UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Entre Teoria e Prática: Um Estudo sobre Pipelines de Visualização 3D e suas Divergências

Trabalho de Conclusão de Curso

Marcos Augusto Campagnaro Mucelini

Cascavel–PR

2025

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Marcos Augusto Campagnaro Mucelini

Entre Teoria e Prática: Um Estudo sobre Pipelines de Visualização 3D e suas Divergências

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel.

Orientador(a): Adair Santa Catarina

Cascavel–PR

2025

MARCOS AUGUSTO CAMPAGNARO MUCELINI

ENTRE TEORIA E PRÁTICA: UM ESTUDO SOBRE PIPELINES DE VISUALIZAÇÃO 3D E SUAS DIVERGÊNCIAS

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Adair Santa Catarina (Orientador)
Colegiado de Ciência da Computação, UNIOESTE

Prof. Edmar André Bellorini
Colegiado de Ciência da Computação, UNIOESTE

Prof. Josué Castro
Colegiado de Ciência da Computação, UNIOESTE

Cascavel, 11 de Março de 2025

Aos meus amigos que estiveram ao meu lado, especialmente ao Matheus, Nathalia, Greice e Barbara

Agradecimentos

Agradeço primeiramente ao professor Carlos José Maria Olguin, a pessoa que esteve ao meu lado como mentor e amigo no início de minha caminhada como cientista da computação; agradeço também ao professor Adair Santa Catarina por me orientar neste trabalho e aos demais professores que dispuseram tempo e conhecimento para que pudesse ter as ferramentas necessárias para escrever este trabalho. Agradeço à Unioeste por me proporcionar bolsas de iniciação científica e extensão durante boa parte de minha graduação. Agradeço a todos os camaradas da Unidade Popular, Jornal A Verdade e Soberana, por contribuírem na minha radicalização com o despertar de minha consciência de classe na ciência imortal do proletariado. E agradeço especialmente aos meus melhores amigos Nathalia Gomes e Matheus de Souza, amigos e camaradas que estiveram ao meu lado em diversos momentos críticos da minha vida. Sem estes dois em minha vida, não a teria mais, e não teria tido forças para lutar pelo justo e pelo correto.

Os chefes da social-democracia encobriram e ocultaram às massas o verdadeiro caráter de classe do fascismo e não lutaram contra as medidas reacionárias cada vez mais graves da burguesia. Sobre eles pesa grande responsabilidade histórica, pelo fato de, nos momentos decisivos da ofensiva fascista, uma parte considerável das massas trabalhadoras da Alemanha e de diversos outros países fascistas não reconhecer no fascismo a fera sedenta de sangue do capital financeiro, seu pior inimigo, e destas massas não estarem preparadas para fazer-lhe frente. (Georgi Dimitrov)

Resumo

Este trabalho apresenta um estudo comparativo entre dois modelos distintos de *pipelines* de visualização 3D, abordando suas diferenças teóricas e práticas, com foco nas nuances que surgem ao implementar as etapas geométricas e de rasterização. O objetivo principal é identificar e analisar as variações nos métodos descritos pela literatura, discutindo como essas escolhas afetam a fidelidade visual e a eficiência computacional. Para tanto, o estudo utiliza o *pipeline* proposto por Santa Catarina e uma versão simplificado do *pipeline* proposto por Alvy Ray Smith, realizando uma implementação prática e uma análise comparativa detalhada. Além de documentar as decisões de desenvolvimento, o trabalho propõe uma ferramenta didática para alunos de Computação Gráfica. Os resultados obtidos mostram divergências literárias e metodológicas, ressaltando o impacto das subjetividades envolvidas em implementações computacionais. O estudo conclui com reflexões sobre a aplicação prática das teorias e suas implicações para a visualização realista de cenas 3D, propondo direções futuras para aprimoramento e pesquisa.

Palavras-chave: *Pipeline* de visualização 3D, Computação Gráfica, Rasterização, Análise comparativa.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

CG	(Computação Gráfica)
MVC	(Model-View-Controller)
FOV	(Field of View)
CAD	(Computer Aided Design)
SRU	(Sistema de referencia do universo)
SRC	(Sistema de referencia da câmera)
SRT	(Sistema de referencia da tela)
SRN	(Sistema de referencia normalizado)
VRP	(View Reference Point)
MSB	(Most significant bit)
VSCode	(Visual Studio Code)
IDE	(Interface de desenvolvimento)
POO	(Programação orientada a objetos)
STL	(tandard Template Library)
SDL2	(Simple DirectMedia Layer 2)
MSVC	(Microsoft Visual C++ Compiler)
API	(Application Programming Interface)
LLVM	(Low Level Virtual Machine)
LTS	(Long-term support)
GUI	(Graphical user interface)
GTest	(Google Test)
MVC	(Model view controller)
FPS	(Frames per second)
CV	(Coeficiente de Variação)
ANOVA	(Analysis of variance)

Lista de símbolos

$\in$	Pertence
$\emptyset$	Vazio
Δ	Variação
σ	Sigma

Sumário

1 Introdução

Assim como os artistas renascentistas aperfeiçoavam técnicas de luz e sombreamento para alcançar a perfeição na representação do mundo, a computação gráfica persegue um objetivo semelhante. Um marco importante neste campo foi o trabalho de Alvy Ray Smith que, em *The Viewing Transformation* Smith (1983), propôs um modelo inovador de visualização para representar cenas e objetos 3D em uma tela de computador. Contudo, da mesma forma que os artistas enfrentam desafios ao transportar uma cena para a tela, os cientistas da computação lidam com limitações físicas dos pixels, que impõem restrições específicas, de maneira análoga ao processo artístico.

A computação gráfica atualmente permeia diversas áreas, desde jogos eletrônicos a aplicações médicas que produzem modelos tridimensionais detalhados do corpo humano. Com essa expansão e diversificação nas aplicações da CG, surge uma vasta gama de requisitos e desafios técnicos. Por exemplo, um jogo de computador moderno pode necessitar de renderização em tempo real, com ênfase em efeitos visuais dinâmicos, enquanto uma aplicação médica pode priorizar a precisão e a fidelidade dos detalhes em uma reconstrução 3D.

A produção de cenas sintéticas em 3D segue um processo algorítmico bem fundamentado na teoria, conhecido como *pipeline* de visualização 3D. Na prática, no entanto, a implementação computacional do *pipeline* revela nuances oriundas em divergências apresentadas pela literatura, bem como em decisões do desenvolvedor.

O *pipeline* de visualização consiste em uma estrutura organizada e sequencial de processos que transformam dados tridimensionais em imagens bidimensionais visíveis em telas ou impressões. Esse processo envolve uma série de etapas, incluindo modelagem, transformações geométricas, iluminação, mapeamento e rasterização. A função primordial do *pipeline* é traduzir a informação tridimensional de objetos e cenários em uma representação visual coerente e realista, permitindo que os espectadores interajam com ambientes virtuais ou desfrutem de conteúdo gráfico em filmes, jogos, simulações médicas e outras aplicações.

Como mencionado anteriormente, a implementação de um *pipeline* de visualização 3D admite personalizações. Diante desse cenário, destaca-se a relevância de se estudar e compreender diferentes modelos de *pipeline*, em seus algoritmos e métodos matemáticos, com o intuito de explorar a corretude e a eficiência computacional de diferentes alternativas. Essas investigações nos permitirão entender mais profundamente as divergências e determinar suas vantagens quanto ao desempenho e fidelidade visual nos resultados obtidos.

Como exemplo de diferença em fontes literárias temos um problema comum na computação gráfica, que é o cálculo do vetor médio unitário empregado nos modelos de iluminação propostos por Gouraud (1971) e Phong (1973).

O método matemático descrito para se obter o vetor unitário médio em um vértice compartilhado por diversas faces é, segundo Foley et al. (1995), calculado pela equação 1, onde o vetor unitário médio $\hat{N}$ em um vértice **V** qualquer é dado pela soma dos vetores normais $\vec{N}_i$ das faces que compartilham o vértice **V**, seguido de sua normalização.

$$\hat{N} = \frac{\sum_{i=1}^{n} \vec{N}_i}{|\sum_{i=1}^{n} \vec{N}_i|} \tag{1}$$

Entretanto Conci, Azevedo e Leta (2003) apresentam que o vetor normal unitário médio num vértice **V** qualquer é calculado pela equação 2, o que corresponde à média simples das componentes dos **k** vetores normais das faces que compartilham o vértice.

$$\hat{N} = \frac{\sum_{i=1}^{k} \vec{N}_i}{k} \tag{2}$$

Além desta diferença de cálculo, não está claro na literatura se os vetores normais das faces consideradas no cálculo do vetor normal unitário médio em $\vec{V}$ devem ou não estar normalizados.

Ademais, a expectativa de corretude total nos processos algorítmicos de síntese de cenas 3D pode ser equivocada. Observa-se, na realidade, uma corretude parcial: algoritmos podem gerar saídas distintas, mesmo com entradas idênticas (objetos modelados), dependendo dos métodos utilizados na construção da cena e ou decisões do desenvolvedor. Vale ressaltar que, embora os resultados obtidos por diferentes métodos possam divergir, eles ainda podem ser considerados válidos dentro de seus respectivos contextos, de acordo com os requisitos impostos para a construção de uma cena. Adicionalmente, o processo de conversão de uma cena 3D em uma imagem 2D gera, por consequência, uma perda de realismo, já que simulamos o 3D através de perspectivas e pontos de fuga, semelhante ao processo que um artista usa para realizar uma pintura.

Além disso, a subjetividade inerente aos métodos de visualização 3D contribui para o desafio da replicabilidade em pesquisas científicas. Em muitos campos, a replicabilidade é

um critério essencial para validar a robustez e a credibilidade de um estudo. No entanto, na computação gráfica, a replicabilidade pode se tornar problemática, pois pequenas variações nos métodos ou nas implementações podem resultar em diferenças, ainda que sutis, nos resultados. Isso se potencializa caso o desenvolvedor não tenha discorrido acerca das suas decisões ao longo do processo de implementação do *pipeline* de visualização 3D.

Em diversos casos, pesquisadores podem seguir metodologias descritas em artigos, mas, sem uma compreensão plena dos aspectos subjetivos e das decisões tomadas durante a implementação, os resultados replicados podem diferir dos originais. Essas discrepâncias não apenas questionam a validade dos resultados, como também podem dificultar o avanço do conhecimento, uma vez que os esforços subsequentes podem se basear em uma compreensão incorreta dos métodos empregados.

1.1 Objetivos

1.1.1 Objetivo geral

Este trabalho aborda a implementação e comparação de dois modelos distintos de *pipeline* de visualização, com o objetivo de identificar as diferenças entre os métodos descritos na literatura, implementá-los e realizar análises comparativas para avaliar se as variações nos resultados são significativas. Além disso, o estudo aborda a subjetividade presente na aplicação prática das teorias, destacando as diferenças notáveis entre os métodos propostos na literatura e os resultados práticos obtidos.

1.1.2 Objetivos específicos

Este trabalho tem como objetivos específicos documentar as decisões tomadas durante a fase de desenvolvimento, criando um guia de implementação dos *pipelines*. Também se propõe a desenvolver uma ferramenta didática que possa ser utilizada por alunos da disciplina de computação gráfica do curso de Ciência da Computação na Unioeste, como suporte em seu aprendizado.

1.2 Estrutura do Texto

Além deste capítulo introdutório, que apresenta e contextualiza o problema e define os objetivos, o presente trabalho está dividido em mais **sete capítulos**. O Capítulo 2 fornece o referencial teórico utilizado nesta monografia, abordando conceitos essenciais de geometria

analítica e álgebra linear, bem como as etapas do *pipeline* de visualização e seus processos intermediários, incluindo alguns aspectos de implementação.

O Capítulo 3 descreve os materiais necessários e o ambiente no qual os testes foram realizados, oferecendo uma visão detalhada das condições experimentais.

O Capítulo 4 explora a metodologia adotada no desenvolvimento do projeto, destacando as principais etapas de implementação, assim como a definição das métricas utilizadas para avaliar os testes. Já o Capítulo 5 é dedicado à apresentação e discussão dos resultados obtidos, com exposições textuais e visuais, por meio de imagens.

O Capítulo 6 apresenta as conclusões do trabalho, Por fim o capitulo 7 apresenta sugestões de trabalhos a fim de complementar este estudo.

2

Referencial Bibliográfico

O processo de visualização 3D é uma jornada complexa e multifacetada que se desdobra em duas etapas principais, **cálculos geométricos** e **rasterização** Foley et al. (1995). Os cálculos geométricos envolvem a modelagem matemática e as transformações geométricas de transformação de câmera, projeção e mapeamento; a rasterização inclui a ocultação de superfícies, os cálculos de iluminação e o recorte 2D, culminando com o desenho propriamente dito em um reticulado de pixels. Cada etapa tem sua importância única e, embora possam existir variações e simplificações, especialmente nas etapas intermediárias, a sequência dessas fases é crucial para garantir uma representação visual eficaz e realista.

Todos estes processos empregam conceitos elementares de geometria analítica e álgebra linear. Como o objetivo deste trabalho é realizar uma comparação entre modelos de *pipeline* distintos, os fundamentos necessários em geometria analítica e álgebra linear encontram-se disponíveis no Anexo A.

2.1 Modelagem 3D

A modelagem 3D consiste no conjunto de técnicas utilizadas para a criação de representações tridimensionais de objetos e ambientes em espaços virtuais. Sua aplicação abrange diversas áreas como design, engenharia, entretenimento e simulação científica. Fundamentalmente, a modelagem 3D se baseia em representações matemáticas para descrever as formas geométricas de um objeto, utilizando-se de estruturas como malhas poligonais, superfícies NURBS (*Non-Uniform Rational B-Splines*) e técnicas de subdivisão Hearn e Baker (1997).

O processo de criação de objetos 3D envolve a aplicação de transformações geométricas, tais como translações, rotações e escalonamentos, com o objetivo de definir a posição e o tamanho dos objetos no espaço tridimensional Foley et al. (1995). Adicionalmente, o uso de modelos

hierárquicos é frequente na organização da estrutura de objetos complexos, o que facilita a manipulação de componentes em sistemas mais sofisticados Hearn e Baker (1997).

A representação de objetos tridimensionais pode ser implementada por meio de diversas técnicas. Detalhes sobre a geração procedural de formas geométricas simples, técnica empregada neste trabalho, podem ser consultados no Apêndice A.

2.2 Malhas Poligonais

As malhas poligonais são a técnica mais comum para representação de objetos tridimensionais, especialmente em aplicações que exigem renderização em tempo real, como videogames e simulações interativas. Uma malha poligonal consiste em uma rede de polígonos — geralmente triângulos ou quadriláteros — que define a superfície do objeto. Cada polígono é composto por vértices conectados por arestas, formando faces planas que se unem para formar a geometria completa Hearn e Baker (1997).

Uma das práticas mais comuns na construção de malhas é a **triangulação** processo que subdivide a superfície em triângulos. A vantagem dessa abordagem é que os triângulos garantem uma representação consistente de superfícies, incluindo aquelas com curvaturas complexas, e são altamente compatíveis com a maioria dos hardwares gráficos, que são otimizados para processar triângulos de maneira eficiente Akenine-Mooller et al. (2018). Esse método é amplamente adotado em gráficos de tempo real, onde a velocidade de processamento é essencial para o desempenho da aplicação

No entanto, malhas construídas com um número reduzido de triângulos podem resultar em superfícies com baixa resolução, comprometendo a suavidade visual. Para resolver essa questão, utilizam-se técnicas de **subdivisão de superfícies**, que aumentam o número de polígonos, aprimorando o nível de detalhe e suavidade da malha. Métodos como Catmull-Clark e Loop são especialmente populares em modelagem de personagens e ambientes detalhados, sendo amplamente aplicados em animação e efeitos visuais para adicionar realismo às superfícies Foley et al. (1995).

Além da subdivisão, a ténica de **nível de detalhe (LoD)** desempenha um papel crucial na eficência de renderização de malhas poligonais. O LoD permite ajustar dinamicamente a quantidade de polígonos de um objeto com base na distância entre o objeto e a câmera, reduzindo a complexidade da malha quando o objeto está distante e aumentando-o quando ele está próximo. Essa adaptação dinâmica é fundamental para otimizar o desempenho de cenas complexas, equilibrando a carga de processamento com a necessidade visual Hearn e Baker (1997).

Embora de as malhas poligonais ofereçam uma solução eficiente para a representação de objetos tridimensionais, elas podem apresentar limitações ao retratar superfícies curvas ou detalhadas, a menos que um número um número significativo de polígonos seja utilizado.

Nesse contexto, as técnicas de subdivisão, juntamento com o uso de texturas são essenciais para aumentar o realismo visual das superfícies sem sobrecarregar o processamento, permitindo que as malhas poligonais continuem sendo uma escolha versátil e eficiente para uma ampla variedade de aplicações.

2.3 Armazenamento de Dados e Estruturas para Malhas Poligonais

As malhas poligonais constituem uma das representações mais difundidas para objetos tridimensionais, particularmente em aplicações que demandam renderização em tempo real. Elas são definidas por um conjunto de vértices, arestas e faces que, em conjunto, delimitam a superfície do objeto. A complexidade dessas malhas pode variar significativamente, desde representações simples até modelos altamente detalhados. Essa variação impõe a necessidade de estruturas de dados eficientes para assegurar o desempenho na manipulação e renderização, conforme discutido em Hearn e Baker (1997).

A eficiência na manipulação de malhas poligonais é intrinsecamente ligada às estruturas de dados empregadas para organizar seus componentes: vértices, arestas e faces. Dentre as diversas estruturas disponíveis, a estrutura ***half-edge*** (meia-aresta) se destaca pela sua capacidade de representar e navegar pelas conexões entre vértices e arestas de forma eficiente.

2.3.1 Estrutura Half-Edge

A estrutura *half-edge* é uma representação amplamente utilizada para malhas poligonais, especialmente em malhas com muitos triângulos ou polígonos complexos. Nessa estrutura, cada aresta é dividida em duas "meias-arestas" (*half-edges*), com cada meia-aresta apontando para um dos vértices adjacentes e conectada à face da qual é fronteira (Akenine-Moller et al. (2018), Hearn e Baker (1997)). Essa organização permite realizar operações como percorrer faces adjacentes, encontrar vizinhos e modificar a topologia da malha, de maneira eficiente. A Figura 1 apresenta uma representação visual da estrutura, na qual observamos quatro vértices interligados por linhas **azuis** e **vermelhas**, que representam as meias-arestas. Cada elemento da estrutura possui um identificador único: vértices (v_1, v_2, v_3 e v_4), arestas ($e_0, e_1, ..., e_9$) e faces (f_0 e f_1).

Na tabela 1, podemos ver como as meias-arestas são organizadas, onde cada meia-aresta armazena referências à sua gêmea, à face incidente, e às meias-arestas "anterior" e "próxima" ao longo da mesma face ou de um buraco. A estrutura permite busca eficiente por elementos em tempo constante $O(1)$. Esta representação simplifica a navegação na estrutura e melhora o desempenho em operações de busca e modificação topológica.

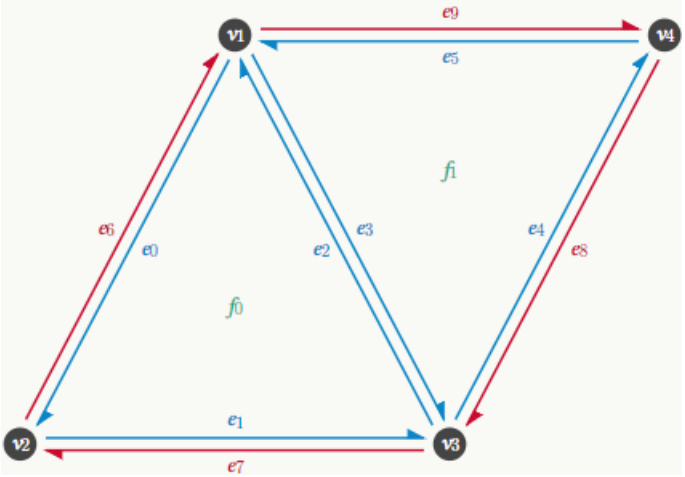


Figura 1 – Exemplo de representação visual da estrutura *half-edge*.

Tabela 1 – Lista de meias-arestas.

Meia-aresta	Origem	Gêmeo	Face incidente	Próximo	Anterior
e0	v1	e6	f0	e1	e2
e1	v2	e7	f0	e2	e0
e2	v3	e3	f0	e0	e1
e3	v1	e2	f1	e4	e5
e4	v3	e8	f1	e5	e3
e5	v4	e9	f1	e3	e4
e6	v2	e0	∅	e9	e7
e7	v3	e1	∅	e6	e8
e8	v4	e4	∅	e7	e9
e9	v1	e5	∅	e8	e6

Na tabela 1, algumas meias-arestas contêm o símbolo ∅ (vazio), que representa um ponteiro nulo, utilizado para indicar arestas de fronteira — arestas sem face adjacente ou que delimitam buracos na malha. A implementação de arestas de fronteira na estrutura *half-edge* pode seguir duas abordagens: usar uma única meia-aresta com ponteiro de gêmeo nulo ou empregar um par de meias-arestas, sendo que a meia-aresta que define a fronteira possui um ponteiro de face nulo (∅). A segunda abordagem facilita a codificação da estrutura por eliminar a necessidade de condições especiais, tratando automaticamente todas as meias-arestas com gêmeos não nulos.

Tabela 2 – Lista de faces.

Face	Meia-aresta incidente
f0	e0
f1	e3

Para manter a estrutura eficiente e organizada, cada vértice armazena sua posição e uma referência a uma meia-aresta incidente, como ilustrado na tabela 3. Esse design simplifica o

acesso aos vértices e facilita a navegação pela estrutura. Adicionalmente, conforme mostrado na tabela 2, cada face armazena uma referência a uma meia-aresta arbitrária pertencente ao seu perímetro, o que possibilita uma navegação consistente e eficiente dentro da estrutura *half-edge*.

A estrutura *half-edge* oferece diversas vantagens em relação a representações mais simples, como listas explícitas de vértices e arestas. Sua organização facilita operações complexas, como a inserção de novos vértices, remoção de arestas e subdivisão de faces, ao mesmo tempo em que permite consultas em tempo constante. Essa eficiência garante um meio consistente de percorrer as arestas e faces de um objeto, o que é particularmente útil em aplicações que exigem modificações topológicas frequentes, como subdivisão de superfícies e simplificação de malhas Akenine-Mooller et al. (2018).

Outro ponto essencial da estrutura *half-edge* é a sua capacidade de organizar a **topologia** da malha. Cada meia-aresta armazena referências para o vértice de origem, a face à qual pertence e a meia-aresta gêmea na face adjacente. Essa configuração permite a navegação eficiente pela malha e é especialmente vantajosa em algoritmos de sombreamento como Gouraud (1971) e Phong (1973), onde a conectividade topológica entre faces e arestas influencia diretamente a aplicação do sombreamento suave.

