PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DEFERRED RENDERING EM *ENGINE* PRÓPRIA: DESENVOLVIMENTO E OTIMIZAÇÃO

EDGAR DA SILVA RODRIGUES

GOIÂNIA

2025

EDGAR DA SILVA RODRIGUES

DEFERRED RENDERING EM *ENGINE* PRÓPRIA: DESENVOLVIMENTO E OTIMIZAÇÃO

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade de Goiás, como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Orientador:
Prof. Rafael Leal Martins

Banca examinadora:
Prof.ª Dra.ª Solange da Silva
Prof. Me. Fernando Gonçalves Abadia

GOIÂNIA

2025

EDGAR DA SILVA RODRIGUES

DEFERRED RENDERING EM *ENGINE* PRÓPRIA: DESENVOLVIMENTO E OTIMIZAÇÃO

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Ciência da Computação, em: __ /__/__.

_____

Orientador: Prof. Me. Rafael Leal Martins

_____

Prof.ª Dra.ª Solange da Silva

_____

Prof. Me. Fernando Gonçalves Abadia

GOIÂNIA

2025

AGRADECIMENTOS

Primeiramente, expresso minha gratidão à minha mãe, que sempre foi minha maior incentivadora e exemplo de dedicação. Seu apoio foi fundamental para que eu chegasse até aqui. Sem sua compreensão e suporte diário, esta conquista não seria possível.

Registro também meus agradecimentos ao meu professor orientador, Rafael Leal, por ter aceitado me acompanhar na construção deste trabalho. Sua disponibilidade para esclarecer dúvidas e contribuições valiosas foram essenciais para o desenvolvimento desta pesquisa. Agradeço pela paciência, pela confiança em meu trabalho e pelos conhecimentos compartilhados ao longo deste percurso.

Estendo meus agradecimentos aos meus amigos e colegas da faculdade, que de diversas formas contribuíram para esta jornada, seja através de conversas, trocas de conhecimento, apoio emocional ou motivação nos momentos de dificuldade. A convivência com eles tornou o processo acadêmico mais leve, enriquecedor e significativo.

Por fim, agradeço a todos que, direta ou indiretamente, fizeram parte desta etapa. Cada palavra ou gesto de apoio foram essenciais para que este trabalho se concretizasse.

RESUMO

Este trabalho apresenta o desenvolvimento de um renderizador gráfico em tempo real construído com a biblioteca BGFX, explorando conceitos fundamentais da computação gráfica moderna, como *Forward Rendering*, *Deferred Rendering*, *Physically Based Rendering* (*PBR*), *Shadow Mapping* e *Level of Detail* (*LOD*). O projeto teve como objetivo compreender e implementar, de forma progressiva, as principais etapas do pipeline gráfico, desde a exibição do primeiro triângulo até a integração de uma interface gráfica interativa utilizando *Dear ImGui*. Foram abordadas técnicas de otimização, mapeamento de texturas, iluminação dinâmica e pós-processamento, priorizando o equilíbrio entre qualidade visual e desempenho. O processo de implementação reforçou a importância de arquiteturas modulares e escaláveis, além de demonstrar a aplicabilidade prática dos conceitos teóricos estudados na literatura de computação gráfica. Concluiu-se que o resultado obtido foi um renderizador funcional e extensível, capaz de reproduzir cenas tridimensionais com iluminação realista e controle interativo, aproximando-se das estruturas encontradas em motores gráficos profissionais.

Palavras-chave: Computação gráfica; Renderização em tempo real; *Deferred Rendering*; *PBR*; BGFX.

ABSTRACT

This work presents the development of a real-time graphics renderer built with the BGFX library, exploring key concepts of modern computer graphics such as Forward Rendering, Deferred Rendering, Physically Based Rendering (PBR), Shadow Mapping, and Level of Detail (LOD). The project aimed to progressively implement and understand the main stages of the graphics pipeline, from displaying the first triangle to integrating an interactive graphical interface using Dear ImGui. Optimization techniques, texture mapping, dynamic lighting, and post-processing effects were applied to achieve a balance between visual quality and performance. The implementation process highlighted the relevance of modular and scalable architectures, as well as the practical applicability of theoretical concepts from computer graphics literature. In conclusion, the final result was a functional and extensible renderer capable of reproducing three-dimensional scenes with realistic lighting and interactive control, closely resembling the architecture of professional game engines.

Keywords: Computer graphics; Real-time rendering; Deferred Rendering; PBR; BGFX.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS E ABREVIATURAS

CG – Computação Gráfica

API – Application Programming Interface (Interface de Programação de Aplicações)

CPU – Central Processing Unit (Unidade Central de Processamento)

GPU – Graphics Processing Unit (Unidade de Processamento Gráfico)

FPS – Frames per Second (Quadros por Segundo)

G-buffer – Geometry Buffer

PBR – Physically Based Rendering

BGFX – Biblioteca gráfica multiplataforma utilizada no projeto

SDL3 – Simple DirectMedia Layer 3

BX – Biblioteca auxiliar de matemática e utilidades

LOD – Level of Detail

MSAA – Multisample Anti-Aliasing

FXAA – Fast Approximate Anti-Aliasing

SMAA – Subpixel Morphological Anti-Aliasing

TAA – Temporal Anti-Aliasing

VRS – Variable Rate Shading

UI – User Interface

FOV – Field of View

SUMÁRIO

1. INTRODUÇÃO

A indústria de jogos digitais evoluiu a ponto de exigir sistemas capazes de gerar imagens realistas em tempo real, conciliando qualidade visual e alto desempenho. Em essência, um jogo é composto por uma sequência de quadros (*Frames*) exibidos a taxas elevadas normalmente de 30, 60 ou até mais vezes por segundo criando a ilusão de movimento contínuo, assim como no cinema. A diferença fundamental é que, enquanto o cinema exibe quadros previamente gravados, nos jogos cada *Frame* deve ser calculado instantaneamente, refletindo a interação do jogador com o ambiente virtual (Akenine-Möller; Haines; Hoffman, 2018).

A tarefa de gerar um *Frame* é conduzida pela *Game Engine, Software* que integra múltiplos subsistemas como física, áudio, inteligência artificial e, sobretudo, renderização o processo que converte a descrição matemática de um mundo tridimensional em *pixels* exibidos na tela (Gregory, 2018). A renderização é reconhecida como a etapa mais intensiva em termos de processamento, explorando ao máximo os recursos da *GPU* e definindo diretamente a qualidade visual da experiência interativa.

Historicamente, o método predominante foi o *Forward Rendering*, no qual cada objeto da cena é processado integralmente para cada fonte de luz. Essa abordagem é adequada em cenários com poucas luzes, mas apresenta sérios gargalos de desempenho quando se busca maior realismo visual, como em ambientes internos complexos ou cenas noturnas com múltiplos pontos de iluminação (Ferko, 2012; Kim, 2016).

Como alternativa, as *Game Engines* modernas passaram a adotar o *Deferred Rendering*. Nesse modelo, a geometria da cena é processada apenas uma vez e armazenada em estruturas auxiliares denominadas *G-Buffers*. Na etapa seguinte, cada luz lê os dados do *G-Buffer* e aplica apenas o cálculo de iluminação que lhe compete, combinando os resultados até compor a imagem final (Petrescu et al., 2016). Essa técnica possibilita a utilização de dezenas ou centenas de luzes com custo computacional controlado, embora imponha novos desafios, como maior consumo de memória, dificuldades na implementação de transparência e limitações

na aplicação de técnicas de antisserrilhamento (*Anti-Aliasing*) convencionais (Fridvalszky; Tóth, 2020).

O desenvolvimento de uma *Engine* própria, em vez da utilização de ferramentas prontas como Unity ou Unreal, pode oferecer vantagens significativas. A liberdade para modificar o código permite maior controle sobre desempenho e flexibilidade na implementação de soluções específicas, além de representar uma oportunidade de aprendizado profundo sobre o funcionamento do *Pipeline* gráfico (Pharr; Jakob; Humphreys, 2023). Nesse contexto, a construção de um *Pipeline Deferred* em uma *Engine* própria contribui não apenas para aplicações práticas, mas também para o avanço acadêmico na área de computação gráfica.

Justifica-se o desenvolvimento de uma *Game Engine* própria por três razões principais que se complementam. Em primeiro lugar, a independência frente a soluções comerciais evita riscos associados a mudanças unilaterais de políticas por parte de grandes empresas, como ocorreu em 2023 quando a Unity anunciou taxas por instalação, provocando protestos generalizados e posterior recuo da medida (Stuart, 2023). Além disso, a criação de um renderizador próprio oferece flexibilidade e controle total sobre o código-fonte, permitindo ajustes finos em subsistemas críticos, como o de renderização, de modo a otimizar o desempenho e implementar recursos específicos como *Physically Based Rendering* e técnicas avançadas de iluminação de forma mais eficiente do que em motores genéricos (Chebanyuk; Mushynskyi, 2021). Finalmente, sob a perspectiva acadêmica, construir uma *Engine* é um exercício que amplia significativamente a compreensão de sistemas em tempo real, *Pipelines* gráficos e arquitetura de *Software*, conforme defendido na literatura especializada (Gregory, 2018; Petrescu et al., 2016). Assim, a iniciativa alia confiabilidade, flexibilidade e valor formativo, consolidando-se como um projeto de relevância tanto prática quanto científica.

Este trabalho concentra-se exclusivamente no *Pipeline* de *Deferred Rendering*. A matemática de cada *Shader*, assim como aspectos arquiteturais de *CPUs* e *GPUs*, será abordada de forma simplificada, visando justificar conceitos e escolhas técnicas. Assim, este projeto busca responder à seguinte questão de pesquisa:

Seria possível construir, em uma *Engine* relativamente simples, um *Pipeline* de *Deferred Rendering* capaz de exibir cenas interativas cheias de luzes mantendo desempenho aceitável em *Hardware* de consumo?

Para tanto, serão revisados fundamentos de computação gráfica, projetado um esquema modular de passagens de geometria, iluminação e pós-processamento, além da implementação prática em C++ com o uso de bibliotecas que abstraem APIs gráficas modernas. O desempenho será avaliado em cenários de teste com diferentes níveis de complexidade, de forma a analisar tanto a qualidade visual quanto a eficiência computacional.

O objetivo geral deste trabalho é implementar uma *Engine* gráfica com foco na técnica de renderização denominada *Deferred Rendering*, de modo que possa servir como base para projetos pessoais e acadêmicos, contribuindo tanto para a compreensão prática do *Pipeline* gráfico quanto para a otimização de desempenho em ambientes interativos.

Os objetivos específicos deste trabalho são:

- Integrar bibliotecas (BGFX, SDL3, BX) em C++ para gerenciamento de janelas, importação de modelos 3D e abstração de *APIs* gráficas, facilitando o uso de OpenGL, Direct*3D*, Vulkan, entre outras.
- Implementar o funcionamento de *Deferred Rendering*, incluindo a criação do *G-Buffer* (armazenamento de informações dos objetos), a realização do *Lighting Pass* (etapa de cálculo de iluminação) e a aplicação de efeitos básicos de pós-processamento, como correção de cor e *Bloom*.
- Aplicar técnicas de *Physically Based Rendering* (PBR) para aumentar o realismo na interação entre luz e materiais.
- Avaliar o desempenho da *Engine* em cenas com diferentes quantidades de luzes dinâmicas, incluindo fontes móveis e projeção de sombras em tempo real.

Espera-se que este trabalho possa contribuir com:

- A documentação de etapas do desenvolvimento da parte gráfica de *engines* modernas.
- Uma base sólida para um futuro *software* comercial para o desenvolvimento de jogos.

O presente Trabalho de Conclusão de Curso está estruturado de maneira a proporcionar uma compreensão progressiva do tema e das etapas de desenvolvimento do projeto. O Capítulo 2 apresenta o referencial teórico, abordando os conceitos fundamentais de *Game Engines* e de renderização gráfica, que servem

de base para a compreensão técnica do estudo. O Capítulo 3 trata dos principais tipos de renderização empregados na computação gráfica, com ênfase nas abordagens *Forward*, *Forward+* e *Deferred Rendering*, discutindo as vantagens e limitações de cada uma. O Capítulo 4 descreve a arquitetura do renderizador desenvolvido, explicando o funcionamento do G-*Buffer*, do *Lighting Pass* e das etapas de composição da imagem. No Capítulo 5, são apresentadas técnicas avançadas e otimizações aplicadas à implementação, como o uso de *Physically Based Rendering* (*PBR*), *Culling* e *Anti-Aliasing*. O Capítulo 6 expõe o desenvolvimento prático do projeto, detalhando a integração das bibliotecas utilizadas e as etapas de construção da *Engine*. Em seguida, o Capítulo 7 apresenta os resultados obtidos e as análises de desempenho realizadas. Por fim, o Capítulo 8 reúne as considerações finais, destacando as contribuições do trabalho, as dificuldades enfrentadas e as possibilidades de aprimoramento para versões futuras da *Engine*.

