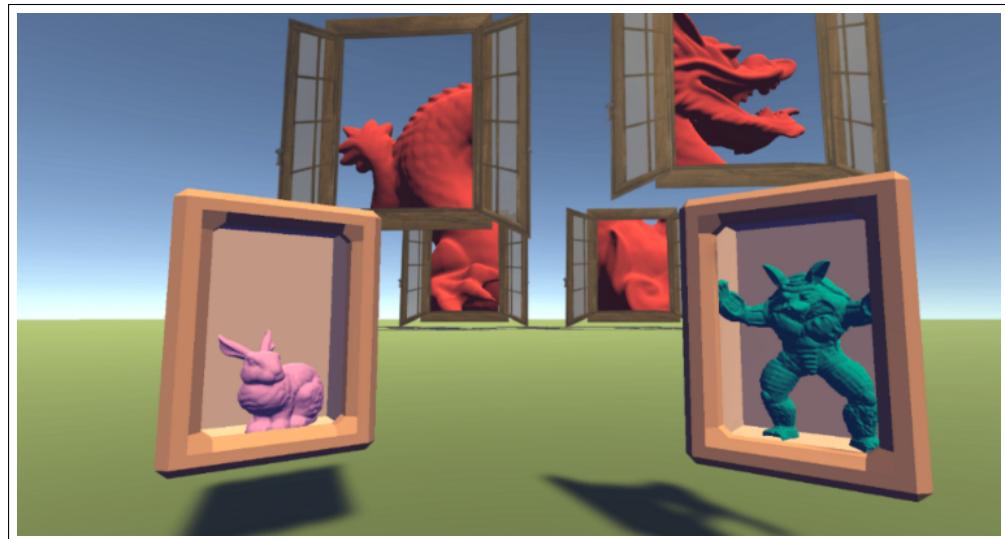
posterior para que o processo de renderização seja realizado em plano de fundo e então quando terminar seu conteúdo é trocado com o do buffer frontal para exibir o resultado final e iniciar a nova renderização. Isso ajuda a conseguir animações suaves à taxas interativas (ROST, 2006).

Figura 3 – O buffer de profundidade é mostrado em tons de cinza sendo que objetos próximos ficam com tonalidade mais escura enquanto objetos distantes assumem uma tonalidade mais clara.



Fonte: Larra (2021)

Figura 4 – O stencil buffer permite a customização da forma como objetos 3D são renderizados.



Fonte: <https://www.ronja-tutorials.com/assets/images/posts/022/Result.gif>

No caso do suporte a visualização 3D estéreo, mais dois buffers serão utilizados em conjunto com os dois citados anteriormente para criar uma combinação com quatro buffers de cor que são divididos para cada olho. Se um objeto 3D precisa ser desenhado com remoção de superfície encoberta, o buffer de profundidade entra em ação comparando o valor da profundidade de cada pixel dos objetos em cena para determinar qual será visível ou obscurecido. E há ainda a

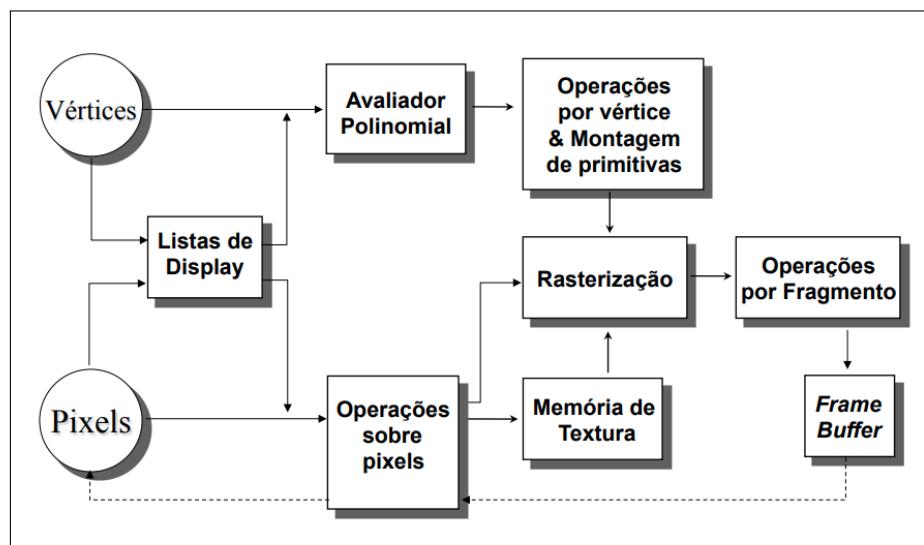
opção do uso de um *stencil buffer* para aplicar operações complexas utilizando máscaras com o objetivo de determinar onde cada pixel deve ser atualizado ou não (ROST, 2006).

O buffer de acumulação é capaz de reproduzir efeitos complexos como suavização em tela cheia de alta qualidade, profundidade de campo e desfoque de movimento. Ele funciona como um buffer de cor, porém com maior precisão, capaz de acumular imagens para produzir uma única imagem composta. Segundo essa linha, o *multisample buffer* é capaz de produzir várias amostras da renderização para realizar suavização sem precisar renderizar a cena mais de uma vez (ROST, 2006). Por último, os buffers auxiliares servem para guardar dados genéricos.

2.1.1.1 Pipeline gráfica do OpenGL

Para que a máquina de estados do OpenGL possa operar corretamente, foi definida uma ordem específica em que as operações envolvidas no processo de renderização precisam ser realizadas, essa padronização é chamada de *pipeline* gráfica (ROST, 2006) e pode ser vista na Figura 7. Todos os dados necessários para desenhar a geometria estão contidos em espaço em memória e podem ser lidos pelo OpenGL de três maneiras diferentes.

Figura 5 – Pipeline gráfico do OpenGL.



Fonte: Anselmo (2021)

A primeira seria enviar um vértice de cada vez utilizando alguns comandos intermitentes para manipular atributos dos vértices. A segunda seria utilizar matrizes de vértices, o que oferece melhor performance devido a forma de organização dos dados, pois são utilizados ponteiros e mais dados podem ser processados de uma vez. Esses dois casos citados acima fazem uso do modo imediato pois as primitivas são renderizadas assim que são especificadas (ROST, 2006).

O terceiro modo seria utilizar algum dos dois procedimentos citados acima implementando uma lista de exibição, que é uma estrutura de dados que guarda comandos para execução futura. Algumas vantagens desse método no que diz respeito a performance seria a possibilidade de otimizar os comandos contidos na lista, ou ainda guardar os comandos na memória do acelerador gráfico para uma melhor performance de desenho (ROST, 2006).

Como todas as primitivas geométricas podem ser descritas por vértices, as curvas e as superfícies podem ser descritas pelas funções polinomiais chamadas funções base. A função do avaliador polinomial nesse caso é derivar os vértices para conseguir representar superfícies e curvas. Isso é feito através do método de mapeamento polinomial, que produz as normais da superfície, as coordenadas da textura, as cores, e valores de coordenadas espaciais dos pontos de controle (VIEIRA, 2017).

A próxima etapa é o estágio das operações por vértice que converte os vértices em primitivas. Alguns dados do vértice são transformados em matrizes de pontos flutuantes. Nesta etapa ocorre a projeção de coordenadas do espaço do mundo para o espaço da tela. Inclui algumas etapas como geração e transformação de coordenadas de textura, e também cálculos de luz para produção dos valores de cor (VIEIRA, 2017). Por isso é normal que essa etapa exija mais recursos computacionais.

Logo em seguida ocorre a montagem das primitivas por meio do *clipping* (eliminação de parte da geometria desnecessária para a renderização), que testa se a primitiva está totalmente dentro do plano de visualização, caso sim ela é repassada para o devido processamento. Caso ela esteja totalmente fora do plano de visualização ela é rejeitada. Se a primitiva estiver parcialmente visível no plano, ela é dividida para que somente a porção visível siga para processamento.

Outra operação desse estágio é a projeção das coordenadas da perspectiva para coordenadas da janela. Além disso, há uma etapa opcional de *culling* onde os polígonos são testados para saber quais faces serão descartadas (ROST, 2006). Paralelamente a esse processo, os dados de pixels contidos em uma matriz na memória do sistema são empacotados e processados por um mapa de pixels. Os resultados, que serão então empacotados em um formato apropriado e retornados a uma matriz de memória do sistema, podem ser escritos na memória da textura ou emitidos à etapa de rasterização.

Rasterização é a etapa de conversão de dados tanto geométricos como de pixel em fragmentos. Cada fragmento passa por mais algumas operações como: texturização; aplicação do valor de cor e profundidade; cálculos de névoa; testes de profundidade, transparência e remoção de faces ocultas (VIEIRA, 2017). Ao final são armazenados os valores no *frame buffer*.

Apesar de possuir muitos processos, é uma etapa relativamente simples e pode ser executada eficientemente para milhões de pixels por segundo com o hardware disponível atualmente.

Na etapa de texturização a API tem capacidade de trabalhar com quatro tipos de texturas. Texturas de uma dimensão (vetor de pixels), texturas 2D (matriz mxn de pixels), texturas 3D (matriz com uma dimensão a mais para guardar informações adicionais e.g. profundidade), e mapas cúbicos (normalmente usado para simular reflexões de ambiente). A API também pode trabalhar com formatos de imagem compactados, esses usam significativamente menos memória e melhoram a performance (ROST, 2006).

É importante destacar que a API também fornece a possibilidade de utilizar texturas do tipo *mipmap* — várias representações da mesma imagem, porém cada uma tem metade da resolução da anterior — em conjunto com o parâmetro de nível de detalhe que será detalhado mais adiante, mas que de forma simplificada permite a otimização do processo de renderização de objetos que estão distantes da câmera.

$$f((x,y), \mathbf{v}) = p \circ r(g \circ h(\mathbf{v}), (x,y)) \quad (2.1)$$

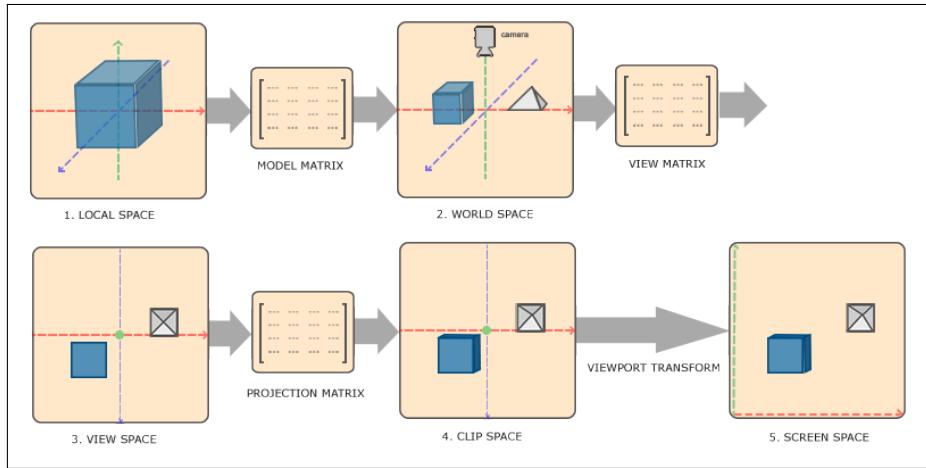
Uma descrição matemática da pipeline de renderização está na Equação 2.1 onde (x, y) é a posição na tela de cada pixel, \mathbf{v} é um conjunto de primitivas, p é o shader de fragmentos, g é o shader de geometria, h é o shader de vértice e r é o estágio de rasterização onde as primitivas são convertidas em pixels com atributos de geometria interpolados (WANG *et al.*, 2014).

Por fim, é importante mencionar que existem dois modos principais de renderização: o modo direto calcula as luzes e materiais para cada geometria visível e depois resolve qual está mais próxima da câmera para então decidir quais exibir. Já o modo diferido realiza várias passadas para as geometrias e só então realiza os cálculos, dessa forma apenas os fragmentos visíveis são considerados. O segundo modo é mais eficiente para lidar com muitas luzes, porém não oferece suporte a transparência e suavização (ŠMÍD, 2017).

2.1.1.2 Matrizes de transformação de coordenadas

Esse é um tópico mais complexo, mas que também é importante para entender como a pipeline gráfica do OpenGL transforma descrições de objetos tridimensionais em imagens 2D que são exibidas na tela (Figura 6). Algo semelhante a como uma câmera pode ser usada para criar uma representação em imagem de algo no mundo real. Entretanto nesse caso o primeiro

Figura 6 – Processo de transformação entre os sistemas de coordenadas e seus espaços.



