



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

**ESTUDO COMPARATIVO DE FERRAMENTAS DE SHADERS E FORMAS DE
OTIMIZAÇÃO EM DIFERENTES GAME ENGINES**

ANDERSON ARAUJO MACEDO
MATRÍCULA: 1710512

FORTALEZA – CEARÁ
2021

ANDERSON ARAUJO MACEDO

ESTUDO COMPARATIVO DE FERRAMENTAS DE SHADERS E FORMAS DE
OTIMIZAÇÃO EM DIFERENTES GAME ENGINES

Trabalho de Conclusão de Curso apresentado
como exigência parcial para a obtenção do grau
de bacharel em Engenharia de Computação sob
a orientação de conteúdo do professor André
Lunardi de Souza e orientação metodológica da
professora Adriana Pereira do Nascimento.

FORTALEZA – CEARÁ

2021

Ficha catalográfica da obra elaborada pelo autor através do programa de geração automática da Biblioteca Central da Universidade de Fortaleza

Macedo, Anderson Araujo.

ESTUDO COMPARATIVO DE FERRAMENTAS DE SHADERS E FORMAS DE

OTIMIZAÇÃO EM DIFERENTES GAME ENGINES / Anderson Araujo

Macedo. - 2021

91 f.

Trabalho de Conclusão de Curso (Graduação) - Universidade de Fortaleza. Curso de Engenharia De Computação, Fortaleza, 2021.

Orientação: André Lunardi de Souza.

1. Shaders. 2. Unity. 3. Unreal Engine. 4. Godot. 5. Otimização. I. Souza, André Lunardi de. II. Título.

AGRADECIMENTOS

Primeiramente, agradeço a Deus que me mostrou o caminho a seguir nos momentos difíceis e me deu força para que eu nunca desistisse, não somente no período como universitário, mas também ao longo da vida.

Agradeço a minha mãe por ter me apoiado irrestritamente, abrindo mão de tudo por mim. E por sempre ter me incentivado a estudar e a buscar um futuro melhor. Agradeço ao meu pai pela paciência e por investir para o meu desenvolvimento pessoal e aprendizado, para que eu pudesse realizar meu sonho.

Ao professor Paulo Ricardo por ter me motivado a continuar estudando no momento em que eu mais duvidei se estava no caminho certo. E também aos professores Daniel e Imbiriba por instigarem a minha sede de aprendizado com suas aulas incríveis alinhando perfeitamente teoria e prática.

Aos amigos por ter tido a chance de conhecer todos vocês durante essa jornada; espero poder tê-los por muitos outros anos. Obrigado pelos incontáveis momentos de diversão e pelas risadas que compartilhamos juntos ao longo dos anos.

Ao professor e coordenador André Lunardi por ter insistido em me orientar quando todas as minhas alternativas já estavam esgotadas e por ter me gratificado com essa oportunidade de continuar com um ótimo aproveitamento dos meus estudos.

À Vanessa, minha esposa e amor da minha vida, por estar ao meu lado e ser minha companheira de tudo apesar de todas as dificuldades, pelo carinho e amor de todos os dias. Percebo que a vida é mais bonita contigo, leoinha.

“Nunca deixe que lhe digam que não vale a pena
acreditar no sonho que se tem, ou que os seus
planos nunca vão dar certo, ou que você nunca
vai ser alguém”

(Renato Russo)

RESUMO

Shader é um tipo de programa de computador utilizado para simular como a luz interage com os objetos ou as superfícies. Ele torna possível criar aspectos visuais nas superfícies de objetos 3D. Por exigir bastante recursos computacionais, a performance desses programas é algo importante. Atualmente é exigida a renderização em um curto intervalo de tempo de gráficos cada vez mais realistas. Ao fazer uso de shaders custosos e não otimizados, podem ocorrer alguns problemas como surgimento de artefatos, incompatibilidade com hardwares e o superaquecimento da GPU. Nesse estudo foram utilizados três motores de jogo. O primeiro foi o Godot, um software para produção de jogos 2D e 3D criado no ano de 2007. O segundo foi Unity que 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. O terceiro foi Unreal Engine, da Epic Games, que é o mais robusto e também é muito utilizado tanto por desenvolvedores profissionais quanto iniciantes. A utilidade desse estudo consiste em descobrir qual game engine, utilizando critérios quantizados de performance, apresenta a melhor ferramenta para criação de Shaders. A pesquisa realizada para este trabalho é do tipo aplicada. É um estudo com características exploratórias e descritivas. A abordagem é do tipo quantitativa. Os dados necessários para a realização do *benchmarking* foram obtidos pelas ferramentas de profiling, dada sua natureza de tempo real. Os dados foram analisados para os shaders utilizados no cenário padrão de testes em cada uma das game engines levando em consideração múltiplos níveis de otimização. Foi observado que a Unity mostrou-se como uma solução mais completa. A Unreal também ofereceu uma variedade de *shaders* otimizados, porém suas ferramentas de análise de desempenho e opções de otimização deixaram a desejar. Os resultados obtidos com a Godot foram os piores dentre os três.

Palavras-chave: Shaders. Unity. Unreal Engine. Godot. Otimização

ABSTRACT

A shader is a computer program used to simulate how light interact with objects and surfaces. It makes it possible to create visual aspects on the surfaces of 3D objects. As it requires a lot of computational resources, the performance of these programs is something important. Currently, rendering increasingly realistic graphics in a short time span is required. When using costly and not optimized shaders, some problems can occur such as the appearance of artifacts, incompatibility with hardware and GPU overheating. In this study, three game engines were used. The first was Godot, a 2D and 3D game production software created in the year 2007. The second was Unity that is the most common choice among professional and amateur game developers for its rapid prototyping capability and wide range of target platforms. The third was Unreal Engine, by Epic Games, which is the most robust and is also widely used by both professional developers and beginners. The usefulness of this study is to find out which game engine, using quantized performance criteria, presents the best tool for creating Shaders. The research carried out for this work is of the applied type. It is a study with exploratory and descriptive characteristics. The approach is of the quantitative type. The data obtained to perform the *benchmarking* were obtained by profiling tools, given its real-time nature. Data were identified for the shaders used in the test scenario in each of the game engines taking into account optimization levels. It was observed that Unity proved to be a more complete solution. Unreal also offered a variety of optimized *shaders*, however their performance analysis tools and optimization options falls short. The results obtained with Godot were the worst among the three.

Keywords: Shaders. Unity. Unreal Engine. Godot. Optimization

LISTA DE ILUSTRAÇÕES

Figura 1 – Doom faz uso de 3D (esquerda) enquanto Wolfenstein posiciona imagens em diferentes camadas para simular a profundidade (direita).	18
Figura 2 – Hardware da placa gráfica da NVIDIA.	19
Figura 3 – No buffer de profundidade objetos próximos ficam com tonalidade mais escura enquanto objetos distantes assumem uma tonalidade mais clara.	21
Figura 4 – O stencil buffer permite a customização da forma como objetos 3D são renderizados.	21
Figura 5 – Pipeline gráfico do OpenGL.	22
Figura 6 – Conteúdo das quatro colunas da matriz <i>modelView</i>.	25
Figura 7 – 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>.	25
Figura 8 – Demonstração de um shader simples de linha contorno.	27
Figura 9 – Efeito de contorno obtido com HLSL.	29
Figura 10 – Maiores níveis de tesselação produzem aumento no número de vértices.	30
Figura 11 – Regra da mão direita.	31
Figura 12 – 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.	37
Figura 13 – Estrutura hierárquica da cena na Godot Engine.	38
Figura 14 – A programação lógica e criação de materiais usa o sistema de <i>Blueprint</i>.	40
Figura 15 – Fluxograma do processo	48
Figura 16 – Cena padrão de testes	51
Figura 17 – A ferrovia fica mais estreita e se cruza no horizonte.	70

LISTA DE TABELAS

Tabela 1 – Número de instruções necessárias para operações específicas no OpenGL	44
Tabela 2 – Contagem de objetos presentes na cena de teste	50
Tabela 3 – Dados de <i>profiling</i> de renderização da Godot	52
Tabela 4 – Dados de <i>profiling</i> de renderização na Unreal	54
Tabela 5 – Dados estatísticos de renderização da Unity	58
Tabela 6 – Dados de <i>profiling</i> de renderização da Unity	59
Tabela 7 – Dados de <i>profiling</i> de memória e iluminação da Unity	60

LISTA DE QUADROS

Quadro 1 – Principais componentes das <i>game engines</i>	35
Quadro 2 – Funcionalidades gráficas presentes nas <i>game engines</i>	36

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Transcrição do <i>shader</i> de contorno de GLSL para HLSL	73
Código-fonte 2 – <i>Shader</i> GLSL 3D simples para efeito de contorno	76
Código-fonte 3 – <i>Shader</i> de efeito de musgo na Unity	77
Código-fonte 4 – <i>Shader</i> de efeito de água na Unity	80

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Process Unit
CRT	Cathode Ray Tube
FPS	Frames Por Segundo
FXAA	Fast Approximate Anti-Aliasing
GLSL	OpenGL Shading Language
GPGPU	GPU de propósito geral
GPU	Graphics Processing Unit
GUI	Graphic User Interface
HDR	High Dynamic Range
HDRP	High Definition Render Pipeline
HLSL	High-Level <i>Shader</i> Language
HZB	Hierarchical Z-Buffer
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
MS-DOS	Microsoft Disk Operating System
MSAA	Multi-Sample Anti-Aliasing
NES	Nintendo Entertainment System
OpenGL	Open Graphics Library
SDK	Software Development Kit
SGI	Silicon Graphics International
SO	Sistema Operacional
T&L	Transform & Lighting
TAA	Temporal Anti-Aliasing
VRAM	Video Random Access Memory

SUMÁRIO

1	INTRODUÇÃO	14
1.1	Justificativa	15
1.2	Objetivos	15
1.2.1	Objetivo Geral	15
1.2.2	Objetivos Específicos	16
1.3	Estrutura do trabalho	16
2	REFERENCIAL TEÓRICO	17
2.1	Evolução da Programação de Shaders	17
2.1.1	Como o OpenGL funciona	20
2.1.1.1	Pipeline gráfica do OpenGL	22
2.1.1.2	Matrizes de transformação de coordenadas	24
2.1.2	OpenGL Shading Language	26
2.1.3	Direct3D versus OpenGL	28
2.1.4	High-Level Shader Language	29
2.2	Conceitos Técnicos de Shaders	31
2.2.1	Vertex Shader	33
2.2.2	Fragment Shader	34
2.3	Motores de jogo e suas ferramentas	35
2.3.1	Godot	37
2.3.2	Unity	38
2.3.3	Unreal	40
2.3.3.1	Nós de material	41
2.4	Otimização e performance	42
3	METODOLOGIA	48
3.1	Fluxo de Desenvolvimento	48
3.2	Sobre a pesquisa	48
3.3	Escolha das <i>game engines</i>	49
3.4	Sobre os dados e características	49
4	RESULTADOS	51
4.1	Discussão dos resultados na Godot	51
4.2	Discussão dos resultados na Unreal	53

4.3	Discussão dos resultados na Unity	55
5	CONCLUSÃO	61
	REFERÊNCIAS	63
	GLOSSÁRIO	68
	APÊNDICES	69
	APÊNDICE A – Coordenadas Homogêneas	70
	APÊNDICE B – Código-fonte do <i>shader</i> de contorno em HLSL	73
	ANEXOS	75
	ANEXO A – Código-fonte do <i>shader</i> de contorno em GLSL	76
	ANEXO B – Código-fonte do <i>shader</i> de musgo	77
	ANEXO C – Código-fonte do <i>shader</i> de água	80

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

Nesta seção 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, 2021).