2. PROCEDIMENTOS METODOLÓGICOS

Este trabalho, quanto à natureza, é um resumo de assunto, e quanto aos procedimentos técnicos, é uma pesquisa bibliográfica e experimental desenvolvida em um ambiente real de programação de *Software*. O processo ocorreu de forma iterativa, permitindo que o protótipo da *Engine* fosse aprimorado continuamente enquanto dados de desempenho e qualidade visual eram coletados e analisados. O uso dessa metodologia prática possibilitou acompanhar a evolução do projeto de maneira controlada, garantindo que cada etapa servisse como base para decisões técnicas nas fases subsequentes.

Na etapa de planejamento, optou-se pela linguagem C++ devido à sua alta performance em operações de renderização em tempo real e pela ampla utilização na indústria de jogos. Para compor o ecossistema de desenvolvimento foram escolhidas bibliotecas consolidadas: o BGFX, responsável pela abstração de diferentes APIs gráficas como OpenGL, Direct*3D* e Vulkan; o SDL3, voltado para o gerenciamento de janelas e dispositivos de entrada; e o BX, utilizado para operações matemáticas e funções utilitárias. A seleção dessas ferramentas foi

guiada pela documentação consistente, portabilidade e compatibilidade com os objetivos de modularidade e desempenho do projeto. O cronograma foi estruturado em ciclos curtos de implementação e avaliação, o que permitiu reagir rapidamente a gargalos de desempenho identificados durante os testes.

O desenvolvimento seguiu um processo incremental. Inicialmente, foi construído um protótipo mínimo, capaz de exibir apenas um objeto estático sem iluminação avançada, com o objetivo de validar o *pipeline* básico. A cada iteração, novos recursos foram incorporados, como a implementação do G-*Buffer*, o passe de iluminação e efeitos de pós-processamento. Em paralelo, cada incremento era avaliado em *Hardware* de consumo, com o registro de resultados de modo a garantir rastreabilidade e documentação contínua das escolhas técnicas.

Os procedimentos de coleta de dados incluíram tanto métricas quantitativas quanto qualitativas. Entre as métricas numéricas, foi monitorado o *Framerate* (*FPS*) em cenas de teste que variavam a quantidade de luzes dinâmicas, enquanto a análise qualitativa consistiu na avaliação visual de materiais renderizados com *PBR*, da integridade das sombras e da ausência de artefatos gráficos. Todas as observações foram registradas para orientar ajustes de desempenho e qualidade, assegurando que os resultados pudessem ser reproduzidos em diferentes execuções, já que os cenários de teste foram armazenados no repositório do projeto.

Por fim, a gestão de versões foi realizada com a ferramenta Git, utilizada para controlar o código-fonte, modelos 3D e configurações de cena. Esse recurso também permitiu a execução de *rollbacks* sempre que uma modificação comprometia o desempenho ou gerava inconsistências visuais, assegurando assim a confiabilidade e a continuidade do desenvolvimento da *Engine*.

3. REFERENCIAL TEÓRICO

Este capítulo consolida os conceitos centrais de desenvolvimento de jogos e computação gráfica que fundamentam o projeto, buscando alinhar terminologia,

escopo técnico e posicionar a implementação em relação às práticas adotadas pela indústria e pela pesquisa acadêmica.

3.1. *Game Engine*

Do ponto de vista da engenharia de *Software*, uma *Game Engine* pode ser compreendida como um arcabouço de *Middleware* que integra, sob uma mesma arquitetura, diversos subsistemas responsáveis pela renderização, áudio, física, animação, inteligência artificial, rede, interface e ferramentas de autoria. A principal função dessa integração é abstrair as particularidades do *Hardware* subjacente, impor padrões de comunicação entre os módulos e oferecer um ambiente de desenvolvimento mais acessível, com editores visuais, *visual scripting* e sistemas de *hot-reload*, permitindo que os profissionais de *design* de jogos concentrem seus esforços na experiência interativa e não na infraestrutura técnica (Gregory, 2018). Em analogia ao setor fabril, a *Engine* funciona como uma linha de montagem: modelos tridimensionais, texturas, sons e *scripts* atravessam estágios de importação, otimização e compilação até resultarem em um executável multiplataforma.

A literatura especializada distingue dois grandes domínios que compõem uma *Engine*: o *Runtime*, que corresponde ao código executado em tempo real no dispositivo-alvo (PCs, consoles, dispositivos móveis ou plataformas de realidade virtual), e as ferramentas de autoria e *Pipeline* de conteúdo, que englobam editores, processadores de *Assets*, *Profiler*s e depuradores voltados ao ambiente de produção (Akenine-Möller; Haines; Hoffman, 2018). *Engines* comerciais como Unity, Unreal e Godot expandiram esse ecossistema ao adicionar serviços integrados como lojas de plugins, sistemas de *Cloud Build* e telemetria, consolidando-se como plataformas completas que extrapolam a função de motor gráfico (UNITY TECHNOLOGIES, 2021; EPIC GAMES, 2021).

Entre as responsabilidades essenciais de uma *Engine*, destaca-se em primeiro lugar o gerenciamento de recursos, cuja função é centralizar o carregamento, o cache e o descarte de elementos como texturas, malhas, sons e *shaders*. Esse gerenciamento é crucial para manter o consumo de memória dentro dos limites do *hardware* disponível e evitar a duplicação de dados, o que impacta

diretamente na eficiência e no desempenho do jogo (Gregory, 2018). Outra responsabilidade central é a renderização em tempo real, processo no qual vértices, índices e estados de material são transformados em pixels por meio do *Pipeline* gráfico da *GPU*. Nas *APIs* modernas, como Direct*3D* 11/12, Vulkan, Metal e OpenGL, esse fluxo contempla estágios como montagem de vértices, sombreamento programável, rasterização, testes de profundidade e fusão final (*Output-Merger*). A própria documentação do Direct*3D* 11 descreve esse *Pipeline* como composto por módulos programáveis, como *Vertex Shader* e *Pixel Shader*, e etapas fixas, como o *rasterizer*, que podem ser adaptadas ou ignoradas conforme as necessidades do projeto (MICROSOFT, 2022).

Por fim, o gerenciamento de entrada representa outro subsistema vital, responsável por capturar e normalizar sinais de dispositivos como teclado, mouse, gamepads, sensores inerciais e interfaces de toque. Esse processo é fundamental para garantir responsividade e consistência na jogabilidade. Bibliotecas de baixo nível, como o *SDL3*, fornecem camadas unificadas que traduzem eventos do sistema operacional para estruturas internas da *Engine*, o que permite desacoplar dispositivos lógicos dos comandos de jogo por meio de action mapping, aplicar filtros contra ruídos em sinais analógicos e oferecer suporte a múltiplos controladores simultâneos, possibilitando experiências como *Split-screen* local e *Cross-play* entre plataformas (SDL, 2024).

Assim, a *Game Engine* deve ser compreendida como muito mais que um simples motor gráfico: trata-se de uma plataforma multifacetada que unifica aspectos técnicos, criativos e de otimização, servindo como alicerce tanto para a indústria quanto para a pesquisa acadêmica em computação gráfica e desenvolvimento de jogos digitais.

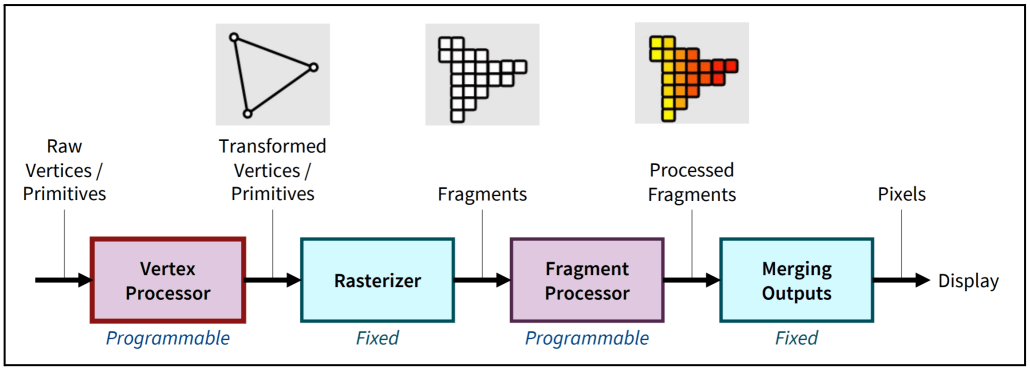
3.2. *Pipeline* de renderização em tempo real

O *Graphics Pipeline* é a cadeia de estágios pela qual a *GPU* converte descrições geométricas tridimensionais, vértices, texturas, parâmetros de material e dados de iluminação em pixels finais gravados no *Frame-Buffer*. Esse processo, que se repete dezenas ou até centenas de vezes por segundo, sustenta a ilusão de movimento contínuo nos jogos digitais. Sua eficiência decorre da execução

massivamente paralela de *Shaders* em milhares de núcleos de processamento, capazes de mascarar latências de memória por meio de técnicas de comutação de contexto em *Hardware*. Segundo Akenine-Möller, Haines e Hoffman (2018), trata-se da espinha dorsal da computação gráfica em tempo real, e seu domínio é essencial tanto para o desenvolvimento de motores gráficos quanto para a compreensão das limitações e potencialidades do *Hardware*.

A documentação oficial do Direct*3D*, referência consolidada na indústria, resume esse fluxo em cinco macropassos lógicos: entrada de dados, processamento de vértices, rasterização, processamento de fragmentos e composição final (MICROSOFT, 2024). Cada etapa é configurável por estados das *APIs* gráficas modernas (Direct*3D*, Vulkan, OpenGL, Metal), reforçando a flexibilidade desse modelo (KHRONOS GROUP, 2025). A Figura 1 ilustra esse processo de forma esquemática, mostrando a sequência pela qual a *GPU* transforma dados geométricos em *pixels* exibidos na tela.

Figura 1 - Estrutura simplificada do Pipeline gráfico em tempo real.



Fonte: leeyngdo

O primeiro estágio, entrada de dados (*Input Assembly*), é responsável por transferir informações da memória principal para a *GPU*. Nessa fase, dados de vértices e índices são organizados em fluxos que descrevem atributos como posição, normais, cores, coordenadas de textura e tangentes. O uso de *index Buffers* permite reaproveitar vértices redundantes em modelos complexos, otimizando o tráfego de memória. Além disso, técnicas como *Instancing* viabilizam o reuso eficiente de geometrias, recurso amplamente empregado em *Engines*

modernas para lidar com múltiplas instâncias de objetos em cena (KHRONOS GROUP, 2025).

O segundo estágio, o processamento de vértices, ocorre no *Vertex Shader*, que aplica transformações de espaço aos vértices utilizando o conjunto de matrizes *Model-View-Projection* (*MVP*). Essa etapa também é responsável por transportar variáveis intermediárias (como coordenadas de textura e vetores *TBN*) para os estágios posteriores. Além disso, pode incluir cálculos de iluminação simplificada em vértices técnica útil em contextos de *Hardware* limitado, como jogos móveis e a geração de saídas para sistemas auxiliares, como partículas ou *Level of Detail* dinâmico (Akenine-Möller et al., 2018; MICROSOFT, 2024). O padrão Vulkan 1.3 define restrições práticas como número máximo de bindings e stride, garantindo portabilidade entre diferentes classes de *GPU* (KHRONOS GROUP, 2025).