Tabela 3 – Lista de vértices.

Vértice	**Coordenadas**	**Meia-aresta incidente**
v1	(2, 4, 0)	e0
v2	(0, 0, 0)	e1
v3	(4, 0, 0)	e2
v4	(6, 4, 0)	e5

Na *half-edge* as informações de vértices, arestas e faces são armazenadas em vetores na memória. Os valores de cada elemento são mantidos uma única vez e referenciados conforme necessário, minimizando a redundância e tornando a implementação mais eficiente. Cada vértice possui uma referência a uma meia-aresta incidente, enquanto as faces fazem referência a uma meia-aresta que define seu perímetro Foley et al. (1995).

O uso da estrutura *half-edge* oferece uma abordagem flexível e eficiente para manipulação de malhas poligonais, sendo ideal para renderizações em tempo real, onde o rápido acesso à conectividade e à topologia da malha é crucial. Essa estrutura otimizada possibilita a realização de modificações topológicas complexas de forma ágil, permitindo alto desempenho em simulações gráficas e aplicações interativas Akenine-Mooller et al. (2018).

2.4 Elementos de uma Cena 3D

A descrição de uma cena 3D consiste da especificação de seus elementos, de como eles estarão organizados, além de informações de exibição (configurações). Os componentes de uma cena 3D são os objetos que habitam uma representação virtual. De acordo com Velho e Gomes (2007) estes objetos se enquadram em 3 categorias:

Modelos de Objetos - que descrevem a geometria, a topologia e as propriedades visuais dos objetos.

Fontes de Luz - que descrevem a iluminação.

Câmera virtual - que descreve o observador.

A estrutura dos objetos em uma cena 3D corresponde a grupos de objetos que possuem relações entre si, podendo ser estruturados de forma hierárquica. Já as informações de exibição são compostas por vários outros parâmetros que especificam como a descrição da cena pode ser manipulada. Alguns exemplos que serão abordados com maior profundidade, posteriormente, são: tipo de projeção, cor(es) da(s) luz(es), posição do observador e modelo de iluminação.

2.4.1 Objetos 3D

Os **objetos 3D** são a base de qualquer cena tridimensional, representando tanto os elementos visíveis quanto os elementos funcionais da cena. Esses objetos podem variar em complexidade, desde formas geométricas simples como cubos, esferas, cilindros e cones, até modelos detalhados de personagens, edificações ou ambientes virtuais inteiros. Cada objeto 3D é descrito por um conjunto de atributos que especificam sua geometria, topologia e propriedades de materiais.

Cada objeto 3D também pode ser manipulado por uma série de transformações geométricas como translação, rotação e escalonamento, permitindo que os objetos sejam posicionados, orientados e redimensionados dentro da cena. Essas transformações são aplicadas através de matrizes de transformação que operam no espaço tridimensional Smith (1983).

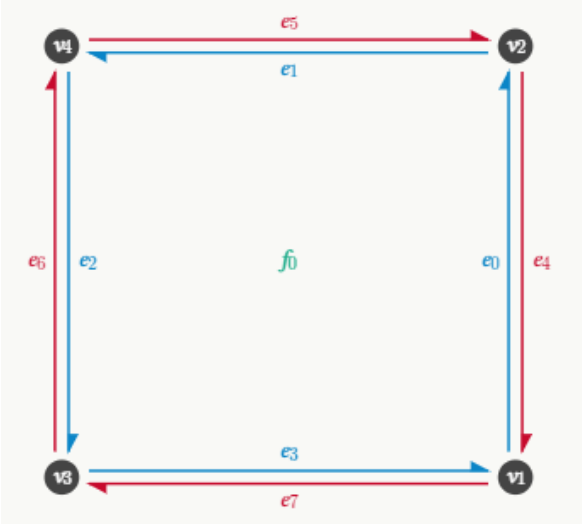
2.4.1.1 Geometria e Topologia

Geometria é a descrição matemática da forma de um objeto, definida por vértices, arestas e faces. Em uma representação geométrica, os objetos 3D são frequentemente modelados como malhas poligonais, nas quais as superfícies são subdivididas em polígonos (geralmente triângulos) para otimizar a renderização. A figura 2 ilustra esse conceito, mostrando uma malha

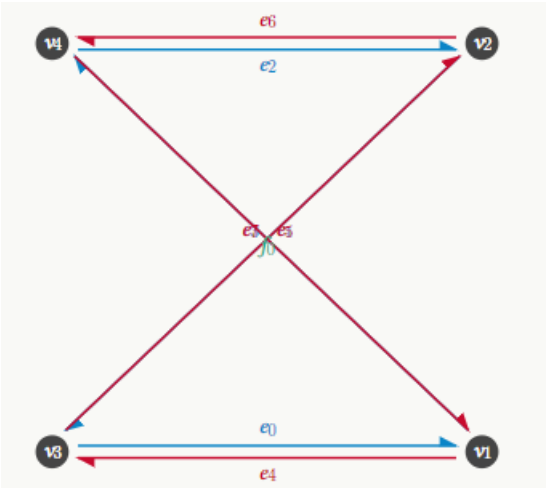
quadrilateral composta por vértices (marcados pelos círculos cinzas). Cada vértice possui uma posição específica, contribuindo para a forma geral do objeto Hearn e Baker (1997).

Figura 2 – Exemplo de uma malha quadrilateral representada geometricamente por seus vértices.

Entretanto, a geometria por si só não fornece todas as informações necessárias para determinar a configuração exata do objeto. Para isso, também é preciso considerar a **topologia**.



(a) Exemplo de uma malha quadrilateral com uma única face f_0.



(b) Exemplo da mesma malha quadrilateral com uma única face f_0 porém com uma topologia alterada.

Figura 3 – Comparação de topologias de malhas quadrilaterais.

A **topologia** se refere à conectividade entre os elementos geométricos do objeto, ou seja, como os vértices, arestas e faces estão interconectados para formar a estrutura da malha. A topologia define a "estrutura" ou "esqueleto" do objeto, independentemente de sua forma geométrica ou tamanho, sendo essencial para operações como subdivisão de superfícies e manipulação da malha. Na figura 3a vemos uma representação de uma malha quadrada com uma única face (representada por f_0), enquanto a figura 3b mostra a mesma malha, mas com topologia distinta da figura 3a. Isso exemplifica como a topologia define a forma geométrica

sem alterar o número de faces ou vértices, apenas modificando a conectividade dos elementos Akenine-Moller et al. (2018).

A topologia é particularmente importante em animações e modelagens complexas, onde manter a conectividade das faces e arestas é crucial para preservar a integridade da forma durante a manipulação. Assim, enquanto a geometria determina a posição e a forma dos vértices e arestas, a topologia define as relações e conexão entre esses elementos, garantido a consistência estrutural do objeto.

2.4.1.2 Materiais

Além da geometria e da topologia, os **materiais** também são uma parte essencial da representação de um objeto 3D, pois é através destes parâmetros que podemos controlar a reflexão, opacidade e brilho de um objeto Foley et al. (1995). Estes parâmetros influenciam também como cada componente da luz (azul, vermelho e verde) interage com o objeto, permitindo criar uma ampla variedade de objetos, com características como maior polimento (reflexão e brilho) ou opacidade, por exemplo.

Para representar os materiais de um objeto, são utilizados coeficientes de reflexão ambiente (***ka***), coeficiente de reflexão difusa (***kd***) e coeficiente de reflexão especular (***ks***) variando de 0 (nenhuma luz refletida) a 1 (toda a luz refletida) Hearn e Baker (1997). Além disso temos um número real que controla o brilho do objeto (***n***) Akenine-Moller et al. (2018).

2.4.1.2.1 Coeficientes de reflexão e Brilho

A interação da luz com os materiais de um objeto tridimensional é controlada por diferentes coeficientes de reflexão, que determinam como cada tipo de luz será refletido pela superfície. Esses coeficientes permitem que um modelo de iluminação seja aplicado ao objeto, conferindo realismo e diversidade visual em função das características de cada material.

Primeiramente, o **coeficiente de reflexão ambiente**, denotado como K_a, define a quantidade de luz ambiente refletida uniformemente em toda a superfície do objeto. Esse coeficiente é uma constante que independe da posição da fonte de luz e é essencial para evitar que objetos em áreas com pouca iluminação fiquem completamente escuros, garantindo uma base de iluminação geral Foley et al. (1995).

Além disso, o **coeficiente de reflexão difusa** K_d controla a quantidade de luz que é refletida de forma difusa pela superfície. Ao contrário da luz ambiente, a luz difusa depende da orientação da superfície em relação à fonte de luz, mas é independente da posição do observador. Essa reflexão difusa cria uma aparência suave e é fundamental para simular materiais opacos, que dispersam a luz uniformemente em várias direções Hearn e Baker (1997).

Já o **coeficiente de reflexão especular** K_s é responsável pela quantidade de luz refletida de forma especular em superfícies brilhantes, como metais e vidros. Esse tipo de reflexão depende da posição do observador e cria os pontos de brilho (ou *highlights*) característicos desses materiais. Superfícies com alta reflexão especular tendem a parecer mais brilhantes e reflexivas, contribuindo para um efeito visual mais realista Akenine-Moller et al. (2018).

Para controlar a concentração do brilho especular, utilizamos o **parâmetro de brilho** n, que ajusta a distribuição espacial da luz especular refletida. Valores elevados de n produzem reflexos mais concentrados, típicos de materiais como metais, enquanto valores baixos geram reflexos mais difusos, comuns em superfícies não metálicas e opacas Santa Catarina (2024).

A representação dos materiais também pode variar de acordo com o tipo de luz utilizado. No caso de uma luz monocromática, cada coeficiente de reflexão – K_a, K_d e K_s – é representado por um único valor de intensidade. Em cenários com luz colorida, onde são consideradas as componentes RGB (vermelho, verde e azul), cada coeficiente é expandido para uma tupla com três valores, permitindo que cada componente de cor seja ajustada individualmente para simular com maior precisão as características do material.

Esses coeficientes desempenham um papel fundamental na simulação de uma variedade de materiais como plásticos, metais, madeira e superfícies brilhantes ou opacas, ajustando a aparência do objeto em diferentes condições de iluminação. Além disso, o controle das componentes RGB permite que a cor do objeto seja influenciada pela interação entre as luzes da cena e as propriedades do material.

Para enriquecer ainda mais as características visuais, texturas podem ser aplicadas às superfícies dos objetos. O mapeamento de texturas utiliza coordenadas UV, associando pontos da textura bidimensional aos vértices da superfície tridimensional. Dessa forma, padrões e cores podem ser aplicados com precisão em áreas específicas da geometria, proporcionando detalhes adicionais e aprimorando o realismo do objeto Foley et al. (1995).

2.4.2 Modelos de Luz e Iluminação

A iluminação é um dos elementos mais importantes em uma cena 3D, pois define a interação dos objetos com as fontes de luz, influenciando diretamente a percepção visual do observador. Os modelos de iluminação simulam o comportamento da luz ao atingir superfícies, determinando a geração de cores, sombras e reflexos. Dentre as diversas classificações de fontes de luz, destacam-se a **lâmpada omnidirecional** (ou luz pontual) e a **luz ambiente**, cada qual com um papel distinto na composição da cena. A lâmpada omnidirecional, por exemplo, emite luz em todas as direções a partir de um único ponto no espaço 3D, simulando fontes de luz como lâmpadas incandescentes Hearn e Baker (1997). Já a luz ambiente, por sua vez, não tem posição definida no espaço e provê uma iluminação uniforme e não direcional, preenchendo as sombras e suavizando o contraste na cena.

A modelagem das luzes em uma cena 3D exerce influência direta na percepção dos objetos e do ambiente como um todo. Nesse contexto, dois modelos de luz amplamente utilizados são as luzes **monocromáticas** e as luzes baseadas no modelo de cores **RGB**.

2.4.2.1 Luzes Monocromáticas

As luzes monocromáticas representam a luz utilizando apenas uma intensidade, variando do preto (ausência de luz) ao branco (intensidade máxima), sem considerar variações de cor. São fontes de luz que emitem apenas um comprimento de onda, resultando em uma cor única e pura. Esse tipo de iluminação é menos comum em simulações visuais realistas, mas pode ser usado em cenas que necessitam de um controle preciso sobre a cor, como em simulações científicas ou experimentos com luz e cor. No caso da luz monocromática a intensidade de iluminação é uniforme em toda a cena, e os objetos assumem tonalidades baseadas apenas nessa única frequência de luz Hearn e Baker (1997).

2.4.2.2 Luzes RGB

As luzes RGB (*Red*, *Green*, *Blue*) são o padrão na computação gráfica para a geração de cores. Através da combinação de diferentes intensidades de luz vermelha, verde e azul é possível criar uma vasta gama de cores. Este modelo é baseado no princípio da síntese aditiva de cores, onde as três cores primárias são combinadas para gerar qualquer outra cor perceptível pelo olho humano Akenine-Mooller et al. (2018). Esse modelo é amplamente utilizado em cenas 3D, pois simula a luz do mundo real que contem todas as cores do espectro de luz visível.

Cada componente de cor no modelo RGB é representado por um valor de intensidade que varia normalmente de 0 a 255 (ou 0.0 a 1.0, em valores normalizados). A combinação desses três componentes resulta em diferentes cores. Por exemplo, ao combinar valores máximos de todos os componentes (R=255, G=255, B=255) obtemos luz branca. Se todos os componentes forem 0, o resultado é a ausência total de luz, ou seja, preto. A combinação parcial desses componentes pode resultar em uma infinidade de outras cores Foley et al. (1995).

2.4.2.3 Luz Ambiente

A **luz ambiente** é uma fonte de luz amorfa, distribuída uniformemente por toda a cena, iluminando todos os objetos de forma constante, independentemente de suas posições ou orientações, conforme mostra a figura 4. Essa luz simula a iluminação indireta, que resulta da reflexão e dispersão da luz em outras superfícies antes de atingir o objeto. Sua função é garantir que nenhuma parte da cena fique completamente escura, mesmo sem uma fonte de luz direta. Esse efeito é especialmente importante para manter o realismo em cenas que envolvem renderização Hearn e Baker (1997).

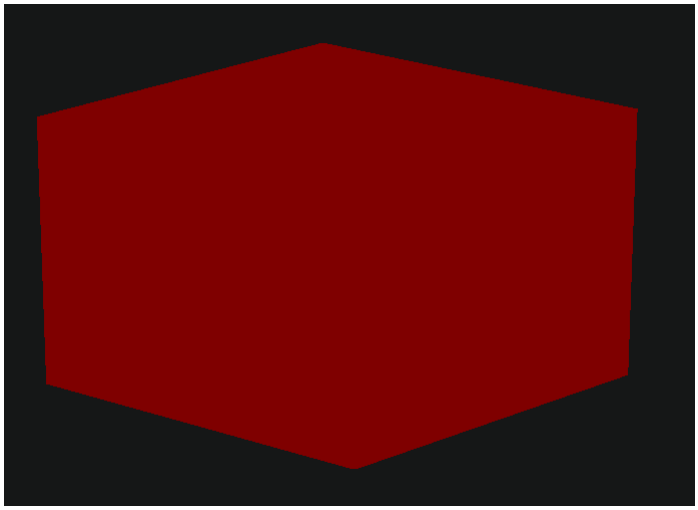


Figura 4 – Exemplo de iluminação ambiente

A intensidade da luz ambiente que um objeto reflete depende de seus coeficientes de reflexão ambiente (**Ka**). Esses coeficientes controlam a quantidade de luz ambiente refletida pela superfície do objeto, variando de 0 (nenhuma luz refletida) a 1 (toda a luz ambiente refletida) Hearn e Baker (1997). A equação para calcular a luz ambiente refletida é dada pela equação 3:

$$I_{\text{amb}} = K_a \cdot I_a \tag{3}$$

Onde:

I_{amb} é a intensidade da luz ambiente refletida pelo objeto.

K_a é o coeficiente de reflexão ambiente do material.

I_a é a intensidade da luz ambiente global na cena.

Conforme a figura 4 demonstra, a luz ambiente é constante, não produz sombras ou brilhos e ilumina homogeneamente todo o objeto; ela é essencial para fornecer o efeito de iluminação indireta, ou seja, a luz refletida nas paredes e/ou outras superfícies de objetos na cena. Ela é geralmente somada com outras componentes da iluminação, como a reflexão difusa e especular, para criar maior realismo à cena 3D Hearn e Baker (1997).

2.4.2.4 Reflexão Difusa

Na reflexão difusa, apresentado na figura 5 a luz incidente sobre uma superfície rugosa é dispersa igualmente em todas as direções. Esse comportamento simula a forma como a luz interage com materiais opacos e não reflexivos como madeira ou pedra, conferindo às superfícies um sombreamento suave e uniforme.

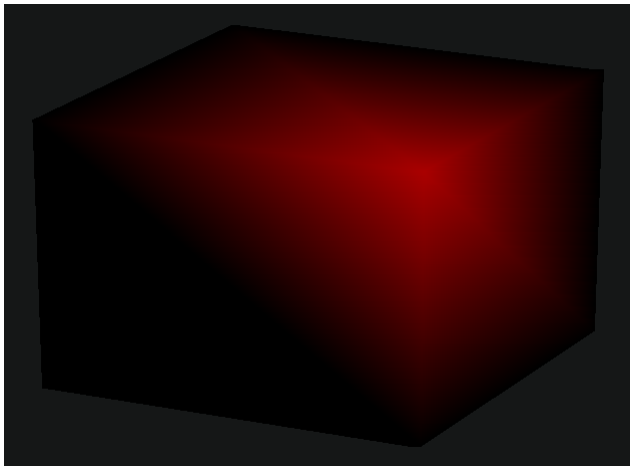


Figura 5 – Exemplo de reflexão difusa

A intensidade da luz difusa refletida por uma superfície depende do ângulo de incidência da luz (θ), sendo máxima quando a luz incide perpendicularmente à superfície. Esse comportamento é descrito pela **Lei de Lambert**, que estabelece que a intensidade da luz refletida é proporcional ao cosseno do ângulo formado entre o vetor de luz e a normal da superfície Foley et al. (1995), Hearn e Baker (1997). A figura 6 ilustra esse princípio.

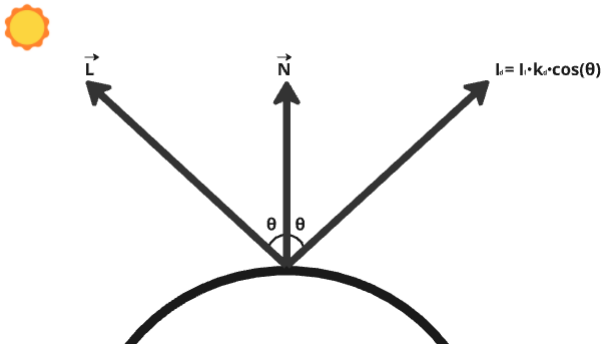


Figura 6 – Ângulo de incidência e a Lei de Lambert

A reflexão difusa pode ser calculada usando a equação 4 sobre um ponto da superfície, com base na posição e intensidade da luz, nos coeficientes de reflexão difusa do material e na orientação da superfície.

$$I_d = K_d \cdot I_l \cdot (\hat{N} \cdot \hat{L}) \tag{4}$$

Onde:

I_d é a intensidade da luz difusa refletida.

K_d é o coeficiente de reflexão difusa do material.

I_l é a intensidade da luz incidente.

$\hat{L}$ é o vetor unitário que aponta da superfície para a fonte de luz.

$\hat{N}$ é o vetor normal unitário à superfície.

A reflexão difusa é o principal componente responsável pelo sombreamento suave e pela definição de forma e volume dos objetos em cenas tridimensionais. Isso fica evidente ao comparar os diferentes componentes de iluminação (iluminação ambiente, reflexão difusa e reflexão especular), como mostrado nas figuras 4, 5 e 7.

2.4.2.5 Reflexão Especular

A reflexão especular, apresentado na figura 7 ocorre quando a luz atinge superfícies lisas e brilhantes, sendo refletida em uma direção específica. Esse tipo de reflexão gera pontos de brilho concentrados (*highlights*) em materiais como metais ou superfícies polidas. Diferente da reflexão difusa, a intensidade da reflexão especular depende não apenas da posição da fonte de luz e da normal da superfície, mas também da posição do observador Hearn e Baker (1997), como mostrado na figura 8.

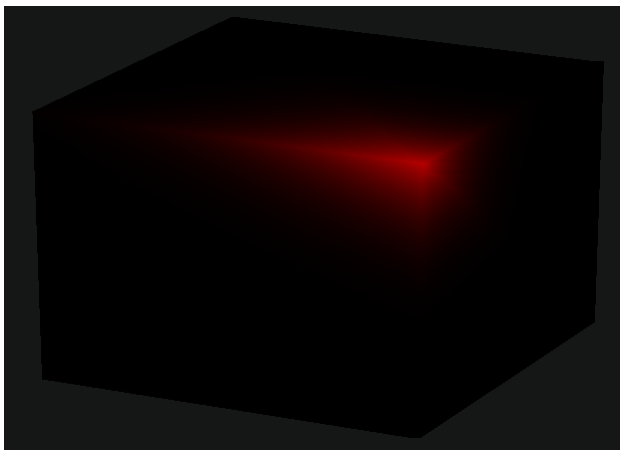


Figura 7 – Exemplo de reflexão especular

Na figura 8 o ângulo de incidência (θ) é igual ao ângulo de reflexão. A direção do observador é representada pelo vetor $\hat{\mathbf{S}}$. Em uma superfície refletora ideal, como um espelho perfeito, a luz é refletida apenas na direção do vetor de reflexão $\hat{\mathbf{R}}$. Nesse caso, o reflexo só é visível quando os vetores $\hat{\mathbf{R}}$ e $\hat{\mathbf{S}}$ formam entre si um ângulo α próximo de zero.

Superfícies que não são refletores ideais apresentam reflexão especular dentro de uma faixa angular finita. Superfícies brilhantes possuem uma faixa menor de reflexão especular, enquanto superfícies foscas têm uma faixa maior Santa Catarina (2024).

A reflexão especular é calculada pela equação 5, que determina a intensidade do brilho especular:

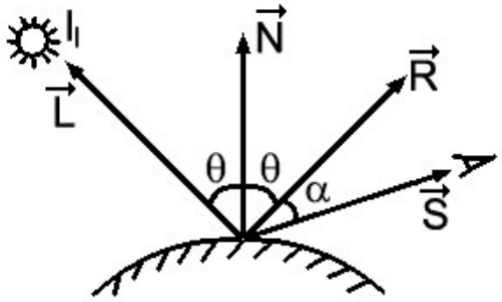


Figura 8 – Reflexão especular

Fonte: Santa Catarina (2024, p. 136)

$$I_s = K_s \cdot I_l \cdot (\hat{R} \cdot \hat{S})^n \tag{5}$$

Onde:

I_s é a intensidade da luz especular refletida.