Fonte: Learn OpenGL (2021)

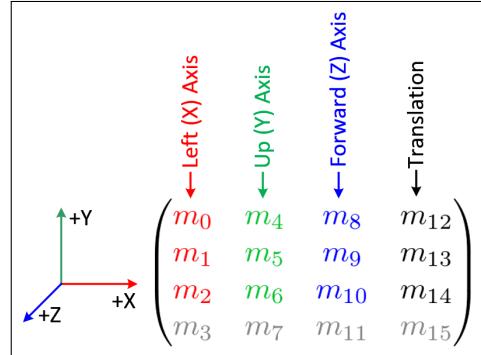
passo consiste em pegar as informações do modelo do objeto 3D, como posição dos vértices e normais da superfície, para interpretá-las como coordenadas do espaço do objeto (ROST, 2006).

Como cada objeto tem suas próprias características é necessário definir um sistema de coordenadas uniforme para que seja possível usar vários objetos em uma única cena. Para isso é utilizado o sistema de coordenadas global, e aqui a API vai um passo além e realiza mais uma conversão para o sistema de coordenadas de olho levando em consideração a posição da câmera na cena, seu ponto focal (para onde a câmera está olhando) e o vetor de direção para cima (e.g. a orientação da câmera) (ROST, 2006).

$$\begin{bmatrix} x_{olho} \\ y_{olho} \\ z_{olho} \\ w_{olho} \end{bmatrix} = M_{modelView} \cdot \begin{bmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{bmatrix} \quad (2.2)$$

Na equação 2.1, a matriz *modelView* é uma multiplicação das matrizes de conversão de coordenadas de espaço de objeto para espaço global e de espaço global para espaço de olho. O cálculo das normais é semelhante, a diferença é que é utilizada a matriz transposta da inversa da matriz *modelView* para multiplicar um vetor de normais. Como é possível ver na Figura 7 os três elementos mais à direita (m_{12}, m_{13}, m_{14}) são para transformação de translação. O m_{15} é uma coordenada homogênea (Apêndice — A) usada para transformação para o espaço de projeção. Os três conjuntos de elementos (m_0, m_1, m_2), (m_4, m_5, m_6) e (m_8, m_9, m_{10}) são usados para rotação e escala e representam os três eixos ortogonais x, y e z (AHN, 2013).

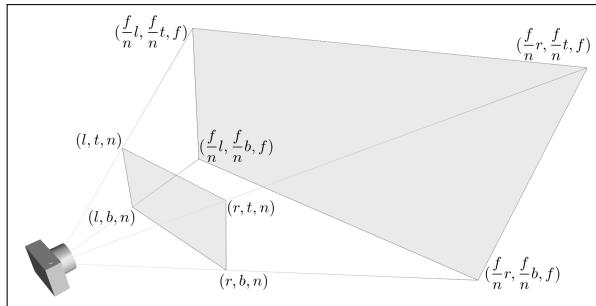
Figura 7 – Conteúdo das quatro colunas da matriz *modelView*.



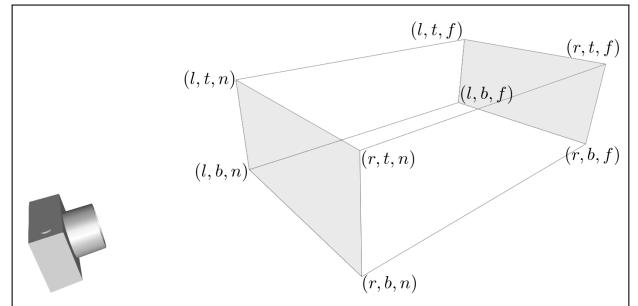
Fonte: Ahn (2021)

Após essa conversão, as coordenadas obtidas são multiplicadas pela matriz de projeção para definir como os vértices serão projetados na tela, os valores dessa matriz dependem se modo de projeção utilizado é em perspectiva ou ortográfico (Figura 8) e são mostrados nas equações 2.3 e 2.4 respectivamente.

Figura 8 – Modos de projeção de câmera. As letras correspondem a *left*, *right*, *bottom*, *top*, *near* e *far*.



(a) Viewing Frustum em perspectiva



(b) Viewing Frustum ortográfico

Fonte: Ahn (2021)

$$M_{persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (2.3)$$

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

Os valores obtidos nessa operação são normalizados em uma região cúbica definida pelos pontos (-1, -1, -1) e (1, 1, 1) para espaço de coordenadas de dispositivo normalizado, isso significa que os valores passam a ser algum valor entre -1 e 1. Essa etapa é necessária para que a área de visualização seja apropriadamente mapeada em uma janela de exibição de tamanho arbitrário. Por último, as coordenadas são convertidas para o sistema de coordenadas de tela. A partir desse ponto elas continuam para o processo de rasterização da pipeline do OpenGL (AHN, 2013).

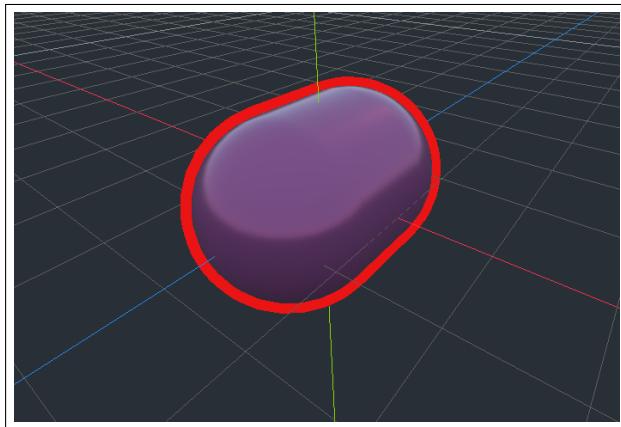
2.1.2 OpenGL Shading Language

Devido à necessidade crescente de substituir funcionalidades fixas por programabilidade em áreas que ficavam cada vez mais complexas, como processamento de vértices e fragmentos, foi desenvolvida uma solução que adicionou estágios programáveis para resolver esse problema. Essa solução foi a introdução da linguagem de sombreamento GLSL, feita para ser executada nos dois processadores programáveis existentes no OpenGL: o processador de vértices e o processador de fragmentos (portanto os respectivos nomes *vertex shader* e *fragment shader*) (ROST, 2006).

Um shader pode então ser definido como um código escrito em uma linguagem de sombreamento (HLSL, GLSL, RSL e etc) com o propósito de ser executado por um dos processadores programáveis do OpenGL. Um programa de shader é então um conjunto de shaders compilados executáveis (ROST, 2006). A Figura 9 mostra a implementação do código-fonte 2 para criar um efeito de linha de contorno em volta de um objeto 3D utilizando a linguagem GLSL.

A linguagem de sombreamento GLSL faz uso de uma sintaxe bastante similar a linguagem de programação C. Seus tipos incluem vetores e matrizes por serem estruturas fundamentais para cálculos matemáticos com operações para gráficos 3D. O uso de números de ponto flutuante (*float*) também é fundamental para conseguir altos níveis de precisão a troco de

Figura 9 – Demonstração de um shader simples de linha contorno.



Fonte: Elaborado pelo autor (2021)

performance, por isso é possível especificar o nível de precisão desejado ao utilizá-los. Além disso ela oferece suporte a laços, chamadas a sub-rotinas, expressões condicionais e conta com funções embutidas próprias para o desenvolvimento de shaders (ROST, 2006).

Essa linguagem possibilitou aos desenvolvedores implementar um conjunto de diferentes técnicas para conseguir obter uma variedade enorme de efeitos visuais; não somente isso mas o fato de que essas técnicas são implementadas com aceleração via hardware pela GPU (com processamento paralelo) proporciona um aumento drástico de performance e libera carga da CPU para realizar outras tarefas (ROST, 2006).

O processador de vértices é uma unidade programável que realiza operações nos valores de vértices recebidos e seus dados associados. Essas operações consistem em transformação de vértices, transformação e normalização das normais, geração e transformação das coordenadas de textura, iluminação e aplicação de cor. Shaders feitos para rodar nesse processador são chamados de shaders de vértice (ROST, 2006).

Variáveis de atributo (variável global somente leitura alterada por vértice) são utilizadas para passar valores da aplicação para o processador de vértices. Já as variáveis uniformes (variável global somente leitura alterada por primitiva) são utilizadas para passar dados tanto para o processador de vértices como de fragmentos. Por último há as variáveis variantes cuja função é transportar informação do processador de vértices para o processador de fragmentos (ROST, 2006).

O processador de vértices atua em um vértice por vez e uma implementação pode ter múltiplos processadores operando em paralelo (o mesmo vale para o processador de fragmentos). Logo, o shader de vértice é executado uma vez para cada vértice, sendo que há uma possibilidade de perda de performance caso um shader de vértice precise calcular mais variáveis variantes do

que o que é necessário pelo shader de fragmentos. Por outro lado, o processador de fragmentos é responsável por realizar algumas operações como interpolação de valores, acesso e aplicação de texturas, névoa e soma de cor (ROST, 2006).

Cabe ressaltar que, em termos de performance, normalmente os desenvolvedores preferem utilizar um shader de vértice mais genérico em conjunto com um shader de fragmento, pois assim é possível utilizar apenas um subconjunto das variáveis contidas no shader de vértice e ainda sim reduzir tempo e custos de desenvolvimento e manutenção para uma grande quantidade de shaders (ROST, 2006).

2.1.3 Direct3D (HLSL) *versus* OpenGL (GLSL)

Direct3D é uma API para desenvolvimento de aplicações gráficas nativas para plataformas proprietárias da Microsoft. Ela evoluiu muito durante os anos 90 e superou a OpenGL. Conceitualmente, a pipeline gráfica de ambas APIs são bem semelhantes, mas uma diferença importante se dá em termos de design de gerenciamento dos estágios de shaders, onde OpenGL faz uso de um objeto (programa de shader) que contém múltiplos shaders enquanto que Direct3D expõe um contexto de renderização diretamente para a criação de shaders. Quanto às linguagens (GLSL e HLSL), são muito parecidas e os desenvolvedores conseguem transcrever instruções facilmente de uma para a outra (MICROSOFT, 2020).

Essa interface faz parte da biblioteca de APIs do DirectX e é usada em consoles Xbox e sistemas Windows. DirectX também inclui algumas outras bibliotecas (e.g. Direct2D e DirectSound). Com o passar do tempo o foco voltou-se para APIs de acesso de baixo nível aos hardwares gráficos como é o caso do DirectX 12. Entretanto sua versão anterior ainda é mais utilizada devido a questões de compatibilidade (HASU, 2018).

Considerando a pouca diferença em capacidade de renderização existente entre essas duas interfaces, a escolha sobre qual usar depende muito da plataforma alvo de desenvolvimento. Direct3D é específica para plataformas da Microsoft e é amplamente suportada por fornecedores de hardware gráfico, especialmente em computadores desktop. OpenGL é open source e possui bastante aceitação no espaço de desenvolvimento mobile, principalmente devido ao desenvolvimento da OpenGL ES — uma subseção do OpenGL projetada especialmente para sistemas embarcados como smartphones e consoles portáteis (VARCHOLIK, 2014).

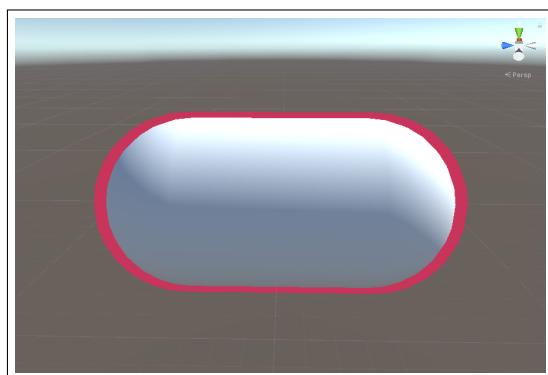
Uma das principais diferenças é o ambiente de execução. O compilador HLSL traduz o código para linguagem de máquina que é processada pelo driver do DirectX. Enquanto que

no caso do OpenGL os fornecedores de hardware, por serem responsáveis pela implementação do compilador, possuem muito mais liberdade para realizar otimizações em shaders. Para mitigar essa diferença a Microsoft fornece uma solução (DirectX Effects Framework) para desenvolvedores elaborarem programas de shaders iguais para hardwares com menor e maior capacidade de processamento (ROST, 2006).

2.1.4 High-Level Shader Language

Essa é uma linguagem de shader criada em 2002 (acompanhou o DirectX 9) que também assemelha-se à linguagem C. Ao longo dos anos foram adicionadas melhorias como suporte a *Multithread*, adição de uma API para uso de GPGPU (GPU de propósito geral) e suporte a tesselação. Na Figura 10 é mostrada a implementação de um shader de contorno (ver código-fonte 1) similar ao anterior para realçar as diferenças e semelhanças entre as duas linguagens.