Na sequência, a rasterização converte primitivas geométricas em fragmentos. O processo envolve interpolação baricêntrica de atributos e a criação de máscaras de cobertura sub-pixel que determinam a contribuição de cada triângulo sobre os *pixels* da tela. Essa etapa é fundamental para técnicas de suavização, como o *Multisample Anti-Aliasing* (MSAA), que reduz serrilhados com impacto moderado no desempenho (NVIDIA, 2023). Além disso, o rasterizador suporta recursos avançados como *Variable Rate Shading* (*VRS*), que permite variar a taxa de sombreamento por região, equilibrando qualidade e custo computacional. Em dispositivos móveis, arquiteturas *tile-based* dividem a tela em blocos menores, minimizando largura de banda e consumo energético (ARM, 2023).

O quarto estágio é o processamento de fragmentos (*pixel shading*), onde cada *fragmento* gerado na rasterização é tratado pela *GPU*. Nesse ponto, são avaliados os atributos interpolados (posição, normais, coordenadas) juntamente com dados de materiais e luzes. O paradigma dominante é o *Physically Based Rendering* (*PBR*), baseado em modelos *BRDF* que descrevem a interação entre luz e superfícies de forma fisicamente plausível (Pharr; Jakob; Humphreys, 2023). Essa etapa também inclui o cálculo de sombras em tempo real, via *Shadow Maps* ou algoritmos híbridos com *Ray Tracing*, e reflexões por meio de técnicas como *Screen-space Reflections*. Mecanismos como o *early-Z* e o *Stencil Test* garantem que fragmentos ocultos sejam descartados precocemente, evitando desperdício de processamento (MICROSOFT, 2023).

Finalmente, no estágio de composição e pós-processamento, também

chamado *Output-Merger*, a *GPU* combina os fragmentos válidos com os valores já presentes nos *Render Target*s. Operações de *Depth Test* e *Stencil Test* asseguram a correta sobreposição dos elementos da cena, enquanto operações de *Blending* mesclam cores e efeitos (NVIDIA, 2023). Essa fase é igualmente responsável por organizar a cadeia de pós-processamento em tela cheia, onde são aplicadas técnicas como *Temporal Anti-Aliasing* (*TAA*), *Bloom*, *Motion Blur*, *Tone Mapping* e *Color Grading*. Esses efeitos, embora não façam parte da formação geométrica da cena, são determinantes para a qualidade perceptual da imagem final (EPIC GAMES, 2021). Após o processamento, o quadro é transferido ao *Swap-chain* para exibição, coordenado por tecnologias como G-Sync e FreeSync, que asseguram maior fluidez visual e reduzem latência (NVIDIA, 2023).

Portanto, o *Pipeline* de renderização em tempo real constitui uma sequência estruturada de estágios fixos e programáveis, cuja função é transformar descrições matemáticas de uma cena em imagens interativas de alta fidelidade. A literatura especializada aponta que sua compreensão é indispensável não apenas para a otimização de motores gráficos, mas também para o avanço das técnicas de iluminação, sombreamento e pós-processamento na computação gráfica moderna (Akenine-Möller et al., 2018; Pharr; Jakob; Humphreys, 2023).

3.3. Tipos de Renderização

Os métodos de renderização em tempo real têm como objetivo organizar o cálculo de iluminação e sombreamento dentro do *Graphics Pipeline*, equilibrando qualidade visual e desempenho. A escolha da técnica impacta diretamente a escalabilidade da *Engine*, sobretudo em cenários com grande número de luzes dinâmicas. A literatura especializada classifica três abordagens principais: *Forward Rendering*, *Forward+ Rendering* e *Deferred Rendering*, cada qual com *trade-offs* próprios em termos de custo computacional e compatibilidade com efeitos visuais avançados (Pharr; Jakob; Humphreys, 2023).

3.3.1. Forward Rendering

O *Forward Rendering* é o modelo mais tradicional e direto de renderização. Nele, cada objeto da cena é processado em um único *pass*, no qual a *GPU* avalia materiais, texturas e calcula a contribuição de todas as luzes que afetam o objeto naquele momento. Trata-se de uma técnica conceitualmente simples, historicamente associada às primeiras gerações de *Engines* comerciais como id Tech e Unreal *Engine* 2 (Gregory, 2018).

Apesar da simplicidade, o custo computacional cresce rapidamente à medida que o número de luzes aumenta, pois cada pixel precisa processar múltiplas equações de iluminação. Isso torna a técnica pouco escalável para jogos modernos que demandam dezenas ou centenas de fontes de luz dinâmicas. Ainda assim, o *Forward Rendering* permanece atraente em contextos onde a cena possui poucas luzes, ou quando efeitos como transparência e *MSAA* são prioritários, já que sua integração é mais natural nesse modelo (NVIDIA, 2023).

3.3.2. Forward+ Rendering

O *Forward+ Rendering* surge como evolução do método tradicional, buscando mitigar sua principal limitação: o alto custo de processar luzes desnecessárias. Nessa abordagem, a tela é dividida em blocos regulares (tiles) ou volumes espaciais (clusters), e a *Engine* associa cada região apenas às luzes que de fato a influenciam (Olsson; Assarsson, 2012). Dessa forma, quando o *Pixel Shader* executa, ele processa apenas as luzes relevantes ao bloco correspondente, reduzindo desperdício de cálculos.

O método ganhou popularidade em *Engines* modernas devido ao seu bom equilíbrio entre flexibilidade e desempenho. Além de suportar centenas de luzes dinâmicas sem perda significativa de performance, o Forward+ mantém a compatibilidade com transparências e MSAA, algo mais complexo em *Deferred Rendering* (Akenine-Möller et al., 2018). Empresas como Square Enix e DICE documentaram o uso dessa técnica em seus motores proprietários, especialmente em jogos com forte ênfase em iluminação dinâmica (FROSTBITE *ENGINE*, 2017).

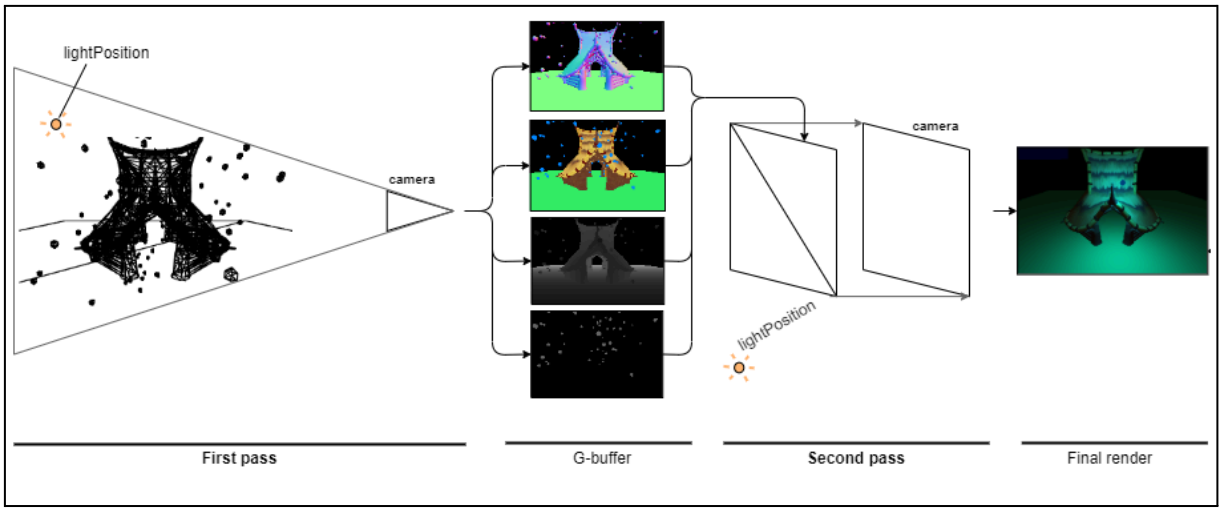
3.3.3. Deferred Rendering

O *Deferred Rendering* representa uma mudança significativa na organização do *Pipeline* gráfico, ao separar a etapa de desenho da geometria da etapa de cálculo da iluminação. Diferentemente do *Forward Rendering*, em que a *GPU* processa objetos e luzes simultaneamente, essa técnica retarda o cálculo de luzes para um segundo momento, permitindo maior escalabilidade em cenas complexas.

O processo inicia-se com o chamado G-*Buffer Pass*, no qual a cena é renderizada em múltiplas texturas auxiliares conhecidas como *G-Buffers*. Nesses *Buffers*, cada *pixel* da tela armazena informações essenciais como posição tridimensional, normal da superfície, cor, profundidade e parâmetros adicionais de material. Esse conjunto de dados substitui a necessidade de reprocessar a geometria quando novas luzes são avaliadas, já que as informações necessárias para o sombreamento permanecem disponíveis em memória (Sanderson; Mitchell, 2009).

Em seguida, ocorre o *Lighting Pass*, no qual a iluminação é calculada a partir dos valores previamente registrados nos *G-Buffers*. Nesse estágio, cada luz é processada individualmente e sua contribuição é aplicada diretamente sobre os *pixels* correspondentes. O custo computacional, portanto, deixa de depender do número de objetos da cena e passa a estar mais relacionado à resolução da tela e à quantidade de luzes, o que torna o método altamente eficiente em cenários com dezenas ou centenas de fontes luminosas (Akenine-Möller; Haines; Hoffman, 2018). A Figura 2 exemplifica graficamente esse funcionamento, destacando a separação entre a etapa de captura dos dados geométricos no G-*Buffer* e a aplicação da iluminação posterior em tela cheia.

Figura 2 - Organização do Pipeline Deferred Rendering.



Fonte: cglearn

A principal vantagem do *Deferred Rendering* é justamente essa escalabilidade: adicionar novas luzes dinâmicas implica em custo marginal reduzido, algo inviável em *Forward Rendering*. Não por acaso, *Engines* amplamente utilizadas, como a Frostbite (usada pela DICE em títulos como Battlefield) e a Unreal *Engine* 4, adotaram variações desse modelo devido à sua eficiência em ambientes realistas e altamente iluminados (EPIC GAMES, 2021). Entretanto, essa abordagem também traz desafios técnicos relevantes. A manipulação de transparências, como superfícies de vidro, líquidos ou gases, é consideravelmente mais complexa, uma vez que a cena não é composta em um único passe, mas sim a partir de *Buffers* intermediários. Além disso, técnicas tradicionais de *Anti-Aliasing* baseadas em múltiplas amostras (*MSAA*) são difíceis de integrar, exigindo a adoção de alternativas como *FXAA* ou *TAA*, que embora mais leves, podem introduzir artefatos visuais (NVIDIA, 2023).

Dessa forma, o *Deferred Rendering* consolidou-se como uma solução de compromisso: oferece desempenho e escalabilidade em cenários ricos em iluminação dinâmica, ao custo de maior complexidade na implementação de efeitos avançados. Sua adoção pela indústria demonstra não apenas sua relevância prática, mas também o quanto a escolha do *Pipeline* gráfico está intrinsecamente ligada às prioridades de cada projeto, equilibrando fidelidade visual, desempenho e viabilidade técnica.

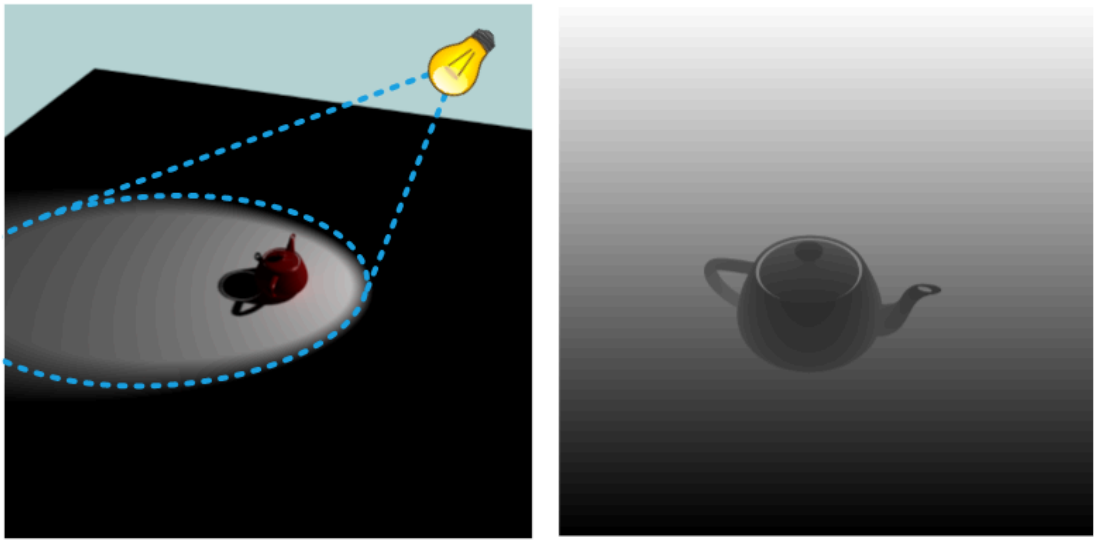
3.4. Arquitetura do Renderizador

O renderizador desenvolvido neste trabalho foi estruturado segundo o modelo de *Deferred Rendering*, técnica utilizada em *Engines* modernas devido à sua eficiência no tratamento de múltiplas fontes de luz. O fluxo de produção das imagens é dividido em três etapas principais: a criação do *G-Buffer*, o cálculo de iluminação (*Lighting Pass*) e a composição final com efeitos de pós-processamento.