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 Instituto de Tecnologia de Massachusetts (MIT) para visualizar a saída de um programa instantaneamente (LUTEN, 2021).

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, 2021). 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 International Business Machines (IBM).

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 – Doom faz uso de 3D (esquerda) enquanto Wolfenstein posiciona imagens em diferentes camadas para simular a profundidade (direita).



Fonte: Retro Refurbs (2021)

Paralelamente, a Silicon Graphics (SGI) — companhia especializada em computação gráfica 3D — trabalhava no lançamento da Open Graphics Library (OpenGL), uma Application Programming Interface (API) *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, 2021).

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, 2021). 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 possuia 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 (LUTEN, 2021). Enquanto isso, OpenGL também trouxe a sua própria linguagem de alto nível chamada OpenGL Shading Language (GLSL).

2.1.1 Como o OpenGL funciona

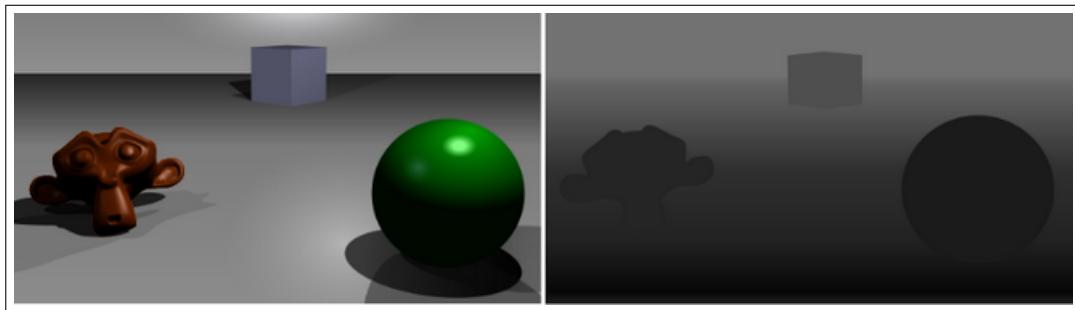
Para Rost (2006), a API do OpenGL desenha gráficos em uma memória especializada em quadros de imagem (*frame buffer*). Ela oferece suporte tanto a geometrias 3D quanto a imagens simples. O modelo de funcionamento dessa API pode ser descrito como cliente-servidor, pois a aplicação (cliente) faz solicitações por meio de comandos que são interpretados e processados pela implementação OpenGL (servidor). 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 (2021), 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).

O princípio 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 (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).

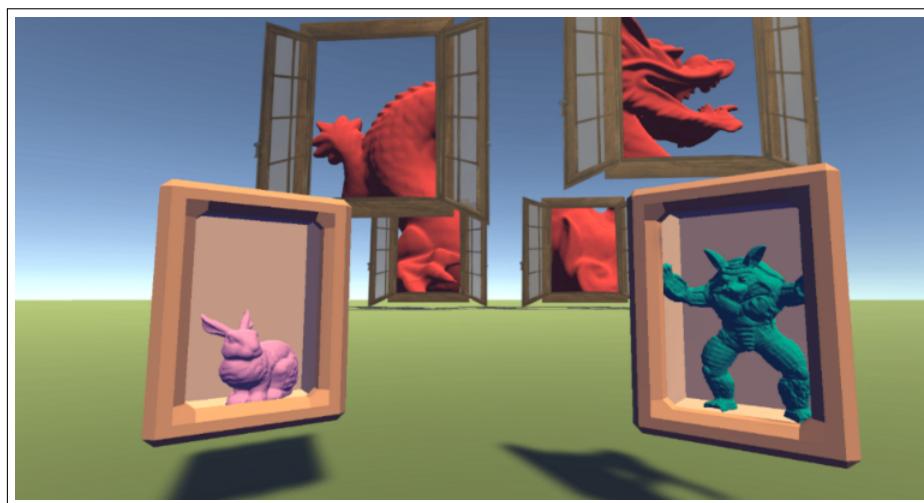
Para que uma janela suporte a renderização ela precisa de alguns elementos: até quatro *buffers* para as cores, um *buffer* de profundidade (Figura 3), um *stencil buffer* (Figura 4), um *buffer* de acumulação, um *multisample buffer* e um ou mais *buffers* auxiliares. A maioria dos *hardwares* suporta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

Figura 3 – No buffer de profundidade 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: Ronja Tutorials (2021)

Para suporte a visualização 3D estéreo, mais dois *buffers* são 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 precisar ser desenhado com remoção de superfície encoberta, o *buffer* de profundidade compara o valor da profundidade de cada *pixel* dos objetos em cena para determinar qual será visível ou obscurecido. 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).

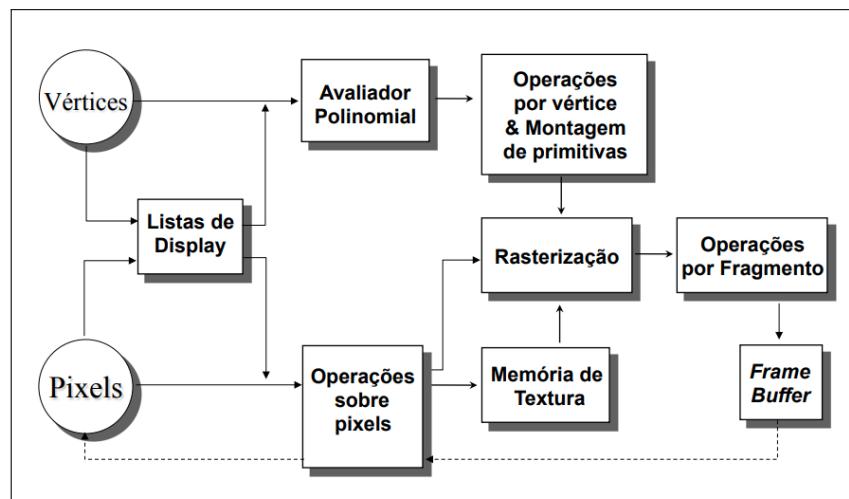
O *buffer* de acumulação é capaz de reproduzir efeitos complexos como suavização em tela cheia de alta qualidade, profundidade de campo e desfoque de movimento. Ele funciona como um *buffer* de cor, porém com maior precisão, capaz de acumular imagens para produzir uma única imagem composta. Já o *multisample buffer* é capaz de produzir várias amostras da

renderização para realizar suavização sem precisar renderizar a cena mais de uma vez. Por último, os *buffers* auxiliares servem para guardar dados genéricos (ROST, 2006).

2.1.1.1 Pipeline gráfica do OpenGL

A máquina de estados do OpenGL possui uma ordem específica em que as operações do processo de renderização precisam ser realizadas. Essa padronização é chamada de *pipeline* gráfica e é mostrada na Figura 5. Todos os dados necessários para desenhar a geometria estão contidos em espaço em memória e podem ser lidos pelo OpenGL de três maneiras diferentes (ROST, 2006).

Figura 5 – Pipeline gráfico do OpenGL.



Fonte: Montenegro (2021)

A primeira maneira é enviar um vértice de cada vez utilizando comandos para manipular atributos dos vértices. A segunda é 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 métodos fazem usam o modo imediato pois as primitivas são renderizadas assim que são especificadas. O terceiro método utiliza um dos dois procedimentos citados acima com uma lista de exibição — uma estrutura de dados que guarda comandos para execução futura. Um ganho de performance desse método é a possibilidade de otimizar os comandos contidos na lista, ou ainda guardar os comandos na memória do acelerador gráfico (ROST, 2006).

A função do avaliador polinomial 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. Isso 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. 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 artefatos resultantes podem ser escritos na memória da textura ou emitidos à etapa de rasterização (ROST, 2006).

Rasterização é a etapa de conversão de dados tanto geométricos como de *pixel* em fragmentos. Cada fragmento passa por 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 simples e pode ser executada eficientemente para milhões de *pixels* por segundo.

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, e mapas cúbicos. 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.

Uma descrição matemática da *pipeline* de renderização está na Equação 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).

$$f((x, y), \mathbf{v}) = p \circ r(g \circ h(\mathbf{v}), (x, y)) \quad (1)$$

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 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

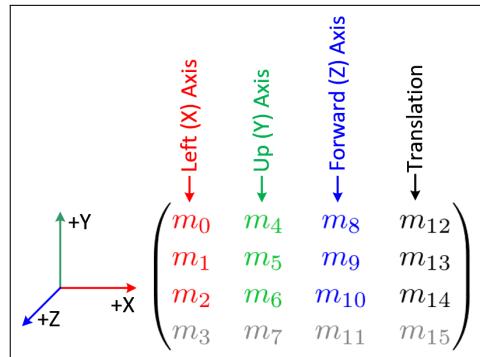
Para o OpenGL transformar descrições de objetos tridimensionais em imagens 2D ele utiliza as informações do modelo do objeto, como posição dos vértices e normais da superfície, para interpretá-las como coordenadas do espaço do objeto. Como cada objeto tem suas próprias características, o sistema de coordenadas global é usado para que seja possível colocar vários objetos em uma única cena. Em seguida a API realiza mais uma conversão para o sistema de coordenadas de olho levando em consideração a posição e o ponto focal da câmera e o vetor de direção para cima (ROST, 2006).

Na equação 2, 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.

$$\begin{bmatrix} x_{olho} \\ y_{olho} \\ z_{olho} \\ w_{olho} \end{bmatrix} = M_{modelView} \cdot \begin{bmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{bmatrix} \quad (2)$$

Como é possível ver na Figura 6 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, 2021).

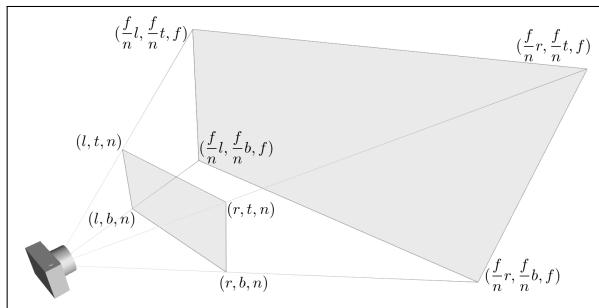
Figura 6 – Conteúdo das quatro colunas da matriz *modelView*.



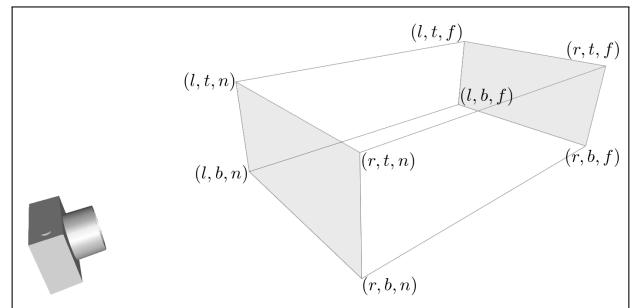
Fonte: Ahn (2021)

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 7) e são mostrados nas equações 3 e 4 respectivamente.