K_s é o coeficiente de reflexão especular do material.

I_l é a intensidade da luz incidente.

$\hat{R}$ é o vetor de reflexão da luz.

$\hat{S}$ é o vetor que aponta da superfície para o observador.

n é o expoente que aproxima a distribuição espacial da luz refletida especularmente, também chamado de coeficiente de brilho (*shininess*) .

A reflexão especular adiciona realismo às cenas, simulando o comportamento de materiais reflexivos, como plásticos e metais, permitindo controlar a intensidade e a nitidez dos reflexos.

2.4.3 Sistema de Câmera em Cenas 3D

Em uma cena 3D, a **câmera virtual** desempenha o papel de observador da cena, determinando o ponto de vista a partir do qual os objetos são visualizados. Diferente de uma câmera física, a câmera em um ambiente virtual é uma entidade matemática que define como a geometria tridimensional será projetada em uma tela bidimensional. O sistema de câmera é responsável por configurar a projeção, a orientação e a posição do observador, além de definir parâmetros importantes, como o campo de visão, planos de recorte e profundidade de campo Hearn e Baker (1997).

2.4.3.1 Modelos de Projeções

Existem dois tipos principais de projeções usadas no sistema de câmera de uma cena 3D: **projeção perspectiva** e **projeção ortográfica**.

Na projeção perspectiva os objetos parecem menores à medida que se afastam da câmera, simulando a maneira como os humanos enxergam o mundo real. Esse tipo de projeção é governado por um ou mais pontos de fuga, onde as linhas paralelas parecem convergir à distância. A câmera define um *campo de visão* (FOV - *Field of View*) que controla o ângulo da visão capturada Foley et al. (1995). Em termos matemáticos, a projeção perspectiva é representada por uma matriz de projeção que transforma as coordenadas dos objetos tridimensionais em coordenadas de tela, levando em consideração a profundidade.

Por outro lado, na projeção ortográfica, os objetos mantêm suas dimensões independentemente da distância da câmera. Esse tipo de projeção é frequentemente usado em aplicações técnicas, como design assistido por computador (CAD) e visualizações arquitetônicas, onde a precisão geométrica é importante. A projeção ortográfica desconsidera o efeito de perspectiva, resultando em uma imagem "achatada" onde as linhas paralelas permanecem paralelas Hearn e Baker (1997).

2.4.3.2 Posicionamento e Orientação da Câmera

O Sistema de Referência da Câmera (SRC) permite, em uma cena 3D, controlar a **posição** e a **orientação** da câmera no espaço tridimensional. A posição da câmera é definida por um ponto no espaço 3D, enquanto a orientação é especificada pelos vetores que determinam a direção em que a câmera está "apontando" e a sua orientação "para cima". Esses parâmetros são descritos pela transformação de **visualização**, que converte as coordenadas do mundo tridimensional para o sistemas de coordenadas da câmera Smith (1983). Essa transformação é frequentemente representada por uma matriz de visualização, que corresponde à mudança da base canônica do Sistema de Referência do Universo (SRU), onde os objetos foram modelados, para a base ortonormal do SRC, obtida a partir da posição da câmera, do seu vetor de orientação e do ponto focal. Mais informações sobre mudança de bases estão descritas no anexo A.3.

A transformação de visualização é determinada por três parâmetros principais Foley et al. (1995):

Posição da Câmera (*eye*): O ponto de vista do observador na cena. Este é o ponto no espaço 3D de onde a cena é observada e define a origem do SRC.

Ponto de Focal (*look-at*): O ponto no espaço 3D para o qual a câmera está direcionada. Ele define a direção da linha de visão da câmera e é usado para determinar o vetor de direção da câmera.

Vetor *view-up* (cima): Define a orientação vertical da câmera, determinando qual direção é considerada "para cima" no enquadramento da cena. Este vetor é fundamental para garantir que a câmera não esteja inclinada e para definir o eixo vertical do SRC.

Esses três parâmetros são usados para construir a matriz de visualização que posiciona a câmera e define como os objetos são enquadrados no plano de visão. A transformação de visualização é aplicada após as transformações de modelo, permitindo que os objetos sejam corretamente posicionados em relação ao ponto de vista da câmera. Essa abordagem cria uma clara distinção entre o SRU (onde os objetos estão posicionados) e o SRC (a partir do qual a cena é observada e renderizada) Foley et al. (1995).

Além disso, no processo de renderização, a transformação de visualização é combinada com a transformação de projeção, que converte a cena 3D em uma imagem 2D, preservando a perspectiva ou aplicando uma projeção ortográfica, dependendo da configuração da câmera Hearn e Baker (1997). A câmera virtual permite, portanto, que o observador "navegue" pela cena tridimensional, mudando o ponto de vista e controlando a orientação da visualização.

2.4.3.3 Planos de Recorte e Campo de Visão

O sistema de câmera também define o **campo de visão** (FOV - *Field of View*), que controla o ângulo de visão da câmera. Um campo de visão mais amplo captura uma maior porção da cena, mas pode resultar em distorção perspectiva nas bordas. Um campo de visão mais estreito reduz a distorção, mas limita a quantidade de cena visível Akenine-Moller et al. (2018).

Além disso, a câmera virtual utiliza **planos de recorte** (*clipping planes*) para determinar a faixa de distâncias da câmera em que os objetos são renderizados. Existem dois tipos de planos de recorte:

Plano de recorte próximo (*near clipping plane*): Define a distância mínima da câmera a partir da qual os objetos começam a ser renderizados. Objetos mais próximos que essa distância são ignorados.

Plano de recorte distante (*far clipping plane*): Define a distância máxima a partir da qual os objetos ainda são renderizados. Objetos além desse plano são descartados.

Esses planos são essenciais para otimizar a renderização, limitando a quantidade de geometria processada com base na profundidade da cena Hearn e Baker (1997).

2.5 *Pipeline* de Visualização 3D

O *pipeline* de visualização 3D é um processo algorítmico que transforma dados tridimensionais em imagens bidimensionais, permitindo a representação visual de cenas complexas em ambientes computacionais. Este *pipeline* é fundamental na computação gráfica e abrange uma sequência de etapas interdependentes, cada uma desempenhando um papel essencial na criação da imagem final que será apresentada ao observador Foley et al. (1993).

O modelo original de *pipeline* proposto por Smith (1983) estabeleceu as bases para a visualização 3D. Suas etapas principais incluem:

1. **Transformação para Coordenadas Normalizadas de Visualização:** A cena é inicialmente transformada para um sistema de coordenadas normalizadas. Essa normalização simplifica o processo de recorte, otimizando as operações subsequentes.

2. **Recorte no Espaço Normalizado:** O recorte é realizado dentro do frustum de visualização (volume de visualização 3D) no espaço de coordenadas normalizadas, delimitando a região da cena que será efetivamente projetada na tela.

3. **Projeção Perspectiva e Mapeamento para o Espaço de Exibição:** Após o recorte, os pontos restantes são projetados, aplicando-se a projeção perspectiva e, em seguida, mapeados para o espaço de exibição 2D do dispositivo de visualização.

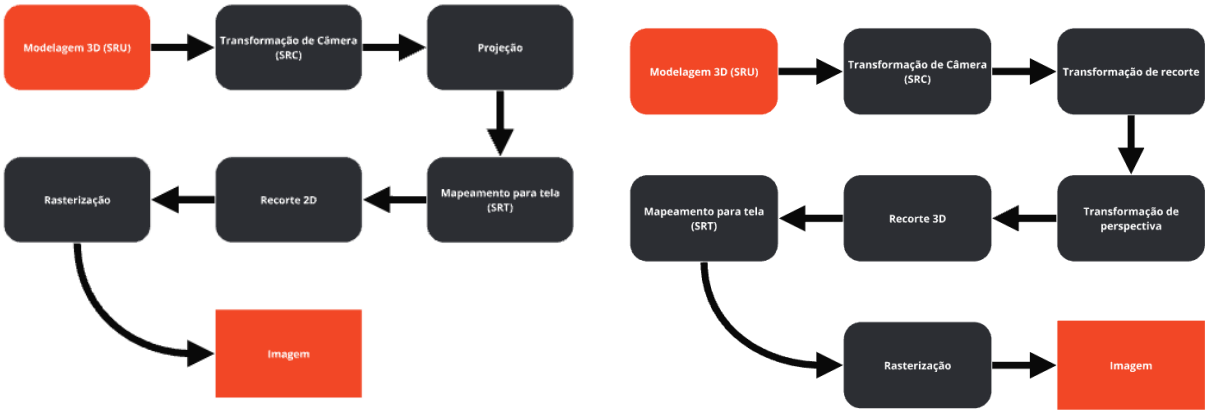
Este trabalho analisa dois *pipelines* de visualização inspirados no modelo de Smith (1983) propostos por Santa Catarina Santa Catarina (2024). Além das transformações geométricas presentes no modelo original, o autor utiliza etapas como ocultação de faces, recorte, rasterização e aplicação de efeitos visuais, expandindo o escopo do *pipeline*.

2.5.1 Visão Geral Comparativa dos *Pipelines*

Esta seção apresenta uma comparação direta entre os *pipelines* de visualização 3D proposto por Santa Catarina (2024), contrastando-os com o modelo clássico de Smith (1983). O primeiro *pipeline* (Figura 9a - **Pipeline A**) adota uma abordagem mais simplificada, enquanto o segundo (Figura 9b - **Pipeline B**) se aproxima mais da estrutura original de Smith, caracterizando-se por maior complexidade e granularidade nas etapas.

A Figura 9 oferece uma visão geral comparativa das etapas de cada *pipeline*, permitindo uma análise direta das similaridades e diferenças entre as abordagens.

A principal divergência entre os *pipelines* de Santa Catarina, em relação ao modelo de Smith, reside no tratamento do recorte. Smith realiza o recorte no espaço 3D normalizado, dentro do frustum de visualização, antes da projeção. O *pipeline* B segue esta mesma abordagem,



(a) *Pipeline* A (b) *Pipeline B*

Figura 9 – Comparativo das Etapas dos *Pipelines* de Visualização

realizando o recorte também no espaço 3D, dentro de um frustum normalizado com dimensões unitárias, o qual é posteriormente convertido em um paralelepípedo normalizado, que define o Sistema de Referência Normalizado (SRN), visando simplificar a operação. Essa estratégia torna o recorte mais eficiente, eliminando primitivas não visíveis antes da projeção e da conversão para o Sistema de Referência da Tela (SRT). Consequentemente, assim como no modelo de Smith, as transformações geométricas no *pipeline* não podem ser completamente concatenadas, pois o algoritmo de recorte 3D deve ser realizado em um sistema de coordenadas intermediário, o SRN.

Em contraste, o *pipeline* A posterga a operação de recorte para o espaço 2D, realizando-a após o mapeamento, diretamente na *viewport*. Essa simplificação tem implicações importantes. Enquanto o *pipeline* B eliminam primitivas não visíveis antes da transformação para o espaço de exibição, o *pipeline* A transforma todos os objetos, mesmo aqueles fora da *viewport*, antes de realizar o recorte 2D. Embora a concatenação das matrizes de transformação geométrica e de visualização seja simplificada com essa abordagem, o custo computacional aumenta devido à necessidade de transformar objetos que não serão exibidos. Em resumo, o recorte 2D na *viewport* pode diminuir o número de cálculos devido à concatenação das matrizes, mas transforma todos os objetos para o SRT, mesmo os que não serão exibidos Santa Catarina (2024).

Portanto, o *pipeline* B adota uma abordagem mais complexa e alinhada ao modelo clássico de Smith, priorizando a eficiência do recorte 3D em detrimento da concatenação completa das transformações. Já o *pipeline* A opta por uma abordagem mais simplificada, priorizando a concatenação das matrizes (A.3.3), mesmo que isso implique em transformar objetos que posteriormente serão descartados pelo recorte 2D. As seções subsequentes detalharão cada etapa dos *pipelines*, comparando-as em maior profundidade.

2.5.1.1 Transformação de Câmera

A transformação de câmera é uma etapa fundamental em qualquer *pipeline* de visualização 3D. O objetivo principal é posicionar e orientar a câmera no espaço da cena, de forma que os objetos possam ser visualizados a partir do ponto de vista desejado. Conceitualmente, esta etapa é quase igual para ambos os *pipelines* propostos por Santa Catarina (2024) e consiste em uma translação seguida de uma rotação. A translação move o sistema de coordenadas do mundo de forma que a origem coincida com a posição da câmera, enquanto a rotação alinha os eixos do sistema de coordenadas da câmera com a orientação desejada. A diferença é que no *pipeline A* o SRC é orientado pela regra da mão-direita, enquanto no *pipeline* B o SRC é orientado pela regra da mão-esquerda.

De acordo com Foley et al. (1995), essa transformação é geralmente representada por uma matriz de visualização M_{view} que combina translação e rotação em uma única matriz 4x4. Esta matriz, quando aplicada aos vértices dos objetos na cena, converte suas coordenadas do SRU para o SRC.

A matriz de visualização é, portanto, um elemento fundamental no *pipeline* de visualização. Ela garante que todos os objetos da cena sejam adequadamente ajustados para que, na etapa de projeção, eles sejam transformados do espaço 3D para a tela 2D, preservando a perspectiva e a relação espacial entre os objetos Hearn e Baker (1997).

2.5.1.1.1 Coordenadas de Câmera

As coordenadas de câmera são essenciais para determinar como uma cena 3D será visualizada a partir de uma posição específica do observador (**View Reference Point** - VRP ou P_{eye}). Esse sistema de coordenadas permite transformar o espaço tridimensional do universo em um espaço de visualização que reflete o ponto de vista da câmera. Conforme descrito em Hearn e Baker (1997), a transformação para o SRC é uma das etapas cruciais do pipeline de visualização, pois ajusta a cena de acordo com a posição, orientação e direção da câmera.

A transformação do espaço 3D para o espaço de visualização é composta de duas fases principais: a definição do sistema de referência da câmera e a transformação de coordenadas no SRU para o SRC Foley et al. (1995).

2.5.1.1.2 Regra da Mão-Direita vs. Mão-Esquerda

A definição do SRC (Sistema de Referência de Coordenadas) está diretamente relacionada à escolha entre duas convenções espaciais fundamentais: a regra da mão direita e a regra da mão esquerda, como ilustrado na Figura 10. Essa decisão afeta não apenas a orientação dos eixos

coordenados, mas também a direção considerada positiva para medidas de profundidade (*eixo z*) e a ordem dos vetores em operações de produto vetorial.

Na regra da mão direita, amplamente adotada em sistemas como OpenGL e em contextos matemáticos formais, o polegar, o indicador e o dedo médio da mão direita apontam, respectivamente, para as direções positivas dos eixos x, y e z. Nessa configuração, a câmera costuma estar voltada ao longo do eixo z negativo, de modo que o valor de profundidade diminui conforme os objetos se afastam do observador.

Em contraste, a regra da mão esquerda, utilizada em ambientes como DirectX e Unity, inverte a orientação do eixo z. Nesse caso, os dedos da mão esquerda definem as direções positivas de x, y e z, e a câmera se alinha com o eixo z positivo, de forma que a profundidade aumenta na direção em que os objetos se afastam do observador.

A adoção de uma ou outra dessas regras impacta algoritmos que dependem da orientação espacial, como o cálculo de vetores normais às superfícies e o descarte de faces ocultas (*backface culling*).

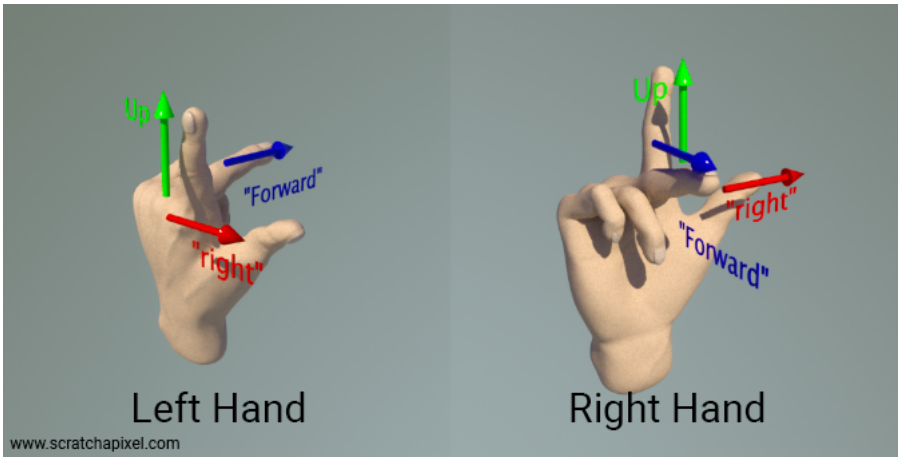


Figura 10 – Regra da mão esquerda e regra da mão direita.

Fonte: Prunier (2025)

2.5.1.1.3 Especificação do SRC

O SRC é descrito por três vetores ortogonais: o vetor de direção ($\hat{\mathbf{n}}$), que depende diretamente da posição da câmera (*eye*) e do ponto focal (*look-at*), e define o eixo da profundidade; o vetor "para cima" ($\hat{\mathbf{v}}$), que define a orientação vertical da câmera em relação à cena; e o vetor"direita" ($\hat{\mathbf{u}}$), que é ortogonal aos dois primeiros e define a direção horizontal da câmera Akenine-Mooller et al. (2018). A relação entre esses vetores é governada pela convenção de coordenadas adotada (Seção 2.5.1.1.2), determinando a ordem dos produtos vetoriais e a orientação dos eixos.

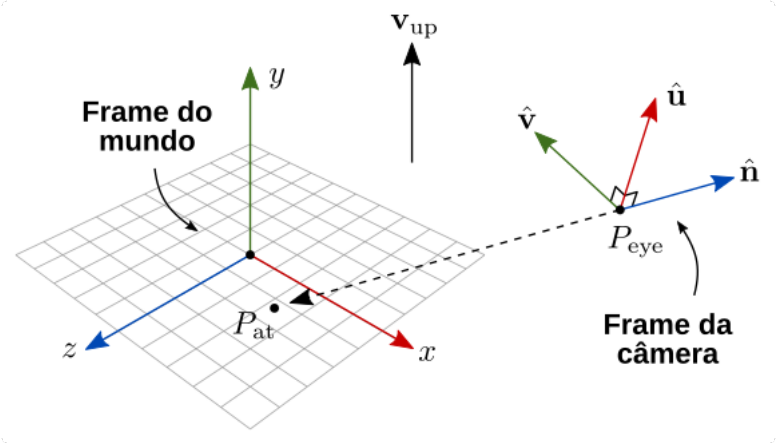


Figura 11 – Representação de uma câmera virtual

Fonte: Dorta (2024)

Na figura 11, que segue a regra da mão direita, o vetor $\hat{u}$ é calculado por meio da equação 9. Essa operação garante que o sistema de coordenadas resultante mantenha a ortogonalidade entre os eixos, com a direção de visualização alinhada ao eixo z negativo. Em sistemas que adotam a regra da mão esquerda, o vetor $\hat{n}$ terá invertido seu sentido, preservando a relação espacial entre os vetores mas invertendo a orientação do eixo z. Em outas palavras, ao concluir a especificação do SRC, $\hat{n} = -\hat{n}$.

Além disso, a figura 11 também nos mostra os eixos x, y e z do SRU, que é o sistema nos quais os objetos 3D foram modelados. O ponto P_{eye} representa a posição da câmera no espaço e o ponto focal (P_{at}) define para onde a câmera está apontada. Estes pontos são essenciais para calcular os vetores $\hat{n}$, $\hat{v}$, e $\hat{u}$ que definem o SRC.

2.5.1.1.4 Transformação de coordenadas do SRU para coordenadas do SRC

Antes que os objetos possam ser projetados é necessário converter as coordenadas dos vértices do SRU para o SRC, por meio de duas transformações geométricas: uma translação e uma rotação, descritas nos seguintes passos:

Transladar o ponto P_{eye} (posição da câmera) para a origem do SRU.

Aplicar rotações para alinhar os eixos do SRC ($\hat{u}$, $\hat{v}$, $\hat{n}$) com os eixos do SRU (x, y, z).

Se as coordenadas da posição da câmera P_{eye} no SRU forem (x_0, y_0, z_0), a matriz de translação correspondente será $T(-x_0, -y_0, -z_0)$. A sequência de rotações pode exigir até três rotações em torno dos eixos coordenados, dependendo da orientação desejada para o vetor de direção $\hat{n}$ Santa Catarina (2024).

Outra abordagem para gerar a matriz de rotação é calcular a base ortonormal $\hat{u}, \hat{v}, \hat{n}$ do SRC. O processo para determinar esses vetores, segundo Santa Catarina (2024) é realizado em três passos.

Passo 1 - Obtenção do vetor $\hat{n}$: O vetor $\hat{n}$ é calculado pela diferença entre os pontos P_{eye} e P_{at}, conforme mostrado na equação 6:

$$\hat{n} = \frac{P_{eye} - P_{at}}{|P_{eye} - P_{at}|} \tag{6}$$

O vetor $\hat{n}$ indica a direção para a qual a câmera está voltada, sendo perpendicular ao plano de projeção, e define o eixo z do SRC.

Passo 2 - Obtenção do vetor $\hat{v}$: O vetor $\vec{V}$ pode ser obtido por meio da diferença entre o vetor *view-up* $\vec{Y}$ e sua projeção sobre o vetor $\hat{n}$, conforme a equação 7. Para simplificar, $\vec{Y}$ pode ser definido como $(0, 1, 0)$, de acordo com Santa Catarina (2024).

$$\vec{V} = \vec{Y} - (Y \cdot \hat{n}) \cdot \hat{n} \tag{7}$$

Finalmente, normalizamos o vetor $\vec{V}$ para obter o vetor $\hat{v}$, que define o eixo vertical do SRC, conforme a equação 8.

$$\hat{v} = \frac{\vec{V}}{|\vec{V}|} = (v_x, v_y, v_z) \tag{8}$$

Passo 3 - Obtenção do vetor $\hat{u}$: O vetor $\hat{u}$, que define o eixo horizontal do SRC, é obtido através do produto vetorial entre $\hat{v}$ e $\hat{n}$, garantindo que ele seja perpendicular a ambos. O cálculo é dado pela equação 9.

$$\hat{u} = \hat{v} \times \hat{n} \tag{9}$$

2.5.1.1.5 Matriz de transformação M_{view}

A matriz M_{view}, que transforma coordenadas do SRU para o SRC, é composta por uma combinação de operações de translação e rotação que alinham as origens e os eixos dos dois sistemas, transformando os objetos da cena em relação à posição e orientação da câmera, preparando-os para a projeção no plano de visualização.

A matriz de translação M_{trans} pode ser expressa como:

$$M_{trans} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

onde $P_{eye} = (x_0, y_0, z_0)$ corresponde à posição da câmera no SRU. A translação move a câmera para a origem do SRU, preparando o espaço para o alinhamento angular.