Na primeira etapa, o *G-Buffer* Pass, a cena não gera apenas uma textura de cor, mas sim um conjunto de *Buffers* especializados que armazenam informações fundamentais para o sombreamento. Entre esses dados estão a cor difusa, o vetor normal da superfície, a posição em espaço de câmera, além de parâmetros físicos de material, como metalicidade e rugosidade. Esse armazenamento diferenciado permite que cada pixel da tela carregue os atributos necessários para um cálculo de iluminação posterior mais detalhado e flexível (Akenine-Möller; Haines; Hoffman, 2018).

Na sequência ocorre o *Lighting Pass*, no qual cada pixel do G-*Buffer* é processado de acordo com a equação de iluminação definida. Diferentemente do *Forward Rendering*, os objetos da cena não precisam ser redesenhados a cada fonte luminosa, já que todas as informações relevantes já estão registradas nos *Buffers*. Isso torna possível aplicar a iluminação de dezenas ou até centenas de luzes dinâmicas em tempo real, com custo computacional proporcional ao número de pixels e não ao número de objetos da cena (NVIDIA, 2023). É também nessa etapa que entram técnicas de sombreamento em tempo real, como o *Shadow Mapping*, no qual a cena é renderizada da perspectiva da luz para gerar um mapa de profundidade usados posteriormente na determinação das áreas iluminadas e sombreadas. A Figura 3 ilustra esse processo, evidenciando como o mapa de profundidade auxilia na projeção coerente das sombras em relação às fontes luminosas.

Figura 3 - Funcionamento do Shadow Mapping em tempo real.



Fonte: docs.techsoft3d

Concluído o cálculo de iluminação, o resultado segue para a etapa de composição final. Nesse estágio, o sistema integra técnicas de pós-processamento que aprimoram a qualidade visual da imagem. Entre as mais comuns estão o *Bloom*, que simula o efeito de espalhamento de luz em áreas muito intensas, e o *Tone Mapping*, responsável por adaptar a faixa dinâmica de iluminação para o espaço de cor da tela, evitando saturações e preservando detalhes em regiões muito claras ou muito escuras (Pharr; Humphreys, 2023).

Apesar de suas vantagens, a arquitetura baseada em *Deferred Rendering* também apresenta desafios técnicos. Entre os benefícios mais notáveis estão a eficiência no cálculo de múltiplas luzes, já que as equações de iluminação são aplicadas de forma conjunta, e a flexibilidade para implementar modelos de iluminação avançados, incluindo variantes de *Physically Based Rendering* (PBR). Contudo, duas limitações se destacam. A primeira refere-se ao tratamento de transparências: superfícies como vidro, líquidos ou gases exigem abordagens específicas baseadas em *Forward Rendering*, já que os *G-Buffers* não lidam bem com múltiplas camadas sobrepostas. A segunda limitação diz respeito ao antisserrilhamento (*Anti-Aliasing*). Técnicas clássicas como *MSAA* apresentam dificuldade de integração nesse *Pipeline*, o que leva à adoção de alternativas baseadas em pós-processamento, como *FXAA*, *SMAA* ou *TAA*, que embora

eficazes, podem introduzir artefatos e demandam ajustes finos conforme o contexto (Karras, 2022).

Assim, a arquitetura do renderizador aqui implementado reflete a natureza de compromisso típica do *Deferred Rendering*: ao mesmo tempo em que oferece escalabilidade e flexibilidade superiores no manejo de luzes e materiais, exige soluções adicionais para lidar com transparência e suavização de bordas, o que demanda escolhas cuidadosas durante o processo de desenvolvimento.

3.5. Técnicas Avançadas e Otimizações

A implementação de um renderizador baseado em *Deferred Rendering* pode ser potencializada por um conjunto de técnicas avançadas que equilibram qualidade visual e desempenho computacional. Essas abordagens surgem da necessidade de lidar com restrições de *Hardware*, otimizar o uso de memória e possibilitar que cenas complexas sejam exibidas em tempo real sem comprometer a taxa de quadros.

3.5.1. Physically Based Rendering (PBR)

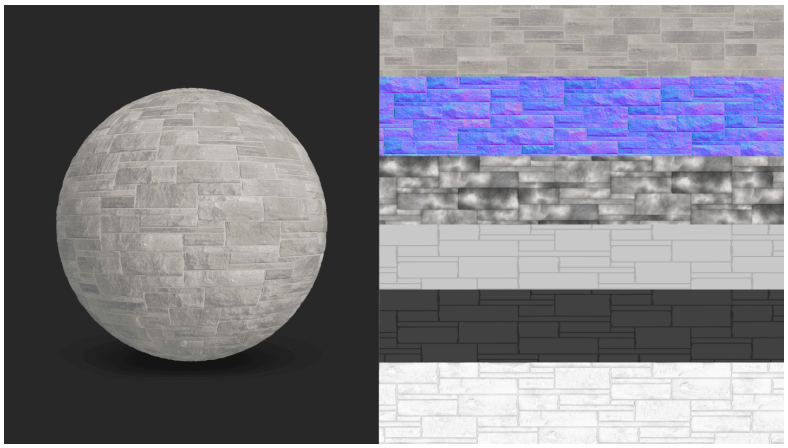
O *Physically Based Rendering* (*PBR*) é um conjunto de modelos matemáticos voltados para reproduzir de forma realista a interação da luz com diferentes tipos de superfícies. Diferentemente de métodos empíricos utilizados em renderizadores mais antigos, o *PBR* se fundamenta em princípios da óptica e na conservação de energia, garantindo que a luz refletida por um material nunca ultrapasse a quantidade de luz incidente. Essa abordagem assegura consistência física, fazendo com que materiais mantenham uma aparência previsível em diferentes condições de iluminação, independentemente da intensidade da luz ou do ambiente em que estão inseridos (Pharr; Humphreys, 2023).

O realismo alcançado pelo *PBR* depende da combinação de diversos mapas de textura que descrevem as propriedades físicas do material. O mapa de *Albedo* define a cor base sem considerar efeitos de iluminação, enquanto o mapa de normais adiciona a sensação de relevo e irregularidades à superfície. Já o mapa de rugosidade controla se a reflexão da luz será difusa ou especular, determinando se o

material terá um aspecto mais polido ou áspero. O mapa de metalicidade diferencia materiais metálicos de não metálicos, alterando profundamente o modo como a luz interage com a superfície. Em alguns casos, outros parâmetros, como o índice de refração, também podem ser incorporados para simular materiais translúcidos, como vidro ou líquidos.

A Figura 4 exemplifica esse processo, mostrando à esquerda o resultado final aplicado em uma esfera com textura de pedra realista, enquanto à direita estão representados os diferentes mapas de textura utilizados para construir o material. Cada um deles contribui para a forma como a luz será refletida ou absorvida, permitindo que superfícies virtuais se aproximem cada vez mais do comportamento observado em objetos reais.

Figura 4 - Exemplo de material PBR, diferentes texturas.



Fonte: lightbeans

Graças a essa previsibilidade e fidelidade visual, o *PBR* tornou-se um padrão amplamente adotado em *Engines* modernas como Unreal e Unity, que disponibilizam fluxos de trabalho baseados nesse modelo para artistas e desenvolvedores. Com isso, é possível criar materiais que mantêm consistência visual em diferentes projetos, melhorando tanto a imersão gráfica quanto a eficiência na produção de jogos (EPIC GAMES, 2021).

3.5.2. Occlusion Culling e LOD

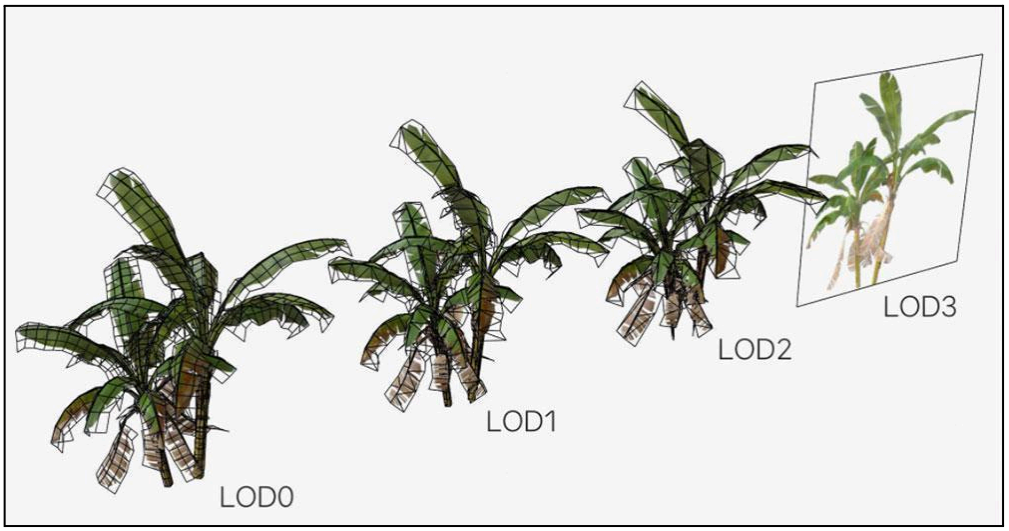
Occlusion Culling – Uma das otimizações clássicas em gráficos em tempo real é o descarte de objetos que não contribuem para a imagem final por estarem completamente ocultos por outros elementos. Essa técnica reduz significativamente o número de *Draw Call*s e libera a *GPU* para processar apenas os elementos visíveis. Métodos modernos utilizam algoritmos hierárquicos baseados em árvores de volumes (*Bounding Volume Hierarchies*) ou *Depth Buffers* para determinar a visibilidade de forma eficiente (Baik et al., 2020).

Level of Detail (*LOD*) – Uma das formas mais comuns de otimizar a renderização em tempo real é reduzir a complexidade dos modelos *3D* conforme eles se afastam da câmera. Essa técnica é chamada de *Level of Detail* (*LOD*), ou nível de detalhe. A lógica é simples: quando um objeto está muito distante, o jogador não consegue perceber seus detalhes finos, como o número exato de polígonos de uma folha ou de uma parede. Assim, não faz sentido desperdiçar recursos de processamento para desenhar um modelo altamente detalhado que, na prática, aparecerá na tela como um pequeno ponto.

Para lidar com isso, os desenvolvedores criam diferentes versões do mesmo objeto: uma muito detalhada para quando o objeto está próximo ao observador, e outras progressivamente simplificadas para distâncias maiores. Esse processo pode incluir reduzir a quantidade de polígonos, trocar texturas por versões de menor resolução ou até substituir o modelo por uma simples imagem (*billboard*). *Engines* modernas, como Unity e Unreal, já oferecem ferramentas automáticas para gerar essas versões simplificadas, garantindo que a transição entre os níveis de detalhe ocorra de forma suave e quase imperceptível (Akenine-Möller; Haines; Hoffman, 2018).

A Figura 5 mostra um exemplo prático dessa técnica: um mesmo modelo de planta é exibido em quatro níveis de detalhe (LOD0, LOD1, LOD2 e LOD3). No nível mais próximo (LOD0), o objeto possui uma malha completa e detalhada, enquanto nos níveis seguintes há reduções progressivas na quantidade de polígonos. No último nível (LOD3), o modelo é substituído por uma imagem plana, já que àquela distância o jogador não distinguiria diferenças relevantes.