Figura 10 – O mesmo resultado de contorno obtido com HLSL.

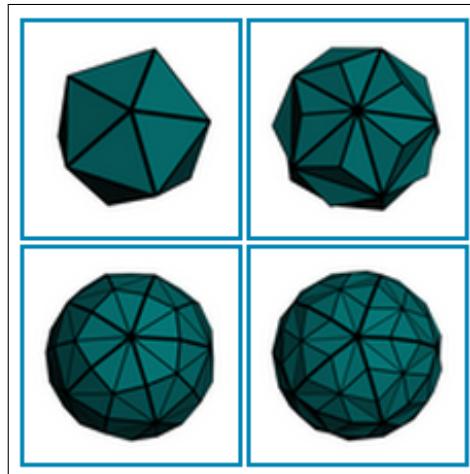


Fonte: Elaborado pelo autor (2021)

Uma GPU de propósito geral é uma unidade de processamento gráfico que realiza cálculos genéricos que normalmente seriam feitos pela CPU. São utilizadas para realizar tarefas custosas como cálculos de física, criptografia e computações científicas, pois é possível tirar proveito do paralelismo. Da mesma forma que um núcleo pode ser utilizado para renderizar múltiplos pixels simultaneamente, ele também é capaz de processar múltiplos fluxos de dados ao mesmo tempo (TECHTARGET, 2015).

Tesselação é mais um processo na pipeline gráfica responsável por adicionar detalhes a objetos diretamente pela GPU. De maneira geral, mais detalhes geométricos (mais vértices), resultam em uma renderização "mais bonita". Seu modo de funcionamento consiste em subdividir um objeto dinamicamente e sem o custo adicional de reprocessamento de geometria. Isso permite

Figura 11 – Maiores níveis de tesselação produzem aumento no número de vértices.



Fonte: Wikimedia (2021)

um sistema de nível de detalhe dinâmico e menos utilização do barramento de gráficos, o que melhora a performance (VARCHOLIK, 2014).

Esse processo é relativamente novo e ocorre na etapa do shader de geometria, que adiciona um passo de criação de geometria na pipeline gráfica após o shader de vértices. Isso permite que o programador implemente tesselação automática para geometrias complexas, ou realize operações gráficas dependentes de geometria como silhuetas e sombras (BAILEY; CUNNINGHAM, 2007).

Um shader de geometria pode ter vários usos, por exemplo adicionar ou gerar mais primitivas (pontos, linhas, triângulos ou quadriláteros). Porém eles aceitam apenas uma quantidade limitada de topologias. Sua saída consiste em pontos, linhas ou triângulos e segue continuamente na pipeline. Basicamente, são utilizados os produtos dos shaders de vértice e tesselação para montagem de primitivas (HASU, 2018).

Shaders de tesselação interpolam geometria para acrescentar detalhes geométricos que permitem aos desenvolvedores realizar subdivisões adaptáveis, utilizar modelos mais "grosseiros" que serão refinados pela GPU, aplicar mapas de deslocamento detalhados sem fornecer detalhes de geometria, adaptar qualidade visual exigindo nível de detalhe e criar silhuetas suaves. Resumindo, tesselação é um processo que divide uma superfície em uma malha suavizada de triângulos e pode aumentar a qualidade da imagem final (HASU, 2018).

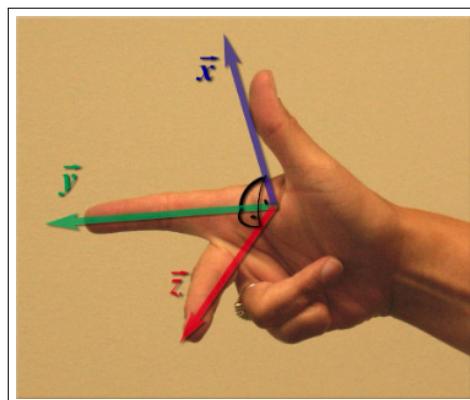
Eles possuem acesso a todas as informações na pipeline. São portanto capazes de escolher parâmetros de tesselação dinamicamente dependendo da informação lida. A principal diferença em relação aos shaders de vértice é que enquanto esses modificam os vértices individualmente sem referência às primitivas, o shader de tesselação amplifica uma

única primitiva (HASU, 2018).

Nível de detalhe é um fator chave de otimização utilizado por game engines para alcançar renderização de alta qualiadade com melhor performance. Além disso é desejável evitar mudanças frequentes em shaders e chamadas de desenho utilizando um número reduzido de shaders. Isso minimiza a sobrecarga da CPU e ajuda a GPU a desenhar mais objetos em um frame com shaders com nível de detalhe (HE *et al.*, 2015).

Em relação ao sistema de coordenadas, Direct3D faz uso do sistema de mão esquerda enquanto OpenGL utiliza o sistema de mão direita (Figura 12). Porém as aplicações são livres para utilizar seu próprio sistema de coordenadas. Um exemplo é o software de modelagem 3D Blender que usa o sistema de mão direita, enquanto ambas Unity e Unreal utilizam o sistema de mão esquerda (HASU, 2018).

Figura 12 – Regra da mão direita.



Fonte: Luiz (2021)

2.2 CONCEITOS TÉCNICOS DE SHADERS

A introdução do HLSL trouxe uma linguagem semelhante a C com estruturas extrar para tratar vetores, matrizes e outros elementos relacionados a graficos. Com a melhoria da legibilidade e da produtividade, o seu uso permite aos desenvolvedores focar no código e reutilizar otimizações de alto nível (como fazer uso de tipos correspondentes e funções embutidas). O compilador HLSL contém truques de otimização de baixo nível que podem ajudar (RIGUER, 2002).

Ao estudar computação gráfica a dúvida mais comum ao se deparar com certos termos utilizados é "o que é um shader". Essa palavra pode causar uma certa estranheza no início mas sua definição não é nenhum bicho de sete cabeças. Shaders são apenas pequenos programas

(assim como um reproduutor de mídia ou uma calculadora de um computador) que são executados diretamente pela GPU ao invés da CPU. Isso permite a redução da carga de trabalho gráfico da CPU pelo redirecionamento das tarefas para a GPU que possui hardware especializado para isso (LUTEN, 2014).

Um shader é um procedimento que computa alguns aspectos visuais de um objeto, como cor e deslocamento de superfície, tonalidade e direção da luz e efeitos volumétricos. São utilizados para especificar transformações de vértices e cores de pixels. Pode-se pensá-los como uma função f que recebe um conjunto de valores (como coordenadas de posição, normais e texturas) v_i e um conjunto de parâmetros u_i (informação de cores e luz) e retorna a cor C_{xy} de cada objeto em cada pixel xy da imagem final (PELLACINI, 2005).

Shaders programáveis são uma das ferramentas mais impressionantes desenvolvidas para computação gráfica nos últimos anos. Através de seu uso, programadores ganharam flexibilidade para aplicar efeitos vértice-por-vértice e pixel-por-pixel com o processamento paralelo em gráficos interativos entre as mais diversas áreas como ciência, arte, engenharia, entre outras (BAILEY; CUNNINGHAM, 2007).

Figura 13 – Demonstração de como é possível criar visuais únicos utilizando shaders.



Fonte: Adaptado de Hologram Planet (2021)

Tecnicamente falando, um shader contém um conjunto de instruções que são executadas concorrentemente para cada pixel desenhado na tela. Essa forma de operação abre um leque de possibilidades, onde é possível por exemplo atribuir um comportamento para cada pixel baseado na sua posição na tela. Em uma comparação com programação procedural, ele funcionaria como uma função que recebe uma posição e retorna uma cor, sendo que após a compilação seu tempo de execução é extremamente rápido (VIVO; LOWE, 2015).

Uma metáfora para ajudar a compreender a dimensão da complexidade do processamento de um shader seria imaginá-lo como um bloco de várias tarefas que passa por uma linha de produção industrial. As tarefas podem ser pequenas ou grandes e consequentemente podem demandar mais processamento e energia. No caso da CPU cada trabalho seguinte teria que esperar o término do atual para começar (VIVO; LOWE, 2015). É interessante ressaltar que hoje em dia existe a tecnologia de multiprocessamento, onde os computadores normalmente possuem grupos de quatro processadores que atuam em conjunto para realizar as tarefas.

Considerando uma tela com resolução de 800x600, significa que 480.000 pixels precisam ser processados a cada frame sendo que normalmente é utilizada uma taxa de 30 frames por segundo (FPS), então será necessário fazer 14.400.000 cálculos por segundo. Isso explica o fato de video games e outras aplicações gráficas exigirem muito mais poder de processamento que outros programas. Seu conteúdo gráfico implica em inúmeras operações por cada pixel, pois cada pixel na tela precisa ser computado, e também em perspectivas e geometrias de jogos 3D (VIVO; LOWE, 2015).

Esse cenário pode ser suficiente para sobrecarregar um microprocessador comum e fica pior quando leva-se em consideração as tecnologias que fazem uso seja de taxa de FPS maior, seja de resoluções maiores como 2K, e acima. Para resolver esse problema utiliza-se processamento paralelo. A GPU possui vários pequenos microprocessadores que funcionam concorrentemente, além disso ela possui funções matemáticas específicas aceleradas via hardware para realizar operações matriciais e trigonométricas rapidamente (VIVO; LOWE, 2015).

A dificuldade relativa de programar shaders levou ao desenvolvimento de ferramentas visuais que auxiliam a criação desses programas através do uso nós funcionais conectados entre si que remetem a uma estrutura de árvore. Esse tipo de ferramenta está presente na Unreal Engine desde 2005. Ao mesmo tempo que possuem utilidade, é importante garantir que shaders gerados por tais ferramentas seja otimizados. Otimização é importantíssima para encorajar desenvolvedores a criar shaders maiores e mais complexos (JENSEN *et al.*, 2007).

Conforme descrito em estudo por Wang *et al.* (2014) vários estudos sobre otimização de shaders já foram conduzidos, porém com efetividade apenas para o estágio de fragmentos. De certa forma a qualidade dos shaders depende bastante da experiência dos programadores e pode ser um processo bem demorado para níveis de complexidade maiores. Normalmente a computação mais custosa ocorre no shader de fragmentos.

Placas gráficas atuais oferecem suporte a quatro níveis de paralelismo: de dispositivo, de núcleo, de tarefa e de dados. O primeiro significa que vários processadores ou placas podem

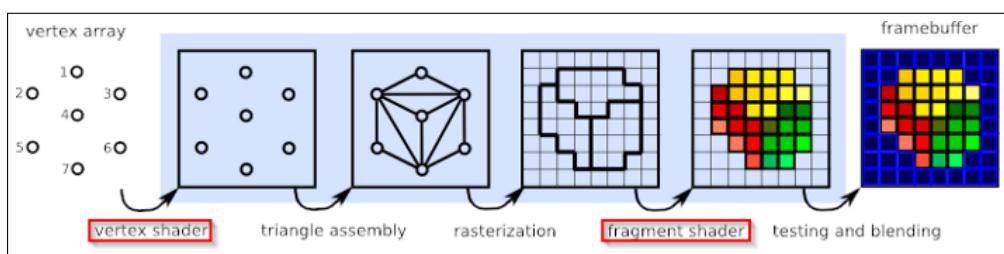
operar no mesmo sistema. O segundo significa que os múltiplos núcleos são independentes. O terceiro quer dizer que cada núcleo pode ter várias tarefas. Por fim, paralelismo de dados significa que múltiplas instruções podem agir em vários elementos de dados de uma vez (HASU, 2018).

Também é importante mencionar que os shaders de computação são um estágio fora da pipeline de renderização que são utilizados para calcular informações arbitrárias. Então são bem distintos quando comparados aos outros tipos de shaders. São ótimos para implementar algoritmos que fazem uso da GPGPU. Podem também ser utilizados para acelerar partes da renderização. Suas entradas e saídas são genéricas e a única definição é do espaço de execução que é abstrato. Os dados processados são agrupados em grupos de trabalho (menor unidade de processamento) de tamanho definido pelo programador que são executados aleatoriamente e em paralelo (HASU, 2018).

2.2.1 Vertex Shader

Um shader de vértice é um programa executado uma vez para cada vértice cujo é atribuído. As operações mais importante dessa etapa são as que envolvem cálculo de transformação e luzes. A aplicação desses shader permite uma paleta ilimitada de efeitos visuais renderizadas em tempo real. Dentre as principais aplicações merecem destaque o suporte a criação de animações realistas e a possibilidade de deformar superfícies para criar efeitos realistas de ondas (NVIDIA, 2019).

Figura 14 – Como os estágios de shaders se relacionam.