A matriz de rotação M_{rot}, que alinha os eixos do SRC com os eixos do SRU, é construída a partir dos vetores ortonormais $\hat{u}, \hat{v}, \hat{n}$, seguindo a convenção de coordenadas escolhida:

$$M_{rot} = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z & 0 \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & 0 \\ \hat{n}_x & \hat{n}_y & \hat{n}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

Na regra da mão direita (adotada na Figura 11), o determinante da submatriz 3×3 de M_{rot} é +1, caracterizando uma rotação pura. Para sistemas de mão esquerda, o determinante será −1, indicando uma reflexão combinada com rotação. Nesses casos, a terceira linha de M_{rot} (correspondente a $\hat{n}$) é multiplicada por −1 para compensar a inversão do eixo z. Por fim, a matriz de visualização final é obtida pela multiplicação das duas matrizes, de acordo com a equação 12.

$$M_{view} = M_{rot} \cdot M_{trans} = \begin{bmatrix} \hat{u}_x & \hat{u}_y & \hat{u}_z & -P_{eye} \cdot \hat{u} \\ \hat{v}_x & \hat{v}_y & \hat{v}_z & -P_{eye} \cdot \hat{v} \\ \hat{n}_x & \hat{n}_y & \hat{n}_z & -P_{eye} \cdot \hat{n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

Note-se que, em sistemas de mão esquerda como o DirectX, a componente z da profundidade é invertida posteriormente durante a projeção, requerendo sincronização entre as etapas de visualização e projeção.

2.5.1.2 Transformação de Recorte

A etapa de transformação de recorte é crucial no pipeline B proposto por Santa Catarina, especialmente por adotar uma abordagem de recorte 3D. Essa transformação tem como objetivos centralizar o volume de visualização em relação ao eixo z e converter o volume original em um volume canônico com coordenadas variando entre -1 e 1 em cada eixo. Este volume em projeções em perspectiva assume a forma de um *frustum* (pirâmide com o topo cortado) e, em projeções ortográficas, assume a forma de um paralelepípedo. Esse procedimento foi descrito por Smith (1983), onde o autor realiza essa etapa juntamente com a transformação de câmera (2.5.1.1).

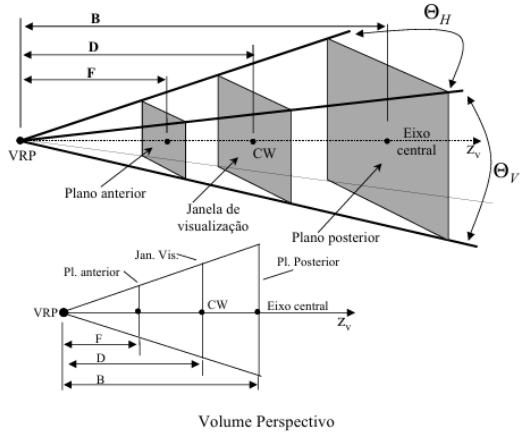
O principal benefício desse processo reside na simplificação das operações de recorte 3D, pois elimina a complexidade de calcular a interseção com um volume arbitrário, substituindo-a pela tarefa mais simples de determinar a interseção com um cubo alinhado aos eixos de coordenadas. Essa abordagem contrasta com o pipeline A, que realiza o recorte em 2D após a operação de mapeamento, operando diretamente sobre as coordenadas da tela para, em seguida, restringir a renderização aos limites da viewport.

Para obter a transformação de recorte é necessário concatenar duas transformações geométricas – uma de cisalhamento e outra de escala – representadas pelas matrizes apresentadas nas equações 13 e 14.

$$D_c = \begin{bmatrix} 1 & 0 & \frac{-cu}{D} & 0 \\ 0 & 1 & \frac{-cv}{D} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

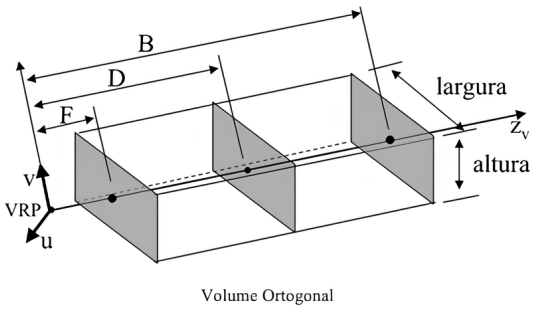
A matriz de cisalhamento D_c é definida de modo que cu e cv representem, respectivamente, as coordenadas x e y do centro da janela de imagem. O valor D corresponde à distância entre a posição da câmera (P_{eye}) e o plano de projeção. Conforme observado em Smith (1983), essa matriz não altera as proporções do cubo unitário durante o cisalhamento, atuando exclusivamente para centralizar a janela em relação ao eixo z, o que é indispensável, visto que a janela nem sempre se encontra perfeitamente centrada.

Em seguida, aplica-se uma transformação de escala (matriz E) que converte o volume de visualização de um *frustum* genérico para um paralelepípedo canônico, conforme demonstrado nas figuras 12a e 12b.



(a) *Frustum* genérico de visualização

Fonte: Pereira (2013, p. 13)



(b) Paralelepípedo canônico de visualização

Fonte: Pereira (2013, p. 13)

Figura 12 – Transformação de volumes

$$E = \begin{bmatrix} \frac{d}{su \cdot F} & 0 & 0 & 0 \\ 0 & \frac{d}{sv \cdot F} & 0 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{14}$$

onde *su* e *sv* correspondem, respectivamente, à metade da largura e da altura da janela de visualização, e F representa a distância entre P_{eye} até o plano anterior (*near*).

Ao concatenar as matrizes E e D_c, é possível aplicar ambas as transformações simultaneamente (A.3.3), obtendo a matriz de transformação de recorte C.

$$C = E \times D = \begin{bmatrix} \frac{d}{su \cdot F} & 0 & \frac{-cu}{su \cdot F} & 0 \\ 0 & \frac{d}{sv \cdot F} & \frac{-cv}{sv \cdot F} & 0 \\ 0 & 0 & \frac{1}{F} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{15}$$

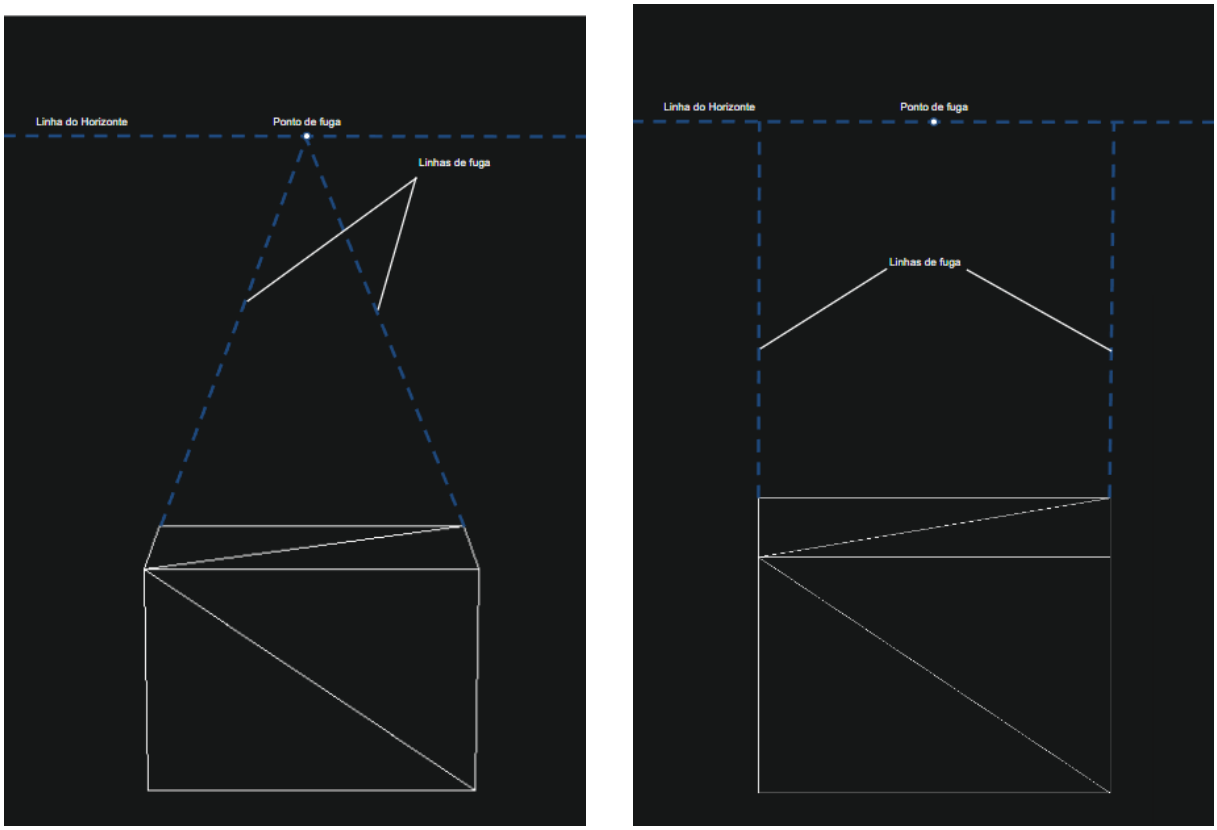
2.5.1.3 Projeções

A projeção, embora presente em ambos os pipelines, ocorre em momentos distintos. No *pipeline A*, a projeção é a segunda etapa, conforme ilustrado na Figura 9a. Em contraposição, no *pipeline B*, a projeção ocorre na terceira etapa (transformação de perspectiva), como demonstrado na Figura 9b. Esta diferença ocorre devido ao *pipeline B* precisar transformar o volume de visualização num paralelipípedo canônico antes de projetar as coordenadas.

Para além dessa diferença técnica entre os pipelines, a etapa de projeção do *pipeline B*, da maneira como o autor propõe, realiza mais duas transformações no volume de visualização, sendo estas transformações uma translação e uma escala.

No contexto do *pipeline* de visualização, a projeção desempenha um papel fundamental ao converter coordenadas 3D dos objetos em coordenadas 2D de um plano de projeção, permitindo que a cena seja exibida de forma apropriada em uma tela, que naturalmente é bidimensional. Esse processo é essencial para a renderização de gráficos, já que ele traduz a posição e profundidade dos objetos no espaço tridimensional para uma representação que pode ser compreendida visualmente pelo observador Hearn e Baker (1997). Existem dois tipos principais de projeção utilizados para realizar essa conversão: a projeção perspectiva e a projeção ortográfica.

A projeção perspectiva simula como o olho humano ou uma câmera enxergaria o mundo, fazendo com que objetos mais distantes pareçam menores, criando uma sensação de profundidade e realismo Akenine-Mooller et al. (2018). Em contraste, a projeção ortográfica não leva em conta a distância dos objetos da câmera, de modo que todos os objetos mantêm o mesmo tamanho, independentemente da profundidade.



(a) Projeção em perspectiva de um cubo (b) Projeção ortogonal de um cubo

Figura 13 – Comparação entre projeção em perspectiva e projeção ortogonal

Na figura 13a observamos um cubo projetado em perspectiva, onde as projetantes convergem para um ponto de fuga e criam a sensação de profundidade à medida que os objetos mais distantes parecem menores. As projetantes direcionam-se todas para o ponto de fuga na linha do horizonte, simulando a maneira como os olhos humanos percebem o espaço tridimensional no mundo real. Esse tipo de projeção é amplamente utilizado por criar um efeito de realismo Foley et al. (1995).

Em contraste, a figura 13b mostra o mesmo cubo projetado em projeção ortográfica. Neste tipo de projeção, as projetantes são perpendiculares ao plano de projeção, resultando em objetos que mantêm suas proporções independentemente da distância da câmera. Para fins comparativos, foi mantido o mesmo ponto de fuga da projeção perspectiva mas, diferentemente da projeção perspectiva, não há linhas de fuga convergindo para esse ponto. A linha do horizonte permanece presente, mas sem impacto na visualização, já que o cubo não sofre distorções de tamanho. Esse tipo de projeção é frequentemente utilizado em contextos técnicos e arquitetônicos, onde a preservação exata das proporções dos objetos é crucial Hearn e Baker (1997).

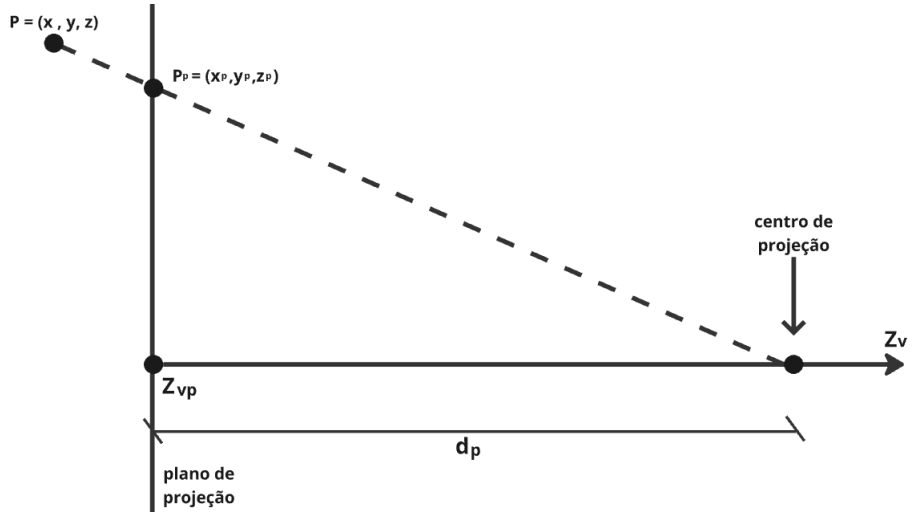


Figura 14 – Projeção em perspectiva do ponto $P = (x, y, z)$ na posição (x_p, y_p, z_p) sobre o plano de projeção

2.5.1.3.1 Projeção em perspectiva no *pipeline* A

Para calcular a projeção de um objeto 3D, os pontos desse objeto são transformados ao longo de linhas chamadas de projetantes, que convergem para o centro de projeção. Suponha que o centro de projeção esteja posicionado no ponto z_{prp} ao longo do eixo z_v, e que o plano de projeção seja perpendicular ao mesmo eixo e localizado em z_{vp}. A figura 14 ilustra essa disposição geométrica. Nosso objetivo é determinar as novas coordenadas (x_p, y_p, z_p) do ponto $P = (x, y, z)$ projetado no plano de projeção Santa Catarina (2024).

O conjunto de equações 16 descrevem as coordenadas (x_p, y_p, z_p) de qualquer ponto ao longo da linha de projeção em função do ponto original, da posição do plano de projeção e do centro de projeção. Essas equações expressam matematicamente como o ponto 3D é transformado em uma representação 2D, levando em conta a profundidade relativa do ponto em relação à câmera. Esse processo cria a distorção típica da perspectiva, na qual objetos mais próximos ao observador parecem maiores do que os que estão mais distantes.

$$\begin{aligned} x_p &= x - xu \\ y_p &= y - yu \\ z_p &= z - (z - z_{prp}) \cdot u \end{aligned} \tag{16}$$

O parâmetro u assume valores no intervalo entre $[0, 1]$; quando $u = 0$ estamos em P, quando $u = 1$ estamos no centro de projeção $(0, 0, z_{prp})$. As projeções em perspectiva e as ortográficas podem ser definidas através de matrizes 4×4, assim como todas as etapas de transformações geométricas do objeto, isso nos garante a possibilidade de concatenarmos estas operações, ou seja, realizarmos todas elas de uma única vez só. A matriz de projeção perspectiva é dada pela equação 17.

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-z_{vp}}{d_p} & z_{prp}(\frac{z_{prp}}{d_p}) \\ 0 & 0 & \frac{-1}{d_p} & \frac{z_{prp}}{d_p} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{17}$$

onde d_p é a distância entre o plano de projeção e a câmera. E nesta representação o fator homogêneo é dado pela equação 18.

$$h = \frac{z_{prp} - z}{d_p} \tag{18}$$

As coordenadas do ponto projetado no plano de projeção são obtidas a partir das coordenadas homogêneas dividindo-se por h, como mostra a equação 19.

$$x_p = \frac{x_h}{h}, y_p = \frac{y_h}{h} \tag{19}$$

Observando a equação 19, percebemos que o valor de z é preservado nas coordenadas de projeção para uso em algoritmos de ocultação de superfícies. Isso ocorre porque o valor original da coordenada z contém informações cruciais sobre a profundidade relativa dos objetos na cena, o que permite ao algoritmo identificar quais objetos ou partes de objetos estão atrás de outros e, portanto, devem ser ocultados.

Adicionalmente, observamos que o centro de projeção não precisa estar obrigatoriamente alinhado ao longo do eixo z_v. As equações de projeção podem ser generalizadas para acomodar centros de projeção em qualquer ponto do espaço, ajustando as relações de projeção para considerar essa variabilidade Santa Catarina (2024). Um caso especial a ser considerado é quando o centro de projeção coincide com a posição da câmera no SRU que, após conversão para o SRC, representa a origem deste sistema, o que implica em $z_{prp} = 0$ e $z_{vp} = -d_p$. Essas considerações são importantes para ajustar as equações de projeção e garantir que o algoritmo de remoção de linhas ocultas funcione corretamente em diferentes cenários de configuração do centro de projeção.

Além disso, como apontado por Akenine-Mooller et al. (2018), a projeção perspectiva introduz distorções, especialmente em cenas com campos de visão amplos, onde as linhas retas convergem para um ponto de fuga no horizonte. Essas distorções são mais evidentes quando se utilizam câmeras com ângulos muito abertos, resultando no fenômeno de “distorção de perspectiva”.

2.5.1.3.2 Projeção em perspectiva no pipeline B

Após a transformação de recorte explicada na seção 2.5.1.2, o volume de visualização é convertido de um *frustum* genérico para um paralelepípedo canônico. Entretanto, conforme mencionado no início da seção de projeções 2.5.1.3, no *pipeline B* a projeção não apenas transforma as coordenadas para simular o efeito de perspectiva, mas também modifica o volume canônico por meio de duas operações geométricas.

Inicialmente, procedemos com o deslocamento do volume de visualização de modo que o plano anterior (*near*) seja trazido para a origem do sistema (lembrando que o volume está alinhado ao eixo z). Para isso, utiliza-se a matriz de translação F, definida pela equação 20.

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{F}{B} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{20}$$

onde F representa a distância entre a câmera e o plano anterior (*near*) e B a distância entre a câmera e o plano posterior (*far*). A razão $\frac{F}{B}$, denominada z_{min}, define a divisão das distâncias aos planos. Essa operação já confere um efeito parcial de projeção, pois objetos mais próximos à câmera aparentam ser maiores que os distantes.

Em seguida, aplica-se uma transformação de escala ao longo do eixo z de modo que o plano posterior seja levado para $z = 1$. Essa transformação é descrita pela matriz G (equação 21):

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1-z_{min}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{21}$$

Posteriormente, define-se a matriz H (equação 22) responsável pela transformação efetiva que gera o efeito de perspectiva:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1-z_{min}}{z_{min}} & 1 \end{bmatrix}. \tag{22}$$

Ao concatenar as matrizes F, G e H obtém-se a seguinte matriz composta, conforme apresentada na equação 23.

$$H \cdot G \cdot F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1-z_{min}} & \frac{-z_{min}}{1-z_{min}} \\ 0 & 0 & \frac{1}{z_{min}} & 0 \end{bmatrix}. \tag{23}$$

Todavia, a projeção não se completa com essa concatenação. É necessário, ainda, multiplicar as três primeiras linhas da matriz (equação 23) pelo escalar $\frac{1}{z_{min}}$ e, em seguida, multiplicar toda a matriz pelo escalar z_{min}. Como resultado, obtém-se a matriz final P, apresentada na equação 24.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1-z_{min}} & \frac{-z_{min}}{1-z_{min}} \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{24}$$

Notamos que, ao multiplicar as três primeiras linhas por $\frac{1}{z_{min}}$ e depois toda a matriz por z_{min}, o efeito nas primeiras linhas equivale a uma multiplicação por 1, alterando unicamente o elemento $a_{4,3}$, que se torna unitário. Esse ajuste faz com que o volume de visualização possua limites de $2z_{min}$ nos eixos x e y, e que o plano $z = z_{min}$ delimite o eixo z. Ademais, o VRP (posição da câmera) é efetivamente levado ao infinito, fazendo com que a projeção perspectiva se converta, ao desprezar a coordenada z, em uma projeção paralela ortográfica.

2.5.1.3.3 Desenvolvimento matemático para projeções paralelas

Neste tipo de projeção, as projetantes são paralelas entre si, e a direção da projeção é definida por um vetor de projeção. A projeção é dita ortográfica quando o vetor de projeção é perpendicular ao plano de projeção, e oblíqua quando o vetor forma um ângulo diferente de 90º com o plano Santa Catarina (2024).

Para definir matematicamente uma projeção paralela ortográfica, consideramos o ponto $P = (x, y, z)$ em coordenadas do SRC. Se o plano de observação estiver posicionado em z_{vp} ao longo do eixo z, a projeção paralela transforma as coordenadas tridimensionais para coordenadas de projeção da seguinte forma:

$$x_p = x, \quad y_p = y, \quad z_p = z_{vp} \tag{25}$$

A matriz de transformação que realiza a projeção ortográfica, conforme mostrado na equação 26 corresponde a uma matriz identidade. Esse tipo de projeção preserva as proporções

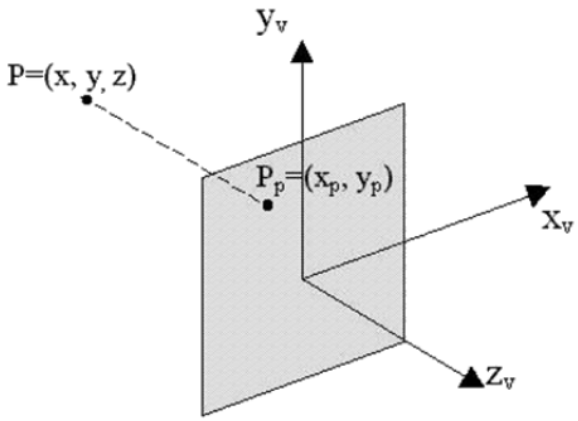


Figura 15 – Projeção ortográfica de um ponto no plano de projeção

Fonte: Santa Catarina (2024, p. 67)

dos objetos tornando-se especialmente útil para a geração de vistas frontais, laterais e superiores Santa Catarina (2024). Um ponto importante a ser destacado é que $z_p = z$, podendo ser empregada nos algoritmos de ocultação de superfícies pois preserva a noção de profundidade dos objetos da cena.

$$M_{ortog} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{26}$$

2.5.1.4 Transformação do SRC para o SRT

A transformação do SRC para o SRT, chamada de mapeamento, é uma etapa comum aos *pipelines*, entretanto, ela ocorre em momentos distintos. No *pipeline* A o mapeamento é realizado após a projeção, se valendo da concatenação de operações. Em contrapartida, no *pipeline* B, o mapeamento é realizada após o recorte 3D, abrindo mão da concatenação de operações, ao ganho de uma eliminação de primitivas que não serão processadas pelas etapas posteriores ao recorte.