Figura 5 - Exemplo de Level of Detail



Fonte: 3dstudio.co

3.5.3. Instancing e Anti-Aliasing

Instancing – O *Instancing* é um recurso essencial para cenas com grande repetição de elementos semelhantes, como florestas, multidões ou partículas. Em vez de enviar cada objeto individualmente à *GPU*, o método desenha múltiplas cópias de um mesmo modelo variando apenas parâmetros como posição, escala e rotação. Essa técnica minimiza a sobrecarga de chamadas de desenho e aproveita o paralelismo massivo da *GPU*, permitindo renderizar milhares de instâncias em um único *Draw Call* (NVIDIA, 2023).

Anti-Aliasing – Um dos problemas visuais mais comuns em gráficos em tempo real é o chamado aliasing, que aparece como bordas serrilhadas em objetos *3D*. Esse efeito ocorre porque a tela é composta por pixels quadrados que não conseguem representar perfeitamente linhas diagonais ou curvas. Para o jogador, isso resulta em imagens com contornos pouco naturais, prejudicando a imersão.

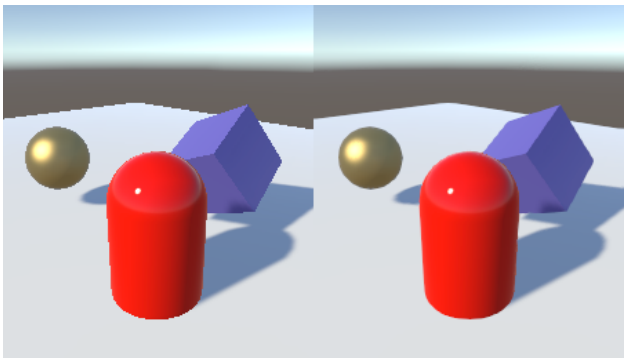
A suavização dessas bordas é chamada de *Anti-Aliasing* e, em *Pipeline*s forward, é normalmente feita com *MSAA* (*Multi-Sample Anti-Aliasing*). Porém, em *Pipeline*s deferred, a integração do *MSAA* é mais complexa devido à forma como os *G-Buffers* armazenam as informações da cena. Por esse motivo, técnicas de

pós-processamento se tornaram a solução mais comum.

Entre elas, destaca-se o *FXAA* (*Fast Approximate Anti-Aliasing*), conhecido por sua rapidez e baixo custo computacional, embora às vezes suavize em excesso os detalhes finos da cena. Já o *SMAA* (*Subpixel Morphological Anti-Aliasing*) oferece uma melhor definição ao combinar análise morfológica das bordas com correções em nível *Subpixel*. Por fim, o *TAA* (*Temporal Anti-Aliasing*) vai além, utilizando informações de quadros anteriores para suavizar não apenas as bordas estáticas, mas também reduzir a cintilação em movimento. Apesar da eficácia, o *TAA* pode introduzir artefatos de *ghosting* quando mal configurado (Karras, 2022).

A Figura 6 ilustra o efeito do Anti-Aliasing. À esquerda, é possível observar objetos 3D com bordas serrilhadas, sem qualquer técnica de suavização aplicada. À direita, os mesmos objetos aparecem com o Anti-Aliasing ativado, apresentando contornos mais suaves e realistas, o que melhora significativamente a qualidade visual da cena.

Figura 6 - Exemplo de Anti-Aliasing



Fonte: vr.arvilab

4. DESENVOLVIMENTO DA *ENGINE*

O processo de desenvolvimento da *Engine* foi realizado de forma incremental, adotando uma abordagem iterativa que possibilitou a validação de cada novo recurso antes da introdução de funcionalidades mais complexas. Essa estratégia é

comum em projetos de computação gráfica e desenvolvimento de *Game Engines*, pois reduz a complexidade do depuramento e facilita a identificação de gargalos de desempenho (Gregory, 2018).

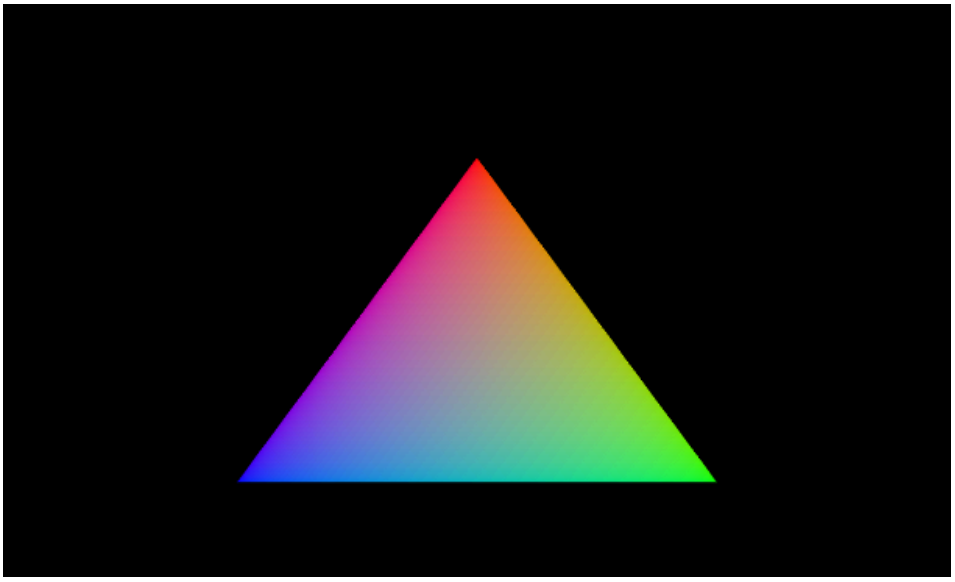
4.1. Teste de renderização com BGFX: o primeiro triângulo

A primeira etapa consistiu em validar a configuração da biblioteca BGFX, escolhida como abstração gráfica devido à sua capacidade de oferecer suporte multiplataforma para APIs modernas como Direct*3D*, OpenGL, Vulkan e Metal (BGFX, 2024). O teste inicial correspondeu à renderização de um triângulo estático, considerado o equivalente ao "*Hello World*" da computação gráfica.

Esse procedimento, embora simples, possui valor pedagógico e técnico significativo. Ele comprova que os estágios fundamentais do *pipeline*, a criação da janela, inicialização da *GPU*, configuração dos *Shaders* e envio de vértices ao vertex *Buffer* foram corretamente implementados (Akenine-Möller; Haines; Hoffman, 2018). A exibição correta do triângulo validou a comunicação entre *CPU* e *GPU*, estabelecendo a base sobre a qual os demais recursos seriam construídos.

O primeiro experimento realizado consistiu na renderização de um triângulo colorido utilizando a biblioteca BGFX. Este teste teve como objetivo validar o funcionamento do *Pipeline* gráfico básico, incluindo o envio de vértices à *GPU*, a compilação de *Shaders* e o gerenciamento de *Buffers*. A figura 7 ilustra o resultado obtido, demonstrando a correta interpolação de cores entre os vértices.

Figura 7 - Primeiro triângulo



Fonte: autoria própria

4.2. Renderização de modelos 3D compilados

Com o *Pipeline* básico funcionando, avançou-se para a renderização de modelos tridimensionais. Essa etapa demandou a implementação de um sistema de importação e compilação de modelos para o formato aceito pelo BGFX, envolvendo a conversão de geometrias em *Vertex Buffers* e *Index Buffers*.

Do ponto de vista teórico, a renderização de modelos *3D* é uma das principais responsabilidades de uma *Game Engine*. É nesse estágio que conceitos como malhas poligonais, hierarquias de objetos e sistemas de coordenadas tornam-se centrais (Pharr; Jakob; Humphreys, 2023). Ao carregar modelos compilados, a *Engine* passou a lidar com estruturas mais próximas de aplicações reais, incluindo múltiplos objetos, diferentes materiais e a necessidade de otimização do tráfego de memória entre *CPU* e *GPU*.

A validação dessa etapa representou a transição do teste puramente didático (triângulo estático) para a manipulação de ativos gráficos (*Assets*) típicos de um jogo ou aplicação interativa.

4.3. Introdução do conceito de câmera

O próximo passo foi a introdução da câmera virtual, responsável por definir a perspectiva sob a qual a cena é exibida. Esse conceito é fundamental na computação gráfica, pois permite a navegação em ambientes tridimensionais a partir da combinação de duas matrizes principais: *View Matrix* (responsável pela posição e orientação da câmera no espaço) e *Projection Matrix* (responsável pela transformação da cena em coordenadas de tela, seja por projeção ortográfica ou perspectiva).

A implementação contemplou controles básicos de interação por teclado e mouse, permitindo translação e rotação da câmera. Essa adição transformou o renderizador em um ambiente verdadeiramente interativo, no qual o usuário pode explorar os modelos sob diferentes ângulos e perspectivas.

Segundo Akenine-Möller, Haines e Hoffman (2018), a abstração da câmera é indispensável para a percepção espacial em ambientes virtuais, sendo parte estrutural de qualquer *Engine* moderna. Além disso, sua correta implementação garante consistência entre os sistemas de movimentação, iluminação e renderização.

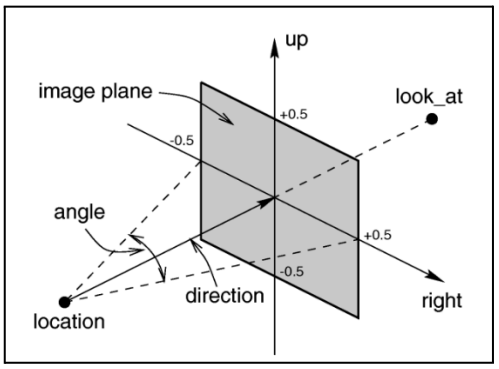
Após a validação da renderização básica, foi implementado o conceito de câmera virtual, elemento essencial em qualquer sistema gráfico tridimensional. A câmera define o ponto de vista do observador dentro do ambiente *3D*, determinando a posição, a direção de observação e o enquadramento da cena.

O modelo adotado segue a projeção em perspectiva, onde objetos mais distantes aparecem menores, simulando a percepção visual humana. A câmera é representada por um conjunto de vetores que definem sua orientação no espaço:

- **location** (posição) indica o ponto de origem da visualização;
- **look_at** define o ponto para o qual a câmera está direcionada;
- **up** determina o vetor vertical da câmera, necessário para calcular sua rotação;
- **right** representa o vetor horizontal, completando o sistema de coordenadas local.

Além disso, parâmetros como o campo de visão (*Field of View – FOV*) e a distância do plano de projeção afetam diretamente a sensação de profundidade e a composição visual. A Figura 8 ilustra o funcionamento da câmera em perspectiva, mostrando como o plano de imagem é posicionado em relação ao observador e ao ponto focal.

Figura 8 - Câmera perspectiva



Fonte: f9.ijs.si

4.4. Renderização de texturas em modelos 3D

Com a renderização básica de modelos implementada, a etapa seguinte consistiu em habilitar o uso de texturas sobre as superfícies, recurso essencial para conferir realismo visual e diversidade estética às cenas tridimensionais. A texturização permite mapear imagens bidimensionais (2D) sobre malhas tridimensionais (3D), atribuindo padrões visuais de cor, detalhe e relevo. Esse processo possibilita que modelos relativamente simples apresentem aparência rica e complexa, sem a necessidade de aumentar o número de polígonos (Akenine-Möller; Haines; Hoffman, 2018).

A implementação foi realizada por meio do carregamento de imagens em *Buffers* de textura, vinculados diretamente aos *Shaders* responsáveis pelo processamento de fragmentos (*Pixel Shaders*). Durante a renderização, cada vértice do modelo contém coordenadas *UV* — armazenadas nos vertex *Buffers* — que definem como os pontos da imagem (texels) são mapeados sobre a superfície tridimensional. Esse mapeamento garante que cada *Fragmento* da malha receba a

informação visual correspondente à área exata da textura.

Segundo Pharr, Jakob e Humphreys (2023), a aplicação de texturas constitui um dos pilares da computação gráfica moderna, pois permite substituir complexidade geométrica por complexidade visual. Assim, detalhes como rugosidades, fissuras, ferrugem ou imperfeições naturais são simulados visualmente, reduzindo o custo computacional e preservando o desempenho em tempo real. Além da textura de cor (*Albedo*), o sistema também pode empregar mapas adicionais — como *Normal Map*s, *Metallic Maps* e *Roughness Map*s —, formando a base para a implementação de técnicas de *Physically Based Rendering* (*PBR*).