Fonte: Medium (2021)

O processador de vértices realiza as principais transformações de coordenadas descritas na Seção 2.1.1.2. Como nessa etapa há bastante informação sobre a geometria e o processador é capaz de realizar inúmeras operações, essa é uma ótima etapa para inserir código. Quando essas coordenadas deixam o estágio de processamento, elas são cortadas e mapeadas para o sistema de coordenadas de tela, prontas para serem rasterizadas (BAILEY; CUNNINGHAM,

2007).

O shader de vértices prepara o ambiente de shader para o processamento de vértices, a tesselação e os shaders de geometria e também para a rasterização e para o shader de fragmentos. Aqui também podem ocorrer mudanças de coordenadas. Os dados recebidos são enviados para o estágio de processamento de vértices da pipeline. Isso inclui dados que as aplicações enviam aos shaders (HASU, 2018).

Podem ser passados coordenadas de vértice para o shader de fragmentos utilizando geometria de espaço de objeto ou de olho. Por exemplo, para tesselação, o shader de vértices pode passar primitivas conectadas com os dados que controlam a subdivisão que deve ser realizada, enquanto a saída do shader de tesselação consiste em uma coleção de vértices para a nova geometria. No fim o principal objetivo dos shaders de vértice é pré-processar os vértices e gerenciar os atributos que seguirão para a pipeline (HASU, 2018).

2.2.2 Fragment Shader

Um shader de fragmento é um programa executado uma vez para cada pixel. Fragmentos são estruturas de dados (contidas em cada pixel) que são criadas pela rasterização das primitivas. Um fragmento contém todos os dados necessários para atualizar seu espaço no *frame buffer*. O processamento desses fragmentos consistem em operações feitas em cada pixel, sendo que as mais notáveis são a leitura da memória de textura e a aplicação do valor de textura para cada fragmento (ROST, 2006).

Nessa etapa o processador de fragmentos recebe as informações de cada pixel (seus valores de vermelho, verde, azul, transparência e coordenadas de textura). Cada pixel também possui informação recebida do processador de vértices e interpolada na rasterização. Esse processador também pode acessar informações globais como a posição das luzes e seu trabalho final é processar essas informações e produzir a cor final de cada pixel (ou descartá-lo). Sua grande utilidade consiste na possibilidade de customização da aparência dos pixels de acordo com as necessidades do programador (BAILEY; CUNNINGHAM, 2007).

Esse é o shader responsável por produzir a cor final para cada pixel a partir de variáveis de estado e valores interpolados através de polígonos. Também são responsáveis por computar a cor e a intensidade da luz de cada fragmento. Além disso, são capazes de lidar com tipos diferentes de propriedades de vértices (os principais são coordenadas de textura e profundidade de pixels) (HASU, 2018).

Dependendo da resolução definida, algo em torno de 2 milhões de pixels podem precisar ser processados e renderizados a cada frame (60 FPS). Isso facilmente gera uma carga computacional enorme. Felizmente, com a evolução da tecnologia, os desenvolvedores podem facilmente implementar programas que controlam a iluminação, o sombreamento e a cor de cada pixel (NVIDIA, 2019).

Como foi mostrado por Bilodeau (2019), apesar de ser possível realizar computações de propósito geral no shader de fragmentos, vale a pena destacar a tecnologia dos shaders de computação e suas vantagens como maior controle sobre as *threads*, acesso à memória compartilhada, dispensabilidade de renderização de polígonos. Além disso, utilizar oclusão de ambiente em alta definição é muito custosa e pode ser substituída por meia resolução em conjunto com desfoco para melhorar a performance.

Shaders de fragmentos modernos são capazes de realizar centenas ou milhares de operações aritméticas, incorporando funções trigonométricas custosas e vários acessos a mapas de textura para produzir a cor final de cada pixel. Consequentemente, seu custo de execução pode facilmente dominar a capacidade computacional por quadro (SITTHI-AMORN *et al.*, 2008).

2.3 MOTORES DE JOGO E SUAS FERRAMENTAS

Um conjunto de ferramentas para criar um jogo é geralmente chamado de motor de jogo. Elas variam desde uma IDE (Ambiente de Desenvolvimento Integrado) até um pacote de software com capacidade de simulação física, renderização, rede e inteligência artificial. Podem ser classificadas pelas dificuldade de uso, plataformas de destino ou gênero de jogo que podem criar (JÓNSDÓTTIR, 2010).

Criadores de jogos atualmente contam com motores de jogo para desenvolver as principais partes de software para seus jogos. Sua principal função é simplificar as tarefas dos desenvolvedores oferecendo abstrações convenientes para os sistemas operacionais e seus hardwares nos quais o jogo funcionará. Isso somado ao propósito de explorar ao máximo a capacidade das máquinas dos usuários para que seja proporcionada uma experiência mais imersiva possível (MESSAOUDI; SIMON; KSENTINI, 2015).

Conforme definido por Barczak, Woźniak (2019) motores de jogo são softwares (proprietários ou de código aberto) voltados para facilitar a criação de jogos eletrônicos. Seus sistemas permitem a integração de vários elementos como *Scripts*, malhas, animações e audios que juntos podem formar um jogo eletrônico que pode ser distribuído para diferentes plataformas

Quadro 1 – Principais componentes das game engines

	Godot	Unity	Unreal
Shading	GLSL	ShaderLab (HLSL) / Shader Graph	Material Nodes
Sistema de Partículas	Built-in	Built-in / VFX Graph	UnrealCascade
Motor de física	Bullet	PhysX	PhysX
Programação	GDScript / C#	C#	C++ / Blueprint
Audio	Bus System	Audio Mixer	Sound Cue

Fonte: Elaborado pelo autor (2021)

por indivíduos ou companhias.

Atualmente motores de jogo são utilizados não somente como ferramentas de desenvolvimento de jogos, mas também em comunidades científicas para estudos que envolvem simulações nas áreas de medicina, arquitetura (para design de ambientes), meteorologia, geologia (com simulação e estudo de topologias) e educação, principalmente através do uso de Realidade Virtual e Realidade Aumentada (ŽUKOWSKI, 2019).

Quadro 2 – Funcionalidades gráficas presentes nas game engines

	Unity	Unreal	Godot
1. Texture			
1.1. Basic	✓	✓	✓
1.2. Procedural	✓	✓	✓
2. Lighting			
2.1. Per-vertex	✓	✓	✓
2.2. Per-pixel	✓	✓	✓
2.3. Light Mapping	✓	✓	✓
2.4. Gloss/Specular Maps	✓	✓	X
2.5. Basic	✓	✓	✓
3. Shadows			
3.1. Shadow Mapping	✓	✓	✓
3.2. Projected	✓	✓	✓

Fonte: Adaptado de Christopoulou e Xinogalos (2017)

Cada engine conta com um motor de renderização com funcionalidades que facilitam o processamento das informações da cena (câmera, luzes, materiais e texturas). Esse motor é uma parte fundamental de uma game engine e consome a maior parte dos recursos (90% dos cálculos dizem respeito a renderização). Como os jogadores esperam visuais melhores e os jogos precisam rodar em hardwares inferiores, há um conflito que faz com que os motores de renderização precisem ser otimizados e configuráveis durante o desenvolvimento (ŠMÍD, 2017).

Ambas as engines Unity e Unreal contém um sistema de iluminação global que simula operação em tempo real. Boa parte dos cálculos de renderização de luz são feitos antes de exibir o primeiro frame e seus resultados são utilizados durante a execução do jogo. As duas

engines oferecem suporte a sombreamento baseado na física (simulação da interação física entre as luzes e as superfícies). Além disso, destaca-se o suporte a algumas técnicas avançadas como suavização de bordas, reflexões em tempo real, oclusão de ambiente, profundidade de campo, entre outras (BARCZAK; WOŹNIAK, 2019).

Em termos de qualidade de gráficos, Godot e Unity são considerados inferiores a Unreal, que é mais adequada para renderizar gráficos empolgantes e realistas. Sendo que Unreal Engine se destaca pelos seus notáveis efeitos de pós-processamento e por implementar um ótimo efeito de dispersão de subsuperfície (Figura 15) (BARCZAK; WOŹNIAK, 2019).

Figura 15 – A luz que penetra na superfície de um objeto translúcido é espalhada pela interação com o material e sai da superfície em um ponto diferente.



Fonte: Mr Blue Summers (2021)

A escolha final das game engines requer a ponderação de alguns fatores como tipo de licença, possibilidade de modificação do código, possível uso comercial, especificações de hardware, plataforma alvo, habilidade da equipe de desenvolvimento, ferramentas de suporte, estabilidade da engine e público alvo (NAVARRO; PRADILLA; RIOS, 2012).

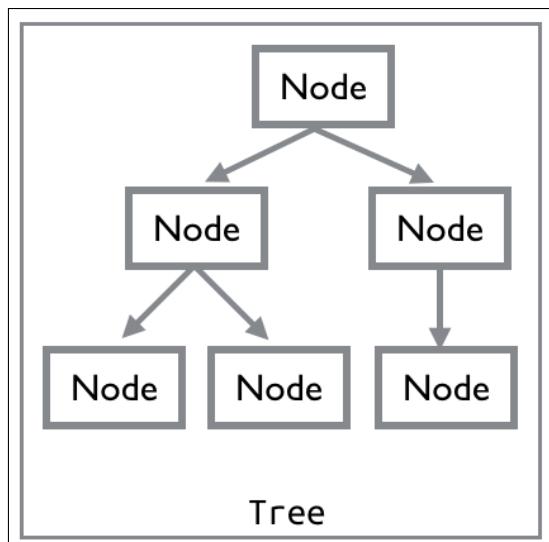
Por serem ferramentas complexas, sua comparação é uma tarefa complicada. A maneira mais apropriada seria implementar o mesmo projeto com complexidade apropriada e de maneira similar em cada game engine utilizando critérios mensuráveis para avaliação (ŠMÍD, 2017).

2.3.1 Godot

Godot é uma engine recente que atualmente encontra-se na versão 3 (a versão 4 ainda está em desenvolvimento) com diferenças consideráveis de arquitetura em relação às outras duas engines. Nesse caso cada sistema central é gerenciado por um servidor de forma assíncrona. Esses servidores operam em um alto nível de abstração. Apesar de soar complexo, sua interface é bem amigável. O sistema de árvore de cenas permite organizar o conteúdo do jogo da forma que humanos estão acostumados a visualizar o mundo ao redor (MANZUR; MARQUES, 2018).

Todos os jogos criados nessa engine são baseados no uso de nós e cenas. Nós podem ser considerados como átomos de funcionalidade de jogo e são utilizados em conjunto para formar cenas que podem ser combinadas para formar outras cenas mais complexas. Nós são parte fundamental dos jogos e consistem em um bloco funcional com nome, propriedades e uma função especial. (MANZUR; MARQUES, 2018).

Figura 16 – Estrutura hierárquica da cena na Godot Engine.



Fonte: Packtpub (2021)

Uma funcionalidade que merece destaque é a de sinais, que são uma implementação do padrão de projeto de observador. O conceito básico é de que um objeto notifica outros objetos que possuem interesse em suas ações. O desenvolvedor não precisa se preocupar quando ou onde as ações serão solicitadas. Então o objeto emite o sinal e seus observadores são notificados (MANZUR; MARQUES, 2018).

Para programar, essa engine possui a própria linguagem chamada GDScript (mas também há suporte para C# e programação visual), que foi criada especificamente para a engine

e possui uma sintaxe similar a Phyton. Cada objeto possui um tipo constituído de hierarquias de classe, variáveis são chamadas de membros e funções são chamadas de métodos (MANZUR; MARQUES, 2018).

2.3.2 Unity

Unity é um motor de jogo desenvolvido e mantido pela empresa Unity Technologies que permite a criação de jogos 2D e 3D. Apresentada em 2005, sua primeira versão era simples e compatível apenas com Mac OS, porém com sua evolução foi adicionado suporte a outras plataformas (PC, Linux, Android, iOS, PS4, etc) (BARCZAK; WOŹNIAK, 2019). Esse foi um dos principais fatores que tornou essa engine tão popular. Já que por suportar mais plataformas, pode proporcionar acesso a um público mais amplo.

Ela faz uso do DirectX como pipeline de renderização padrão, mas além disso ela utiliza quatro atividades de renderização adicionais. A pipeline de renderização direta é dividida em passagem de ambiente para objetos não afetados pela luz, passagem de transparência e passagem de luz para objetos opacos. Já a pipeline de renderização diferida é baseada em um modelo inteligente que primeiro computa a geometria e depois aplica a luz (MESSAOUDI; SIMON; KSENTINI, 2015).

A pipeline de renderização de pré-passagem é direcionada para as restrições no uso de diferentes shaders no processo diferido. As informações de luz são guardadas em um buffer para melhorar a performance dos cálculos. Por último é realizado o processo de renderização de vértices iluminados (cada objeto é renderizado com a iluminação de todas as fontes de luz calculada nos vértices) que é o mais rápido (MESSAOUDI; SIMON; KSENTINI, 2015).