Figura 7 – 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)

$$M_{persp} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (3)$$

$$M_{orto} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

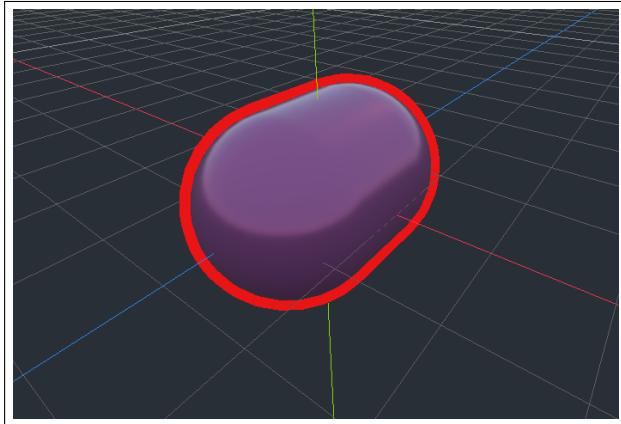
Os valores obtidos nessa operação são normalizados em uma região cúbica definida pelos pontos (-1, -1, -1) e (1, 1, 1) para espaço de coordenadas de dispositivo normalizado, ou seja, os valores passam a ser algum valor entre -1 e 1. Essa etapa é necessária para que a área de visualização seja apropriadamente 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 (AHN, 2021).

2.1.2 OpenGL Shading Language

Pela necessidade de substituir funcionalidades fixas por programabilidade em áreas que ficavam cada vez mais complexas (e.g. processamento de vértices e fragmentos) foram adicionados estágios programáveis por meio da 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. 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 8 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.

Figura 8 – Demonstração de um *shader* simples de linha contorno.



Fonte: Elaborado pelo autor (2021)

A linguagem GLSL faz uso de uma sintaxe similar a linguagem de programação C. Seus tipos incluem vetores e matrizes por serem estruturas fundamentais para cálculos matemáticos com operações para gráficos 3D. Os números de ponto flutuante (*float*) também são fundamentais para conseguir altos níveis de precisão a troco de performance, por isso é possível especificar o nível de precisão ao utilizá-los. Além disso ela oferece suporte a laços, chamadas a sub-rotinas, expressões condicionais e possui funções embutidas (ROST, 2006).

Essa linguagem possibilitou aos desenvolvedores implementar um conjunto de diferentes técnicas para conseguir obter uma variedade de efeitos visuais; outrossim essas técnicas são implementadas com aceleração via *hardware* pela GPU (com processamento paralelo) proporcionando um aumento drástico de performance e liberando carga da CPU para realizar outras tarefas.

O processador de vértices é uma unidade programável que realiza operações nos vértices 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. Variáveis de atributo são utilizadas para passar valores da aplicação para o processador de vértices. Já as variáveis uniformes 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. 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 *versus* OpenGL

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. 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*. As linguagens GLSL e HLSL são muito parecidas e os desenvolvedores conseguem transcrever instruções facilmente de uma para a outra (MICROSOFT, 2021).

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).

Devido a semelhança em capacidade de renderização entre as 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).

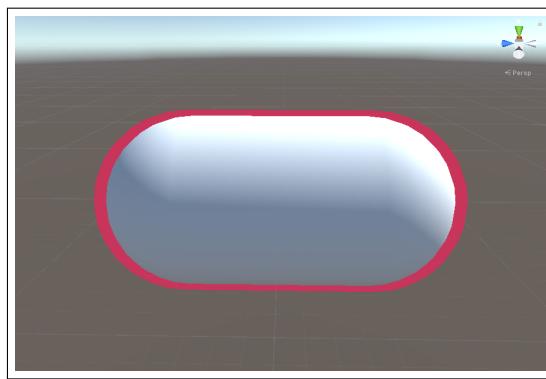
Uma das principais diferenças é o ambiente de execução. O compilador HLSL traduz o código para linguagem de máquina que é processada pelo *driver* do DirectX. Enquanto que

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 ou maior capacidade de processamento (ROST, 2006).

2.1.4 High-Level Shader Language

Essa é uma linguagem de *shader* criada em 2002 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 9 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 9 – Efeito de contorno obtido com HLSL.

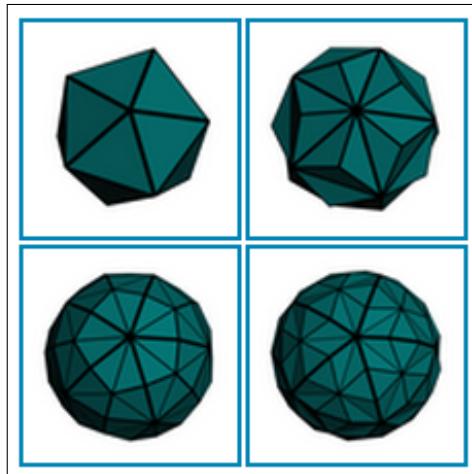


Fonte: Elaborado pelo autor (2021)

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, 2021).

Tesselação é mais um processo na *pipeline* gráfica responsável por adicionar detalhes a objetos diretamente pela GPU como mostrado na Figura 10. Seu modo de funcionamento consiste em subdividir um objeto dinamicamente e sem o custo adicional de reprocessamento de geometria. Isso permite um sistema de nível de detalhe dinâmico e menos utilização do barramento de gráficos, o que melhora a performance (VARCHOLIK, 2014).

Figura 10 – Maiores níveis de tesselação produzem aumento no número de vértices.



Fonte: Wikimedia (2021)

Esse processo 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).

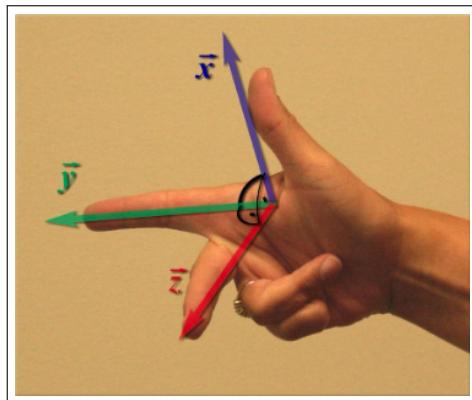
Eles possuem acesso a todas as informações na pipeline. São portanto capazes de escolher parâmetros de tesselação dinamicamente dependendo da informação lida. A principal diferença em relação aos *shaders* de vértice é que enquanto esses modificam os vértices individualmente sem referência às primitivas, 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 11). 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 11 – Regra da mão direita.



Fonte: Luiz (2021)

2.2 CONCEITOS TÉCNICOS DE SHADERS

O HLSL é uma linguagem semelhante a C com estruturas extras para tratar vetores, matrizes e outros elementos relacionados a graficos. Com a melhoria da legibilidade e da produtividade, os desenvolvedores podem 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 os termos utilizados é "o que é um shader". Essa palavra pode causar uma certa estranheza no início mas sua definição não é tão complexa. 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, 2021).

Um *shader* 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 v_i (como coordenadas de posição, normais e texturas) 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).

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).

É possível imaginar um *shader* 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 (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 de programar *shaders* levou ao desenvolvimento de ferramentas visuais que auxiliam a criação desses programas através do uso de 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 sejam 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. 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, o quarto significa que múltiplas instruções podem agir em vários elementos de dados de uma vez (HASU, 2018).

Também é importante mencionar os *shaders* de computação, que são um estágio fora da *pipeline* de renderização utilizados para calcular informações arbitrárias. 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. Dentre suas principais aplicações merecem destaque o suporte a criação de animações realistas e a possibilidade de deformar superfícies para criar efeitos realistas de ondas (NVIDIA, 2021).

O processador de vértices realiza as principais transformações de coordenadas descritas na Subseção 2.1.1.2. Essa é uma ótima etapa para inserir código pois há bastante informação sobre a geometria. 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* (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 de um *shader* 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 consiste 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 *shader* é responsável por produzir a cor final para cada *pixel* a partir de variáveis de estado e valores interpolados através de polígonos. Também é responsável por computar a

cor e a intensidade da luz de cada fragmento. Além disso, é capaz 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, 2021).

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).

Sua principal função é ajudar os 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). O Quadro 1 mostra alguns dos componentes mais importantes dos motores de jogo utilizados nesse estudo.

Quadro 1 – Principais componentes das *game engines*

	Godot	Unity	Unreal
Shading	GLSL	ShaderLab (HLSL) / <i>Shader Graph</i>	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)

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 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 (ŻUKOWSKI, 2019).

Cada *engine* conta com um motor de renderização com funcionalidades que facilitam o processamento das informações da cena (câmera, luzes, materiais e texturas). O Quadro 2 mostra os componentes de renderização dos motores utilizados nesse estudo. Esse motor é uma parte fundamental de uma *game engine* e consome a maior parte dos recursos. 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).

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)

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 e profundidade de campo (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 mostrado na Figura 12.

Figura 12 – 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)

Para escolher qual dos motores de jogos será utilizado em um projeto é preciso ponderar alguns fatores como tipo de licença, possibilidade de modificação do código, possível uso comercial, especificações de hardware, plataforma alvo, habilidade da equipe, ferramentas de suporte, estabilidade da *engine* e público alvo (NAVARRO; PRADILLA; RIOS, 2012).

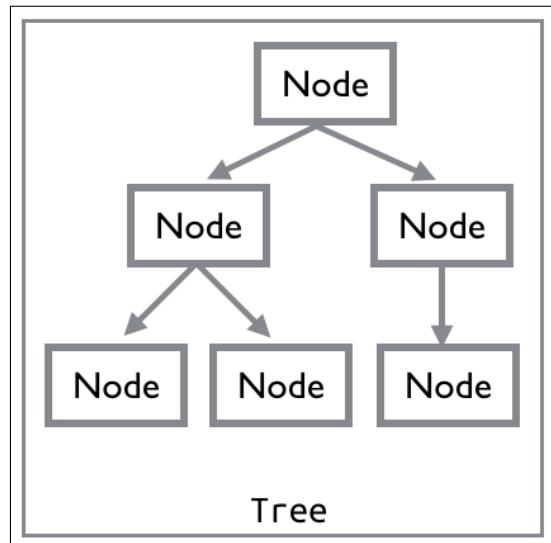
Por serem ferramentas complexas, sua comparação é uma tarefa complicada. A maneira mais apropriada seria implementar o mesmo projeto com complexidade apropriada e de maneira similar em cada *game engine* utilizando critérios mensuráveis para avaliação (ŠMÍD, 2017).

2.3.1 Godot

Para Manzur e Marques (2018), Godot é uma *engine* que difere arquiteturalmente em relação as 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 (Figura 13) de cenas permite organizar o conteúdo do jogo da forma que humanos estão acostumados a visualizar o mundo ao redor.

Figura 13 – Estrutura hierárquica da cena na Godot Engine.



Fonte: Packtpub (2021)

Todos os jogos criados nessa *engines* 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 num bloco funcional com nome, propriedades e uma função especial. (MANZUR; MARQUES, 2018).

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).

2.3.2 Unity

Unity é um motor de jogo desenvolvido e mantido pela empresa Unity Technologies. 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.

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.

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).