A Figura 9 ilustra o resultado prático desse processo, apresentando um modelo tridimensional texturizado. À esquerda, observa-se o modelo final com os materiais aplicados, enquanto à direita o mesmo objeto é exibido em modo *Wireframe*, revelando sua estrutura geométrica subjacente. Essa comparação evidencia como o mapeamento *UV* e a aplicação de texturas podem enriquecer visualmente um modelo sem alterar sua complexidade poligonal.

Figura 9 - Carregamento de modelo 3d com textura

Fonte: learnopengl

A conclusão dessa etapa consolidou a *Engine* como uma plataforma capaz de exibir modelos tridimensionais realistas, aproximando seu funcionamento das soluções adotadas em *Engines* comerciais, como Unreal e Unity. Esse avanço representa um marco no desenvolvimento do renderizador, tornando possível

manipular e visualizar *Assets* com qualidade comparável às aplicações reais da indústria de jogos.

4.5. Introdução de luzes direcionais e pontuais

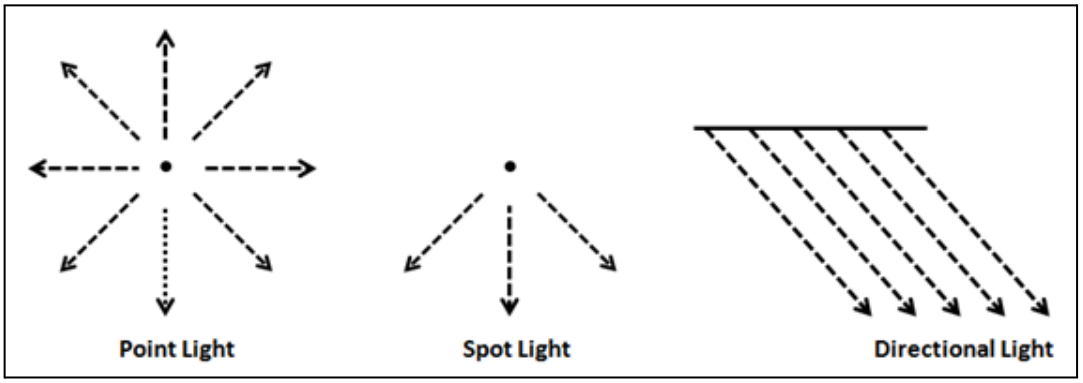
A partir do momento em que os modelos *3D* texturizados passaram a ser exibidos corretamente, tornou-se necessário introduzir o conceito de iluminação, um elemento essencial para criar profundidade, contraste e realismo em ambientes virtuais. É a luz que permite ao observador perceber formas, volumes e materiais, tornando a cena mais próxima da forma como o olho humano enxerga o mundo real.

Nesta etapa, foram implementados dois tipos básicos de fontes de luz: a luz direcional e a luz pontual. A luz direcional simula uma fonte de iluminação muito distante, como o Sol, em que todos os raios de luz incidem paralelamente sobre os objetos. Esse tipo de luz é ideal para iluminar grandes áreas de forma uniforme. Já a luz pontual funciona como uma lâmpada comum, emitindo luz em todas as direções a partir de um ponto fixo no espaço. Assim, os objetos mais próximos da fonte ficam mais claros, enquanto os mais distantes recebem menos luz, criando um efeito natural de atenuação.

Para que essas luzes influenciassem a aparência dos objetos, foram feitas modificações nos *Shaders* — pequenos programas que controlam a cor e o brilho de cada ponto da imagem. Eles passaram a calcular a intensidade luminosa com base na orientação da superfície (as chamadas normais) e na posição da luz em relação a cada *Fragment*o renderizado.

A Figura 10 ilustra os principais tipos de luz utilizados em motores gráficos: a luz pontual (Point Light), que emite em todas as direções; a luz de foco (Spot Light), que concentra o feixe em um cone; e a luz direcional (Directional Light), cujos raios são paralelos. No projeto, foram implementadas as duas primeiras, fundamentais para compreender o comportamento da iluminação básica em 3D.

Figura 10 - Tipos de luzes



Fonte: willievaldez

Segundo Akenine-Möller, Haines e Hoffman (2018), essas duas formas de iluminação são a base para sistemas mais sofisticados, como luzes direcionais com sombras projetadas, spotlights com cone de influência e até modelos de iluminação global. A correta aplicação desses conceitos permite que as superfícies revelem brilho, sombreamento e textura de forma convincente.

A implementação dessa etapa marcou um avanço importante no desenvolvimento do renderizador, pois estabeleceu a primeira integração completa entre geometria, textura e iluminação, transformando as cenas planas em ambientes tridimensionais visualmente coerentes e com sensação realista de profundidade.

4.6. Modelo PBR com múltiplas texturas

A evolução natural do sistema de iluminação foi a adoção do modelo de *Physically Based Rendering* (PBR), atualmente padrão em *Engines* modernas como Unreal e Unity (EPIC GAMES, 2021). O PBR busca simular a interação da luz com superfícies de forma fisicamente plausível, fundamentado em princípios da óptica e da conservação de energia (Pharr; Humphreys, 2023).

Diferentemente de modelos empíricos mais antigos, como Lambert ou Blinn-Phong, o PBR descreve materiais a partir de parâmetros como:

- ***Albedo*** : cor base do material;
- **Rugosidade** (*roughness*): define a dispersão da luz especular;
- **Metalicidade** (*metalness*): controla a proporção entre reflexão metálica e

difusa;

- ***Normal Maps***: simulam irregularidades de superfície sem aumentar a geometria.

Esses mapas de textura adicionais foram integrados ao *Pipeline* da *Engine*, permitindo que cada objeto exibisse comportamentos realistas sob diferentes condições de iluminação. Por exemplo, metais passaram a refletir intensamente o ambiente, enquanto superfícies rugosas dispersam a luz de forma difusa.

A implementação de PBR representou um salto qualitativo na *Engine*, aproximando-a dos padrões da indústria e garantindo consistência visual entre diferentes materiais. Além disso, possibilitou a aplicação prática de conceitos discutidos no referencial teórico, unindo o embasamento científico à experimentação técnica.

4.7. Reestruturação do renderizador: migração de Forward para Deferred Rendering

Até esse ponto, a *Engine* trabalhava com um *Pipeline* baseado no *Forward Rendering*, onde cada objeto da cena era processado em conjunto com todas as luzes relevantes. Embora simples, essa abordagem se torna limitada quando a cena possui muitas fontes de iluminação, pois o custo de renderização cresce de forma proporcional ao número de luzes e objetos.

Diante desse desafio, foi realizada a migração para o modelo de *Deferred Rendering*, já discutido no referencial teórico. Nessa arquitetura, a cena é primeiro renderizada em múltiplos *Buffers* auxiliares (*G-Buffers*), armazenando informações como cor difusa, normais, profundidade e parâmetros de material. Em seguida, as equações de iluminação são aplicadas em um passo separado, diretamente sobre os pixels da tela, sem a necessidade de redesenhar os objetos.

Segundo Akenine-Möller, Haines e Hoffman (2018), essa técnica apresenta um ganho expressivo em desempenho em cenários com dezenas ou centenas de luzes dinâmicas, visto que o custo passa a ser proporcional ao número de pixels processados e não mais à multiplicação entre luzes e objetos. Por essa razão, *Engines* modernas como a Unreal *Engine* e Frostbite utilizam variações do modelo *Deferred* em seus *pipeline*s (EPIC GAMES, 2021).

A reestruturação exigiu uma revisão significativa na organização do código da *Engine*, especialmente no gerenciamento de *Buffers* de renderização e na implementação do *Lighting Pass*. Esse esforço, no entanto, resultou em maior escalabilidade, tornando viável o uso de cenas mais complexas e com maior riqueza de iluminação.

4.8. Implementação de Shadow Mapping

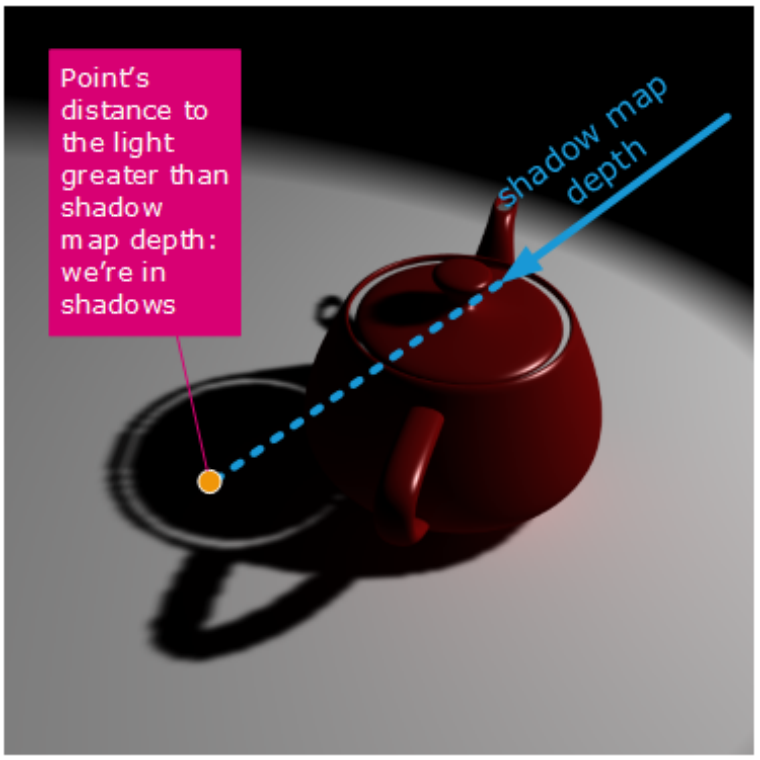
Com a arquitetura *deferred* consolidada, a etapa seguinte consistiu na introdução do *Shadow Mapping*, uma das técnicas mais utilizadas para a geração de sombras dinâmicas em tempo real. As sombras desempenham um papel essencial na percepção de profundidade e na noção de espaço tridimensional, sendo fundamentais para transmitir realismo visual em cenas virtuais.

O princípio do método é relativamente simples, mas extremamente eficaz. Primeiramente, a cena é renderizada a partir da perspectiva da fonte de luz, e as informações de profundidade dos objetos são armazenadas em um *buffer* especial chamado *Shadow Map*. Esse mapa registra a distância entre a luz e os objetos visíveis a partir dela.

Na etapa seguinte, durante o cálculo de iluminação da câmera principal, cada pixel visível é comparado com o valor de profundidade armazenado no *Shadow Map*: se o pixel estiver mais distante da luz do que o valor gravado no mapa, ele está em sombra, caso contrário, ele está iluminado (Williams, 1978).

A Figura 11 ilustra esse processo, mostrando como o valor de profundidade armazenado no mapa é utilizado para determinar se um ponto na superfície está iluminado ou sombreado.

Figura 11 - Exemplo de shadow mapping



Fonte: techsoft3d

Embora o conceito seja direto, o Shadow Mapping apresenta alguns desafios técnicos importantes:

- Aliasing em baixa resolução, que resulta em bordas de sombra irregulares e serrilhadas;
- "Shadow acne", efeito visual causado por imprecisões numéricas que fazem com que as superfícies se auto-sombreiem indevidamente;
- Limitações de alcance, especialmente em cenas amplas, que exigem soluções como *Cascaded Shadow Maps* (*CSM*) para preservar a nitidez das sombras em diferentes distâncias da câmera.

Para minimizar esses problemas, foram aplicadas técnicas de correção, como o ajuste de bias (um pequeno deslocamento na profundidade para evitar sobreposição incorreta) e filtragens simples no mapa de sombras para suavizar as transições. De acordo com Pharr, Jakob e Humphreys (2023), o Shadow Mapping continua sendo uma das soluções mais equilibradas entre qualidade visual, custo computacional e compatibilidade com diferentes arquiteturas de *GPU*, razão pela

qual permanece amplamente adotado na indústria de jogos e visualização 3D.

A implementação dessa técnica no renderizador representou um salto significativo de realismo, adicionando contraste, profundidade e coerência visual às cenas tridimensionais. As sombras dinâmicas não apenas aumentaram a imersão do observador, mas também serviram como validação da robustez do pipeline deferred desenvolvido neste trabalho.