Apesar de sua pipeline ser considerada uma caixa preta (desenvolvedores não possuem muito controle sobre seu funcionamento) por Hasu (2018), há diferentes alternativas baseadas em renderização diferida, direta e legado (iluminação diferida ou por vértice). Cabe mencionar também a pipeline de renderização programável que é uma forma alternativa de permitir que desenvolvedores configurem a renderização.

Há também a pipeline de renderização de alta definição (HDRP) voltada para computadores e consoles de última geração, que oferece iluminação coerente e unificada e melhores ferramentas de debug. Já a pipeline peso-leve LWRP é voltada para dispositivos móveis e realidade virtual que apresenta algumas otimizações de performance (HASU, 2018).

Sua arquitetura consiste em um sistema modular baseado em componentes que

são utilizados para compor os objetos nos jogos. Cada componente possui um conjunto de funcionalidades que afetam o comportamento do objeto, dessa forma não é necessário usar herança. Isso é uma vantagem visto que aumenta a flexibilidade e a eficiência da modificação dos objetos (BARCZAK; WOŹNIAK, 2019).

Cabe ainda destacar que ela possui uma das melhores documentações. A maioria das funções são descritas em profundidade e com bastante uso de exemplos, o que é muito útil principalmente para usuários iniciantes. Além disso ela possui uma vasta quantidade de templates, uma interface limpa e fácil de configurar, uma comunidade ativa, e seu uso de C# ao invés de C++ torna a programação mais simples e agradável (BARCZAK; WOŹNIAK, 2019).

Conforme verificado em estudo por Costa, Gomes, Duarte (2016), dentre os motores de jogo a Unity é uma das ferramentas mais vantajosa do ponto de vista de produção por possuir um formato mais profissional e comercial, voltado para desenvolvimento multiplataforma. Além disso ela é capaz de performar melhor em composições de hardware mais simples.

A escolha da Unity pode-se justificar pelo fato de sua notável popularidade e seu crescimento constante, sendo que mais de 47% dos desenvolvedores de jogos utilizam-na, com aproximadamente 45% de participação no mercado de game engines e mais de 600 milhões de jogadores gastando mais de US\$ 110 bilhões (MESSAOUDI; SIMON; KSENTINI, 2015).

Na Unity os shaders são escritos em uma linguagem chamada *ShaderLab*. Entretanto, o código do shader em si é escrito em HLSL. Ela oferece suporte a alguns tipos de shaders: de superfície (facilita a interação com luz e sombra), de vértice e de fragmento e de funções fixas (KUISMIN, 2020).

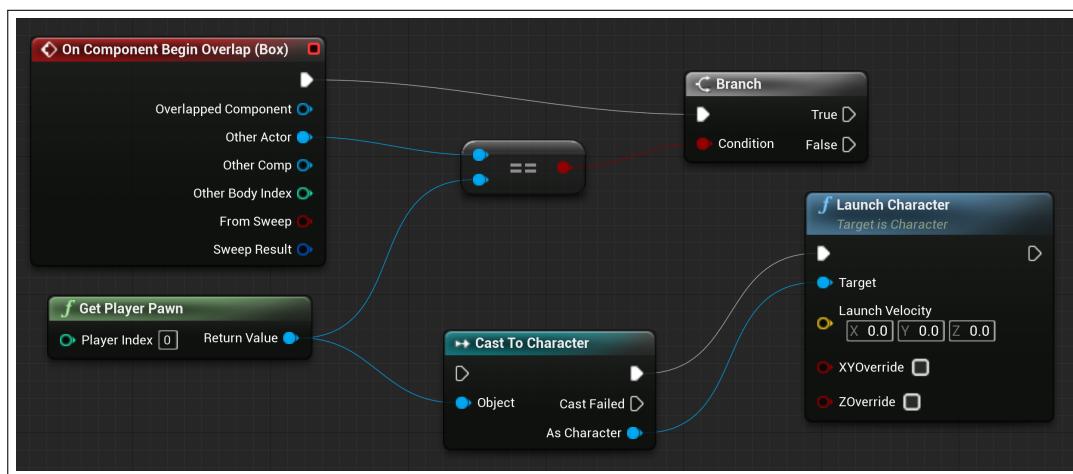
Apesar dos shaders serem escritos na linguagem declarativa *ShaderLab*, o código do programas é escrito como um *snippet HLSLPROGRAM* que é transformado em código HLSL. Cada tipo de shader é então definido usando a instrução *pragma*. Também há suporte para escrita manual de código GLSL, porém isso é recomendado apenas para testes ou casos específicos pois a Unity se encarrega de compilar e otimizar o HLSL para GLSL caso seja necessário para a plataforma alvo (HASU, 2018).

Há ainda alguns shader mais complexos que conseguem modificar a geometria base de malhas. São os shaders de geometria e tesselação, que pode ser utilizados para adaptar a qualidade visual ao nível de detalhe exigido por meio da otimização da malha em tempo real conforme sua distância da câmera (KUISMIN, 2020).

2.3.3 Unreal

Unreal é mais um motor de jogo que está a mais tempo no mercado (desde 1998). Ela também permite a criação de jogos para múltiplas plataformas. Ela possui uma ferramenta específica para criação de scripts chamada *BluePrint* que permite implementar lógica de programação usando blocos, porém também é possível criar scripts em C++ (BARCZAK; WOŹNIAK, 2019).

Figura 17 – A programação da lógica do jogo pode ser feita utilizando o sistema visual de *BluePrint*.



Fonte: Unreal Engine (2021)

É uma engine, em comparação com a Unity, mais difícil de dominar apesar de possuir uma interface amigável, porém com excesso de opções à primeira vista. Seu sistema de programação com *BluePrints* (Figura 17) traz uma vantagem para pessoas que não sabem ou não gostam de escrever código (BARCZAK; WOŹNIAK, 2019).

Merecem destaque seu sistema de renderização Multithread (intitulado Gemini) com 64 bits de HDR (High Dynamic Range), oclusão de ambiente, iluminação por pixel, iluminação especular dinâmica e reflexões, seu sistema de customização de terrenos e ainda o sistema de malhas de navegação para integração com personagens controlados por inteligência artificial (ARMSTRONG, 2013).

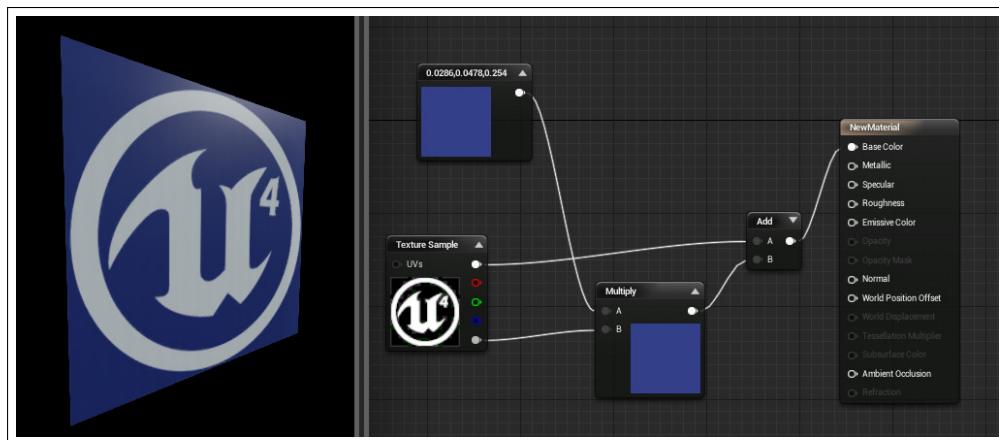
Unreal apresenta uma melhor performance audiovisual de maneira geral. Além disso ela apresenta uma interface mais complexa em comparação com Unity e Godot que oferece todas as ferramentas necessárias em uma única janela. É portanto uma engine voltada para usuários mais experientes e hardwares mais robustos (CHRISTOPOULOU; XINOGALOS, 2017).

2.3.3.1 Nós de material

Unreal provê uma ferramenta de edição visual de grafos de nós para definição de expressões que computam parâmetros de entrada para modelos de materiais pré-definidos pela engine. Cada nó na expressão do grafo corresponde a um *snippet* (pequeno pedaço de código) de shader que é composto com código modelo de material provido pela própria engine durante a compilação (HE; FOLEY; FATAHALIAN, 2016).

Novamente a Unreal se destaca, por implementar uma ferramenta visual de edição de materiais que reutiliza os nós das *BluePrints* mencionadas anteriormente, porém dessa vez voltadas para a modificação das propriedades dos materiais, o que torna o processo de criação de shader bastante amigável e divertido por estimular visualmente a criatividade dos usuários (BARCZAK; WOŹNIAK, 2019).

Figura 18 – Uso de uma textura como base para definir a cor de um material.



Fonte: Unreal Engine (2021)

Seu sistema de materiais é uma das melhores ferramentas. Para cada material é compilado um shader. Então é possível utilizar o mesmo material ou pequenas variações deste. Esse método proporciona aos artistas uma ferramenta poderosa para criação de materiais, porém o lado negativo é que no caso de shaders mais complexos o tempo de compilação pode aumentar bastante (ŠMÍD, 2017).

Por baixo dos panos, essa engine implementa uma otimização de sombreamento chamada cascata de sombras. Ela renderiza múltiplos mapas de sombra baseado na distância da câmera para que objetos mais próximos tenham sombras mais definidas que objetos distantes. Há uma mistura suave entre esses mapas para que a mudança seja quase imperceptível. Isso melhora a performance e é muito eficiente especialmente para grandes cenas (ŠMÍD, 2017).

2.4 OTIMIZAÇÃO E PERFORMANCE

Embora as arquiteturas de shaders forneçam execuções rápidas, a avaliação de shaders gera um custo alto no processo de renderização. Para shaders grandes (como os usados em filmes) pode igualar e exceder os custos associados com remoção de superfícies ocultas e processamento de geometria (PELLACINI, 2005).

Isso torna-se um problema ainda maior em aplicações de tempo real que possuem restrições rígidas de taxa de quadros. Nesse caso, a simplificação geométrica é constantemente utilizada para fornecer uma forma de compensação entre a qualidade da imagem percebida e a velocidade de execução em relação ao tamanho da malha do objeto (PELLACINI, 2005).

Cabe ressaltar que para Riguer (2002), a performance média de um sistema é tão boa quanto o desempenho no pior gargalo. Então a tarefa de otimização pode ser reduzida à encontrar o pior gargalo e removê-lo (ou amenizá-lo). Esse é um processo iterativo que deve ser repetido até que o desempenho esteja em um limite aceitável.

Um gargalo de largura de banda de memória pode ocorrer quando há uso intenso de dados de vértices dinâmicos, uma solução seria reduzir a transferência de dados dinâmicos ou diminuir o tamanho de vértice. Já um gargalo de componente gráfico pode ser causado pelo próprio sistema (RIGUER, 2002).

Continuando, quando há cálculos por vértice ou uso de luzes em excesso pode ocorrer um gargalo de processamento de vértices. Por outro lado, o uso de shaders de fragmento complexos também pode gerar um gargalo. Além disso, o excesso de pixels renderizados por segundo pode gerar um gargalo de taxa de preenchimento (RIGUER, 2002).

Outrossim, o uso de texturas de alta definição e de filtros complexos pode acarretar em um gargalo de carregamento de textura. Por fim, o uso excessivo de recursos de transparência, profundidade e amostragem pode causar um gargalo no processo de rasterização. Na maioria das vezes o que se percebe é uma combinação de vários gargalos atuando em conjunto (RIGUER, 2002).

Enquanto é considerada uma prática comum usar geometria dinâmica, não é bom abusar. Uma quantidade muito maior de polígonos pode ser obtida com geometria estática. Transformar geometria dinâmica em estática e aproveitar shaders de vértice para mover animações para a GPU pode ajudar a balancear a carga de trabalho (RIGUER, 2002).

Devido à sua natureza de operação que requer simulação em tempo real, os jogos eletrônicos são softwares que exigem bastante recursos de hardware. Ainda mais atualmente

quando os desenvolvedores cada vez mais objetivam criar produtos que perfaçam 60 frames por segundo (FPS) ou mais. Dessa forma, para atingir o maior público possível, vale a pena usar ferramentas que possibilitem o uso mais otimizado possível desses recursos (SKOP, 2018).

Com o crescimento do mercado de dispositivos móveis, a oportunidade de desenvolver e vender aplicações sofisticadas para esses dispositivos é ainda mais atrativa. Entretanto há também uma enorme quantidade de desafios ao lidar com esse tipo de tecnologia, entre eles: fonte de alimentação, quantidade de poder computacional, tamanho do display e tipos de entrada. Além disso, os processadores de smartphones não tratam números de ponto flutuante, o que reduz muito a precisão dos cálculos (SINGHAL; PARK; CHO, 2010).