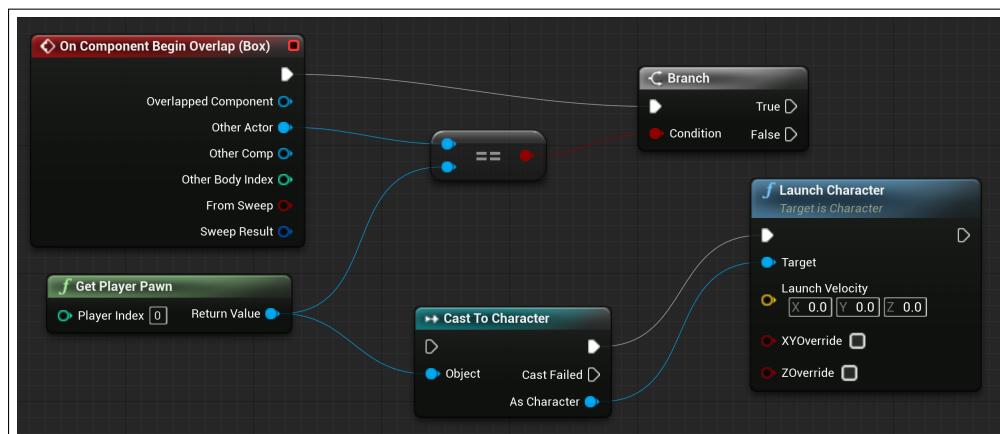
Há ainda alguns *shader* mais complexos que conseguem modificar a geometria base de malhas. São os *shaders* de geometria e tesselação, que pode ser utilizados para adaptar a qualidade visual ao nível de detalhe exigido por meio da otimização da malha em tempo real conforme sua distância da câmera (KUISMIN,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

2.3.3 Unreal

Unreal, lançado em 1998, é um motor de jogo que também permite a criação de jogos para múltiplas plataformas. Ele 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).

É uma *engine* difícil de dominar por possuir excesso de opções à primeira vista, apesar de possuir uma interface amigável. Seu sistema de programação com *BluePrints* (Figura 14) traz uma vantagem para pessoas que não sabem ou não gostam de escrever código (BARCZAK; WOŹNIAK, 2019).

Figura 14 – A programação lógica e criação de materiais usa o sistema de *BluePrint*.



Fonte: Unreal Engine (2021)

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. 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).

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. Shaders grandes (como os usados em filmes) podem 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).

O excesso de cálculos por vértice ou de uso de luzes pode causar 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).

Usar geometria dinâmica é uma prática comum, porém 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 em tempo real, os jogos eletrônicos são softwares que exigem bastante recursos de hardware. Principalmente 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 otimizado desses recursos (SKOP, 2018).

O crescimento do mercado de dispositivos móveis trouxe ampliou o desenvolvimento e a venda de aplicações sofisticadas para esses dispositivos. 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 (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 determina 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).

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. A Tabela 1 mostra a contagem do número de instruções para operações de shaders.

Tabela 1 – 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)

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).

Além disso, como o número de vértices processados em operações de processamento de imagens é muito mais baixo que o de fragmentos, é recomendado realizar cálculos por vértice ao invés de por fragmento por aqueles serem menos custosos. 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 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.

Para medir a performance, é necessária uma forma de Benchmark para comparar a performance com mais precisão e objetividade. Alguma 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).

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 enorme para motores de jogos modernos, que podem conter vários *shaders* únicos, implementando uma vasta coleção de materiais e múltiplos níveis de detalhe que podem exigir diferentes procedimentos de otimização.

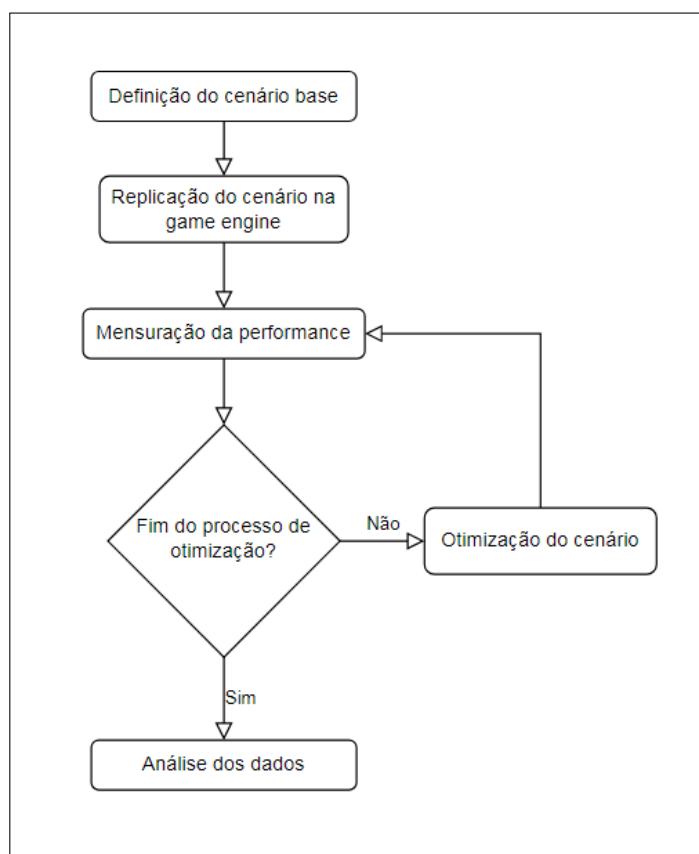
3 METODOLOGIA

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 15 mostra o fluxo utilizado para o desenvolvimento deste trabalho.

Figura 15 – 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 E CARACTERÍSTICAS

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.

A cena padrão de testes foi estabelecida por meio do uso de um conjunto de objetos 3D para caracterizar um teste de estresse pelo uso de várias malhas com bastantes triângulos. A Tabela 2 contém as quantidades de elementos contidos na cena.

Tabela 2 – Contagem de objetos presentes na cena de teste

	Vértices	Triângulos	Qtd.
Casa	6901	4264	72
Plano	4	2	144
Árvores	794	368	288
Cercas	7156	3632	360
Cogumelos	2208	1056	360
Rochas	—	—	—
Tipo 1	576	338	432
Tipo 2	554	338	144
Tipo 3	690	426	288
Tipo 4	512	280	216
Tipo 5	120	62	72
Água	8	12	4
Soma	19523	10778	2380
Total	46464740	25651640	—

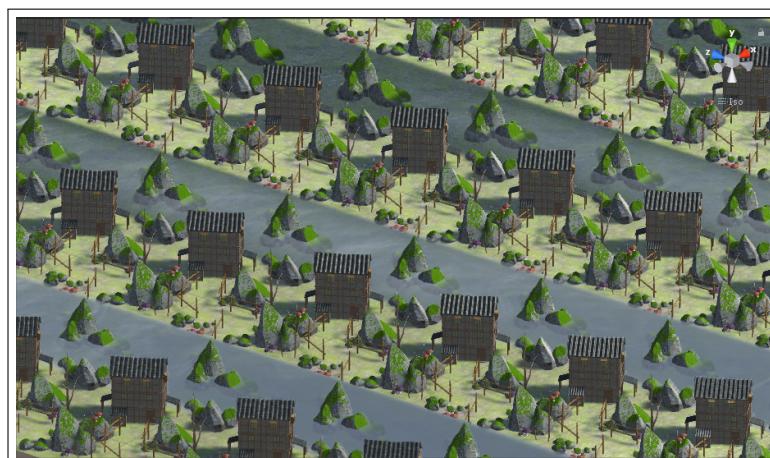
Fonte: Elaborado pelo Autor (2021)

4 RESULTADOS

A seguir são apresentados os resultados do estudo conduzido para cada um dos motores de jogo, aplicando as recomendações mencionadas na documentação de cada um e utilizando o cenário base de testes. Como cada motor utiliza o seu próprio formato de arquivo para manipular os *assets*, para que fosse possível replicar o mesmo cenário nos três motores foi necessário utilizar um cenário pré-modelado, onde os objetos 3D foram posicionados fixamente.

A cena de testes consiste em várias cópias de um conjunto de objetos, dentre eles uma casa, algumas pedras, uma malha que representa a água, um plano para o chão, algumas árvores e cogumelos como mostrado na Figura 16. Esses objetos foram duplicados até que fosse obtido um valor médio de taxa de quadros dentro do padrão aceitável para jogos eletrônicos de 30 FPS.

Figura 16 – Cena padrão de testes



Fonte: Elaborado pelo Autor (2021)

4.1 DISCUSSÃO DOS RESULTADOS NA GODOT

A cena de teste foi replicada na Godot *engine* preservando as características fundamentais de número de triângulos (vértices) e materiais (shaders). Obviamente, houve necessidade de adaptação nos *shaders* de musgo sobre rochas e no *shader* de água devido ao fato do motor de renderização da *engine* fazer uso de GLSL.

Aqui é interessante destacar que a Godot executa automaticamente o culling para evitar a renderização de objetos que estão fora da janela de exibição. Isso funciona bem para jogos que acontecem em uma pequena área, no entanto, isso pode rapidamente se tornar problemático

para níveis maiores.

Seguindo a recomendação da documentação da Godot, foi utilizado o cozimento de iluminação. Os mapas de luz cozidos são um fluxo de trabalho alternativo para adicionar iluminação indireta a uma cena.

Lightmaps preparados funcionam bem em PCs de baixo custo e dispositivos móveis, visto que eles quase não consomem recursos em tempo de execução. Os mapas de luz preparados podem ser usados opcionalmente para armazenar iluminação direta, o que fornece ainda mais ganhos de desempenho (LINIETSKY; MANZUR, 2021).

A iluminação de objetos é uma das operações de renderização mais caras. Iluminação em tempo real, sombras (especialmente luzes múltiplas) e GI são especialmente caros. Eles podem demandar muito de dispositivos móveis de baixa potência. O uso de iluminação cozida tem a desvantagem de não ser dinâmico. Em geral, se várias luzes precisam afetar uma cena, é melhor usar mapas de luz cozidos. O cozimento também pode melhorar a qualidade da cena, adicionando reflexos indiretos de luz (LINIETSKY; MANZUR, 2021).

A próxima etapa consistiu em desativar o filtro das texturas. Ler texturas é uma operação cara, especialmente ao ler várias texturas em um sombreador de fragmento. Além disso, a filtragem pode desacelerar ainda mais esse processo (filtragem trilinear entre mipmaps e cálculo da média). Ler texturas também é caro em termos de uso de energia, o que é um grande problema em celulares.

A Tabela 3 mostra os resultados obtidos para a aplicação de cada etapa de otimização.

Tabela 3 – Dados de *profiling* de renderização da Godot

	FPS	Chamadas de desenho	Vértices desenhados	Uso de memória de vídeo
Configuração padrão	24	26143	11079162	338.7 MB
Etapa 1				
Uso de luz cozida (Low)	26	26143	11079162	446.7 MB
Uso de luz cozida (Medium)	24	26143	11079162	446.7 MB
Uso de luz cozida (High)	25	26143	11079162	446.7 MB
Etapa 2				
Filtro desativado	19	36437	15995958	338.7 MB
Filtro desativado (Sem sombras)	28	26143	11079162	338.7 MB

Fonte: Elaborado pelo Autor (2021)

A partir dos dados obtidos fica nítido que o cozimento de luzes não se mostrou como uma solução efetiva para melhoria de performance, pois não aumentou de maneira considerável a taxa de quadros por segundo.

Também foi possível notar que o uso de memória aumentou, o que já era esperado tendo em vista que o cozimento de luzes processa as sombras em um arquivo de textura que é salvo em memória para ser lido em tempo de execução.