O objetivo principal do mapeamento é converter as coordenadas dos objetos projetados no plano de projeção para o espaço bidimensional da tela, preparando os objetos para a rasterização. Apesar desta etapa estar presente em ambos os *pipelines*, cada um adota uma abordagem diferente.

2.5.1.4.1 Mapeamento para o SRT no *Pipeline* A

O mapeamento de coordenadas de Janela (*Window*) para Porta de Visão (*Viewport*), apresentado na figura 16, nos mostra que ele é responsável por transformar as coordenadas do

espaço de visualização (SRC), ainda representadas em unidades arbitrárias, para as coordenadas no SRT, que são medidas em pixels. Dessa forma, a janela de visualização determina qual parte da cena será exibida na tela e como ela será escalada e posicionada na Porta de Visão Santa Catarina (2024).

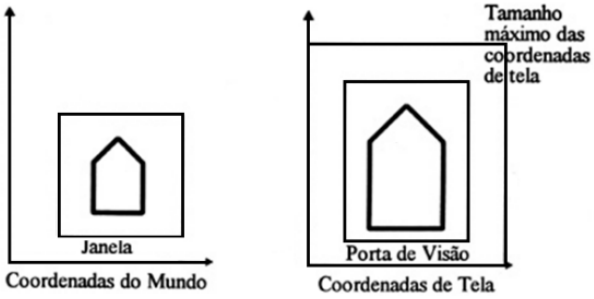


Figura 16 – Exemplo de mapeamento das coordenadas da janela para a porta de visão

Fonte: Santa Catarina (2024, p. 74)

O processo de mapeamento envolve uma série de operações geométricas que garantem que os objetos sejam corretamente posicionados e dimensionados na tela. A figura 17 ilustra as operações envolvidas neste processo.

A primeira etapa no processo de mapeamento envolve uma translação que desloca a janela de visualização para a origem do sistema de coordenadas. Esse alinhamento inicial é fundamental para simplificar as operações subsequentes, garantindo que a janela de visualização esteja apoiada na origem. Com a janela devidamente posicionada, o próximo passo é escalonar suas dimensões para que correspondam às da Porta de Visão (Viewport). Esse escalonamento ajusta a cena ao espaço da tela, preservando as proporções entre a janela de visualização e a Porta de Visão.

Em seguida espelhamos as coordenadas ao longo do eixo horizontal ($\vec{O}_x$), invertendo o sinal das coordenadas y. Esse espelhamento ocorre devido à convenção adotada em alguns sistemas gráficos, nos quais o ponto de origem $(0, 0)$ está no canto superior esquerdo da tela,

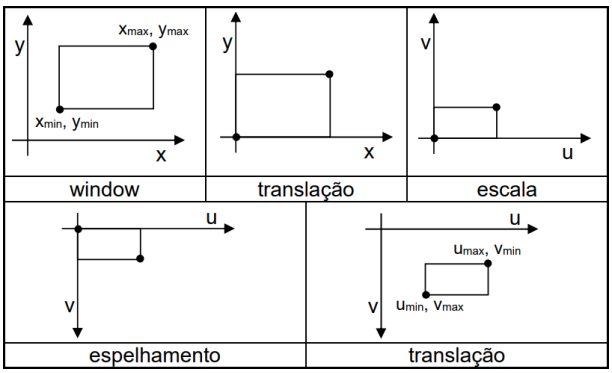


Figura 17 – Transformações geométricas para transformar coordenadas do SRC para o SRT

Fonte: Santa Catarina (2024, p. 76)

resultando em um eixo v invertido. Caso essa inversão não esteja presente, essa etapa pode ser ignorada. No entanto, é essencial estar atento a essa particularidade ao converter coordenadas de câmera para a *Viewport*, especialmente em sistemas interativos, para evitar a renderização invertida do conteúdo Santa Catarina (2024).

Por fim, uma translação adicional é aplicada para posicionar a Porta de Visão na tela, determinando o local exato onde a imagem será exibida em coordenadas de pixel. Essa translação final ajusta o conteúdo da janela de visualização para o ponto correto na tela, garantindo que a imagem seja renderizada com precisão no SRT. Todo esse conjunto de operações pode ser descrito através da equação 27.

$$M_{jp} = T(u_{min}, v_{max}, 0) \cdot E(\vec{O_x}) \cdot S(\frac{u_{max} - u_{min}}{x_{max} - u_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}, 1) \cdot T(-u_{min}, -v_{min}, 0) \quad (27)$$

Multiplicando as matrizes das operações da equação 27 obtemos a matriz apresentada na equação 28:

$$M_{jp} = \begin{bmatrix} \frac{u_{max}-u_{min}}{x_{max}-x_{min}} & 0 & 0 & -x_{min} \cdot \frac{u_{max}-u_{min}}{x_{max}-x_{min}} + u_{min} \\ 0 & \frac{v_{min}-v_{max}}{y_{max}-y_{min}} & 0 & y_{min} \cdot \frac{v_{max}-v_{min}}{y_{max}-y_{min}} + v_{max} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (28)$$

2.5.1.4.2 Mapeamento para o SRT no *Pipeline* B

A essência do mapeamento para o SRT no *pipeline B* é a mesma descrita na sessão anterior; seu objetivo final é transformar coordenadas medidas em unidades arbitrárias para coordenadas medidas em pixels no SRT. A principal diferença entre elas são as transformações aplicadas.

A primeira etapa deste mapeamento é uma escala nos eixos x e y, de modo que, seu comprimento seja dividido pela metade. Isto é importante, pois estamos trabalhando com um volume normalizado, ou seja, tudo o que estiver dentro deste volume terá coordenadas variando de $[-1, 1]$ nos eixos x e y. A escala aplicada reduz estes limites pela metade, garantindo valores entre $[-0.5, 0.5]$, ou seja, tenham um comprimento unitário. Para realizarmos esta escala, precisamos utilizar a equação 29. Note que o fator de escala em y tem sinal invertido devido ao SRT ter eixo vertical orientado de cima para baixo.

$$K = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

Na próxima etapa do mapeamento realizamos um escalonamento e uma translação com base nos limites da porta de visão (*viewport*). Para isto precisamos calcular as variações nos eixos x, y e z, de modo a obter o comprimento destes eixos, utilizando as equações 30

$$\begin{aligned} \Delta x &= x_{max} - x_{min} \\ \Delta y &= y_{max} - y_{min} \\ \Delta z &= F - B \end{aligned} \tag{30}$$

Onde $x_{min}, x_{max}, y_{min}$ e y_{max} os limites da porta de visão. F e B são as distâncias aos planos anterior *near* e posterior *far*.

Como fatores de escala utilizamos os valores Δ_x, Δ_y e Δ_z e, como fatores de translação, usamos x_{min}, y_{min} e F. Ao combinarmos estas duas transformações geométricas obtemos a matriz L, apresentada na equação 31

$$L = \begin{bmatrix} \Delta x & 0 & 0 & x_{min} \\ 0 & \Delta y & 0 & y_{min} \\ 0 & 0 & \Delta z & F \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{31}$$

Por fim, devemos fazer uma translação de 0.5 nos três eixos para arredondar as coordenadas dos pixels que serão empregadas na rasterização. Para isso utilizamos a matriz M, descrita na equação 32

$$M = \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{32}$$

Ao concatenarmos as matrizes das equações 29, 31 e 32, obtemos a matriz Q, apresentada na equação 33

$$M \cdot L \cdot K = Q = \begin{bmatrix} 0.5 \cdot \Delta x & 0 & 0 & 0.5 \cdot \Delta x + x_{\min} + 0.5 \\ 0 & 0.5 \cdot \Delta y & 0 & 0.5 \cdot \Delta y + y_{\min} + 0.5 \\ 0 & 0 & \Delta z & F + 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{33}$$

2.5.1.5 Eliminação de superfícies visíveis

Assim como as etapas de transformação do SRU para o SRC (2.5.1.1), projeção (2.5.1.3) e mapeamento para o SRT (2.5.1.4), a eliminação de superfícies visíveis é uma etapa fundamental e comum aos pipelines, sendo realizada antes a rasterização em ambos os *pipelines*. A eliminação de superfícies visíveis otimiza o processo de rasterização ao garantir que apenas as superfícies voltadas para o observador sejam desenhadas na tela.

Existem diferentes métodos para determinar quais superfícies estão visíveis e quais devem ser ocultadas. Um dos métodos mais comuns é a eliminação de faces ocultas pelo cálculo da normal (*back-face culling*), onde as faces dos polígonos que estão voltadas para o lado oposto à câmera devem ser descartadas, pois não são visíveis.

Outro algoritmo amplamente utilizado é o *Depth-Buffer* (ou *z-Buffer*), que compara a profundidade de cada pixel da cena, assegurando que apenas os objetos, ou partes deles, que estejam mais próximos do observador sejam rasterizadas. O Z-Buffer armazena a profundidade de cada pixel e atualiza esses valores conforme novos objetos são processados, garantindo que objetos mais distantes sejam ocultados pelos mais próximos.

Conforme descrito em Santa Catarina (2024), a eliminação de superfícies visíveis é uma etapa essencial do pipeline de visualização, pois reduz em até 75% o volume de cálculos necessários e contribui para que a imagem final represente corretamente a cena tridimensional.

No presente trabalho, a ocultação de superfícies visíveis foi implementada em duas etapas distintas. Primeiro determinamos a visibilidade das faces pelo cálculo da normal e, posteriormente, na rasterização, é aplicado o algoritmo Z-Buffer para resolver conflitos visuais. O Z-Buffer garante que, em cenários onde duas ou mais faces visíveis estão sobrepostas ou em outros casos de conflitos visuais, apenas a face mais próxima do observador seja renderizada. Esse método eficaz assegura que apenas a geometria visível e relevante seja desenhada, melhorando a precisão do processo de renderização.

2.5.1.5.1 Eliminação de Faces Ocultas pelo Cálculo da Normal

A eliminação de faces ocultas pelo cálculo da normal é uma técnica simples e eficiente, amplamente utilizada em poliedros convexos fechados. Nesse método, a visibilidade de uma face

é determinada pela orientação de sua normal, o vetor que é perpendicular à face. Se a normal estiver apontada em direção ao observador, a face será considerada visível e renderizada. Caso contrário, será considerada oculta e descartada da renderização Santa Catarina (2024).

Na figura 18 é ilustrada uma superfície definida por três pontos P_1, P_2 e P_3, que formam um triângulo em um plano. Esses três pontos são essenciais para o cálculo da equação do plano, uma vez que permitem a definição dos vetores $\vec{A}$ e $\vec{B}$, onde $\vec{A} = P_1 - P_2$ e $\vec{B} = P_3 - P_2$. Esses vetores estão representados na superfície e descrevem as direções ao longo das arestas que conectam os pontos. A seta circular na superfície, no sentido anti-horário, indica a orientação dos pontos, o que é crucial para garantir que a normal calculada esteja corretamente direcionada em relação ao observador.

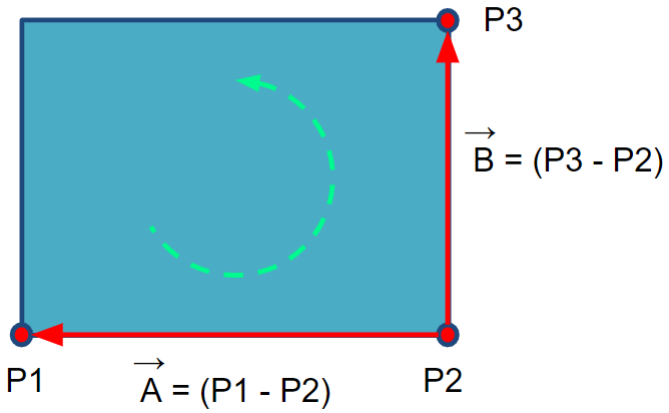


Figura 18 – Determinando a equação do plano

Ao calcular o produto vetorial $\vec{B} \times \vec{A}$, conforme é mostrado na equação 34, é possível obter as componentes do vetor normal ao plano, $\vec{N} = (a, b, c)$.

$$\vec{N} = \begin{vmatrix} i & j & k \\ B_x & B_y & B_z \\ A_x & A_y & A_z \end{vmatrix} = (a, b, c) \tag{34}$$

Para determinar o valor de d, da equação do plano, aplicamos um dos P_i da figura 18 na equação do plano. Com isso, temos que:

$$d = -(a \cdot P_i x) - (b \cdot P2y) - (c \cdot P_i z) \tag{35}$$

A equação $ax + by + cz + d = 0$ define o plano que contém a face, e os coeficientes a, b e c representam as componentes do vetor normal à face. Para determinar a posição relativa do observador em relação ao plano, aplicamos a posição do observador $VRP(x, y, z)$, como descrito na subseção 2.5.1.1.1, na equação do plano. O cálculo segue a equação 36:

$$D = (a \cdot VRP_x) + (b \cdot VRP_y) + (c \cdot VRP_z) + d \tag{36}$$

Onde o valor de D indica a posição relativa do observador ao plano:

Se $D = 0$, o observador está localizado no próprio plano.

Se $D > 0$, o observador está antes do plano e, portanto, ele é visível ao observador.

Se $D < 0$, o observador está depois do plano e, portanto, ele está oculto ao observador.

Outra forma de se determinar a visibilidade de uma face emprega o produto escalar entre seu vetor normal ($\vec{N}$) e um vetor definido entre o centroide da face e a posição da câmera ($\vec{O} = VRP - Centroide$). Se $(\vec{O} \cdot \vec{N}) > 0$, então o ângulo θ formado entre os dois vetores está no intervalo $0 \leq \theta \leq \frac{\pi}{2}$, o que significa que a face está voltada para o observador.

Embora essa técnica não seja suficiente para lidar com com objetos sobrepostos (figura 19), ela é extremamente eficaz em eliminar superfícies desnecessárias em um estágio preliminar, reduzindo o tempo de processamento durante a rasterização Santa Catarina (2024).

2.5.1.5.2 O algoritmo *z-Buffer*

O *z-Buffer* é um algoritmo amplamente utilizado na computação gráfica, especialmente em cenas 3D para a eliminação de superfícies ocultas. Sua principal função é resolver conflitos de visibilidade, garantindo que apenas os objetos mais próximos do observador sejam renderizados em cada ponto da tela. O algoritmo funciona armazenando, para cada pixel da tela, a menor profundidade (z) encontrada até o momento, permitindo comparar a profundidade de novos objetos que ocupam a mesma posição na tela.

A figura 19 ilustra um exemplo de conflito visual, onde dois triângulos — um verde e um vermelho — estão parcialmente sobrepostos, com diferentes partes de cada um à frente do outro. Em situações como essa, o *z-Buffer* atua pixel por pixel, verificando a profundidade de cada ponto na tela e comparando os valores de profundidade dos dois triângulos para determinar qual deles será desenhado em cada área sobreposta. Nas áreas em que o triângulo verde está mais próximo do observador, seus pixels serão renderizados, enquanto nas áreas em que o triângulo vermelho está mais próximo, os pixels do vermelho serão desenhados. O *z-Buffer* garante que em cada pixel apenas o objeto mais próximo ao observador seja visível, eliminando os conflitos de visibilidade entre os objetos que estão parcialmente sobrepostos.

Para que o algoritmo funcione corretamente é essencial que o cálculo de profundidade (z) ocorra durante a etapa de rasterização. Isso porque os valores de profundidade precisam ser interpolados incrementalmente ao longo das superfícies dos objetos à medida que são convertidos

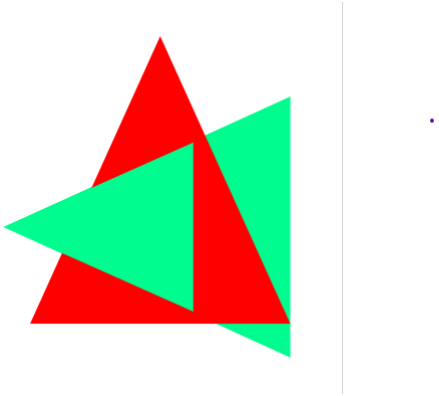


Figura 19 – Exemplo de conflito visual

em pixels. A interpolação de z é realizada em conjunto com a interpolação de outras propriedades, conforme descrito nas seções *Interpolação entre e dentro das linhas de varredura* (2.5.1.7.2 e 2.5.1.7.3) e *Interpolação da coordenada z* (2.5.1.7.4). Cada pixel recebe um valor interpolado de z que é então comparado ao valor armazenado no *z-Buffer*. Caso o valor interpolado seja menor (indicando que o pixel está mais próximo do observador), o *z-Buffer* é atualizado com o novo valor de profundidade, e a cor do pixel na tela é ajustada para representar o objeto mais próximo.

A principal vantagem do Z-Buffer é sua simplicidade e eficiência, sendo amplamente suportado em arquiteturas de hardware gráfico modernas. Contudo, ele requer memória adicional para armazenar os valores de profundidade para cada pixel da tela e requer ajustes para operar com superfícies transparentes Santa Catarina (2024).

2.5.1.6 Recortes

Embora presente em ambos os pipelines, a etapa de recorte apresenta diferenças fundamentais quanto ao momento de execução em que é realizado. No *Pipeline A*, o recorte é intencionalmente postergado para depois da transformação de mapeamento, operando exclusivamente no SRT (seção 2.5.1.4). Nesta abordagem, os vértices projetados são recortados contra a *viewport* em 2D, estratégia que viabiliza a concatenação completa das matrizes de transformação geométrica e elimina a necessidade de normalização do volume de visualização. Contudo, essa postergação implica no processamento de primitivas que, eventualmente, serão completamente descartadas na rasterização.

Em contraste, o *Pipeline B* adota a metodologia clássica proposta por Smith (1983), realizando o recorte pré-projetivo no SRC (seção 2.5.1.1.4). Esta estratégia opera sobre um volume canônico 3D normalizado (cubo unitário), permitindo o descarte antecipado de geometria fora do campo de visão. A vantagem reside na redução de cálculos nas etapas subsequentes, embora exija a normalização do volume de visualização e impeça a concatenação total das matrizes de transformação.

A divergência entre as abordagens reflete objetivos de projeto distintos. A estratégia

empregada no *Pipeline A* prioriza a eficiência na transformação geométrica através da composição matricial integral, aceitando como contrapartida o processamento de primitivas não visíveis. Já a implementação do *Pipeline B* sacrifica parte da eficiência computacional, relativa à concatenação das matrizes de transformações, em prol da otimização global do fluxo de processamento, filtrando geometricamente as primitivas antes do mapeamento.

Como estabelecido na Visão Geral (seção 2.5.1), este *trade-off* técnico constitui a principal diferença entre os modelos: enquanto um privilegia a economia de operações matriciais, o outro busca minimizar o processamento de fragmentos. A seleção entre essas estratégias depende criticamente dos requisitos de arquitetura do sistema gráfico e do balanço desejado entre custo computacional distribuído e concentrado.

2.5.1.6.1 Recorte 2D

O recorte 2D é uma operação fundamental no pipeline gráfico, pois garante que apenas as partes da cena que estão dentro da área visível sejam processadas e renderizadas. Uma das técnicas mais utilizadas para realizar essa operação é o algoritmo de recorte de linha Cohen-Sutherland, desenvolvido especificamente para recortar linhas que estão parcialmente ou completamente fora da janela de visualização Santa Catarina (2024).

O algoritmo Cohen-Sutherland divide o espaço em nove regiões com base na posição relativa de um ponto em relação à janela de recorte. Essas regiões incluem a área interna à janela e oito regiões externas: acima, abaixo, à esquerda, à direita e nas combinações diagonais de cada uma dessas direções. Cada ponto de uma linha é codificado com um *código de região* (ou código *outcode*), que é um valor binário de 4 *bits*. Esses *bits* indicam a posição do ponto em relação à janela, com cada bit representando uma direção: o *bit* mais significativo (MSB) corresponde à parte acima da janela, o segundo bit corresponde à parte abaixo, o terceiro à direita e o quarto à esquerda. Se todos os bits de um ponto forem 0000, o ponto está dentro da janela de recorte. Por outro lado, se um ou mais bits forem 1, o ponto está fora da janela em uma ou mais direções.

A figura 20 ilustra a codificação binária dessas 9 regiões associadas com a janela de recorte. Através dessa codificação o algoritmo consegue determinar rapidamente se uma linha está totalmente dentro ou totalmente fora da janela de recorte, por meio de operações lógicas OR (OU) e AND (E), respectivamente Foley et al. (1995).

Para qualquer ponto extremo (x, y) de uma linha, pode ser determinado o código que identifica em que região o ponto extremo está. Os *bits* são setados conforme as seguintes condições:

Quarto bit setado para 1: O ponto está à esquerda da janela $\Rightarrow x < x_{min}$;

Terceiro bit setado para 1: O ponto está à direita da janela $\Rightarrow x > x_{max}$;

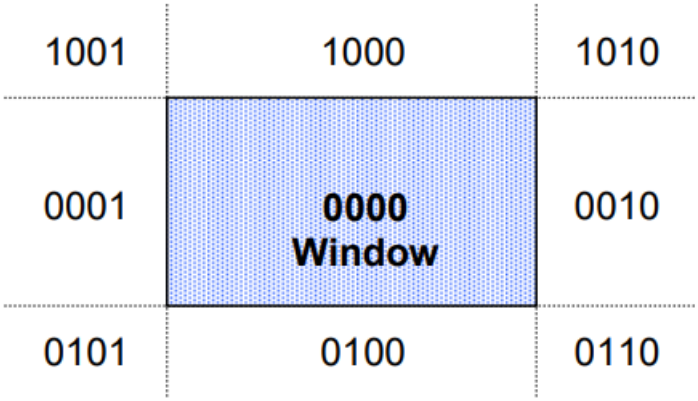


Figura 20 – Codificação binária das 9 regiões associadas com a janela de recorte

Fonte: Santa Catarina (2024, p. 115)

Segundo bit setado para 1: O ponto está abaixo da janela $\Rightarrow y > y_{max}$;

Primeiro bit setado para 1: O ponto está acima da janela $\Rightarrow y < y_{min}$.

A sequência de leitura dos códigos é **TBDE** (topo, baixo, direita, esquerda).

2.5.1.6.2 Implementação do Algoritmo de Cohen-Sutherland

O algoritmo de Cohen-Sutherland utiliza uma estratégia de divisão e conquista para recortar segmentos de linha com base nos limites da janela de visualização. Primeiramente, os extremos de cada segmento de linha são testados para verificar se a linha pode ser trivialmente aceita ou rejeitada. Caso a linha não possa ser completamente aceita ou rejeitada de forma trivial, as interseções da linha com os limites da janela são calculadas, e o processo de teste de aceitação ou rejeição é repetido até que a linha esteja dentro da janela Santa Catarina (2024).