4.9. Reestruturação do projeto com suporte a construção de cenas

Após a implementação das bases de renderização e iluminação, tornou-se necessário organizar o projeto em torno de um sistema de gerenciamento de cenas. Esse módulo passou a ser responsável por centralizar e estruturar os elementos que compõem um ambiente 3D, incluindo modelos, luzes, câmeras e parâmetros de renderização.

A adoção desse sistema possibilitou a implementação de funcionalidades como:

- Hierarquia de objetos (*Scene Graph*): cada entidade pode ser vinculada a outra, permitindo transformações relativas (ex.: uma lâmpada presa a um poste).
- Gerenciamento centralizado de recursos: controle de carregamento, descarte e reutilização de modelos e texturas.
- Organização para testes e *Benchmark*s: criação de múltiplos cenários, cada qual voltado a experimentos específicos, como desempenho com luzes dinâmicas ou análise de materiais PBR.

Segundo Gregory (2018), a organização em *Scene Graphs* é uma das práticas mais comuns em *Engines* comerciais, pois fornece escalabilidade e flexibilidade para lidar com ambientes cada vez mais complexos. Essa reestruturação foi, portanto, essencial para dar maior robustez ao projeto e permitir que evoluísse além de simples demonstrações gráficas isoladas.

4.10. Adição de interface gráfica com ImGUI

Por fim, foi incorporada à *Engine* uma interface gráfica interativa, desenvolvida com a biblioteca *Dear ImGui*. Esse recurso foi fundamental para facilitar o processo de desenvolvimento, testes e depuração do motor gráfico, oferecendo um meio visual e intuitivo para interagir com os parâmetros internos da aplicação em tempo real.

A biblioteca *Dear ImGui* é amplamente utilizada na indústria de desenvolvimento de jogos e aplicações gráficas devido à sua leveza, flexibilidade e fácil integração com *APIs* como OpenGL, Vulkan e BGFX. Diferentemente de interfaces tradicionais voltadas ao usuário final, o ImGui é voltado ao desenvolvedor, permitindo criar janelas, botões, menus e sliders diretamente dentro do contexto da renderização. Essa característica o torna ideal para o controle e depuração de *Pipeline*s gráficos, especialmente durante a fase de implementação e otimização.

A interface adicionada à *Engine* permitiu diversas funcionalidades práticas, como:

- Alterar parâmetros de materiais em tempo real, incluindo *Albedo,* rugosidade e metalicidade, o que possibilitou observar instantaneamente os efeitos das mudanças sobre os modelos *3D*;
- Ativar e desativar fontes de luz, facilitando a análise de desempenho e a avaliação de diferentes configurações de iluminação;
- Visualizar separadamente os *G-Buffers*, como mapas de normais, profundidade e cor, permitindo a inspeção detalhada das etapas do processo de renderização deferred;
- Monitorar métricas de desempenho, como o número de quadros por segundo (*FPS*), tempo de renderização por *Frame* e uso de *GPU*, auxiliando na detecção de gargalos e otimizações possíveis.

De acordo com Spenke e Beilken (2019), ferramentas de interface gráfica voltadas a desenvolvedores são indispensáveis em *Pipeline*s iterativos, pois reduzem o tempo de experimentação, aumentam a eficiência do teste e simplificam a análise de resultados. No contexto deste projeto, o uso do ImGui demonstrou-se essencial para criar uma rotina de experimentação rápida e controlada, característica típica de motores gráficos profissionais como Unity e Unreal.

A Figura 12 ilustra um exemplo de interface desenvolvida com a biblioteca *Dear ImGui*, demonstrando controles interativos, janelas de depuração, seletores de cor e painéis de ajuste de parâmetros. Esse tipo de ferramenta representa um componente-chave no desenvolvimento moderno de *Engines*, pois permite que programadores e artistas colaborem diretamente, sem a necessidade de recompilar o código a cada ajuste.

Figura 12 - Exemplo de UI com imgui

Fonte: ocornut

A introdução do *Dear ImGui*, portanto, não apenas tornou o desenvolvimento da *Engine* mais dinâmico e produtivo, mas também consolidou a estrutura do projeto como uma base flexível para expansões futuras. Com essa ferramenta, tornou-se possível a implementação de um editor de cenas interativo, bem como a criação de ferramentas internas de depuração, aproximando ainda mais o projeto da arquitetura de motores gráficos utilizados profissionalmente no mercado.

5. ANÁLISE DOS RESULTADOS OBTIDOS

Os resultados obtidos com o desenvolvimento da *Engine* gráfica, em termos de performance, foram altamente satisfatórios em *hardware* de consumo. Foram desenvolvidos dois cenários distintos de teste, cada um projetado para avaliar aspectos específicos do *pipeline*, tais como capacidade de lidar com grandes quantidades de luzes, e *Shadow Mapping* em pontos de luz omnidirecionais. Em ambos os casos, as taxas de quadros foram superiores a mil por segundo, mesmo sob condições claramente não ideais e com margem para otimização.

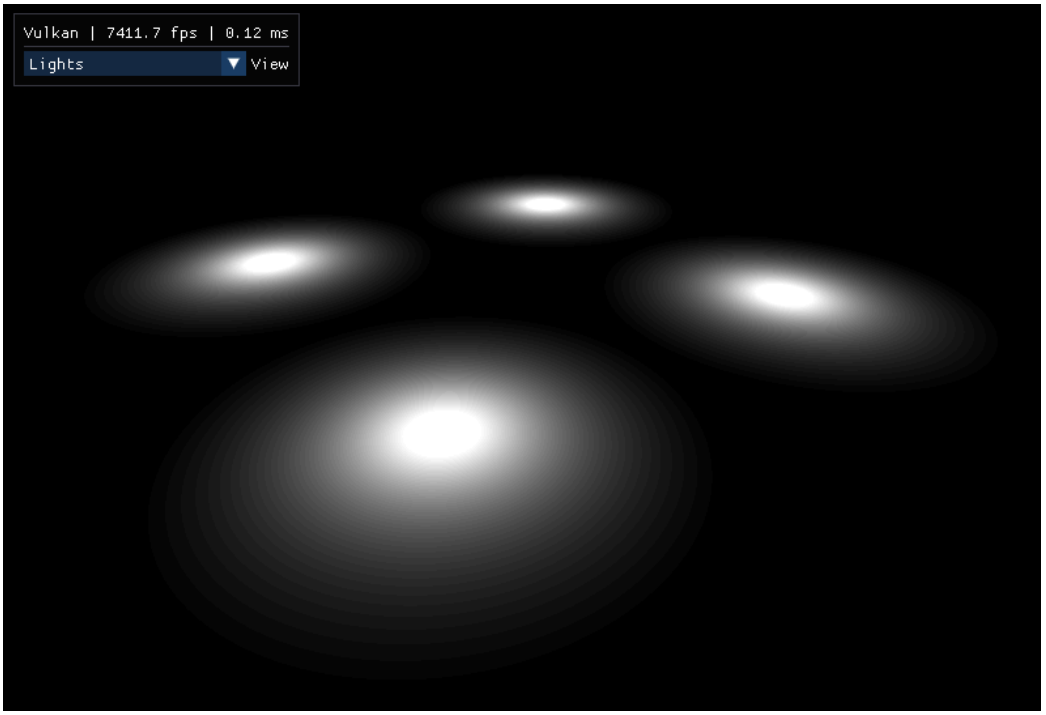
O primeiro cenário de avaliação foi projetado para testar a escalabilidade do *Deferred Rendering* em situações com grande quantidade de luzes. Um modelo 3D foi posicionado no ambiente atuando como uma superfície para a visualização de luzes, e um grande conjunto de pontos de luz foi colocado imediatamente acima desse modelo. Nesse teste, o principal objetivo foi verificar até que ponto seria possível aumentar o número de fontes luminosas antes que o desempenho fosse drasticamente comprometido.

Um aspecto importante desse cenário é a possível alteração do espaçamento entre as luzes. Se estiverem posicionadas muito próximas entre si, o cenário acaba por não representar uma situação ideal de renderização. Em condições reais de produção, distribuições espaciais mais amplas tendem a reduzir o número de fragmentos afetados por cada luz, o que diminui significativamente o custo de processamento. Nesse contexto, foi possível posicionar 2500 pontos de luz dinâmicos com desempenho consistente, apresentando uma queda na performance em cenários de pouco espaçamento entre as luzes.

Cada uma dessas luzes teve sua influência calculada considerando o material *PBR* aplicado no modelo 3D, o que aumenta consideravelmente a complexidade computacional da iluminação. Apesar disso, a *Engine* manteve altas taxas de quadros, demonstrando que o pipeline implementado é eficiente e escalável em diferentes resoluções.

A Figura 13 apresenta um dos quadros capturados deste primeiro experimento, renderizando apenas 4 pontos de luz dinâmicos na resolução de 1400 por 800 *pixels* e atingindo cerca de 7400 quadros por segundo, com cada quadro sendo gerado em aproximadamente 0,12 milissegundos. Um segundo teste com a resolução de 1920 por 1080 *pixels* (*full HD*) demonstrou cerca de 5000 quadros por segundo na mesma cena, com cada quadro sendo gerado em aproximadamente 0,18 milissegundos.

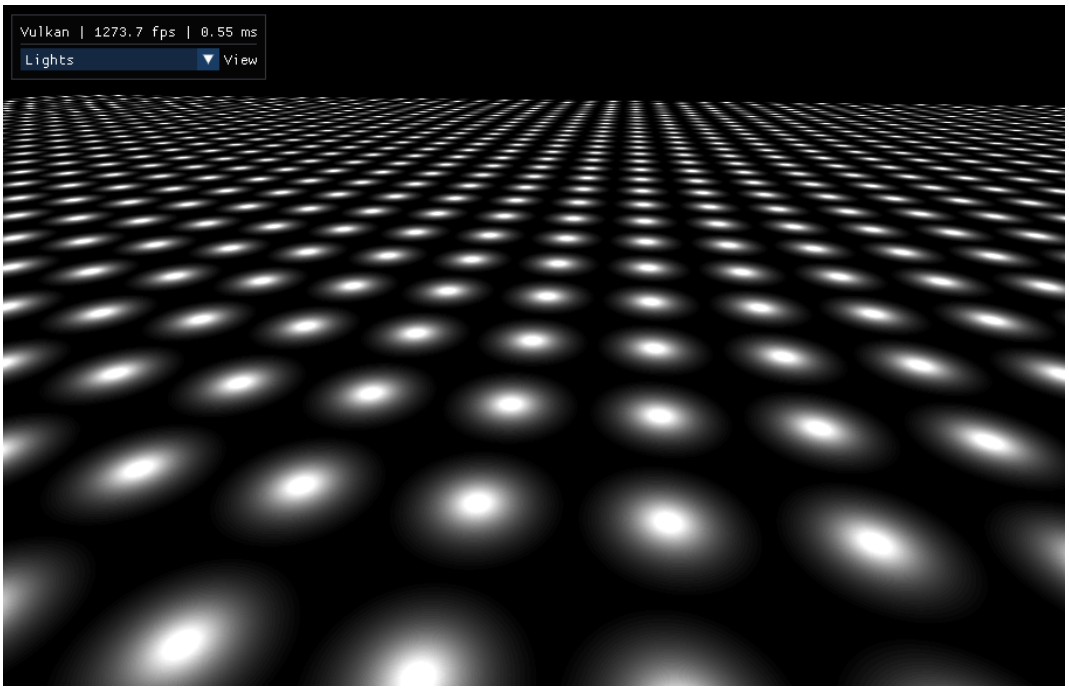
Figura 13 - Exemplo de renderização com 4 luzes



Fonte: autoria própria

A Figura 14 apresenta um outro quadro capturado, com a renderização de 2500 pontos de luz dinâmicos na resolução 1400 por 800 *pixels*, atingindo cerca de 1300 quadros por segundo, dependendo da quantidade de luzes visíveis simultaneamente, com cada quadro sendo gerado em aproximadamente 0,55 milissegundos. Um próximo teste com resolução *full HD* demonstrou cerca de 1200 quadros por segundo na mesma cena, com cada quadro sendo gerado em aproximadamente 0,50 milissegundos.