A etapa mais efetiva para ganho de desempenho acabou sendo o desativamento das sombras, com isso foi possível chegar mais próximo do objetivo de 30 FPS. Por último, desativar o filtro não se mostrou como uma medida viável visto que prejudicou a performance.

4.2 DISCUSSÃO DOS RESULTADOS NA UNREAL

Na Unreal, um ponto positivo vai para a existência de materiais de ambiente (Gramas, tijolos, pedras e até água) já prontos para serem utilizados, e inclusive com otimizações de performance como nível de detalhe e tesselação, dentro da pasta de conteúdo inicial do template de projeto padrão. Isso ajuda a reduzir bastante o trabalho dos desenvolvedores ao implementar *shaders* em seus jogos.

A primeira recomendação na documentação da Unreal sugere desativar a projeção de sombra, que é uma opção que vem habilitada por padrão. As sombras fazem com que os objetos pareçam estar ancorados no mundo e dão ao observador uma sensação de profundidade e espaço.

Sombras estáticas não são custosas no que diz respeito à renderização, mas sombras dinâmicas podem ser um dos maiores problemas de desempenho. Em média, as luzes de projeção de sombras dinâmicas móveis são as mais custosas (EPIC GAMES, 2021).

Para a realização desta etapa, foi desativa a projeção de sombra individualmente para cada malha e em seguida, com a projeção das malhas reativada, foi desativado apenas na fonte de luz direcional.

Por padrão, o Unreal Engine usa um renderizador diferido, pois ele fornece maior versatilidade e concede acesso a mais recursos de renderização. A renderização diferida não é apenas mais rápida, mas também oferece melhores opções de anti-aliasing, o que gera melhores aspectos visuais.

A renderização direta fornece uma linha de base com passagens de renderização mais rápidas. A segunda etapa então consistiu em mudar, nas configurações do projeto, o modo de renderização de diferido para direto.

A próxima etapa consistiu em ajustar a oclusão de HZB (Hierarchical Z-Buffer) que tem um custo constante alto, mas um custo por objeto menor. A oclusão HZB funciona da mesma forma que a oclusão padrão, exceto que é mais conservadora na maneira que seleciona objetos, o

que significa que menos objetos são eliminados como resultado (EPIC GAMES, 2021).

Ele usa uma versão *mipmap* do alvo de renderização de profundidade de cena para verificar os limites de um ator. Também requer menos buscas de textura ao amostrar de um *mipmap* menos detalhado (EPIC GAMES, 2021).

A Tabela 4 mostra os resultados obtidos para a aplicação de cada etapa de otimização.

Tabela 4 – Dados de *profiling* de renderização na Unreal

	Valor médio do maior gargalo de renderização	Chamadas de desenho	Uso máximo de memória de renderização	Triângulos desenhados
Configuração padrão	17.01 ms	3654	408.26 MB	870023
Etapa 1				
Sem projeção de sombra	16.69 ms	3654	408.26 MB	870019
Etapa 2				
Sem redução de serrilhamento	16.65 ms	3654	408.01 MB	870023
FXAA	17.78 ms	3654	393.68 MB	870035
TAA	16.77 ms	3654	402.68 MB	870023
MSAA	17.19 ms	3654	402.68 MB	870023
Etapa 3				
Sem redução de serrilhamento	24.59 ms	5330	363.97 MB	1190199
FXAA	24.05 ms	5330	363.97 MB	1190235
TAA	23.56 ms	5330	369.31 MB	1190223
MSAA	23.59 ms	5340	390.64 MB	1193082
Etapa 4				
HZBO	22.85 ms	5330	361.59 MB	1190235
Etapa 5				
Resolução 50%	22.73 ms	5337	366.92 MB	1191363
Resolução 75%	22.77 ms	5334	361.59 MB	1190779
Resolução 50%	22.42 ms	5341	361.59 MB	1192500
Qualidade média	23.55 ms	5341	383.01 MB	1193025
Qualidade alta	23.74 ms	5468	389.02 MB	1195871
Qualidade épica	24.07 ms	5473	399.02 MB	1196494
Etapa 6				
Qualidade média (Materiais)	23.84 ms	5341	361.59 MB	1192760
Qualidade alta (Materiais)	23.02 ms	5341	361.59 MB	1192724

Fonte: Elaborado pelo Autor (2021)

A análise dos dados obtidos permitiu concluir que a melhor performance de taxa de quadros por segundo (39.67 FPS) ocorreu utilizando o modo de renderização direto com resolução de 75%. Apesar de que esse modo adicionou aproximadamente 7 ms de tempo extra de renderização. Em compensação o uso máximo de memória foi reduzido em cerca de 12%.

Além disso o modo direto permitiu que a Unreal fosse capaz de renderizar mais triângulos (e consequentemente realizar mais chamadas de desenho) sem que houvesse uma perda extrema de performance.

A aplicação de técnicas de otimização de performance na Unreal se mostra mais

difícil por exigir um conhecimento da ferramenta de console da *engine* para aplicar comandos necessários para alterar aspectos que afetam a performance.

Um detalhe importante para levar em consideração é que ao invés de fornecer alternativas embutidas na própria engine, ela recomenda e oferece soluções de terceiros para realizar benchmark e otimizações de gráficos. Além disso, a sua documentação, diferente da Unity, não aborda tantas opções de melhorias de performance em um clique.

Por isso ela é considerada um motor de jogo mais apropriado para usuários avançados, por entender que esses já possuem o conhecimento das técnicas necessárias para melhorar a performance e conseguem aplicá-las sem o auxílio de opções simples embutidas na engine.

Por fim, é importante salientar que desempenho é um tópico omnipresente na criação de jogos em tempo real. Para criar a ilusão de imagens em movimento, uma taxa de quadros de pelo menos 15 quadros por segundo é necessária. Dependendo da plataforma e do jogo, 30, 60 ou até mais quadros por segundo podem ser o alvo (EPIC GAMES, 2021).

A Unreal Engine oferece muitos recursos e eles têm diferentes características de desempenho. A fim de otimizar o conteúdo ou código para atingir o desempenho necessário, é preciso ver onde o desempenho é gasto. Cada caso é diferente e algum conhecimento sobre os componentes internos de *hardware* e *software* é necessário (EPIC GAMES, 2021).

4.3 DISCUSSÃO DOS RESULTADOS NA UNITY

Na Unity foi criado um novo projeto com template 3D. Em seguida foram importados os modelos dos objetos para compor a cena. Ao importar os objetos foram removidas as importações de materiais e animações, pois estes não serão utilizados. A cena final é composta por uma câmera com posição fixa e contém aproximadamente 3 milhões e 300 mil triângulos (5 milhões e 500 mil vértices) resultando em 7211 *batches* (chamadas de desenho).

Para a criação dos materiais, foram utilizados os padrões da *engine* que consistem no *shader* de superfície padrão, exceto para a implementação de um *shader* de musgo que afeta as pedras e para o *shader* de água. Ambos foram escritos utilizando a linguagem *ShaderLab* e constam nos Anexos B e C respectivamente.

Por padrão, a Unity importa um rig genérico para modelos de não personagens. Isso faz com que um componente Animator seja adicionado se o modelo for instanciado no tempo de execução. Se o modelo não for animado por meio do sistema de animação, isso adiciona sobrecarga desnecessária ao sistema de animação, porque todos os animadores ativos devem ser

marcados uma vez por quadro (UNITY TECHNOLOGIES, 2021).

Desativar o rig em modelos não animados para evitar esta adição automática de um componente Animator e possível adição inadvertida de animadores indesejados a uma cena é uma boa prática para objetos não animados e foi utilizada nessa cena (UNITY TECHNOLOGIES, 2021).

É importante ressaltar que ao criar um material na Unity ele já possui um *shader* padrão embutido, facilitando bastante a implementação de *shaders* básicos (como cores ou texturas). Ou seja, para implementar *shaders* simples não é necessário realizar nenhum tipo de programação.

Foram utilizadas as configurações pré-definidas da Unity (tamanho máximo de 2048x2048, compressão normal e algoritmo de redimensionamento Mitchell) para importação das texturas que compõem os materiais. A única mudança realizada foi a remoção do filtro bilinear.

As etapas de otimização seguem as recomendações descritas na documentação da Unity para melhorias de performance. A cada etapa foram medidas as estatísticas fornecidas pela *engine* para comparar os procedimentos.

A primeira etapa de otimização seguindo a recomendação da documentação da Unity consiste em utilizar tamanhos menores de textura. Para aplicações mobile, 2048x2048 ou 1024x1024 é um tamanho suficiente de atlas de textura, e 512x512 é um tamanho suficiente para texturas individuais aplicadas em modelos 3D. Em contrapartida isso pode gerar uma perda de qualidade visual.

A próxima etapa de otimização consiste em desabilitar o sinalizador de leitura e gravação dos modelos 3D, pois ele é habilitado por padrão para todos os modelos importados. A Unity exige que este sinalizador seja habilitado se um projeto estiver modificando uma malha em tempo de execução via *script*, ou se a malha for usada como base para um componente MeshCollider (o que não é o caso nessa cena padrão). Se o modelo não for usado em um MeshCollider e não for manipulado por scripts, é recomendado desativar este sinalizador para salvar metade da memória do modelo.

Se o formato de compressão da textura selecionado não é adequado para a plataforma de destino, o Unity descompacta a textura quando ela é carregada, consumindo tempo de CPU e uma quantidade excessiva de memória. Esse é um problema mais comum em dispositivos Android, que geralmente oferecem suporte a formatos de compactação de textura amplamente diferentes, dependendo do chipset.

Mais uma etapa de otimização consiste em usar a compressão de malha quando for possível. Habilitar a compactação de malha reduz o número de bits usados para representar os números de ponto flutuante para diferentes canais de dados de um modelo. Isso pode levar a uma pequena perda de precisão e os efeitos dessa imprecisão devem ser verificados pelos artistas antes do uso em um projeto final. É possível usar três diferentes níveis de compressão (Low, Medium, High).

Outra etapa recomendada pela Unity é o uso da funcionalidade *Graphics jobs*, uma opção que determina se serão utilizadas *worker threads* (são *threads* que performam uma única tarefa, como *culling* ou *mesh skinning*). Em plataformas onde essa funcionalidade está disponível, ela pode melhorar consideravelmente a performance.

Uma técnica interessante consiste em reduzir a distância de projeção da câmera usando a propriedade *Far Clip Plane*. Esta propriedade é a distância além da qual os objetos não são mais renderizados pela câmera. Para disfarçar o fato de que objetos distantes não são mais visíveis, é possível usar a névoa. Após reduzir a distância de 1000 para 77 foi possível obter o dobro de taxa de quadros (60 FPS).

Caso a névoa seja um aspecto indesejado no jogo, uma alternativa na próxima etapa consiste em ativar a oclusão para desativar a renderização de objetos que estão ocultos por outros objetos. A seleção de oclusão do Unity não é adequada para todas as cenas, pode levar a sobrecarga de CPU adicional e pode ser complexa para configurar, mas pode melhorar muito o desempenho em algumas cenas.

Focando na próxima etapa, é possível fazer uso da instanciação de GPU para desenhar (ou renderizar) várias cópias da mesma malha de uma vez, usando um número menor de chamadas. Isso é útil para desenhar objetos como edifícios, árvores, grama ou outras coisas que aparecem repetidamente em uma cena.