Para realizar o teste de aceitação ou rejeição trivial, os limites da janela são estendidos, dividindo o plano da janela em nove regiões, como ilustrado na Figura 21. Cada extremo do segmento de linha é associado a um código de 4 bits que indica em qual região ele se encontra. Esses códigos são então usados para determinar se a linha está completamente dentro, fora ou parcialmente dentro da janela.

O funcionamento do algoritmo pode ser descrito pelos seguintes passos:

1. Dados dois pontos extremos de um segmento de linha, $P = (x_1, y_1)$ e $Q = (x_2, y_2)$, calculam-se os códigos de 4 bits para cada um desses pontos.

2. Se ambos os códigos forem 0000 (ou seja, o OR lógico entre os códigos retorna 0000), a linha está completamente dentro da janela e pode ser passada diretamente para a rotina de desenho.

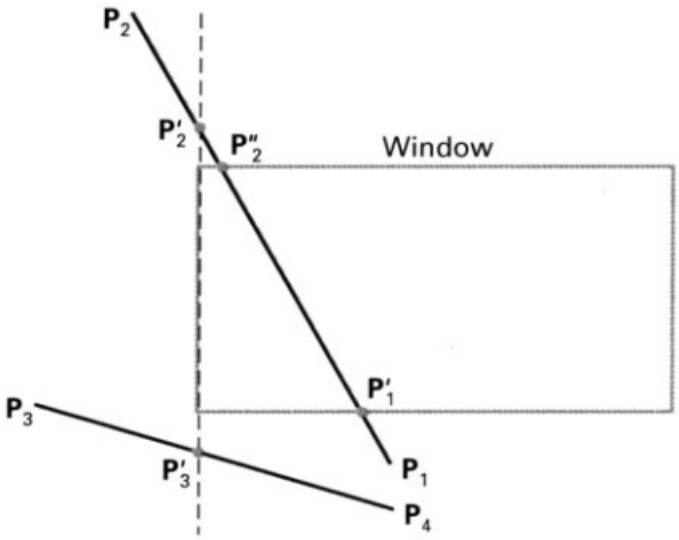


Figura 21 – Exemplo de utilização do algoritmo Cohen-Sutherland

3. Se o AND lógico entre os códigos de ambos os pontos resultar em um valor diferente de zero, a linha está completamente fora da janela e pode ser trivialmente rejeitada.

4. Caso a linha não possa ser trivialmente aceita ou rejeitada, pelo menos um dos extremos está fora da janela e a linha cruza um dos limites. Nesse caso, calcula-se a interseção da linha com o limite da janela e substitui-se o extremo fora da janela pelo ponto de interseção, repetindo o processo até que a linha esteja completamente dentro ou fora da janela Foley et al. (1995).

A Figura 21 ilustra o funcionamento do algoritmo de Cohen-Sutherland. O segmento de linha entre os pontos P_1 e P_2 é recortado à medida que se encontram interseções com os limites da janela de recorte. Inicialmente, o ponto P_1 está fora da janela, abaixo do limite inferior. Calcula-se a interseção P'_1 com a borda inferior e descarta-se o segmento de linha entre P_1 e P'_1. Em seguida, o ponto P_2 também está fora da janela, à esquerda. Calcula-se a interseção P'_2 com a borda esquerda, mas verifica-se que o ponto ainda está fora, acima da janela. Finalmente, calcula-se a interseção final P''_2, e a linha entre P'_1 e P''_2 é mantida.

Um procedimento semelhante é aplicado ao segundo segmento, P_3 a P_4. O ponto P_3 está à esquerda da janela, então a interseção P'_3 com a borda esquerda é calculada, e o segmento entre P_3 e P'_3 é descartado. O segmento restante de P'_3 a P_4 está completamente fora da janela, abaixo do limite inferior, e é completamente rejeitado.

O algoritmo Cohen-Sutherland é amplamente utilizado em sistemas gráficos interativos devido à sua simplicidade e eficiência, uma vez que reduz significativamente o número de cálculos necessários para determinar quais segmentos de linha serão rasterizados, descartando aqueles que estão totalmente fora da janela. Isso resulta em uma melhoria considerável no desempenho gráfico, especialmente durante a etapa de rasterização, pois elimina a necessidade de processar

segmentos invisíveis Foley et al. (1995). No entanto, o algoritmo é projetado para funcionar apenas em um plano de recorte retangular.

2.5.1.6.3 Recorte de Polígonos com o Algoritmo de Sutherland-Hodgman

O algoritmo de Sutherland-Hodgman é um método eficaz para o recorte de polígonos convexos, permitindo ajustar um polígono à área visível definida pela janela de visualização. Este algoritmo examina cada aresta do polígono em relação às bordas da janela e aplica quatro regras principais de acordo com a posição dos vértices da aresta em relação à borda. A Figura 22 ilustra esses casos:

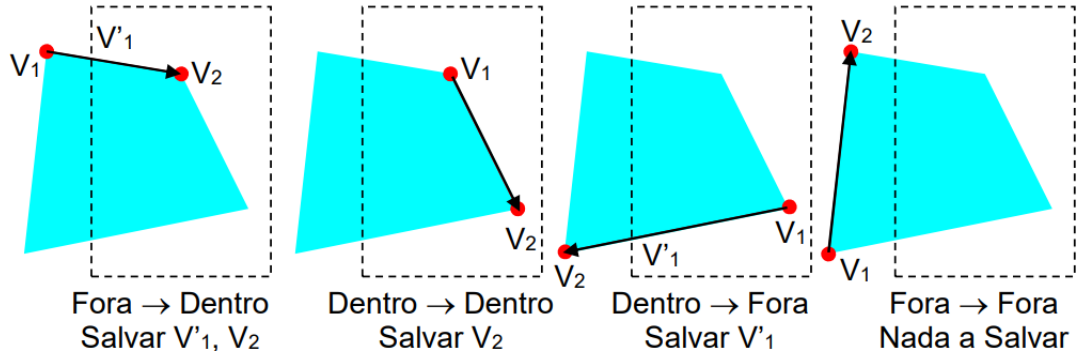


Figura 22 – Casos de recorte do polígono no algoritmo de Sutherland-Hodgman

Fonte: Santa Catarina (2024, p. 119)

No algoritmo, cada aresta do polígono original é processada em relação a cada borda da janela de recorte, conforme os seguintes casos ilustrados na Figura 22:

Caso Fora → Dentro: Se uma aresta cruza de fora para dentro da janela de recorte, o ponto de interseção (V_1') é calculado e armazenado, juntamente com o segundo vértice da aresta (V_2). Esse caso é ilustrado no primeiro quadro da Figura 22, onde a aresta entra na área visível.

Caso Dentro → Dentro: Se ambos os vértices da aresta estão dentro da área visível, apenas o segundo vértice (V_2) é armazenado, pois a aresta está completamente dentro da janela de recorte. Esse cenário é mostrado no segundo quadro da Figura 22.

Caso Dentro → Fora: Quando a aresta sai da área visível, o ponto de interseção (V_1') é calculado e armazenado. O segundo vértice é descartado, pois está fora da janela. A terceira imagem da Figura 22 demonstra essa situação.

Caso Fora → Fora: Se ambos os vértices da aresta estão fora da área de recorte, nenhum ponto é salvo, pois a aresta não contribui para o polígono recortado. Esse caso é ilustrado no último quadro da Figura 22.

O processo de recorte é realizado sequencialmente para cada borda da janela, resultando em um novo conjunto de vértices que definem a parte visível do polígono.

Em polígonos côncavos, o algoritmo de Sutherland-Hodgeman pode falhar, gerando polígonos abertos e arestas incoerentes. Para esta classe de polígonos o algoritmo de recorte adequado é Weiler-Atherton Santa Catarina (2024).

Uma vantagem significativa desse algoritmo é que ele preserva a estrutura do polígono, mantendo a ordem dos vértices e assegurando que o polígono recortado seja descrito por uma sequência de vértices consecutivos. Essa característica torna o Sutherland-Hodgman particularmente útil em sistemas gráficos interativos e sistemas de renderização que exigem operações de recorte em tempo real Foley et al. (1995), Akenine-Mooller et al. (2018).

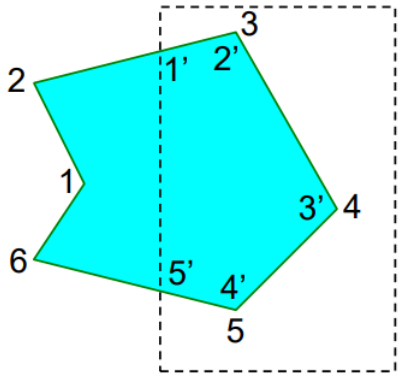


Figura 23 – Exemplo de recorte de um polígono contra uma janela retangular usando o algoritmo de Sutherland-Hodgman

Fonte: Santa Catarina (2024, p. 119)

A Figura 23 ilustra o processo de recorte de um polígono contra uma janela de visualização, destacando os novos vértices introduzidos em cada interseção com as bordas da janela. Esse processo assegura que o polígono final seja totalmente visível dentro da área de visualização especificada, facilitando a renderização da cena.

2.5.1.6.4 Recorte 3D

O recorte 3D é uma generalização do recorte 2D para o espaço tridimensional, onde as cenas são delimitadas por um volume de visualização (*frustum*), criando uma região visível em formato de um trapézio. Em comparação com o recorte 2D, o recorte 3D precisa lidar com coordenadas tridimensionais, o que requer transformações adicionais para normalizar o volume de visão em um paralelepípedo canônico (seções 2.5.1.2 e 2.5.1.4.2).

Enquanto o recorte 2D geralmente utiliza uma abordagem de plano único, o recorte 3D opera em múltiplos planos que definem o frustum, incluindo planos de recorte próximo e distante, além dos planos laterais. Isso permite que o recorte 3D determine que apenas as superfícies

visíveis e dentro do volume de visão sejam mapeados para coordenadas de tela. Assim, o recorte 3D, otimiza o processamento gráfico tornando-se indispensável para o desempenho e a qualidade visual das cenas tridimensionais Foley et al. (1995).

O recorte 3D adota um código de 6 bits, estendendo o padrão TBDE do algoritmo Cohen-Sutherland para TBDEAF. Os bits para qualquer ponto extremo (x, y, z) é setado conforme as condições, considerando o volume normalizado:

Sexto bit setado para 1: O ponto está frente da janela $\Rightarrow z > 1$.

Quinto bit setado para 1: O ponto está atrás da janela $\Rightarrow z < 0$.

Quarto bit setado para 1: O ponto está à esquerda da janela $\Rightarrow x < -1$;

Terceiro bit setado para 1: O ponto está à direita da janela $\Rightarrow x > 1$;

Segundo bit setado para 1: O ponto está abaixo da janela $\Rightarrow y < -1$;

Primeiro bit setado para 1: O ponto está acima da janela $\Rightarrow y > 1$.

Como no recorte 2D (2.5.1.6.1), a linha é trivialmente aceita quando a operação OR entre os códigos de suas extremidades for zero ou trivialmente rejeitada quando a operação AND for diferente de zero Foley et al. (1995).

Se uma linha não puder ser aceita ou rejeitada trivialmente, então devemos prosseguir com o algoritmo de recorte 3D. Durante o processo podemos calcular até seis interseções, uma para cada lado do paralelepípedo canônico. As interseções são calculadas através das equações paramétricas da linha composta por $P_0(x_0, y_0, z_0)$ até $P_1(x_1, y_1, z_1)$.

$$\begin{aligned} x &= x_0 + t(x_1 - x_0) \\ y &= y_0 + t(y_1 - y_0) \\ z &= z_0 + t(z_1 - z_0) \end{aligned} \tag{37}$$

Onde t, variando de 0 a 1, em conjunto com as equações 37, nos dá a coordenada de todos os pontos da linha de P_0 à P_1.

Para calcular a interseção da linha tomamos os planos do paralelepípedo canônico e substituimos as variáveis x, y e z com os valores máximos permitidos dentro do volume canônico: x variando de $[-1, 1]$, y variando de $[-1, 1]$ e z variando de $[0, 1]$. Com isto podemos isolar o valor de t nas equações 37, obtendo o conjunto de equações 38.

$$
\begin{aligned}
&\text{Plano na frente } z = 0 && \Rightarrow t = \frac{-z_0}{dz} \\
&\text{Plano atrás } z = 1 && \Rightarrow t = \frac{1 - z_0}{dz} \\
&\text{Plano da esquerda } x = -1 && \Rightarrow t = \frac{x_0 + (1 - z_0)}{-dx + dz} \\
&\text{Plano da direita } x = 1 && \Rightarrow t = \frac{-x_0 + (1 - z_0)}{dx + dz} \\
&\text{Plano de baixo } y = -1 && \Rightarrow t = \frac{-y_0 + (1 - z_0)}{dy + dz} \\
&\text{Plano do topo } y = 1 && \Rightarrow t = \frac{y_0 + (1 - z_0)}{-dy + dz}
\end{aligned}
\tag{38}
$$

2.5.1.7 Rasterização e Aplicação de efeitos visuais

A rasterização é a ultima etapa em ambos os *pipelines* apresentados neste trabalho. Ela visa converter representações geométricas vetoriais em imagens matriciais, onde cada pixel corresponde a uma amostra da cena a ser renderizada Hearn e Baker (1997). Esse processo envolve a transformação de primitivas gráficas, como linhas, polígonos e curvas, em uma grade discreta de pixels exibíveis em uma tela ou outro dispositivo de saída Akenine-Mooller et al. (2018). A rasterização é amplamente utilizada em gráficos computacionais devido à sua eficiência e capacidade de manipular grandes quantidades de dados de forma rápida, sendo fundamental para renderizações em tempo real, como em jogos e simulações Foley et al. (1993).

Durante a rasterização ambos os pipelines podem aplicar efeitos visuais para aprimorar a imagem final. Esses efeitos podem incluir sombreamento, iluminação, texturização e outros. A implementação específica desses efeitos pode variar, dependendo das escolhas e necessidades que a geração de imagens necessita. Neste trabalho foram aplicados apenas a iluminação local nos objetos.

Na rasterização os vértices dos triângulos, previamente transformados para o espaço da tela, são interpolados para determinar quais pixels pertencem ao interior do triângulo. A interpolação linear é utilizada para calcular atributos como cor, profundidade e coordenadas de textura para cada pixel. Além disso, o teste de profundidade (ou Z-buffer) é empregado para garantir que apenas os pixels visíveis, ou seja, aqueles mais próximos da câmera, sejam renderizados, descartando aqueles ocultos por outros objetos na cena Akenine-Mooller et al. (2018).

Foley et al. (1995) detalham o uso de algoritmos de preenchimento de polígonos como parte essencial do pipeline gráfico, onde o objetivo é determinar quais pixels em uma tela estão dentro dos limites de um polígono. Técnicas como o preenchimento por linhas de varredura (*scanlines*) são amplamente utilizadas para percorrer cada linha da tela e decidir quais pixels

devem ser preenchidos. Segundo Foley et al. (1995) a tarefa de preencher um polígono pode ser dividia em duas etapas:

Decidir que pixels pintar para preencher o polígono

Decidir com que valor pintar o pixel.

Decidir quais pixels devem ser preenchidos geralmente envolve percorres linhas de varredura sucessivas determinando suas interseções com as arestas da primitiva gráfica. Durante essa varredura, os pixels dentro do polígono são preenchidos em blocos adjacentes (spans) da esquerda para a direita. Esse processo ocorre para cada linha que cruza a área delimitada pela primitiva, garantindo que todos os pixels dentro dos limites do polígono sejam corretamente coloridos.

2.5.1.7.1 Interpolação Linear

A interpolação linear é um método matemático fundamental para calcular valores intermediários entre dois pontos conhecidos. No contexto da rasterização ela é utilizada para distribuir suavemente atributos dos vértices (como cor, profundidade e coordenadas de textura) entre os pixels que compõem um polígono. Essa técnica evita descontinuidades visuais, garantindo transições naturais entre os vértices.

A interpolação linear entre dois pontos P_{inicial} e P_{final}, com coordenadas $(x_{\text{inicial}}, y_{\text{inicial}})$ e $(x_{\text{final}}, y_{\text{final}})$, é definida pela taxa de variação m, calculada pela equação 39. A equação 40 permite calcular o valor de x_{i+1} a partir de uma posição x_i.

$$m = \frac{y_{\text{final}} - y_{\text{inicial}}}{x_{\text{final}} - x_{\text{inicial}}} \tag{39}$$

$$x_{i+1} = x_i + \frac{1}{m} \tag{40}$$

A equação 40 permite atualizar x incrementalmente ao longo de incrementos unitários em y (ou vice-versa). O mesmo princípio se aplica a outros atributos, como profundidade (z) ou cor, ajustando a taxa de variação m para cada atributo. A interpolação linear é essencial para algoritmos de preenchimento de polígonos por meio de linhas de varredura *scanlines*.

Por fim a interpolação linear unifica o tratamento de atributos ao longo dos eixos x, y e z, assegurando coerência visual e eficiência computacional. Sua aplicação incremental permite otimizações em GPUs, onde cálculos por pixel são críticos para desempenho e podem ser facilmente paralelizáveis Akenine-Mooller et al. (2018).

2.5.1.7.2 Interpolação ao longo do eixo y: entre linhas de varredura

Durante a rasterização, a interpolação ao longo do eixo y (figura 24) inicia com a identificação dos vértices extremos do polígono (ex., P_1 e P_3, com valores mínimo e máximo de y). Para cada linha de varredura (*scanline*) y_i, os pontos de interseção com as arestas são calculados usando interpolação linear (Seção 2.5.1.7.1). A taxa de variação m de cada aresta é determinada conforme a Equação 39, e o valor de x é atualizado incrementalmente para a próxima *scanline* (y_{i+1}) usando a Equação 40.

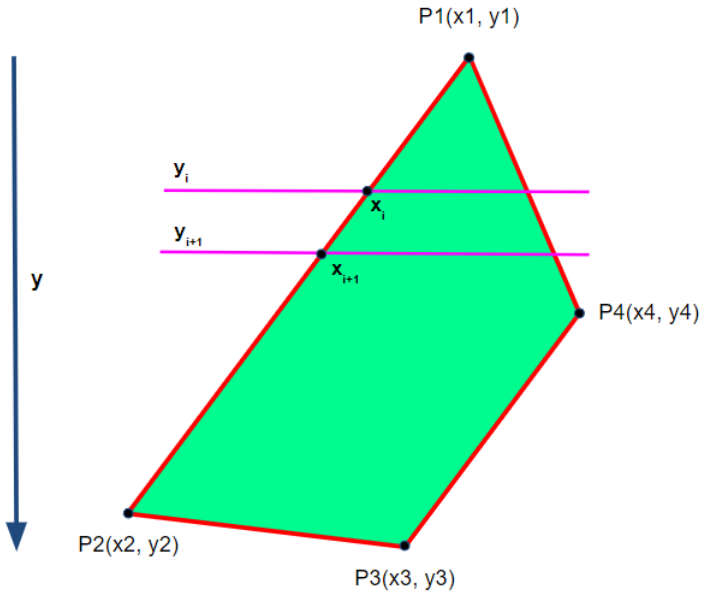


Figura 24 – Interpolação das interseções entre linhas de varredura e arestas ao longo do eixo y.

Esse processo é repetido para todas as arestas não horizontais, percorrendo-as de cima para baixo. A interpolação contínua de x garante que os pixels sejam preenchidos sem saltos ou sobreposições indevidas.

2.5.1.7.3 Interpolação ao longo do eixo x: dentro das linhas de varredura

Após definir os pontos de interseção das linhas de varredura com as arestas (via interpolação em y), a interpolação ao longo do eixo x (figura 25) preenche os pixels entre x_{inicial} e x_{final}. Os valores de x_{inicial} e x_{final} são arredondados para cima e para baixo, respectivamente, evitando pixels fora do polígono.

A interpolação linear também é aplicada a atributos como cor e textura ao longo de X. Por exemplo, a taxa de variação de um atributo A entre x_{inicial} e x_{final} é:

$$T_A = \frac{A_{\text{final}} - A_{\text{inicial}}}{x_{\text{final}} - x_{\text{inicial}}} \tag{41}$$

O valor de A para cada pixel é então atualizado incrementalmente:

$$A_{i+1} = A_i + T_A \tag{42}$$

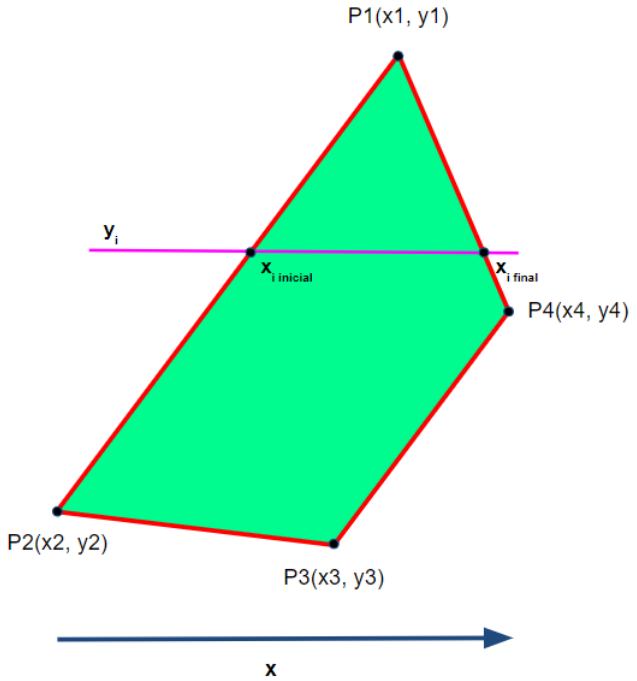


Figura 25 – Interpolação de vértices ao longo do eixo x durante o preenchimento de polígonos.

2.5.1.7.4 Interpolação da coordenada z

A profundidade (coordenada z) é interpolada linearmente e incrementalmente entre e dentro das linhas de varredura. Entre as linhas de varredura a interpolação acontece ao longo do eixo y e dentro das linhas de varreduras, ao longo do eixo x. Entre as linhas de varredura (em y), a taxa de variação T_z é calculada como:

$$T_z = \frac{z_{\text{final}} - z_{\text{inicial}}}{y_{\text{final}} - y_{\text{inicial}}} \tag{43}$$

Dentro das linhas de varredura (em x) ajustamos T_z para:

$$T_z = \frac{z_{\text{final}} - z_{\text{inicial}}}{x_{\text{final}} - x_{\text{inicial}}} \tag{44}$$

Essa *interpolação perspectiva-correta* (*perspective-correct interpolation*) compensa distorções causadas pela projeção 3D, garantindo precisão no cálculo de profundidade para o *z-buffer* (seção 2.5.1.5.2). A Figura 19 (exemplo de conflito resolvido pelo z-buffer) ilustra a importância dessa correção para evitar artefatos de oclusão.