Na GPU de dispositivos móveis o número de espaços de instrução é limitado para os shaders de vértice e fragmento. Enquanto empacotar múltiplos ciclos de renderização em um único shader de fragmento aumenta a contagem de instruções, aumentar a quantidade de ciclos de renderização diminui o fracionamento paralelo. Nesse caso a melhor solução dependerá das necessidades da aplicação (SINGHAL *et al.*, 2011).

Segundo Singhal et al. (2011) algumas formas de otimizar shaders seriam por meio de compressão de texturas para diminuir a sobrecarga de transferência de memória (ou utilizar um formato de pixel de menor precisão), pré-computar coordenadas vizinhas de texturas no shader de vértice e substituir laços por códigos otimizados ou utilizar vetores para realizar operações (diminuir a quantidade de instruções).

Considerando o trabalho de Nusrat *et al.* (2021), cabe aos desenvolvedores simplificar os gráficos para melhorar a performance. Simplificação de shaders e de modelos 3D são os tipos de melhorias mais comuns que podem afetar a experiência visual dos usuários. Por exemplo, ativar o corte de oclusão estático/dinâmico desativa a renderização de objetos cobertos e ativar o Light Baking pré-calcula efeitos de luz durante a compilação.

Nesse sentido, os desenvolvedores devem tentar entender o custo computacional dos shaders e modelos 3D antes de usá-los. Para isso, antes de adicioná-los, devem ser feitos testes pois ao inserir vários modelos e shaders pode ser difícil distinguir quais estão causando problemas de performance (NUSRAT *et al.*, 2021).

Devido a melhoria no poder de processamento gráfico, o uso de técnicas como iluminação 3D, vegetação gerada proceduralmente e fotogrametria aumentou consideravelmente tornando o processo de renderização mais custoso. Métodos comuns para otimização de cenas que fazem uso dessas técnicas são alocação de memória, Multithread, e nível de detalhe. Ou seja, realizar a renderização em uma thread separada da lógica do jogo ajuda bastante a melhorar

a performance (ZHANG *et al.*, 2017).

Nível de detalhe é uma forma de determinar a distribuição dos recursos de renderização entre os objetos (de acordo com a posição e a importância atribuída aos vértices desses), diminuindo o número de dados desnecessários. Outrossim, são gerados modelos simplificados que reduzem a complexidade da cena e permitem uma renderização em tempo real mais eficiente (ZHANG *et al.*, 2017).

Como as GPUs normalmente possuem ótima capacidade de processamento de vértices, a etapa de sombreamento de vértices raramente gerará um gargalo. Entretanto quando muitos pixels precisam ser processados por um shader de fragmentos com muitas instruções é esperado que haja um gargalo. Pode-se dizer que o número de instruções é inversamente proporcional à performance (taxa de frames) (SINGHAL; PARK; CHO, 2010).

O controle de precisão de ponto flutuante — em OpenGL baixo (10 bits), médio (16 bits) e alto (32 bits) — é uma ótima ferramenta para melhorar o desempenho, porém precisa ser usada de forma apropriada, pois um nível de baixa precisão apesar de melhorar a performance pode acabar gerando artefatos indesejados (SINGHAL; PARK; CHO, 2010).

Tabela 2 – Exemplo do número de instruções necessárias para operações específicas no OpenGL

Operação	Número de Instruções	Número de ciclos de renderização	Operação	Número de Instruções	Número de ciclos de renderização
RGB2GRAY	5	1	Gaussian	21	1
RGB2YCbCr	14	1	Sharpening	13	1
YCbCr2RGB	14	1	Gradient	19	2
RGB2HSV	28	1	Bilateral	62	2
HSV2RGB	29	1	Laplacian	14	1
			Box filter	18	1
Bloom	15	1	Sobel	24	2
Skin detection	25	2	Prewitt	16	2
Detail enhancement	13	1	Contrast Stretching	13	1
Edge enhancement	25	2	Median filtering	43	1
Dilation	22	1	Erosion	22	1
Median	43	1	Zero-crossing	22	1
Sepia	21	1	Color gradient	20	1
Radial Blur	21	1	Negative	2	1
Edge overlay	25	2	Gamma	15	1
Gray	5	1	Edge	24	2

Fonte: Singhal, Park e Cho (2010)

Para Crawford e O’Boyle (2018), a quantidade de linhas de código de um shader segue um distribuição de lei de potência, com poucos shaders longos, e vários shaders simples (poucas linhas). Entretanto, até os shaders mais longos possuem em torno de 300 linhas. A

maioria contém menos que 50 linhas. Isso mostra que normalmente shaders são bem menores que softwares.

Além disso, como o número de vértices processados em operações de processamento de imagens é muito mais baixo que o de fragmentos (podem ser da ordem de milhões), é recomendado realizar cálculos por vértice ao invés de por fragmento por aquelas serem menos custosas. Por fim, cabe salientar que maiores velocidades de clock favorecem a performance (SINGHAL; PARK; CHO, 2010).

Como a criação de jogos tornou-se mais acessível, vários jogos são criados e lançados frequentemente. Com o aumento da quantidade de jogos no mercado os desenvolvedores precisam fazer com que eles se destaquem da concorrência. Para isso é comum o uso de modelos gráficos com muitos detalhes, o que pode acabar sobrecarregando o sistema significativamente (MICHAŁ, 2020).

Quando muitos deles estão presentes na tela ao mesmo tempo, o jogo pode não funcionar bem, e isso afeta negativamente a experiência dos jogadores. Uma solução para esse problema consiste em aplicar algoritmos de tesselação para substituir dinamicamente modelos de objetos presentes no jogo. Cada modelo é substituído por um mais simplificado conforme a câmera se afasta, reduzindo a carga no sistema (MICHAŁ, 2020).

A maioria dos motores de jogos disponíveis no mercado oferecem uma ferramenta para geração automática de níveis de detalhe para modelos 3D. A única coisa que o desenvolvedor do jogo deve fazer é indicar como os níveis de detalhe devem ser gerados (caso os parâmetros padrões não se adequem ao seu projeto). A Unreal Engine possui uma ferramenta embutida para gerar níveis de detalhe por padrão com várias configurações prontas preparadas pelos criadores. A ferramenta é capaz de gerar automaticamente o número apropriado de níveis de detalhe (MICHAŁ, 2020).

Uma forma de otimização de shaders já descrita por Rost (2006) consiste na substituição da função de ruído embutida na linguagem de shader por uma função criada pelo próprio desenvolvedor ou por uma textura. A última opção é a melhor em termos de performance. Felizmente, a programabilidade oferecida pela GLSL torna possível pré-computar uma função de ruído e salvar seu resultado em mapas de textura de uma, duas ou três dimensões em cada um de seus quatro componentes.

Outra técnica útil para melhorar a performance ao lidar com várias luzes é o sombreamento diferido, que basicamente determina quais superfícies serão visíveis na cena final e aplica cálculos complexos de efeitos de shader apenas nos pixels que compõem essas

superfícies. Dessa maneira, as operações são adiadas até que sejam estabelecidos os pixels que contribuirão para a imagem final. Essa técnica garante que não haja desperdício de ciclos de hardware com cálculos em pixels que sequer serão exibidos na tela (ROST, 2006).

Mais uma maneira de otimização descrita por Jensen *et al.* (2007) seria garantir que o código de shader seja movido para sua parte menos custosa. Nesse caso existem três possíveis lugares: no programa de vértices, no programa de fragmentos, ou nas declarações constantes. A última é a mais otimizada pois os cálculos são realizados em tempo de compilação. A segunda forma mais otimizada seria utilizar o espaço do programa de vértices, mas se houver muito código nessa parte haverá um desbalanceamento.

Caso seja necessário realizar transformações de coordenadas a melhor opção seria utilizar o espaço do programa de vértices (por exemplo mover transformação da direção da luz em espaço tangente para essa etapa). Ao ponderar-se esses detalhes para as ferramentas de criação de shaders visuais, percebe-se que há uma certa dificuldade em realizar otimizações manuais, já que o código é gerado automaticamente (JENSEN *et al.*, 2007).

Um dos erros mais comuns é usar texturas muito grandes desnecessariamente, o que prejudica bastante a performance, sendo que alguns objetos nunca atingem um tamanho grande na tela. O ideal seria tentar utilizar texturas com dimensões que não são potências de dois para diminuir o consumo de memória. Além disso, deve-se evitar a troca excessiva de texturas (RIGUER, 2002).

Segundo Arnau, Parcerisa e Xekalakis (2014), uma possível técnica de otimização seria o uso de memoização para evitar a execução redundante de computações reutilizando resultados anteriores, o que resulta em aumento de velocidade de execução e economia de energia. De forma simplificada, os cálculos realizados são salvos em uma tabela para que no próximo cálculo os valores sejam reutilizados.

He, Foley e Fatahalian (2016) define o processo de otimização em várias etapas: identificação de componentes de shaders que podem ser transformados em texturas, uso de pré-processamento *offline* para os buffers de parâmetros, seleção da melhor frequência espacial ou espaço de coordenadas para as funções e escolha de técnicas multi-resolução ou formas de reuso.

Esse é um desafio e tanto para game engines modernas, que podem conter vários shaders únicos, implementando uma vasta coleção de materiais e múltiplos níveis de detalhe — por exemplo, o jogo Bungie's Destiny usa mais de 17 mil materiais compilados para 180 mil shaders — que podem exigir diferentes procedimentos de otimização (HE; FOLEY; FATAHALIAN,

2016).

Para medir a performance, é necessária uma forma de Benchmark para comparar a performance com mais precisão e objetividade. Algumas áreas onde o Benchmark é importante são na iluminação global, detecção de colisão, animação, renderização e em áreas onde é preciso medir e comparar a performance (LEXT; ASSARSSON; MOLLER, 2001).

Uma métrica muito comum para sistemas interativos de tempo real é a quantidade de quadros por segundo (taxa de quadros). Ela mede a frequência média de uma aplicação no hardware onde é executada. Assim pode-se obter diferentes valores para diferentes tipos de hardware. Essa é uma medida comum entre várias revistas de jogos de computadores para medir a performance (REHFELD; TRAMBEREND; LATOSCHIK, 2014).

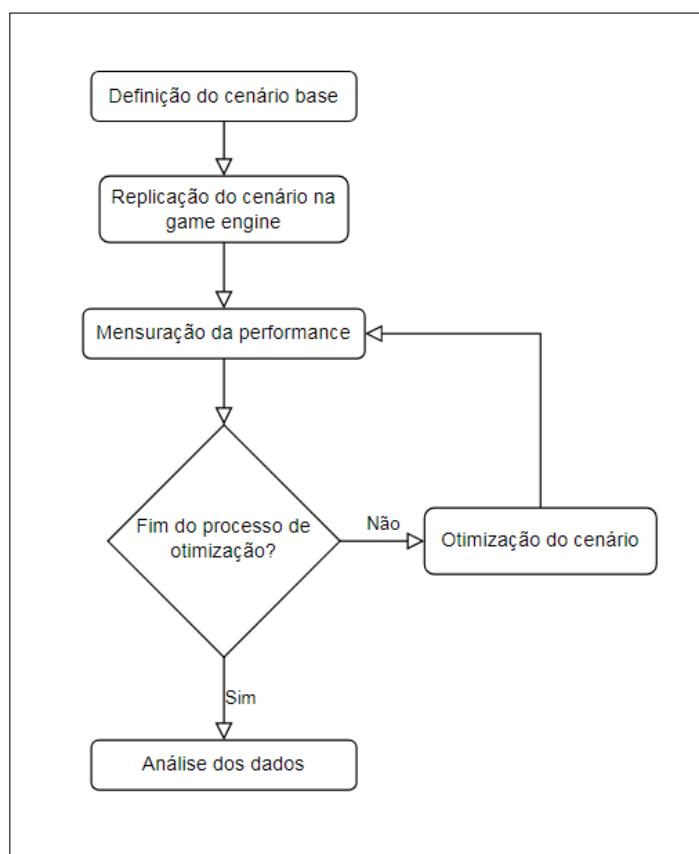
3 METODOLOGIA

O propósito dessa seção é especificar como o trabalho foi realizado, apresentando o tipo de pesquisa, o meio de coleta de dados, o cenário e a análise dos dados coletados a fim de atingir os objetivos propostos.

3.1 FLUXO DE DESENVOLVIMENTO

A figura 19 mostra o fluxo utilizado para o desenvolvimento deste trabalho.