Esse é um método útil para renderizar malhas idênticas e cada instância pode ter parâmetros diferentes (por exemplo, cor ou escala) para adicionar variação e reduzir a aparência de repetição. Isso pode ainda reduzir o número de chamadas de desenho e melhorar significativamente o desempenho de renderização (UNITY TECHNOLOGIES, 2021).

Há ainda uma etapa de definição do nível de cascatas de sombra. Basicamente, quanto mais cascatas forem utilizadas, menos as sombras são afetadas pelo *aliasing* de perspectiva. Aumentar o número aumenta a sobrecarga de renderização. No entanto, essa sobrecarga ainda é menor do que seria no caso de um mapa de alta resolução em toda a sombra.

Outro processo descrito consiste em utilizar os *shaders* móveis otimizados embutidos

na Unity; deve-se experimentar usá-los e ver se isso melhora o desempenho sem afetar a aparência do jogo. Esses *shaders* foram projetados para uso em plataformas móveis, mas são adequados para qualquer projeto. É perfeitamente normal usar *shaders* "móvels" em plataformas não móveis para aumentar o desempenho se eles fornecerem a fidelidade visual necessária para o projeto.

Devido a alta disponibilidade de dados de *profiling* oferecidos pela Unity, os dados foram divididos em mais de uma tabela para melhor visualização. A Tabela 5 mostra os resultados obtidos para a aplicação de cada etapa de otimização para as estatísticas gráficas.

Tabela 5 – Dados estatísticos de renderização da Unity

	Thread de renderização	Batches (Salvo)	FPS
Etapa 1			
Textura 1024x1024	29.6 ms	7211 (15342)	28.1
Textura 512x512	28.4 ms	7211 (15342)	28.9
Etapa 2			
Sem Leitura e Escrita (Malha)	26.5 ms	7226 (15327)	30.1
Compressão baixa (Malha)	28.7 ms	7215 (15338)	28.1
Compressão média (Malha)	31.6 ms	7213 (15340)	25.9
Compressão alta (Malha)	28.7 ms	7199 (15354)	28.6
Etapa 3			
Graphic Job ativado	29.2 ms	7199 (15354)	28.9
Etapa 4			
Névoa ativada	15.8 ms	3397 (6778)	56.7
Culling ativado	27.2 ms	7199 (15354)	29.8
Instanciação de GPU	14.7 ms	2228 (20325)	32.3
Etapa 5			
Cascata de sombra (0)	13.2 ms	2056 (17783)	41.4
Cascata de sombra (2)	14.2 ms	2141 (18913)	39.3
Etapa 6			
Shaders <i>mobile</i>	28.1 ms	7015 (15538)	28.6

Fonte: Elaborado pelo Autor (2021)

Após análise dos dados acima fica bem claro que o uso da névoa foi a medida mais efetiva para melhorar a performance, chegando a marca de 56.7 FPS e um dos menores tempos de *thread* de renderização. Isso também se deve ao fato de que o número de *Batches* acaba sendo reduzido.

Batching é o nome do processo de criação de lotes — grupos de objetos que são enviados como apenas um para a GPU. O Unity faz lotes estáticos e dinâmicos. Objetos estáticos que compartilham exatamente o mesmo material serão enviados como um único objeto (lote). Objetos dinâmicos que compartilham exatamente o mesmo material serão enviados como um único lote sob certas circunstâncias. Sem lote, cada objeto teria que ser enviado como seu próprio lote, criando uma grande sobrecarga de CPU.

Outro ponto importante foi a utilização do nível 0 de cascata de sombra que também diminuiu bastante o tempo de *thread* de renderização.

Já a Tabela 6 contém os dados referentes à memória de renderização obtidos pelo *profiler*. A ferramenta de *profiling* dá informações detalhadas sobre o desempenho do jogo. Se o jogo apresentar baixa taxa de quadros ou alto uso de memória, essa ferramenta pode mostrar o que está causando esses problemas e ajudar a corrigi-los.

A janela do *Profiler*, mostra diferentes aspectos do desempenho do jogo. Por exemplo, uso de memória, quanto tempo de CPU está sendo usado para diferentes tarefas e com que frequência os cálculos físicos estão sendo realizados. É possível usar esses dados para ajudar a encontrar a causa dos problemas de desempenho e medir a eficácia de tentativas de correção (UNITY TECHNOLOGIES, 2021).

É importante entender que há um pequeno custo de desempenho ao usar a janela do Profiler para registrar dados. Isso é comum a todas essas ferramentas; não é possível registrar e exibir dados detalhados como esses sem alguma sobrecarga. Embora seja improvável que isso faça uma diferença significativa na maneira como o jogo é executado (UNITY TECHNOLOGIES, 2021).

Tabela 6 – Dados de *profiling* de renderização da Unity

	Uso de VRAM (MB)	Uso de texturas	Renderização de texturas
Etapa 1			
Textura 1024x1024	39.1 — 62.3	22.1 MB	33.6 MB
Textura 512x512	39.1 — 48.8	8.6 MB	33.6 MB
Etapa 2			
Sem Leitura e Escrita (Malha)	52.5 — 62.2	8.6 MB	47.0 MB
Compressão baixa (Malha)	49.3 — 59	8.6 MB	43.8 MB
Compressão média (Malha)	62.2 — 71.9	8.6 MB	56.7 MB
Compressão alta (Malha)	59 — 68.7	8.6 MB	53.5 MB
Etapa 3			
Graphic Job ativado	51.7 — 61.3	8.6 MB	46.2 MB
Etapa 4			
Névoa ativada	51.6 — 61.3	8.6 MB	46.1 MB
Culling ativado	62.6 — 72.3	8.6 MB	57.1 MB
Instanciação de GPU	62.7 — 72.3	8.6 MB	57.2 MB
Etapa 5			
Cascata de sombra (0)	51.6 — 61.3	8.6 MB	46.1 MB
Cascata de sombra (2)	47.6 — 57.3	8.6 MB	42.1 MB
Etapa 6			
Shaders mobile	55.1 — 61	4.9 MB	49.6 MB

Fonte: Elaborado pelo Autor (2021)

Aqui é possível perceber que em termos de uso de memória, a redução da resolução

das texturas ajudou a diminuir 60% da carga de uso. Por outro lado, a porção de memória destinada a renderização apenas aumentou.

Notou-se também que o melhor intervalo de uso de VRAM (Video Random Access Memory) ocorreu ao usar a menor resolução de textura (a baixa compressão de malha também gerou resultados interessantes). O uso dos *shaders* otimizados para dispositivos móveis embutidos na Unity ajudou a diminuir 3,7 MB de carga de uso de memória. O que no final das contas não fez uma diferença considerável.

A Tabela 7 mostra os dados referentes ao *profiling* de uso de memória e iluminação global.

Tabela 7 – Dados de *profiling* de memória e iluminação da Unity

	Memória de textura	Memória de malha	Memória de material	Tempo entre <i>Updates</i>
Etapa 1				
Textura 1024x1024	72 MB	2.9 MB	94 KB	35.31 ms
Textura 512x512	41.8 MB	2.5 MB	95 KB	40.77 ms
Etapa 2				
Sem Leitura e Escrita (Malha)	41.5 MB	2.8 MB	96 KB	41.07 ms
Compressão baixa (Malha)	41.5 MB	2.4 MB	96 KB	40.45 ms
Compressão média (Malha)	41.5 MB	2.6 MB	96 KB	37.19 ms
Compressão alta (Malha)	41.5 MB	2.6 MB	96 KB	41.38 ms
Etapa 3				
Graphic Job ativado	48.7 MB	2.1 MB	97 KB	36.56 ms
Etapa 4				
Névoa ativada	48.7 MB	2.6 MB	97 KB	19.29 ms
Culling ativado	48.7 MB	2.2 MB	97 KB	38.71 ms
Instanciação de GPU	48.7 MB	2.8 MB	99 KB	9.88 ms
Etapa 5				
Cascata de sombra (0)	48.7 MB	2.1 MB	99 KB	26.74 ms
Cascata de sombra (2)	48.7 MB	2.9 MB	99 KB	9.79 ms
Etapa 6				
Shaders <i>mobile</i>	48.7 MB	2.7 MB	90 KB	38.83 ms

Fonte: Elaborado pelo Autor (2021)

Pela análise dos dados obtidos percebe-se que o uso de *Culling*, de *Graphic Jobs* e a diminuição dos níveis de cascata de sombra são formas efetivas de diminuir o uso de memória de malha.

O uso de *shaders* de dispositivos móveis é uma boa prática para reduzir o uso de memória de material. Além disso o uso de instanciação e a diminuição dos níveis de cascata de sombra foram úteis para reduzir o tempo entre atualização de quadro.

5 CONCLUSÃO

Tendo em vista os aspectos analisados nesse estudo, foi possível comparar as principais ferramentas de desenvolvimento de *shaders* entre os motores de jogo Unity, Unreal e Godot com foco na otimização de performance em cada uma.

Foi observado que a Unity mostrou-se como uma solução mais completa por oferecer, de maneira eficaz, ferramentas de análise de execução de *shaders* e opções de otimização para o usuário. E também por disponibilizar *shaders* otimizados para uso imediato. No final, mostrou-se como um motor de jogo mais amigável a novos usuários por conter uma documentação rica em detalhes e técnicas de otimização.

A Unreal também ofereceu uma variedade de *shaders* otimizados (apesar de não serem voltados para dispositivos móveis), porém suas ferramentas de análise de desempenho e opções de otimização deixaram a desejar quando comparadas as da Unity. Sua documentação mostrou-se voltada para usuários com conhecimento avançado em técnicas de otimização.

A Godot possui uma documentação muito boa, de leitura fácil, porém que peca por trazer exemplos muito generalizados de uso. Isso acabou por criar uma limitação de técnicas que poderiam ser aplicadas para otimizar os *shaders*. Além disso, os resultados obtidos com esse motor de jogo foram os piores dentre os três.

Uma recomendação interessante, que foi além do escopo desse estudo, consiste em desativar os *mipmaps* para objetos que não apresentam grande variação de profundidade no campo de visão da câmera (como texturas com tamanho fixo), para salvar cerca de um terço de memória de carregamento de textura. Por outro lado, caso haja variação é recomendado ativar o uso de *mipmaps* para melhorar a performance.

Cabe ressaltar que os objetos que compartilham configurações semelhantes podem ser combinados na mesma chamada de desenho através da criação de lotes. A CPU cria um pacote de dados para cada chamada. Às vezes, os lotes podem conter outros dados além das chamadas de desenho, mas é improvável que essas situações contribuam para problemas comuns de desempenho.

A CPU coleta informações sobre cada objeto que será renderizado e classifica esses dados em comandos conhecidos como chamadas de desenho. Uma chamada de desenho contém dados sobre uma única malha e como essa malha deve ser renderizada (por exemplo, quais texturas devem ser usadas).

Outrossim, otimizar partes que não possuem relação com os gargalos é uma perda

de tempo e provavelmente irá introduzir novos *bugs* ou até mesmo regressões de desempenho em outros casos. A cada nova etapa de otimização, o ideal é utilizar a ferramenta de *profiling* novamente, pois isso pode revelar um novo gargalo de desempenho que antes estava oculto.