Figura 14 - Exemplo de renderização com 2500 luzes



Fonte: autoria própria

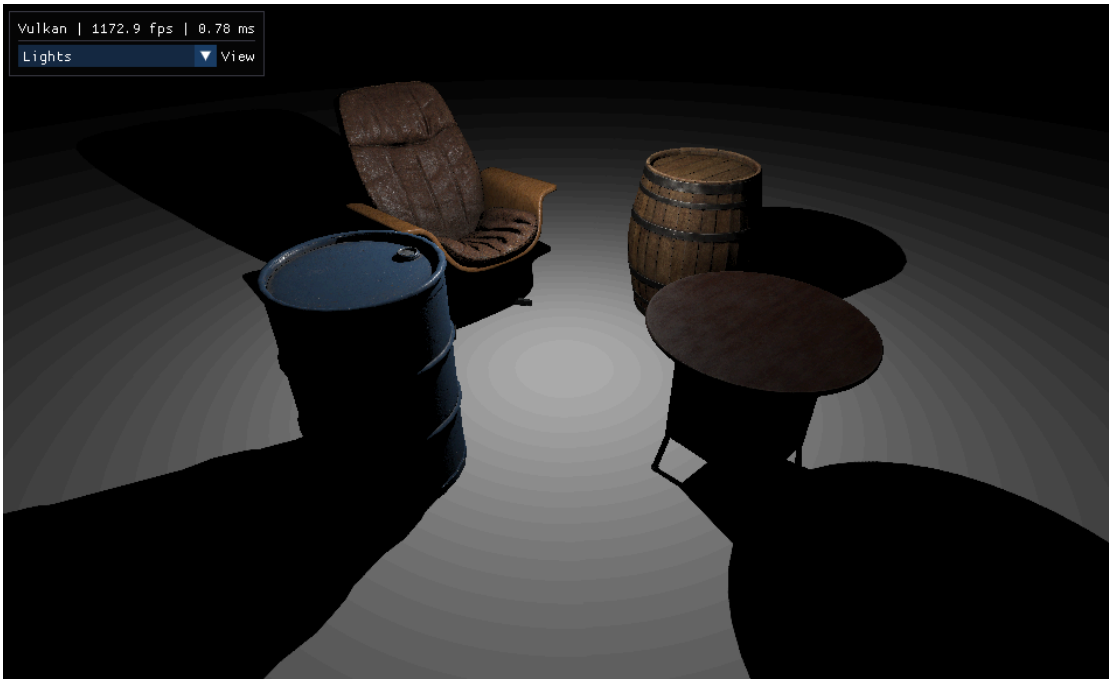
O segundo cenário teve como foco a avaliação do desempenho da *Engine* ao lidar com *Shadow Mapping* em condições de alta carga. Para isso, modelos 3D foram distribuídos ao redor de uma única fonte de luz omnidirecional responsável por projetar sombras dinâmicas. Diferentemente de *pipelines* otimizados que reutilizam mapas de sombra por diversos frames ou que reduzem a resolução para minimizar custo computacional, este teste forçou o sistema ao limite utilizando um mapa de sombras de alta resolução e realizando o cálculo completo de *Shadow Mapping* a cada *frame*.

Este cenário foi configurado de maneira deliberadamente não ideal dada a dinamicidade das sombras e a alta resolução de cada face do mapa de sombras do ponto de luz. Ainda assim, a engine demonstrou comportamento excepcional, registrando alto desempenho, mesmo com o custo elevado de criação do shadow map e do processo de reconstrução de profundidade e iluminação para os objetos na cena.

A Figura 15 apresenta uma captura de quadro representativa deste segundo teste. Com uma resolução de 2048 por 2048 pixels em cada face do mapa de

sombras do ponto de luz foi possível obter, na resolução de tela de 1400 por 800 *pixels,* uma média de 1200 quadros por segundo, cada quadro sendo gerado em cerca de 0,80 milissegundos. Números que poderiam ser melhores com resoluções mais baixas do mapa de sombras e a aplicação de técnicas de otimização nas sombras, que não precisam necessariamente serem calculadas todo frame. Um outro teste com resolução *full HD* demonstrou cerca de 1050 quadros por segundo na mesma cena, com cada quadro sendo gerado em aproximadamente 0,90 milissegundos.

Figura 15 - Exemplo de renderização com modelos 3D e Shadow Mapping



Fonte: autoria própria

Assim, o renderizador desenvolvido para a *engine* se mostra bastante capaz, com desempenho aceitável mesmo em cenários não ideais projetados para testar seus limites.

6. CONSIDERAÇÕES FINAIS

Buscou-se com este trabalho construir, em uma *Engine* relativamente simples, um *Pipeline* de *Deferred Rendering* capaz de exibir cenas interativas cheias de luzes mantendo desempenho aceitável em *Hardware* de consumo.

Concluiu-se que, apesar das dificuldades encontradas estarem relacionadas ao conhecimento das especificidades de cada uma das etapas do *Pipeline* de renderização, foi possível implementar uma *Engine* gráfica básica capaz de exibir uma janela de jogo com as funcionalidades de carregamento de modelos *3D* e texturas, renderização dos modelos em uma *Pipeline Deferred Rendering* que leva em conta luzes direcionais e pontos de luz omnidirecionais, juntamente com a possibilidade de projeção de sombras. Tudo isso em cenas onde o posicionamento dos objetos, a iluminação das luzes e a projeção de sombras são totalmente dinâmicos e calculados todo *Frame*.

A iluminação em particular leva em conta materiais *PBR* aplicados nos modelos 3D, garantindo assim realismo e consistência de cada objeto em diferentes circunstâncias de posicionamento de luzes. Assim é possível a interatividade com o movimento da câmera do jogador, visando observar a cena de vários ângulos e realizar análises de desempenho com base na quantidade e variedade de objetos e luzes renderizadas.

Assim, garante-se uma sólida base para, como sugestão de trabalhos futuros, a implementação de otimizações, novos módulos de funcionalidade que não foram implementados pelo fator tempo na produção deste trabalho, e os demais requisitos para o desenvolvimento de jogos com o *software* a nível de produção.

REFERÊNCIAS BIBLIOGRÁFICAS

3DSTUDIO. **3D LOD (Level of Detail)**. Disponível em: https://3dstudio.co/pt/3d-lod-level-of-detail. Acesso em: 27 out. 2025.

AKENINE-MÖLLER, Tomas; HAINES, Eric; HOFFMAN, Naty. **Real-time Rendering**. 4. ed. Boca Raton: CRC Press, 2018.

ARVILAB. **Anti-Aliasing in VR/AR**. Disponível em: https://vr.arvilab.com/blog/Anti-Aliasing. Acesso em: 27 out. 2025.

CGLEARN. **Deferred Rendering**. Disponível em: https://cglearn.eu/pub/advanced-computer-graphics/deferred-Rendering. Acesso em: 27 out. 2025.

CHEBANYUK, Olena; MUSHYNSKYI, Mykhailo. **Rendering optimization approach for Game Engine development**. Information Theories and Applications, 2021. DOI: 10.54521/ijita28-02-p04. Disponível em: https://consensus.app/papers/Rendering-optimization-approach-for-game-Engine-chebanyuk-mushynskyi/a41fbb3b37675cdbb2f20c7177ab6406. Acesso em: 7 set. 2025.

ENGEL, Wolfgang (org.). **GPU Pro Series (GPU Pro, GPU Zen)**. Wellesley: A K Peters/CRC Press, 2010–2020.

EPIC GAMES. **Unreal Engine Documentation**. 2024. Disponível em: https://docs.unrealEngine.com/. Acesso em: 18 mar. 2025.

FRIDVALSZKY, András; TÓTH, B. **Multisample Anti-Aliasing in Deferred Rendering**. 2020. DOI: 10.2312/egs.20201008. Disponível em: https://consensus.app/papers/multisample-antialiasing-in-deferred-Rendering-fridvalszky-tóth/b2776f4447f2593380d1091e07382314. Acesso em: 7 set. 2025.

GODOT ENGINE. **Godot Documentation – latest**. 2024. Disponível em: https://docs.godotEngine.org/en/stable/. Acesso em: 18 mar. 2025.

GREGORY, Jason. **Game Engine Architecture**. 3. ed. Boca Raton: CRC Press, 2018.

KE-JIAN, Yang. **Rendering Pipeline optimization of deferred shading**. Modern Computer, 2011. Disponível em: https://consensus.app/papers/Rendering-Pipeline-optimization-of-deferred-shading-ke-jian/4d522627edae594089a7a16f0e728dde. Acesso em: 7 set. 2025.

KENWRIGHT, Ben. **Introduction to Computer Graphics and the Vulkan API**. [S. l.]: Independently published, 2021.

KIM, Youngsik. **Comparison of Rendering speeds and PSNRs of 3D game visual effect techniques based on Deferred Rendering according to screen resolutions**. Journal of the Korea Society of Computer Game, v. 29, n. 2, p. 125-135, 2016. DOI: 10.21493/KSCG.2016.29.2.125. Disponível em: https://consensus.app/papers/comparison-of-Rendering-speeds-and-psnrs-of-3d-game-visual-kim/f5add7d7d17d56b3ad2c6afe7043476c. Acesso em: 7 set. 2025.

KIM, Youngsik. **Comparison of transparency techniques in PSNRs based on Deferred Rendering using multiple Render Targets**. International Journal of Multimedia and Ubiquitous Engineering, v. 11, p. 343-352, 2016. DOI: 10.14257/IJMUE.2016.11.11.31. Disponível em: https://consensus.app/papers/comparison-of-transparency-techniques-in-psnrs-based-on-kim/90647bc6bee5550c9d66fcad42c22990. Acesso em: 7 set. 2025.

LEARNOPENGL. **Model Loading – Loading and Rendering 3D Models**. Disponível em: https://learnopengl.com/Model-Loading/Model. Acesso em: 27 out. 2025.

LEE, Yngdo. **Graphics Pipeline**. Disponível em: https://leeyngdo.Github.io/blog/computer-graphics/2024-02-29-graphics-Pipeline. Acesso em: 27 out. 2025.

LIGHTBEANS. **PBR (Physically Based Rendering) explained: a quick guide**. Disponível em: https://lightbeans.com/en/blog/tutorials/pbr-physically-based-Rendering-explained-a-quick-guide. Acesso em: 27 out. 2025.

OCORNUT. **Dear ImGui – Graphical User Interface Library**. Disponível em: https://github.com/ocornut/imgui. Acesso em: 27 out. 2025.

PETRESCU, A. et al. **Virtual Deferred Rendering**. In: 20th International Conference on Control Systems and Computer Science. 2015. p. 373-378. DOI: 10.1109/CSCS.2015.49. Disponível em: https://consensus.app/papers/virtual-deferred-Rendering-petrescu-moldoveanu/28300e400f00573b914a4f3cb8794e5d. Acesso em: 7 set. 2025.

PETRESCU, L. et al. **Analyzing Deferred Rendering techniques**. Journal of Control Engineering and Applied Informatics, v. 18, p. 30-41, 2016. Disponível em: https://consensus.app/papers/analyzing-deferred-Rendering-techniques-petrescu-moldoveanu/6b6d068bf4b455249c5058f7c896522f. Acesso em: 7 set. 2025.

PHARR, Matt; JAKOB, Wenzel; HUMPHREYS, Greg. **Physically Based Rendering: From Theory to Implementation**. 4. ed. Cambridge: Morgan Kaufmann, 2023.

POVRAY. **Camera Perspective – Documentation for POV-Ray 2.25**. Disponível em: https://www-f9.ijs.si/~matevz/docs/PovRay/pov225.htm. Acesso em: 27 out. 2025.

TECHSOFT3D. **Shadow Mapping – Detailed Description**. Disponível em: https://docs.techsoft3d.com/luminate/latest/book/subjects/bk_re/bk_re_rrd/bk_re_shadow_mapping_detailed.html. Acesso em: 27 out. 2025.

TECHSOFT3D. **Shadow Mapping – Detailed Overview (HOOPS Luminate)**. Disponível em: https://docs.techsoft3d.com/hoops/luminate/book/subjects/bk_re/bk_re_rrd/bk_re_shadow_mapping_detailed.html. Acesso em: 27 out. 2025.

UNITY TECHNOLOGIES. **Unity User Manual**. 2024. Disponível em: https://docs.unity3d.com/Manual/index.html. Acesso em: 18 mar. 2025.

VALDEZ, Willie. **Light Attenuation and Types of Lights**. Disponível em: https://willievaldez.Github.io/2020-10-25-Attenuation/. Acesso em: 27 out. 2025.