2.5.1.8 Iluminação

Durante a rasterização a iluminação é aplicada para calcular a aparência final de cada pixel, levando em consideração a interação da luz com as superfícies dos objetos na cena. Foley et al. (1995) descrevem que o modelo de iluminação utilizado na computação gráfica pode incluir componentes de iluminação ambiente, difusa e especular, conforme mencionado anteriormente na seção 2.4.2.

A iluminação é um fator crucial na renderização de cenas 3D, pois é responsável por conferir um certo realismo aos objetos exibidos de acordo com a geometria da cena e com as propriedades dos materiais dos objetos (2.4.1.2). A rasterização, sendo um processo que discretiza a geometria 3D vetorial numa imagem 2D, utiliza a iluminação para calcular como a luz interage com os objetos da cena e como essa interação deve ser representada na tela.

A iluminação pode ser calculada de diversas maneiras, com os modelos de iluminação mais comuns o modelo de sombreamento constante (*flat shading*), o modelo Gouraud (1971) e o modelo Phong (1973). Esses modelos são responsáveis por definir como a luz reflete nas superfícies e como as cores resultantes devem ser aplicadas a cada pixel Akenine-Mooller et al. (2018).

2.5.1.8.1 Sombreamento constante

O sombreamento constante, ou *Flat Shading*, é uma técnica simples de sombreamento utilizada em computação gráfica para simular a iluminação de superfícies. No *Flat Shading* cada face do polígono tem sua iluminação total (cor) calculada em um ponto representativo da face (normalmente o centroide).

O sombreamento constante calcula a iluminação, uma única vez para a face que está sendo rasterizada, utilizando a normal da superfície do polígono, a posição da fonte de luz e a posição da câmera, de acordo com o modelo de iluminação empregado, neste caso, o modelo foi descrito através da soma da luz ambiente (seçao 2.4.2.3) com a reflexão difusa (seção 2.4.2.4) e a reflexão especular (seção 2.4.2.5). O valor resultante de iluminação total é aplicado uniformemente a todos os pixels do polígono, sem realizar interpolação entre e dentro das linhas de varredura. Esse método é computacionalmente eficiente, pois evita o cálculo contínuo da iluminação para cada pixel.

Como observado na Figura 26, o sombreamento constante cria um efeito visual em que cada face é facilmente distinguível, com transições abruptas entre elas. Esse efeito é evidente em superfícies curvas, como a esfera, onde as faces planas são visíveis e as bordas são acentuadas. Embora o *Flat Shading* seja eficiente, ele é limitado em termos de realismo visual, especialmente em superfícies curvas e detalhadas, em que uma transição mais suave de luz e cor seria esperada. Alternativas mais sofisticadas, como o sombreamento *Gouraud* e o sombreamento *Phong*,

oferecem uma interpolação de iluminação que suaviza as transições entre os vértices, resultando em um aspecto mais natural Hearn e Baker (1997).

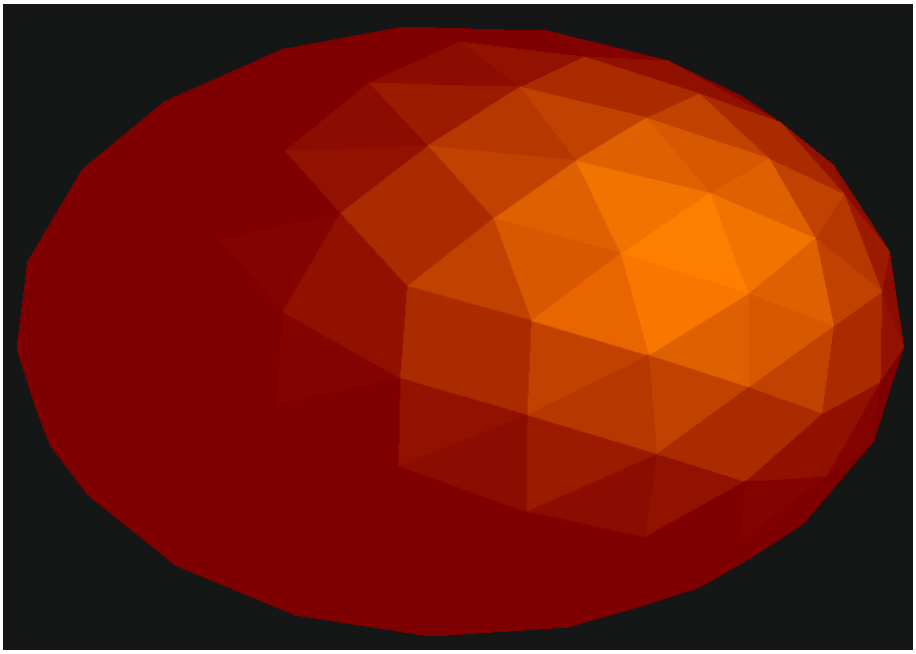


Figura 26 – Exemplo de uma esfera renderizada com sombreamento constante. Cada polígono tem uma cor uniforme, criando um efeito facetado.

Apesar de suas limitações, o sombreamento constante continua a ser uma escolha eficaz em situações onde o desempenho é uma prioridade ou quando se deseja um estilo visual mais geométrico e estilizado. Ele é amplamente utilizado em visualizações de modelos com baixo nível de detalhes, além de ser comum em aplicações que buscam um aspecto estético facetado. Um exemplo notável de seu uso vem da indústria de jogos, com títulos icônicos como *Zharch* da Atari (1987) e *Star Fox* da Nintendo (1993).

Em jogos mais recentes, como *Hotshot Racing* (Figura 27), o sombreamento constante deixou de ser apenas uma solução técnica e passou a ser uma escolha artística consciente. Utilizando-o, o jogo evoca uma estética retrô que remete à era dos gráficos poligonais e ao estilo visual dos primeiros jogos 3D. Em vez de adotar técnicas mais avançadas, como sombreamento *Gouraud*, sombreamento *Phong* ou *Ray Tracing*, os desenvolvedores optaram por um estilo simplificado e geométrico, criando uma identidade visual única. Essa abordagem demonstra como a simplicidade deste sombreador pode ser utilizada de forma deliberada para gerar um apelo nostálgico e artístico.

2.5.1.8.2 Sombreamento *Gouraud*

O sombreamento *Gouraud* foi proposto por Gouraud (1971). Diferente do sombreamento constante, que cria objetos com aparência facetada, o sombreamento *Gouraud* suaviza as diferenças de iluminação entre as faces. Isso é possível porque a cor ou intensidade de luz é

Figura 27 – Cena do jogo *Hotshot Racing*, que utiliza sombreamento constante para reforçar uma estética retrô.

calculada para cada vértice, com base na sua localização e no vetor normal médio do vértice. O resultado é um objeto 3D com um efeito de degradê suave entre as faces, eliminando o efeito de facetas, como observado na Figura 26.

Durante o processo de rasterização, o sombreamento *Gouraud* é aplicado após o cálculo da iluminação para cada vértice do polígono, gerando um par (vértice, cor). Isso permite que a reflexão difusa e a reflexão especular variem para cada vértice, o que pode resultar em vértices mais iluminados ou mais escuros na mesma superfície.

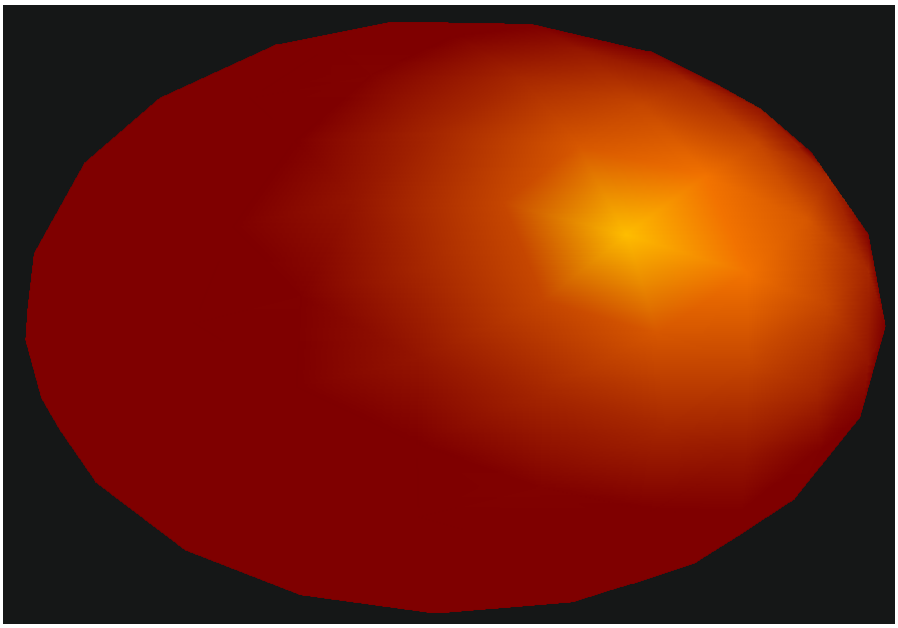


Figura 28 – Exemplo de uma esfera renderizada com sombreamento *Gouraud*, mostrando a suavização entre as faces.

A grande diferença entre o sombreamento *Gouraud* e o sombreamento constante está na maneira como a cor é aplicada: enquanto o segundo utiliza uma única cor para toda a superfície, o primeiro interpola os valores de cor linearmente entre os vértices ao longo da superfície. Esse

processo de interpolação é semelhante ao descrito na sessão 2.5.1.7.1, mas neste caso, ao invés de interpolar coordenadas, os valores de cor de cada vértice são interpolados para produzir um sombreamento suave. A Figura 28 ilustra esse efeito, mostrando como o sombreamento *Gouraud* elimina as transições bruscas entre as faces.

Dado que cada vértice de um polígono possui uma cor associada, a interpolação das cores, em seus canais R, G e B, entre e dentro das linhas de varredura ocorrem da mesma maneira que a interpolação das coordenadas z, conforme apresentado na seção 2.5.1.7.4.

2.5.1.8.3 Sombreamento *Phong*

O sombreamento *Phong*, proposto por Phong (1973), representa uma evolução significativa em relação ao sombreamento constante e ao sombreamento *Gouraud*. A principal diferença está no fato de que as normais dos vértices são interpoladas em cada pixel, e a iluminação é calculada diretamente em nível de pixel, o que resulta em uma suavização significativa nas transições de luz e sombra. Isso é especialmente importante em superfícies especulares, onde os reflexos de luz podem ser mais pronunciados e precisos.

No entanto, ao analisar a Figura 29 notamos que, apesar do aumento da qualidade nos reflexos especulares, a figura ainda apresenta um aspecto facetado. Esse efeito ocorre porque, na implementação realizada, o vetor da direção de observação ($\hat{S}$) é calculado com base no centróide da face e não em cada pixel individual. Como resultado, a reflexão especular, embora visivelmente mais forte e precisa, ainda não é suavizada adequadamente entre os polígonos que compõem o objeto, causando a aparência facetada, que seria corrigida se a direção da observação fosse calculada por pixel, sendo este um dos problemas de subjetividade da aplicação da teoria na prática mencionados na introdução.

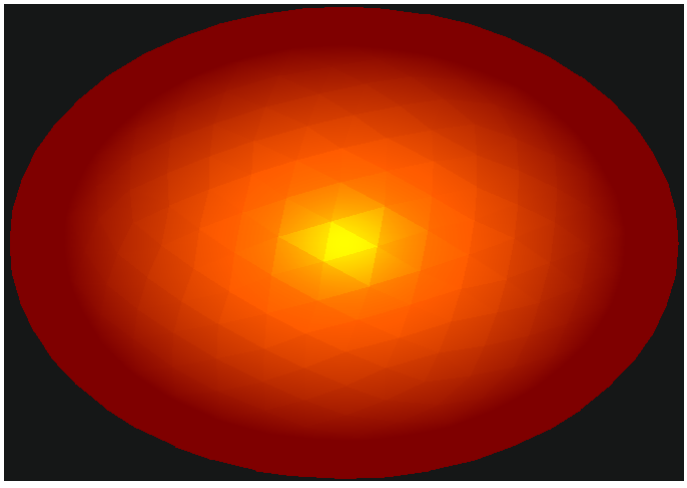


Figura 29 – Exemplo de uma esfera renderizada com sombreamento *Phong*. Note o aumento da reflexão especular, mas com o efeito de facetas devido ao cálculo do vetor de observação com base no centroide da face.

Apesar desse detalhe, o sombreamento *Phong* ainda representa uma solução muito mais realista do que o sombreamento constante ou o sombreamento *Gouraud*, particularmente em superfícies brilhantes e curvas. A interpolação das normais dos vértices entre e dentro das linhas de varredura permite que tanto a reflexão difusa quanto a especular sejam ajustadas com maior precisão para cada pixel, resultando em reflexos mais suaves e transições mais naturais de iluminação, especialmente em objetos especulares, como é visível na reflexão da esfera.

A interpolação dos vetores normais médios nos vértices, em suas coordenadas i, j e k, é realizada da mesma maneira que a interpolação das coordenadas z, conforme apresentado na seção 2.5.1.7.4. Esta interpolação permite que a iluminação seja calculada de forma precisa em cada pixel, resultando em transições suaves de luz e sombra.

3

Ambiente de desenvolvimento e Tecnologias

3.1 Ambiente

O desenvolvimento da aplicação e os testes de execução foram realizados em um notebook com processador Intel Core i5-12500H, com memória RAM DDR4 de 8GB, SSD NVMe com 512GB, placa gráfica NVIDIA GeForce RTX 3050 com VRAM de 4 GB GDDR6. O sistema operacional empregado foi o Windows 11 Home x64.

3.2 Tecnologias utilizadas

Além do hardware gráfico, o desenvolvimento, necessitou também de uma série de softwares e tecnologias para a implementação do presente trabalho. A seguir será discorrido sobre estas tecnologias e softwares utilizadas, bem como o por que da escolha delas.

3.2.1 Interface de Desenvolvimento (IDE)

O **Visual Studio Code** (VSCode) é uma ferramenta bastante popular entre desenvolvedores que utilizam Windows, conhecida pela sua flexibilidade e suporte a diversas linguagens de programação. No contexto deste trabalho, o uso do VSCode se mostrou especialmente vantajoso por algumas razões:

Suporte a múltiplas linguagens: O VSCode permite trabalhar com várias linguagens, como Python, C++ e JavaScript, o que facilita o desenvolvimento de projetos que combinam diferentes tecnologias.

Ampla extensibilidade: Com uma vasta biblioteca de extensões, o VSCode possibilita adicionar funcionalidades específicas para C++, como destaque de sintaxe, depuração avançada e integração fluida com sistemas de controle de versão como o Git.

Personalização da interface: A IDE permite ajustes de temas, atalhos e *layout*, o que torna a experiência de uso mais confortável e adaptada às preferências pessoais de cada desenvolvedor.

Gratuito e de código aberto: Sendo uma ferramenta de código aberto, o VSCode elimina a necessidade de licenças, o que é um ponto positivo em projetos que buscam manter custos baixos.

3.2.2 Linguagem C++

A linguagem **C++** foi escolhida para este projeto por sua eficiência e seu controle refinado de baixo nível, qualidades que a tornam ideal para o desenvolvimento de sistemas que exigem alto desempenho e gerenciamento preciso de recursos (STROUSTRUP, 2013). Entre as principais razões para a utilização do C++ neste trabalho, destacam-se:

Desempenho e Eficiência: C++ permite um controle detalhado sobre o hardware, sem abrir mão de suas abstrações de alto nível, tornando-a uma escolha sólida para aplicações que demandam desempenho elevado.

Programação Orientada a Objetos (POO): Com um suporte robusto ao paradigma de POO, o C++ facilita a organização modular do código e a reutilização de componentes, contribuindo para um desenvolvimento mais estruturado e de fácil manutenção.

Bibliotecas e Ferramentas Disponíveis: A linguagem conta com uma vasta coleção de bibliotecas padrão, como a STL (Standard Template Library), que oferece estruturas de dados eficientes e algoritmos prontos para uso. Adicionalmente, o projeto utiliza bibliotecas como ImGui e SDL2 para suportar a interface gráfica.

Flexibilidade para Programação de Baixo Nível: C++ oferece recursos como ponteiros e manipulação direta de memória, permitindo uma proximidade maior com o hardware sempre que necessário, sem comprometer as abstrações de alto nível.

Além dessas características, o ambiente de desenvolvimento inclui uma ferramenta importante para gerenciar as dependências do projeto, garantindo portabilidade entre sistemas operacionais e arquiteturas e otimizando o processo de compilação (MEYERS, 2017).

A escolha de compiladores foi estrategicamente orientada por critérios de compatibilidade e otimização específicos para cada sistema operacional. Na plataforma Windows, adotou-se o

Microsoft Visual C++ Compiler (MSVC), reconhecido por sua integração nativa com as APIs do sistema e amplo suporte às ferramentas de desenvolvimento da Microsoft (Microsoft, 2022). Para o ambiente Linux, selecionou-se o **Clang/LLVM**, destacando-se tanto por sua rigorosa conformidade com os padrões ISO C++ quanto por seu mecanismo de otimização multinível, particularmente eficaz em ecossistemas *open-source* (LLVM Project, 2021).

3.2.3 Microsoft Visual C++ Compiler (MSVC)

O *Microsoft Visual C++ Compiler (MSVC)* é o compilador padrão para desenvolvimento em C++ na plataforma Windows e está integrado ao Visual Studio, podendo também ser usado com o VSCode (Microsoft, 2022). O MSVC é amplamente empregado em aplicações que requerem integração completa com o ecossistema Windows, oferecendo suporte avançado para as APIs nativas do sistema. Além de aderir aos padrões da linguagem C++, o MSVC fornece uma série de otimizações específicas para o Windows, aumentando o desempenho das aplicações executadas nesse sistema. Outro diferencial é o conjunto de ferramentas de depuração e *profiling*, que facilita a identificação de problemas de desempenho e bugs, permitindo um desenvolvimento mais eficiente. Sua estabilidade e ampla documentação também fazem do MSVC uma escolha confiável e prática. A versão do MSVC utilizada neste projeto é a 14.40.33807

3.2.4 GitHub Copilot

O **GitHub Copilot** foi utilizado como uma ferramenta de suporte ao desenvolvimento, proporcionando sugestões de código e auxiliando na criação de documentação padronizada para as funções. Fornecido em parceria através de uma licença de estudante, o Copilot utiliza inteligência artificial para sugerir trechos de código com base no contexto, ajudando a acelerar o processo de desenvolvimento e a manter a consistência na documentação e implementação das funções.

3.2.5 Clang

O **Clang** é um compilador moderno para C e C++, parte do projeto LLVM, amplamente reconhecido pela alta conformidade com os padrões da linguagem e pelas mensagens de erro e avisos detalhados, que facilitam a identificação e correção de problemas no código (LATTNER; ADVE, 2004). Utilizado em sistemas Unix, como Linux e macOS, o Clang possui um sistema de otimização agressivo e eficiente, o que aumenta o desempenho das aplicações compiladas. Sua modularidade permite fácil integração com outras ferramentas e pipelines de build, tornando-o ideal para projetos que requerem flexibilidade e portabilidade entre plataformas. Além disso, o Clang suporta várias extensões avançadas, como o sanitizador de código, uma ferramenta que

ajuda a identificar erros de memória, garantindo maior segurança e robustez no desenvolvimento em ambientes Linux.

Embora o projeto tenha sido desenvolvido inteiramente no windows, alguns testes foram realizados num ambiente linux Ubuntu 22.04 LTS utilizando o Clang na versão 14.0.0-1ubuntu1.1.

3.2.6 Ferramenta de Build Xmake

Para facilitar o processo de compilação e gerenciamento de dependências no projeto, foi utilizado o **Xmake**, uma ferramenta de *build* moderna e de código aberto. O Xmake se destaca pela simplicidade de uso e pela flexibilidade no suporte a múltiplas plataformas, no contexto deste projeto, foi escolhido como ferramenta de build, pois:

Sintaxe Simples: Xmake utiliza um arquivo de configuração escrito em Lua, o que torna o processo de configuração intuitivo e menos verboso em comparação a outras ferramentas de build, como CMake. Esse arquivo, conhecido como ‘xmake.lua‘, permite definir dependências, configurações de compilação e outras instruções de forma organizada e fácil de manter.

Gerenciamento de Dependências Integrado: Xmake oferece suporte nativo para o gerenciamento de dependências, permitindo buscar e instalar pacotes de bibliotecas diretamente dos repositórios oficiais de grandes projeto. Esse recurso elimina a necessidade de configurações adicionais para integração de bibliotecas externas, simplificando o fluxo de trabalho e reduzindo o tempo de configuração do projeto.

Suporte Multiplataforma: A ferramenta é compatível com diversas plataformas, incluindo Windows, macOS e Linux. Essa compatibilidade facilita a portabilidade do projeto, permitindo que o código seja compilado e executado de forma consistente em diferentes sistemas operacionais.

Modos de Compilação Otimizados: Xmake permite configurar diferentes modos de compilação, como ‘test‘, ‘debug‘ e ‘release‘, possibilitando a personalização de flags de compilação para otimizar o desempenho do binário final ou facilitar a depuração durante o desenvolvimento. Além disso o Xmake possui um sistema de compilação inteligente e caches, que só recompilam aquilo que mudou no código, acelerando o processo de compilação.

A escolha do Xmake como ferramenta de *building* se deu principalmente pela sua capacidade de gerenciar bibliotecas de terceiros, a criação de *scripts* de *building* para diferentes plataformas de maneira simples.

3.2.7 Bibliotecas Utilizadas

Neste projeto foram utilizadas várias bibliotecas para atender às necessidades de interface gráfica, interação com o sistema operacional e testes unitários. A escolha dessas bibliotecas visa otimizar o desenvolvimento e sua escolha foi baseada na usabilidade e integração das soluções, sendo estas soluções amplamente utilizadas no mundo da computação gráfica devido a sua simplicidade de uso, não impondo barreiras técnicas grandes ou requerindo um grande conhecimento técnico para sua utilização. Além disso, sua integração com a ajuda da ferramenta Xmake (3.2.6) e a quantidade de usuários em foruns de programação especializados nestas bibliotecas tornou a adesão delas neste projeto um ponto relevante. A seguir, são detalhadas as principais bibliotecas utilizadas.

3.2.7.1 Dear ImGui

A biblioteca **Dear ImGui** é uma interface gráfica para usuários (GUI) voltada para aplicações que requerem uma interface imediata e leve. Sua principal característica é o uso de uma abordagem de "GUI imediata", onde cada elemento da interface é atualizado e desenhado a cada quadro, o que facilita a criação e manipulação de interfaces dinâmicas. No contexto deste projeto Dear ImGui (versão 1.91.0) foi escolhido pela sua simplicidade e flexibilidade na criação de elementos de interface, como botões, sliders e menus, que são necessários para interagir com o sistema de maneira intuitiva. Além disso, Dear ImGui é altamente customizável e oferece excelente suporte para integração com outras bibliotecas gráficas.