REFERÊNCIAS

- ABDALA, D. D. **Primitivas Gráficas 2D**. 2019. 13 slides, color. Disponível em: <http://www.facom.ufu.br/~abdala/GBC204/03_primitivas2D.pdf>. 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.
- BELAIR, F.; LOVATO, N. **outline3D**. Disponível em: <https://github.com/GDQuest/godot-shaders/blob/master/godot/Shaders/outline3D.shader>. Acesso em: 09 nov. 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.
- 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.
- EPIC GAMES. **Visibility and Occlusion Culling**. Disponível em: <<https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/VisibilityCulling/>>. Acesso em: 08 dez. 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.; TATARUCHUK, 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.
- 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.
- LINIETSKY, J.; MANZUR, A. **Baked lightmaps.** Disponível em: <https://docs.godotengine.org/pt_BR/stable/tutorials/3d/baked_lightmaps.html>. Acesso em: 09 dez. 2021.
- LOGHIN, D.-A. **Water Shader.** Disponível em: <<https://github.com/tuxalin/water-shader/blob/master/shaders/unity/water.shader>>. Acesso em: 08 dez. 2021.
- 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.

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.

ŽUKOWSKI, H. Comparison of 3d games' efficiency with use of cryengine and unity *game engines*. **Journal of Computer Sciences Institute**, 2019.

UNITY TECHNOLOGIES. **GPU instancing**. Disponível em: <https://docs.unity3d.com/Manual/GPUInstancing.html?_ga=2.103728704.1904411092.1638212088-1028174770.1636403299>. Acesso em: 07 dez. 2021.

UNITY TECHNOLOGIES. **Optimizing graphics performance**. Disponível em: <<https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>>. Acesso em: 07 dez. 2021.

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.

WRONSKI, S. **Snow Covered Shader**. Disponível em: <<https://github.com/WorldOfZero/UnityVisualizations/tree/master/SnowTopped>>. Acesso em: 08 dez. 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 17 os trilhos se aproximam até que convergem no horizonte (um ponto no infinito no espaço Euclidiano) distante do observador (AHN, 2021).

Figura 17 – 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, 2021).

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, 2021).

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

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

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

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, 2021).

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

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

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

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

```

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

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  
7 // cor do contorno  
8 uniform vec4 outline_color : hint_color = vec4(1.0);  
9  
10 void vertex() {  
11     // desloca cada vertice na direcao de sua normal vezes  
12     // o fator de espessura  
13     VERTEX += NORMAL * thickness;  
14 }  
15  
16 void fragment() {  
17     /* aplica a cor em cada \textit{pixel} */  
18     ALBEDO = outline_color.rgb;  
19     if(outline_color.a < 1.0) ALPHA = outline_color.a;  
20 }
```

ANEXO B – Código-fonte do *shader* de musgo

Código-fonte 3 – *Shader* de efeito de musgo na Unity

```

1 Shader "Custom/Musgo" {
2     Properties {
3         _Color ("Color", Color) = (1,1,1,1)
4         _MainTex ("Albedo (RGB)", 2D) = "white" {}
5         _Metallic ("Metallic", 2D) = "white" {}
6         _Normal ("Normal", 2D) = "white" {}
7         _Height ("Height", 2D) = "white" {}
8         _Occlusion ("Occlusion", 2D) = "white" {}
9         _MusgoTex ("Musgo Albedo (RGB)", 2D) = "white" {}
10        _MusgoDirection ("Musgo Direction", Vector) = (0,
11                                1, 0, 0)
12        _MusgoAmount ("Musgo Amount", Range(0,1)) = 0.1
13        _Tint("Tint", Color) = (0,1,0,1)
14    }
15    SubShader {
16        Tags { "RenderType"="Opaque" }
17        CGPROGRAM
18        #pragma surface surf Standard fullforwardshadows
19        #pragma target 3.0
20
21        sampler2D _MainTex;
22        sampler2D _MusgoTex;
23        sampler2D _Metallic;
24        sampler2D _Normal;
25        sampler2D _Height;
26        sampler2D _Occlusion;
27
28        struct Input {
29            float2 uv_MainTex;

```

```

30         float3 worldNormal;
31
32     INTERNAL_DATA
33
34     } ;
35
36
37     fixed4 _Color;
38
39     float4 _MusgoDirection;
40
41     float _MusgoAmount;
42
43     float4 _Tint;
44
45
46     void surf (Input IN, inout SurfaceOutputStandard o)
47 {
48
49     fixed3 normal = UnpackNormal(tex2D (_Normal,
50                                         IN.uv_MainTex));
51
52     o.Normal = normal.rgb;
53
54     float3 worldNormal = WorldNormalVector(IN,
55                                         o.Normal);
56
57     float MusgoCoverage = (dot(worldNormal,
58                                 _MusgoDirection) + 1) / 2;
59
60     MusgoCoverage = 1 - MusgoCoverage;
61
62     float MusgoStrength = MusgoCoverage
63
64         < _MusgoAmount;
65
66     fixed4 c = tex2D (_MainTex, IN.uv_MainTex) *
67
68         _Color;
69
70     fixed4 MusgoColor = tex2D (_MusgoTex,
71
72         IN.uv_MainTex) * _Color * _Tint;
73
74     fixed4 metallic = tex2D (_Metallic,
75
76         IN.uv_MainTex);
77
78     fixed4 occlusion = tex2D (_Occlusion,
79
80         IN.uv_MainTex);
81
82     o.Albedo = c * (1 - MusgoStrength) + MusgoColor
83
84         * MusgoStrength;
85
86     o.Metallic = metallic.r;
87
88     o.Occlusion = occlusion.r;
89
90 }
```

```
63         o.Alpha = c.a;
64     } ENDCG
65 } FallBack "Diffuse"
66 }
```

ANEXO C – Código-fonte do *shader* de água

Código-fonte 4 – *Shader* de efeito de água na Unity

```

1 Shader "Water" {
2     Properties{
3         [Header(Features)]
4         [Toggle(USE_DISPLACEMENT)]
5         _UseDisplacement("Displacement", Float) = 0
6         [Toggle(USE_MEAN_SKY_RADIANCEx)]
7         _UseMeanSky("Mean sky radiancEx", Float) = 0
8         [Toggle(USE_FILTERING)]
9         _UseFiltering("Filtering", Float) = 0
10        [Toggle(USE_FOAM)]
11        _UseFoam("Foam", Float) = 0
12        [Toggle(BLINN_PHONG)]
13        _UsePhong("Blinn Phong", Float) = 0
14        [Header(Basic settings)]
15        _AmbientDensity("Ambient Intensity",
16                        Range(0, 1)) = 0.15
17        _DiffuseDensity("Diffuse Intensity",
18                        Range(0, 1)) = 0.1
19        _SurfaceColor("Surface Color",
20                      Color) = (0.0078, 0.5176, 0.7)
21        _ShoreColor("Shore Tint Color",
22                      Color) = (0.0078, 0.5176, 0.7)
23        _DepthColor("Deep Color",
24                      Color) = (0.0039, 0.00196, 0.145)
25        [NoScaleOffset]
26        _SkyTexture("Sky Texture", Cube) = "white" {}
27        [NoScaleOffset]
28        _NormalTexture("Normal Texture", 2D) = "white" {}
29        _NormalIntensity("Normal Intensity",

```

```

30      Range(0, 1)) = 0.5
31      _TextureTiling("Texture Tiling",
32          Float) = 1
33      _WindDirection("Wind Direction", Vector) = (3,5,0)
34      [Header(Displacement settings)]
35      [NoScaleOffset]
36      _HeightTexture("Height Texture",
37          2D) = "white" {}
38      _HeightIntensity("Height Intensity",
39          Range(0, 1)) = 0.5
40      _WaveTiling("Wave Tiling", Float) = 1
41      _WaveAmplitudeFactor("Wave Amplitude Factor",
42          Float) = 1.0
43      _WaveSteepness("Wave Steepness", Range(0, 1)) = 0.5
44      _WaveAmplitude("Waves Amplitude",
45          Vector) = (0.05, 0.1, 0.2, 0.3)
46      _WavesIntensity("Waves Intensity",
47          Vector) = (3, 2, 2, 10)
48      _WavesNoise("Waves Noise",
49          Vector) = (0.05, 0.15, 0.03, 0.05)
50      [Header(Refraction settings)]
51      _WaterClarity("Water Clarity", Range(0, 3)) = 0.75
52      _WaterTransparency("Water Transparency",
53          Range(0, 30)) = 10.0
54      _HorizontalExtinction("Horizontal Extinction",
55          Vector) = (3.0, 10.0, 12.0)
56      _RefractionValues("Refraction/Reflection",
57          Vector) = (0.3, 0.01, 1.0)
58      _RefractionScale("Refraction Scale",
59          Range(0, 0.03)) = 0.005
60      [Header(Reflection settings)]
61      _Shininess("Shininess", Range(0, 3)) = 0.5
62      _SpecularValues("Specular Intensity",

```

```

63      Vector) = (12, 768, 0.15)
64      _Distortion("Distortion", Range(0, 0.15)) = 0.05
65      _RadianceFactor("Radiance Factor",
66          Range(0, 1.0)) = 1.0
67      [HideInInspector] _ReflectionTexture(
68          "Reflection Texture", 2D) = "white" {}
69      [Header(Foam settings)]
70      [NoScaleOffset] _FoamTexture("Foam Texture",
71          2D) = "white" {}
72      [NoScaleOffset] _ShoreTexture("Shore Texture",
73          2D) = "white" {}
74      _FoamTiling("Foam Tiling",
75          Vector) = (2.0, 0.5, 0.0)
76      _FoamRanges("Foam Ranges",
77          Vector) = (2.0, 3.0, 100.0)
78      _FoamNoise("Foam Noise", Vector) =
79          (0.1, 0.3, 0.1, 0.3)
80      _FoamSpeed("Foam Speed", Float) = 10
81      _FoamIntensity("Foam Intensity", Range(0, 1)) = 0.5
82      _ShoreFade("Shore Fade", Range(0.1, 3)) = 0.3
83  }
84  SubShader{
85      Tags{ "IgnoreProjector" = "True"
86          "Queue" = "Transparent"
87          "RenderType" = "Transparent"
88          "LightMode" = "ForwardBase" }
89      GrabPass{ "_RefractionTexture" }
90      Pass{
91          Name "Base"
92          Blend SrcAlpha OneMinusSrcAlpha
93          Cull False
94          ZWrite True
95      }