Figura 19 – Fluxograma do processo



Fonte: Elaborado pelo autor (2021)

3.2 SOBRE A PESQUISA

A pesquisa realizada para a elaboração do presente trabalho é do tipo aplicada, por se tratar de um estudo de análise comparativa e que propõe gerar conhecimento. Esse é um tipo de pesquisa que envolve estudos profundos com a intenção de resolver problemas presentes no em um contexto social semelhante ao dos pesquisadores (GIL, 2017).

Em relação a classificação da pesquisa, pode-se constatar que trata-se de um estudo com características exploratórias por proporcionar maior familiaridade com o problema e descritivas por focar nas características do objeto de estudo (GIL, 2017). Sua finalidade é portanto de criar e ampliar pressuposições, elucidar informações e incertezas sobre o assunto e complementar os entendimentos do explorador.

Já quanto ao tipo de abordagem, seguindo a definição de Hernandez et. al. (2013), entende-se que é do tipo quantitativa por ocorrer por meio da coleta de dados para teste de hipóteses baseando-se na medição numérica e na análise estatística para comprovar teorias e estabelecer padrões. Na abordagem quantitativa, os resultados são propagados por meio de quantitativos adquiridos no processo de coleta dos dados.

Conforme Marconi e Lakatos (2019), a investigação bibliográfica é feita pela pesquisa e exposição de bibliografias públicas (livros, revistas, artigos, teses), para mostrar com profundidade os assuntos delimitados pelo pesquisador. A pesquisa bibliográfica, dessa forma, ajuda a obter resultados expressivos no estudo desenvolvido.

3.3 ESCOLHA DAS GAME ENGINES

Em relação a Unity, é a engine mais popular entre desenvolvedores de jogos (motor mais popular na plataforma de jogos Steam), especialmente para projetos pequenos e médios. Alguns jogos populares desenvolvidos com ela são: Fall Guys, Among Us, Phasmophobia e Cities (DOUCET; PECORELLA, 2021).

Já sobre a Unreal, que cada vez mais reduz as taxas de licenciamento e torna-se mais acessível. Ela é mais propícia para projetos de grande porte (grandes estúdios e jogos AAA). Jogos famosos desenvolvidos com essa engine: ARK, Borderlands, XCOM e PUBG (DOUCET; PECORELLA, 2021).

Por outro lado, a escolha da Godot ocorreu por ser uma engine intuitiva, de código aberto, que apresenta diversas funções que facilitam o desenvolvimento de jogos e por apresentar um crescimento de popularidade e de uso entre desenvolvedores (VARGAS, 2020).

3.4 SOBRE OS DADOS

Neste trabalho, os dados necessários para a realização do *benchmarking*, como taxa de quadros e tempo de execução, foram obtidos a partir da execução das ferramentas de *profiling*. Pois são dados que são obtidos em tempo real. Esses dados foram analisados para os shaders

utilizados no cenário base em cada uma das game engines levando em consideração múltiplos níveis de otimização.

Em relação as características de hardware, foi utilizado um computador com sistema operacional Windows 7 Ultimate, processador Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz, memória RAM de 8 GB, disco rígido de 500 GB, unidade de estado sólido de 120 GB, placa de vídeo PCI Radeon R7 360 e placa mãe com chipset H61.

4 RESULTADOS

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

4.1 RESULTADOS DO EXPERIMENTO A

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

4.2 RESULTADOS DO EXPERIMENTO B

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

5 CONCLUSÕES E TRABALHOS FUTUROS

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris. Nullam eleifend justo in nisl. In hac habitasse platea dictumst. Morbi nonummy. Aliquam ut felis. In velit leo, dictum vitae, posuere id, vulputate nec, ante. Maecenas vitae pede nec dui dignissim suscipit. Morbi magna. Vestibulum id purus eget velit laoreet laoreet. Praesent sed leo vel nibh convallis blandit. Ut rutrum. Donec nibh. Donec interdum. Fusce sed pede sit amet elit rhoncus ultrices. Nullam at enim vitae pede vehicula iaculis.

5.1 CONTRIBUIÇÕES DO TRABALHO

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

5.2 LIMITAÇÕES

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

5.3 TRABALHOS FUTUROS

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

REFERÊNCIAS

- ABDALA, D. D. **Primitivas Gráficas 2D**. [2014?]. 37 slides, color. Disponível em: <<https://www.slideserve.com/vera/efficient-compute-shader-programming>>. Acesso em: 16 nov. 2021.
- AHN, S. H. **Homogeneous Coordinates**. Disponível em: <http://www.songho.ca/math/>. Acesso em: 04 nov. 2021.
- AHN, S. H. **OpenGL**. Disponível em: <http://www.songho.ca/opengl/index.html>. Acesso em: 04 nov. 2021.
- ARMSTRONG, M. S. Game engine review. **NORTH ATLANTIC TREATY ORGANIZATION**, 2013.
- ARNAU, J.-M.; PARCERISA, J.-M.; XEKALAKIS, P. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In: **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2014.
- BAILEY, M.; CUNNINGHAM, S. A hands-on environment for teaching gpu programming. In: **Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education**. [S.l.: s.n.], 2007.
- BARCZAK, A.; WOŹNIAK, H. Comparative study on game engines. **STUDIA INFORMATICA**, 2019.
- BASECOLOR QS. Disponível em: <https://docs.unrealengine.com/4.27/Images/RenderingAndGraphics/Materials/PhysicallyBased/BaseColor_QS.webp>. Acesso em: 11 nov. 2021.
- BELAIR, F.; LOVATO, N. **outline3D**. Disponível em: <https://github.com/GDQuest/godot-shaders/blob/master/godot/Shaders/outline3D.shader>. Acesso em: 09 nov. 2021.
- BILODEAU, B. **Efficient Compute Shader Programming**. Uberlândia: Facom/Ufu, 2019. 74 slides, color. Disponível em: <http://www.facom.ufu.br/~abdala/GBC204/03_primitivas2D.pdf>. Acesso em: 31 out. 2021.
- BPQS 6 Step 4. Disponível em: <https://docs.unrealengine.com/4.27/Images/ProgrammingAndScripting/Blueprints/QuickStart/BPQS_6_Step4.png>. Acesso em: 11 nov. 2021.
- CHRISTOPOULOU, E.; XINO GALOS, S. Overview and comparative analysis of game engines for desktop and mobile devices. **International Journal of Serious Games**, 2017.
- COOKSON, A.; DOWLINGSOKA, R.; CRUMPLER, C. **Unreal Engine 4 Game Development in 24 Hours, Sams Teach Yourself**. Indianapolis: Sams Publishing, 2016.
- COORDINATE Systems. Disponível em: <https://learnopengl.com/img/getting-started/coordinate_systems.png>. Acesso em: 05 nov. 2021.
- COSTA, D. P.; GOMES, F. F. B.; DUARTE, R. Estudo comparativo entre as game engines unity e ogre. **Revista Computação Aplicada**, 2016.

- CRAWFORD, L.; O'BOYLE, M. A cross-platform evaluation of graphics shader compiler optimization. In: **2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2018.
- DOUCET, L.; PECORELLA, A. **Game engines on Steam: The definitive breakdown**. Disponível em: <<https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown>>. Acesso em: 27 nov. 2021.
- GIL, A. C. **Como Elaborar Projetos de Pesquisa**. 6. ed. São Paulo: Atlas, 2017.
- HAAS, J. K. **A History of the Unity Game Engine**. 44 p. Monografia (Graduação) — Worcester Polytechnic Institute, Worcester, MA, Estados Unidos, 2014.
- HASU, J. **Fundamentals of Shaders with Modern Game Engines**. Dissertação (Mestrado) — Lappeenranta University of Technology, 2018.
- HE, Y.; FOLEY, T.; FATAHALIAN, K. A system for rapid exploration of shader optimization choices. **ACM Trans. Graph.**, 2016.
- HE, Y.; FOLEY, T.; TATARCHUK, N.; FATAHALIAN, K. A system for rapid, automatic shader level-of-detail. **ACM Trans. Graph.**, 2015.
- HERNÁNDEZ, R. *et al.* **Metodologia de Pesquisa**. 5. ed. Porto Alegre: Penso, 2013.
- HOLOGRAM planet. Disponível em: <<https://www.youtube.com/watch?v=F0CWzpYY68A&t=2s>>. Acesso em: 29 out. 2021.
- JENSEN, P. D. E.; FRANCIS, N.; LARSEN, B. D.; CHRISTENSEN, N. J. Interactive shader development. In: **Proceedings of the 2007 ACM SIGGRAPH Symposium on Video Games**. [S.l.: s.n.], 2007.
- JÓNSDÓTTIR, R. D. **A comparison of game engines and languages**. 123 p. Monografia (TCC / Graduação em Ciência da Computação) — University Of Akureyri, Akureyri, 2010.
- KL_NVIDIA_GEFORCE_256.JPG. Disponível em: <https://upload.wikimedia.org/wikipedia/commons/c/c1/KL_NVIDIA_Geforce_256.jpg>. Acesso em: 27 out. 2021.
- KUISMIN, A. **Creating Okami Inspired Shader in Unity's Shader Graph**. 33 p. Monografia (TCC) — Universidade de Ciências Aplicadas de Tampere (TAMK), 2020.
- LEXT, J.; ASSARSSON, U.; MOLLER, T. A benchmark for animated ray tracing. **IEEE Computer Graphics and Applications**, 2001.
- LUTEN, E. **OpenGLBook.com**. Disponível em: <https://openglbook.com/the-book.html>. Acesso em: 25 out. 2021.
- MANZUR, A.; MARQUES, G. **Godot Engine Game Development in 24 Hours, Sams Teach Yourself: The Official Guide to Godot 3.0**. Indianapolis: Sams Publishing, 2018.
- MARCONI, M. D. A.; LAKATOS, E. A. **Fundamentos da Metodologia Científica**. 8. ed. São Paulo: Atlas, 2019.
- MARQUES, N. L. R. **Mecânica Geral Básica**. Disponível em: <<https://docplayer.com.br/200330-Mecanica-geral-basica.html>>. Acesso em: 23 nov. 2021.

- MESSAOUDI, F.; SIMON, G.; KSENTINI, A. Dissecting games engines: The case of unity3d. In: **2015 International Workshop on Network and Systems Support for Games (NetGames)**. [S.l.: s.n.], 2015.
- MICHAŁ, T. Comparison of methods and tools for generating levels of details of 3d models for popular game engines. **STUDIA INFORMATICA**, 2020.
- MICROSOFT. **Compare the OpenGL ES 2.0 shader pipeline to Direct3D**. Disponível em: <<https://docs.microsoft.com/en-us/windows/uwp/gaming/change-your-shader-loading-code>>. Acesso em: 07 nov. 2021.
- MONTENEGRO, A. **Computação Gráfica I**. Niterói: Instituto de computação UFF, [2017?]. 99 slides, color. Disponível em: <[http://www.ic.uff.br/~anselmo/cursos/CGI/slidesGrad/CG_aula4\(introducaoOpenGL\).pdf](http://www.ic.uff.br/~anselmo/cursos/CGI/slidesGrad/CG_aula4(introducaoOpenGL).pdf)>. Acesso em: 02 nov. 2021.
- NAVARRO, A.; PRADILLA, J.; RIOS, O. Open source 3d game engines for serious games modeling. In: _____. [S.l.]: InTech, 2012. cap. 6.
- NUSRAT, F.; HASSAN, F.; ZHONG, H.; WANG, X. How developers optimize virtual reality applications: A study of optimization commits in open source unity projects. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2021.
- NVIDIA. **Fragment Shaders**. Disponível em: <<https://www.nvidia.com/en-us/drivers/feature-pixelshader/>>. Acesso em: 12 nov. 2021.
- NVIDIA. **Vertex Shaders**. Disponível em: <<https://www.nvidia.com/en-us/drivers/feature-vertexshader/>>. Acesso em: 12 nov. 2021.
- PACKTPUB. **About nodes and scenes**. Disponível em: <<https://subscription.packtpub.com/book/game-development/9781788831505/1/ch011v11sec07/about-nodes-and-scenes>>. Acesso em: 23 nov. 2021.
- PELLACINI, F. User-configurable automatic shader simplification. In: **ACM SIGGRAPH 2005 Papers**. [S.l.]: Association for Computing Machinery, 2005.
- REHFELD, S.; TRAMBEREND, H.; LATOSCHIK, M. E. Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems. In: **Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology**. [S.l.: s.n.], 2014.
- RENDERING Pipeline. Disponível em: <https://miro.medium.com/max/700/1*_z7Vbb0msvXsUFq3sSMZAg.png>. Acesso em: 12 nov. 2021.
- RIGUER, G. Performance optimization techniques for ati graphics hardware with directx 9.0. **ATI Technologies Inc**, 2002.
- ROST, R. J. **OpenGL Shading Language**. Boston: Addison Wesley, 2006.
- SHEA, R.; LIU, J. On gpu pass-through performance for cloud gaming: Experiments and analysis. In: **2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)**. [S.l.: s.n.], 2013.
- SINGHAL, N.; PARK, I. K.; CHO, S. Implementation and optimization of image processing algorithms on handheld gpu. In: **Proceedings of 2010 IEEE 17th International Conference on Image Processing**. Hong Kong: [s.n.], 2010.