```

```
96      CGPROGRAM
97
98      #pragma vertex vert
99
100     #pragma fragment frag
101
102     #include "UnityCG.cginc"
103
104     #include "conversion.cginc"
105
106     #include "snoise.cginc"
107
108     #include "bicubic.cginc"
109
110     #include "normals.cginc"
111
112     #include "displacement.cginc"
113
114     #include "meansky.cginc"
115
116     #include "radiance.cginc"
117
118     #include "depth.cginc"
119
120     #include "foam.cginc"
121
122     #pragma multi_compile_fog
123
124     #pragma shader_feature USE_DISPLACEMENT
125
126     #pragma shader_feature USE_MEAN_SKY_RADIANCE
127
128     #pragma shader_feature USE_FILTERING
129
130     #pragma shader_feature USE_FOAM
131
132     #pragma shader_feature BLINN_PHONG
133
134     #pragma exclude_renderers d3d11_9x
135
136     #pragma target 3.0
137
138
139     uniform sampler2D _CameraDepthTexture;
140
141     uniform sampler2D _HeightTexture;
142
143     uniform sampler2D _NormalTexture;
144
145     uniform sampler2D _FoamTexture;
146
147     uniform sampler2D _ShoreTexture;
148
149     uniform sampler2D _ReflectionTexture;
150
151     uniform float4 _ReflectionTexture_TexelSize;
152
153     uniform samplerCUBE _SkyTexture;
154
155     uniform sampler2D _RefractionTexture;
156
157     uniform float4x4 _ViewProjectInverse;
158
159     uniform float4 _TimeEditor;
```

```
129     uniform float _AmbientDensity;
130     uniform float _DiffuseDensity;
131     uniform float _HeightIntensity;
132     uniform float _NormalIntensity;
133     uniform float _TextureTiling;
134     uniform float4 _LightColor0;
135     uniform float3 _SurfaceColor;
136     uniform float3 _ShoreColor;
137     uniform float3 _DepthColor;
138     uniform float2 _WindDirection;
139     uniform float _WaveTiling;
140     uniform float _WaveSteepness;
141     uniform float _WaveAmplitudeFactor;
142     uniform float4 _WaveAmplitude;
143     uniform float4 _WavesIntensity;
144     uniform float4 _WavesNoise;
145     uniform float _WaterClarity;
146     uniform float _WaterTransparency;
147     uniform float3 _HorizontalExtinction;
148     uniform float _Shininess;
149     uniform float3 _SpecularValues;
150     uniform float2 _RefractionValues;
151     uniform float _RefractionScale;
152     uniform float _RadianceFactor;
153     uniform float _Distortion;
154     uniform float3 _FoamRanges;
155     uniform float4 _FoamNoise;
156     uniform float2 _FoamTiling;
157     uniform float _FoamSpeed;
158     uniform float _FoamIntensity;
159     uniform float _ShoreFade;
160
161     struct VertexInput {
```

```

162         float4 vertex : POSITION;
163     };
164
165     struct VertexOutput {
166
166         float4 pos : SV_POSITION;
167
168         float2 uv : TEXCOORD0;
169
170         float3 normal : TEXCOORD1;
171
172         float3 tangent : TEXCOORD2;
173
174         float3 bitangent : TEXCOORD3;
175
176         float3 worldPos : TEXCOORD4;
177
178         float4 projPos : TEXCOORD5;
179
180         float timer : TEXCOORD6;
181
182         float4 wind : TEXCOORD7;
183
184         UNITY_FOG_COORDS(8)
185
186     };
187
188
189     VertexOutput vert(VertexInput v) {
190
191         VertexOutput o = (VertexOutput)0;
192
193         float2 windDir = _WindDirection;
194
195         float windSpeed = length(_WindDirection);
196
197         windDir /= windSpeed;
198
199         float timer = (_Time + _TimeEditor) *
200
201             windSpeed * 10;
202
203         float4 modelPos = v.vertex;
204
205         float3 worldPos = mul(unity_ObjectToWorld,
206
207             float4(modelPos.xyz, 1));
208
209         half3 normal = half3(0, 1, 0);
210
211 #ifdef USE_DISPLACEMENT
212
213         float cameraDistance = length(
214
215             _WorldSpaceCameraPos.xyz - worldPos);
216
217         float2 noise = GetNoise(worldPos.xz, timer *
218
219             windDir * 0.5);
220
221         half3 tangent;
222
223         float4 waveSettings = float4(windDir,
224
225             noise.r, noise.g, noise.b);
226
227         o.worldPos = worldPos;
228
229         o.normal = normal;
230
231         o.tangent = tangent;
232
233         o.bitangent = cross(normal, tangent);
234
235         o.projPos = mul(UNITY_MATRIX_VP, projPos);
236
237         o.wind = wind;
238
239         o.timer = timer;
240
241         o.fogCoord = fogCoord;
242
243         return o;
244     }

```

```

195         _WaveSteepness, _WaveTiling);
196
197         float4 waveAmplitudes = _WaveAmplitude *
198             _WaveAmplitudeFactor;
199
200         worldPos = ComputeDisplacement(worldPos,
201             cameraDistance, noise, timer, waveSettings,
202             waveAmplitudes, _WavesIntensity,
203             _WavesNoise, normal, tangent);
204
205         float heightIntensity = _HeightIntensity * (1.0
206             - cameraDistance / 100.0) * _WaveAmplitude;
207
208         float2 texCoord = worldPos.xz * 0.05 *
209             _TextureTiling;
210
211         if (heightIntensity > 0.02) {
212
213             float height = ComputeNoiseHeight(
214                 _HeightTexture, _WavesIntensity,
215                 _WavesNoise, texCoord, noise, timer);
216
217             worldPos.y += height * heightIntensity;
218
219         }
220
221         modelPos = mul(unity_WorldToObject,
222             float4(worldPos, 1));
223
224         o.tangent = tangent;
225
226         o.bitangent = cross(normal, tangent);
227
#endif
228
229         float2 uv = worldPos.xz;
230
231         o.timer = timer;
232
233         o.wind.xy = windDir;
234
235         o.wind.zw = windDir * timer;
236
237         o.uv = uv * 0.05 * _TextureTiling;
238
239         o.pos = UnityObjectToClipPos(modelPos);
240
241         o.worldPos = worldPos;
242
243         o.projPos = ComputeScreenPos(o.pos);
244
245         o.normal = normal;
246
247         UNITY_TRANSFER_FOG(o, o.pos);
248
249         return o;

```

```

228 }
229
230 float4 frag(VertexOutput fs_in, float facing :
231     VFACE) : COLOR {
232     float timer = fs_in.timer;
233     float2 windDir = fs_in.wind.xy;
234     float2 timedWindDir = fs_in.wind.zw;
235     float2 ndcPos = float2(fs_in.projPos.xy /
236                             fs_in.projPos.w);
237     float3 eyeDir = normalize(
238         _WorldSpaceCameraPos.xyz - fs_in.worldPos);
239     float3 surfacePosition = fs_in.worldPos;
240     half3 lightColor = _LightColor0.rgb;
241 #ifdef USE_DISPLACEMENT
242     half3 normal = ComputeNormal(_NormalTexture,
243                                 surfacePosition.xz, fs_in.uv, fs_in.normal,
244                                 fs_in.tangent, fs_in.bitangent, _WavesNoise
245                                 , _WavesIntensity, timedWindDir);
246 #else
247     half3 normal = ComputeNormal(_NormalTexture,
248                                 surfacePosition.xz, fs_in.uv, fs_in.normal,
249                                 0, 0, _WavesNoise, _WavesIntensity,
250                                 timedWindDir
251 );
252 #endif
253     normal = normalize(lerp(fs_in.normal,
254                             normalize(normal), _NormalIntensity));
255     float depth = tex2Dproj(_CameraDepthTexture,
256                             UNITY_PROJ_COORD(fs_in.projPos.xyww));
257     float3 depthPosition = NdcToWorldPos(
258         _ViewProjectInverse, float3(ndcPos, depth)
259 );
260     float waterDepth = surfacePosition.y -

```

```

261         depthPosition.y;
262
263     float viewWaterDepth = length(surfacePosition -
264
265         depthPosition);
266
267     float2 dudv = ndcPos;
268
269     {
270
271         float refractionScale = _RefractionScale *
272
273             min(waterDepth, 1.0f);
274
275         float2 delta = float2(
276
277             sin(timer + 3.0f * abs(depthPosition.y)
278
279                 ),
280
281             sin(timer + 5.0f * abs(depthPosition.y)
282
283                 ));
284
285         dudv += windDir * delta * refractionScale;
286
287     }
288
289     half3 pureRefractionColor = tex2D(
290
291         _RefractionTexture, dudv).rgb;
292
293     {
294
295         INVERSE_FOG_COLOR(fs_in.fogCoord,
296
297             pureRefractionColor);
298
299     }
300
301     float2 waterTransparency = float2(
302
303         _WaterClarity, _WaterTransparency);
304
305     float2 waterDepthValues = float2(
306
307         waterDepth, viewWaterDepth);
308
309     float shoreRange = max(
310
311         _FoamRanges.x, _FoamRanges.y) * 2.0;
312
313     half3 refractionColor = DepthRefraction(
314
315         waterTransparency, waterDepthValues,
316
317         shoreRange, _HorizontalExtinction,
318
319         pureRefractionColor, _ShoreColor,
320
321         _SurfaceColor, _DepthColor);
322
323     float3 lightDir = normalize(
324
325         _WorldSpaceLightPos0);

```

```

294     half fresnel = FresnelValue(
295         _RefractionValues, normal, eyeDir);
296     half3 specularColor = ReflectedRadiance(
297         _Shininess, _SpecularValues, lightColor,
298         lightDir, eyeDir, normal, fresnel);
299 #ifdef USE_MEAN_SKY_RADIANCE
300     half3 reflectColor = fresnel * MeanSkyRadiance(
301         _SkyTexture, eyeDir, normal) *
302         _RadianceFactor;
303 #else
304     half3 reflectColor = 0;
305 #endif // #ifndef USE_MEAN_SKY_RADIANCE
306     dudv = ndcPos + _Distortion * normal.xz;
307 #ifdef USE_FILTERING
308     reflectColor += tex2DBicubic(
309         _ReflectionTexture,
310         _ReflectionTexture_TexelSize.z,
311         dudv).rgb;
312 #else
313     reflectColor += tex2D(_ReflectionTexture,
314         dudv).rgb;
315 #endif // #ifdef USE_FILTERING
316 #ifdef USE_FOAM
317     float maxAmplitude = max(max(_WaveAmplitude.x,
318         _WaveAmplitude.y), _WaveAmplitude.z);
319     half foam = FoamValue(_ShoreTexture,
320         _FoamTexture, _FoamTiling,
321         _FoamNoise, _FoamSpeed * windDir,
322         _FoamRanges, maxAmplitude,
323         surfacePosition, depthPosition, eyeDir,
324         waterDepth, timedWindDir, timer);
325     foam *= _FoamIntensity;
326 #else

```

```

327         half foam = 0;
328 #endif // #ifdef USE_FOAM
329         half shoreFade = saturate(waterDepth *
330             _ShoreFade);
331         half3 ambientColor =
332             UNITY_LIGHTMODEL_AMBIENT.rgb *
333             _AmbientDensity + saturate(
334                 dot(normal, lightDir)) *
335                 _DiffuseDensity;
336         pureRefractionColor = lerp(pureRefractionColor,
337             reflectColor, fresnel * saturate(waterDepth
338             / (_FoamRanges.x * 0.4)));
339         pureRefractionColor = lerp(pureRefractionColor,
340             _ShoreColor, 0.30 * shoreFade);
341         half3 color = lerp(refractionColor,
342             reflectColor, fresnel);
343         color = saturate(ambientColor + color + max(
344             specularColor, foam * lightColor));
345         color = lerp(pureRefractionColor +
346             specularColor * shoreFade, color, shoreFade);
347         UNITY_APPLY_FOG(fs_in.fogCoord, color);
348 #ifdef DEBUG_NORMALS
349             color.rgb = 0.5 + 2 * ambientColor +
350                 specularColor + clamp(dot(normal,
351                     lightDir), 0, 1) * 0.5;
352 #endif
353             return float4(color, 1.0);
354     }ENDCG}
355 }FallBack "Diffuse"

```