- SINGHAL, N.; YOO, J. W.; CHOI, H. Y.; PARK, I. K. Design and optimization of image processing algorithms on mobile gpu. In: **ACM SIGGRAPH 2011 Posters**. [S.l.: s.n.], 2011.
- SITTHI-AMORN, P.; LAWRENCE, J.; YANG, L.; SANDER, P. V.; NEHAB, D.; XI, J. Automated reprojection-based pixel shader optimization. **ACM Trans. Graph.**, 2008.
- SKOP, P. Comparison of performance of game engines across various platforms. **Journal of Computer Sciences Institute**, 2018.
- ŠMÍD, A. Comparison of unity and unreal engine. **Czech Technical University in Prague**, 2017.
- STENCIL Buffers. Disponível em: <<https://www.ronja-tutorials.com/assets/images/posts/022/Result.gif>>. Acesso em: 01 nov. 2021.
- SUB Surface Scattering Example. Disponível em: <<http://www.mrbluesummers.com/wp-content/uploads/2010/07/Sub-Surface-Scattering-Example.jpg>>. Acesso em: 11 nov. 2021.
- TECHTARGET. **GPGPU (general purpose graphics processing unit)**. Disponível em: <<https://whatis.techtarget.com/definition/GPGPU-general-purpose-graphics-processing-unit>>. Acesso em: 08 nov. 2021.
- TESSELATION Level Table. Disponível em: <https://upload.wikimedia.org/wikipedia/commons/f/fc/Tessellation_Level_Table.png>. Acesso em: 08 nov. 2021.
- ŻUKOWSKI, H. Comparison of 3d games' efficiency with use of cryengine and unity game engines. **Journal of Computer Sciences Institute**, 2019.
- VARCHOLIK, P. **Real-Time 3D Rendering with DirectX and HLSL**. Boston: Addison-Wesley Professional, 2014.
- VARGAS, L. **9ª JEPEEx e 3ª Mostra Cultural - IFRS Erechim- Gustavo Guerreiro**. Youtube, 8 nov. 2020. Disponível em: <<https://www.youtube.com/watch?v=-IWzsVbawbA>>. Acesso em: 27 nov. 2021.
- VIEIRA, T. **OpenGL**. Maceió: Instituto de computação UFAL, [2017?]. 9 slides, color. Disponível em: <<https://ic.ufal.br/professor/thales/OpenGL.pdf>>. Acesso em: 02 nov. 2021.
- VISIONTEK_GEFORCE_256.JPG. Disponível em: <https://upload.wikimedia.org/wikipedia/commons/e/e1/VisionTek_GeForce_256.jpg>. Acesso em: 27 out. 2021.
- VIVO, P. G.; LOWE, J. **The Book of Shaders**. Disponível em: <https://thebookofshaders.com/>. Acesso em: 25 out. 2021.
- WANG, R.; YANG, X.; YUAN, Y.; CHEN, W.; BALA, K.; BAO, H. Automatic shader simplification using surface signal approximation. **ACM Trans. Graph.**, 2014.
- WOLFENSTEIN VS DOOM – The battle of the first person shooters! - Retro Refurbs. Disponível em: <<https://www.retrorefurbs.com/wolfenstein-vs-doom-the-battle-of-the-first-person-shooters/>>. Acesso em: 26 out. 2021.
- Z-BUFFERING. Disponível em: <<https://larranaga.github.io/Blog/imagenes/z-buffer.png>>. Acesso em: 01 nov. 2021.

ZHANG, Z.; LUO, X.; VACA, M. G. S.; CASTRO, D. A. E.; CHEN, Y. Vegetation rendering optimization for virtual reality systems. In: **2017 International Conference on Virtual Reality and Visualization (ICVRV)**. [S.l.: s.n.], 2017.

ZUCCONI, A.; LAMMERS, K. **Unity 5.x Shaders and Effects Cookbook**. Birmingham: Packt Publishing, 2016.

GLOSSÁRIO

B

Benchmark: ato de executar um programa, um conjunto de programas ou outras operações, a fim de avaliar o desempenho relativo, normalmente executando uma série de testes padrão e ensaios.

L

Light Baking: Processo de pré-cálculo de realces e sombras para cenas estáticas que cria a ilusão de iluminação em tempo real..

M

Multithread: capacidade que o sistema operacional possui de executar vários conjuntos de tarefas simultaneamente sem que uma interfira na outra.

R

Realidade Aumentada: integração de elementos virtuais a visualizações do mundo real normalmente através de uma câmera.

Realidade Virtual: ambiente gerado por computador com cenas e objetos que parecem reais, fazendo com que os usuários se sintam imersos nessa realidade.

S

Scripts: programas escritos para um sistema de tempo de execução especial que automatiza a execução de tarefas.

V

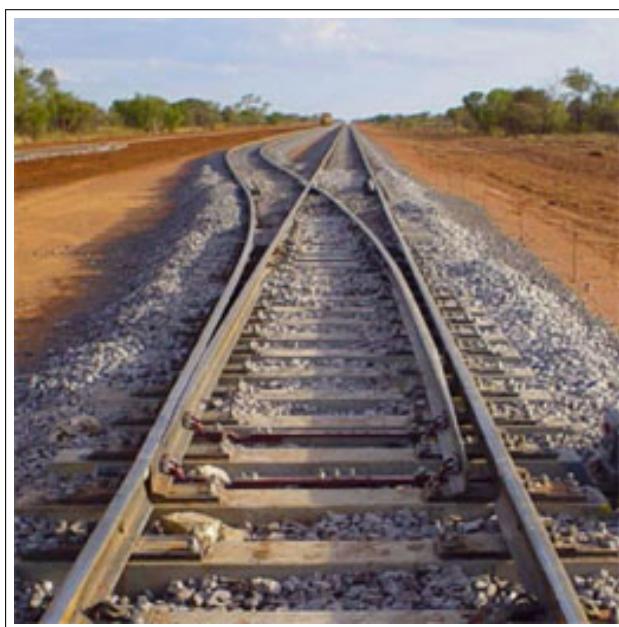
Viewing Frustum: região do espaço no mundo modelado que pode aparecer na tela; campo de visão de um sistema de câmera virtual em perspectiva.

APÊNDICES

APÊNDICE A – Coordenadas Homogêneas

As coordenadas homogêneas foram criadas para solucionar um problema presente na geometria do espaço Euclidiano, onde duas linhas paralelas no mesmo plano nunca poderão se cruzar. Isso só é interessante até o ponto em que precisamos definir o comportamento geométrico no espaço de projeção. Por exemplo na Figura 20 os trilhos se aproximam até que convergem no horizonte (um ponto no infinito no espaço Euclidiano) distante do observador (AHN, 2012).

Figura 20 – A ferrovia fica mais estreita e se cruza no horizonte.



Fonte: <<http://www.songho.ca/math/homogeneous/homogeneous.html>>

Quando um ponto vai para o infinito ele é representado pelas coordenadas (∞, ∞) e isso se torna uma informação inexpressiva. As linhas paralelas deveriam se cruzar no espaço de projeção mas não conseguem no espaço Euclidiano. Para resolver esse problema os matemáticos criaram as coordenadas homogêneas. O motivo de serem chamadas "homogêneas" se dá devido ao fato de que ao convertê-las para coordenadas cartesianas, percebe-se a relação de proporcionalidade explicitada na equação A.5, onde os diferentes pontos representam o mesmo ponto no espaço Euclidiano, ou seja, coordenadas homogêneas são invariantes à escala (AHN, 2012).

Coordenadas homogêneas são utilizadas para fazer a representação de coordenadas N-dimensionais utilizando N+1 números. Então para transformar um ponto de duas dimensões no sistema de coordenadas cartesianas (X, Y) basta simplesmente adicionar uma variável, portanto

(x, y, w) . A correlação matemática entre coordenadas cartesianas e coordenadas homogêneas é exibida abaixo na equação A.1. Utilizando essa lógica fica claro que se um ponto $(1, 2)$, se move em direção ao infinito, se tornando (∞, ∞) , ele passa a ser representado por $(1, 2, 0)$ em coordenadas homogêneas, conforme a equação A.2 (AHN, 2012).

$$X = \frac{x}{w} \quad (A.1)$$

$$Y = \frac{y}{w}$$

$$(1, 2, 0) \rightarrow \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} \approx (\infty, \infty) \quad (A.2)$$

Para provar matematicamente que duas retas paralelas se cruzarão, basta analisar a equação A.3 considerando a geometria Euclidiana. Nesse caso não há solução pois $C \neq D$, apenas se as duas linhas fossem idênticas (sobrepostas) seria possível afirmar que $C = D$. Para que o sistema possua solução é preciso reescrevê-lo como na equação A.4, trocando x e y por suas respectivas coordenadas homogêneas. Então como $(C - D)w = 0 \therefore w = 0$, prova-se que as duas linhas paralelas são capazes de se cruzarem em $(x, y, 0)$ que é um ponto no infinito (AHN, 2012).

$$\begin{cases} Ax + By + C = 0 \\ Ax + By + D = 0 \end{cases} \quad (A.3)$$

$$\begin{cases} A\frac{x}{w} + B\frac{y}{w} + C = 0 \\ A\frac{x}{w} + B\frac{y}{w} + D = 0 \end{cases} \implies \begin{cases} Ax + By + Cw = 0 \\ Ax + By + Dw = 0 \end{cases} \quad (A.4)$$

$$\begin{aligned}
(1,2,3) &\rightarrow \left(\frac{1}{3}, \frac{2}{3} \right) \\
(2,4,6) &\rightarrow \left(\frac{2}{6}, \frac{4}{6} \right) = \left(\frac{1}{3}, \frac{2}{3} \right) \\
(4,8,12) &\rightarrow \left(\frac{4}{12}, \frac{8}{12} \right) = \left(\frac{1}{3}, \frac{2}{3} \right) \\
&\vdots \qquad \rightarrow \qquad \vdots \\
(1a,2a,3a) &\rightarrow \left(\frac{1a}{3a}, \frac{2a}{3a} \right) = \left(\frac{1}{3}, \frac{2}{3} \right)
\end{aligned} \tag{A.5}$$

APÊNDICE B – Código-fonte do shader de contorno em HLSL

Código-fonte 1 – Transcrição do shader de contorno de GLSL para HLSL

```

1   Shader "Unlit/SimpleOutline"
2   {
3       Properties
4       {
5           _OutlineColor("Outline Color", Color) = (1,1,1,1)
6           _OutlineWidth("Outline Width", Float) = 0.5
7       }
8
9       SubShader
10      {
11          Tags {"Queue"="Transparent" "RenderType"="Opaque" }
12          Cull Front
13          Blend SrcAlpha OneMinusSrcAlpha
14
15          Pass
16          {
17              CGPROGRAM
18              #pragma vertex vert
19              #pragma fragment frag
20              #include "UnityCG.cginc"
21
22              struct appdata {
23                  float4 vertex : POSITION;
24              };
25
26              struct v2f {
27                  float4 vertex : SV_POSITION;
28              };
29

```

```
30         float4 _OutlineColor;
31
32
33         v2f vert (appdata v) {
34             v2f o;
35             o.vertex = UnityObjectToClipPos(v.vertex);
36             o.vertex.xyz *= _OutlineWidth;
37             return o;
38         }
39
40         fixed4 frag (v2f i) : SV_Target {
41             return _OutlineColor;
42         }
43     ENDCG
44 }
45 }
46 }
```

ANEXOS

ANEXO A – Código-fonte do shader de contorno em GLSL

Código-fonte 2 – Shader GLSL 3D simples para efeito de contorno

```
1 shader_type spatial;
2
3 render_mode unshaded, cull_front, depth_draw_always;
4
5 uniform float thickness = 0.1; // espessura do contorno
6 uniform vec4 outline_color : hint_color = vec4(1.0); // cor do contorno
7
8 void vertex() {
9     /* desloca cada vertice na direcao de sua normal vezes o fator */
10    VERTEX += NORMAL * thickness;
11}
12
13 void fragment() {
14     /* aplica a cor em cada pixel */
15     ALBEDO = outline_color.rgb;
16     if(outline_color.a < 1.0) ALPHA = outline_color.a;
17}
```

ÍNDICE

AAA, 64