Sua utilização é extremamente simples, com uma sintaxe descomplicada, sendo simples criar interfaces. Entretanto, um ponto negativo é a falta de suporte a criação de *layouts*, já que a biblioteca é focada para a criação de jogos, e suas janelas são flutuantes. Havendo a necessidade de se calcular a posição de cada elemento na tela, para assim criar um *"layout"*.

3.2.7.2 SDL2

A **Simple DirectMedia Layer 2** (SDL2) é uma biblioteca que fornece uma interface de baixo nível para acessar o hardware, permitindo que aplicativos C++ se comuniquem diretamente com o sistema operacional e seus recursos gráficos, sonoros e de entrada. No projeto a SDL2 (versão 2.30.5) foi utilizada para gerenciar janelas, capturar eventos de teclado e mouse e para lidar com as operações gráficas de forma eficiente. Essa biblioteca facilita o desenvolvimento multiplataforma, já que abstrai os detalhes específicos de cada sistema operacional, garantindo que o projeto funcione de maneira consistente em diferentes ambientes.

3.2.7.3 Dear ImGui SDL2 Integration (ImGui-SDL2)

Para integrar Dear ImGui com a SDL2 foi utilizada a biblioteca **ImGui-SDL2**. Essa biblioteca oferece uma camada de compatibilidade que permite utilizar as funcionalidades da SDL2 junto com a interface gráfica de Dear ImGui, facilitando o uso simultâneo de ambas as bibliotecas. Com o ImGui-SDL2 foi possível capturar eventos de entrada e gerenciar o ciclo de atualização e renderização de forma sincronizada entre a SDL2 e a interface do Dear ImGui, proporcionando uma experiência gráfica fluida e interativa.

Na biblioteca Dear ImGui este tipo de integração é chamado de *backend*, e os próprios criadores da biblioteca recomendam utilizar um *backend* para cuidar do desenho na tela, pois, o imgui fornece muito mais um conjunto de ferramentas para a criação de janelas, mas ele não é responsável por desenhar as primitivas em tela. Para isto ele utiliza um backend (*OpenGL*, *SDL*, *Vulkan*, etc).

3.2.7.4 Google Test

Para garantir a qualidade e a confiabilidade do código foi utilizado o **Google Test (GTest)**, uma biblioteca para testes unitários em C++ desenvolvida pelo Google. A versão utilizada no projeto foi a 1.14.0 do GTest, sendo utilizada para cobrir diferentes funcionalidades e módulos do projeto. GTest oferece uma estrutura organizada para definir testes, facilitando a verificação do comportamento esperado de cada componente e acelerando o processo de desenvolvimento, uma vez que erros podem ser detectados rapidamente ainda em estágios iniciais. O uso de testes unitários com GTest contribui para a estabilidade do projeto e assegura que alterações no código não introduzam comportamentos inesperados.

4
Metodologia

Neste capítulo descrevemos a metodologia de desenvolvimento deste trabalho; nele iremos discorrer sobre a implementação realizada e abordaremos as divergências teóricas encontradas na literatura, bem como as subjetividades encontradas em aplicar a teoria na prática. Este capítulo se divide em duas sessões.

Na primeira sessão abordaremos a implementação, detalhes de arquitetura do software, decisões de projetos tomadas durante a implementação, bem como adentrar em mais detalhes técnicos sobre o funcionamento do pipeline de visualização 3D.

Na segunda seção apresentaremos as divergências entre alguns dos métodos utilizados nos pipelines de visualização 3D propostos por diferentes autores, levantaremos algumas hipóteses sobre diferentes métodos de implementação onde a literatura não é clara, além de definirmos critérios de avaliação gerados pelos *pipelines A e B* propostos por Santa Catarina.

4.1 Implementação

A fase da implementação deste trabalho é fulcral, logo não poderia ser feita sem haver antes um planejamento de como estruturar a arquitetura do software, para que resultasse numa organização intuitiva que simplifica a localização de elementos/processos do *pipeline* e, principalmente, facilita as manutenções futuras.

Pensando nisso o sistema foi dividido em "módulos", como demonstra a figura 30. Nela observam-se que há seis módulos principais, que são identificados com o texto no canto inferior direito, sendo eles:

MRX GUI: Responsável pela interface gráfica e pelo processamento de entradas e saídas.

MRX Models: Contém os modelos essenciais para a cena 3D, incluindo câmera, iluminação, cores, malhas poligonais e a própria cena 3D em si.

MRX Core: Abriga as principais estruturas de dados utilizadas para representar objetos 3D, como vértices, vetores físicos, half-edge e faces.

MRX Math: Oferece funções matemáticas para manipulação de vetores e matrizes, além de métodos do pipeline, como transformações geométricas, rasterização, recortes e ocultação de superfícies.

MRX Utils: Inclui funções de desenho que utilizam as bibliotecas gráficas do projeto.

MRX Shapes: Fornece métodos para a geração procedural de formas geométricas usadas na construção da cena 3D.

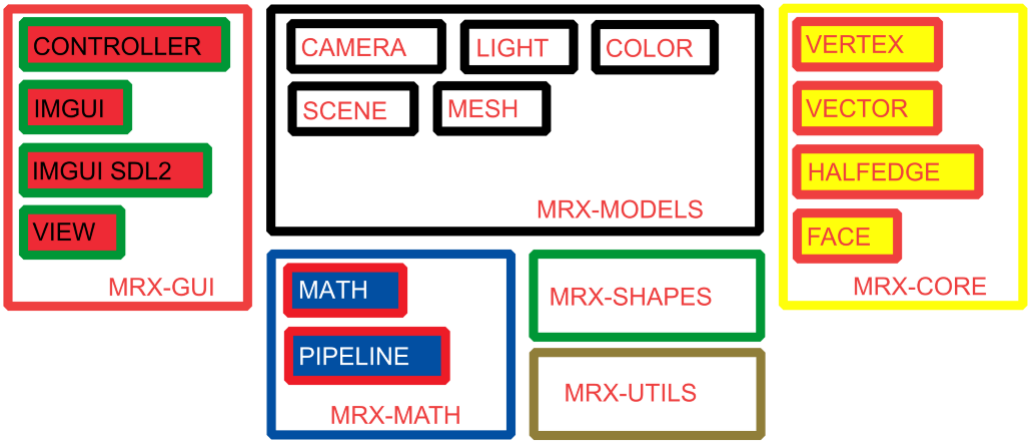


Figura 30 – Arquitetura do software

A separação dos módulos, ilustrada na Figura 30, foi definida com base nas responsabilidades fundamentais de um sistema gráfico, onde cada módulo possui funcionalidade específica. Os nomes dos módulos foram escolhidos para antecipar seu conteúdo ao usuário e o prefixo *"MRX"* foi escolhido para nomear o sistema em homenagem ao filósofo alemão Karl Marx e em similaridade com o nome do autor deste trabalho. A seguir, discutiremos individualmente cada módulo.

4.1.1 Repositório do Projeto

O código-fonte completo deste projeto está disponível em um repositório público no GitHub[1]. Esse repositório contém a estrutura completa do sistema, incluindo os módulos e submódulos discutidos ao longo deste trabalho, além de instruções para instalação e execução. A disponibilização do projeto visa facilitar o acesso ao código, além de incentivar a colaboração e o desenvolvimento contínuo da aplicação.

[1] Repositório do projeto: <https://github.com/Kyhaiu/mrx-scene>

4.1.2 Documentação do Sistema

Para este projeto, as funções foram documentadas utilizando o **Copilot** (3.2.4) como suporte na geração de comentários padronizados. O padrão de documentação adotado inclui descrições detalhadas para cada função, seguindo a estrutura de:

@brief: Uma breve descrição do propósito da função.

@param: Descrição de cada parâmetro da função, especificando seu papel na função.

@return: Explicação sobre o valor de retorno da função.

```
/**
 * @brief Calcula o código de saída de um ponto em relação a uma janela de recorte.
 *
 * @param p Ponto a ser verificado
 * @param min Canto inferior esquerdo da janela de recorte
 * @param max Canto superior direito da janela de recorte
 * @return Código de saída do ponto
 *
 * @note O código de saída é uma combinação de bits que indicam a posição do ponto em relação à janela de recorte.
 * @note O código de saída é calculado da seguinte forma:
 * @note 0000: Ponto dentro da janela
 * @note 0001: Ponto à esquerda da janela
 * @note 0010: Ponto à direita da janela
 * @note 0100: Ponto abaixo da janela
 * @note 1000: Ponto acima da janela
 */
int compute_outcode(core::Vector3 p, core::Vector2 min, core::Vector2 max)
```

Figura 31 – Documentação da função *compute_outcode* utilizada no recorte 2D.

Adicionalmente, para algumas funções, foram incluídos comentários suplementares, como **@note** e **@example**, que oferecem informações sobre o comportamento da função ou exemplos de uso. A Figura 31 ilustra um exemplo de documentação de função, no qual o comentário segue esse padrão e inclui notas para esclarecer aspectos específicos da função ou do valor de retorno.

Embora o Copilot tenha auxiliado na criação da documentação inicial, toda a documentação foi revisada manualmente pelo autor para assegurar precisão e clareza. Durante a revisão, ajustes e correções foram feitos conforme necessário, garantindo que a documentação estivesse correta e atendesse ao código implementado.

Esse padrão de documentação garante consistência e facilita a compreensão do código, especialmente em funções que o significado não é claro devido a nomenclatura. A documentação permite que outros desenvolvedores ou usuários do código compreendam rapidamente o funcionamento e a finalidade de cada função, além de esclarecer a lógica interna quando necessário.

4.1.3 Dependências entre Módulos e Submódulos

A Figura 30 mostra a estrutura do sistema, composta por seis módulos principais divididos em um total de dezessete submódulos. Dada essa quantidade de submódulos é natural que eles não sejam completamente autocontidos (independentes do resto do sistema); pelo contrário, dependências entre eles são esperadas e fazem parte da arquitetura do código.

Para evitar problemas de compilação relacionados a dependências circulares (por exemplo, quando o módulo 1 depende do módulo 2, que por sua vez depende do módulo 1), e otimizar o processo de compilação, foram adotadas duas estratégias importantes.

Primeiramente, utiliza-se a técnica de *forward declaration*, que permite declarar classes e estruturas antes de sua definição completa. Essa abordagem permite que o compilador reconheça referências a essas classes sem exigir a inclusão de todo o código-fonte de cada uma delas. Em complemento foi aplicada a diretiva de compilação *#pragma once*, garantindo que cada arquivo de cabeçalho seja incluído apenas uma vez durante o processo de compilação. Isso evita múltiplas inclusões e possíveis conflitos, assegurando um processo de compilação mais eficiente e organizado.

Para evitar redundâncias nas seções seguintes e orientar o leitor sobre a convenção adotada nas figuras de dependências, explicaremos aqui como interpretar as direções das setas nas ilustrações. Nas figuras, as setas indicam a direção da dependência: elas partem do submódulo solicitado e apontam para o submódulo que depende dele. Por exemplo, na Figura 32, o submódulo *vertex* depende do submódulo *vector*, de modo que a seta parte de *vector* para *vertex*, sinalizando essa relação.

4.1.4 MRX Core

Começando pela parte mais básica e central do software, seu nome ser "MRX Core", não foi uma escolha ao acaso, como o nome indica "Core"do inglês significa "centro", "parte central". E como significado nos diz, este módulo é central a todo o sistema, pois o uma cena tem como elemento principal os objetos que a compõem, sem eles uma cena é só um espaço vazio que não representa nada e não nos diz nada. Este módulo é responsável por conter as estruturas de dados que são utilizadas para representar os objetos 3D.

Na figura 30 temos o "MRX Core"representados pelas cores vermelho e amarelo a direita na figura, onde dentro do módulo podemos identificar vários submódulos sendo eles:

Vertex: Este submódulo contém a representação orientada a objetos do submódulo Vector.

Vector: Fornece estruturas para representar vértices, vetores físicos em 2D, 3D e 4D. Além de algumas funções na criação e manipulação deste vetores, como inicializar um vetor com todos os elementos zerados ou restringir um vetor num intervalo.

Half-Edge: Contem a estrutura half-edge descrita na sessão (2.3.1).

Face: Representa a superfície de um polígono.

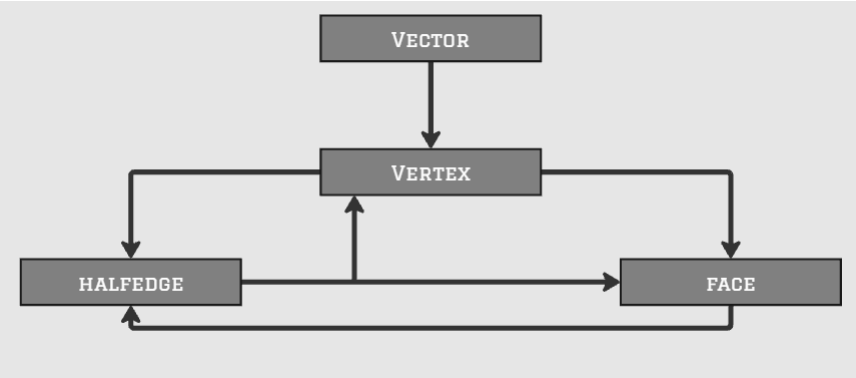


Figura 32 – Diagrama de dependências do módulo MRX Core

Esses submódulos possuem uma forte interdependência, conforme ilustrado na Figura 32. Ainda assim, o módulo é autossuficiente, ou seja, não depende de nenhum outro módulo externo, exceto pelos módulos fornecidos pela STL (biblioteca padrão) do C++.

Essas dependências surgem devido à estrutura de dados Half-Edge, conforme descrito na Seção 2.3.1, onde cada elemento da estrutura mantém uma referência a outros elementos, estabelecendo uma relação interna entre os componentes. Essa interdependência cria um padrão de dependências circulares, como ilustrado na Figura 32. Para evitar erros de compilação decorrentes dessas dependências circulares, empregam-se duas técnicas essenciais descritas na sessão 4.1.3.

4.1.5 MRX Models

Seguindo pelo segundo mais importante modulo do sistema e o maior em termos de submódulos, temos o "MRX Models"que, como o nome indica, é responsável por conter a representação dos modelos que compõem uma cena 3D. Neste módulo temos um total de cinco submódulos, cada um deles responsável por um aspecto da cena, sendo eles:

Color: Submódulo que contém a estrutura de cores, valores padrão de cores e funções pertinentes a cores como conversão da representação de cores do sistema para as estruturas de cores do ImGui.

Light: Contém os modelos de iluminação que o sistema tem suporte (2.5.1.8.1, 2.5.1.8.2, 2.5.1.8.3) além da estrutura que representa as lâmpadas (2.4.2.3).

Câmera: Fornece o modelo de câmera virtual (2.4.3) utilizado na cena, além de métodos de movimentação da câmera.

Mesh: Abriga a estrutura de malha poligonal (**??**) utilizada para representar objetos da cena.

Scene: Submódulo principal do MRX Models, pois ele é utilizado para representar a cena 3D, além de fornecer métodos para modificação da cena.

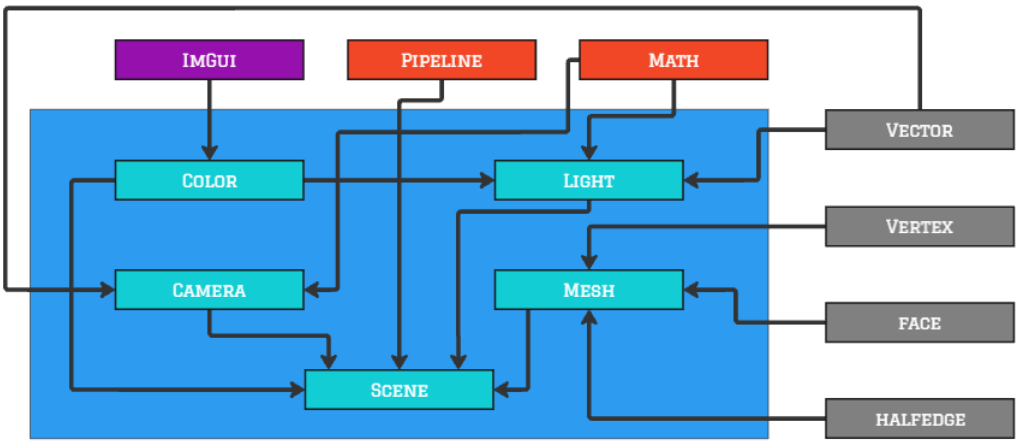


Figura 33 – Diagrama de dependências do módulo MRX Models

A figura 33 ilustra as dependências entre os submódulos do módulo MRX Models, indicando como cada submódulo interage e depende dos outros. Cada caixa representa um submódulo e as setas mostram as relações de dependência. Uma seta partindo do submódulo requerido em direção ao submódulo que depende dele assim como descrito na sessão anterior 4.1.4.

4.1.5.1 Submódulo Color

O submódulo **Color** depende do **ImGui**, localizado no módulo MRX GUI. Esse submódulo é responsável pela representação e manipulação das cores no sistema gráfico. A opção de utilizar uma estrutura própria para cores, em vez das soluções oferecidas diretamente pelo ImGui ou pela SDL, foi motivada pela maior portabilidade de código.

Ao adotar uma representação de cores independente, o sistema gráfico ganha flexibilidade para realizar trocas de bibliotecas de interface gráfica sem precisar de grandes modificações em sua implementação de cores. Dessa forma, se for necessário substituir a biblioteca responsável pela interface gráfica, bastará reescrever a função de mapeamento da estrutura **Color** para a nova representação de cores fornecida pela biblioteca escolhida, evitando alterações extensas no restante do código. Essa abordagem modular e independente facilita a manutenção e a escalabilidade do sistema, garantindo maior adaptabilidade a mudanças futuras nas dependências gráficas.

4.1.5.2 Submódulo Camera

O submódulo **Camera** possui dependências nos módulos **Vector** e **Math**. A dependência de **Vector** é essencial para representar a posição da câmera no espaço tridimensional, enquanto o **Math** fornece as funções matemáticas necessárias para operações avançadas com vetores, indispensáveis para o controle da movimentação da câmera.

4.1.5.3 Submódulo Light

O submódulo **Light** depende de três submódulos: **Math**, **Color** e **Vector**. O submódulo **Math** é utilizado para cálculos relacionados ao modelo de iluminação utilizado, além da movimentação da luz no espaço tridimensional. O submódulo **Color** é utilizado para permitir o ajuste das cores de iluminação, e o submódulo **Vector** auxilia na definição de direções e posições das fontes de luz.

4.1.5.4 Submódulo Mesh

O submódulo **Mesh** depende dos submódulos **Vertex**, **Face** e **Halfedge**. Essas dependências são essenciais para a definição e manipulação de malhas poligonais, uma vez que **Vertex** representa os vértices das malhas, **Face** representa as faces e **Halfedge** fornece a estrutura de dados adequada para representar a conectividade das faces e arestas, garantindo uma topologia consistente.

4.1.5.5 Submódulo Scene

O submódulo **Scene** possui um conjunto mais amplo de dependências, incluindo os submódulos **Pipeline**, **Camera**, **Light**, **Color** e **Mesh**. Essas dependências permitem que o **Scene** integre todos os elementos da cena 3D, configurando a câmera, a iluminação, os objetos e suas propriedades de material e forma. Além disso é necessário utilizar o submódulo Pipeline (4.1.6.2), que é responsável por aplicar o pipeline durante a renderização da cena. A centralização dessas dependências no submódulo **Scene** é fundamental para gerenciar e organizar a renderização completa da cena num único lugar.

4.1.6 MRX Math

Na sequência temos o módulo que cuida da parte matemática e dos processos do pipeline, sendo o MRX Math responsável por conter todos os métodos do pipeline (com exceção a ocultação das superfícies pelo calculo da normal 2.5.1.5.1). Neste módulo temos um total de dois submódulos, sendo eles:

Math: Este submódulo é responsável por realizar todos os cálculos vetoriais e matriciais do sistema.

Pipeline: Cotém todos os métodos do pipeline de visualização descritos na sessão 2.5

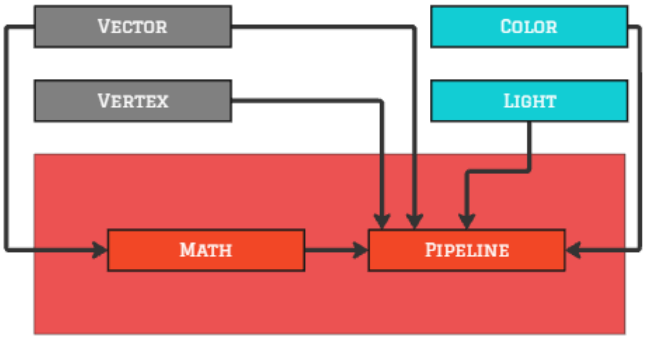


Figura 34 – Diagrama de depêndencias do módulo MRX Math

4.1.6.1 Submódulo Math

A figura 34 mostra as dependências entre os submódulos descritos anteriormente. O submódulo **Math** possui funções de cálculo autocontidas, ou seja, não depende de funções externas, exceto aquelas fornecidas pela biblioteca padrão de matemática ("math.h") da linguagem C. No entanto, ele ainda depende do submódulo **Vector**, presente no módulo MRX Core, pois é lá que estão definidas as estruturas de vetores e matrizes utilizadas por **Math**.

4.1.6.2 Submódulo Pipeline

Para o submódulo **Pipeline** temos as funções que executam etapas do pipeline, descritos na sessão 2.5. O módulo depende dos submódulos Math, Light (4.1.5.3), Color (4.1.5.3) e os submódulos Vector e Vertex contidos no módulo MRX Core (4.1.4). Como o pipeline possui cálculos vetoriais durante a conversão de coordenadas do SRU para o SRC, conforme descrito em 2.5.1.1.4, é necessário utilizar o submódulo Math.

Para o submódulo Light, ele é utilizado para aplicar o modelo de iluminação (2.4.2) durante a rasterização (2.5.1.7), além disso para se utilizar das definições de tipagem contidas no submódulo, assim como para Color, Vector e Vertex.

4.1.7 MRX Utils

O módulo MRX Utils é um módulo utilizado para funções de utilidade do projeto, neste contexto, este módulo é responsável por conter as primitivas gráficas (funções de desenho de pixels, linhas e superfícies) que desenham na memórias de cor e profundidade (2.5.1.5.2), e

também para desenhar a memória de cor na tela utilizando funções do ImGui, além de possuir as funções de manipulação de arquivos necessárias para se salvar e carregar uma cena no projeto A.4.

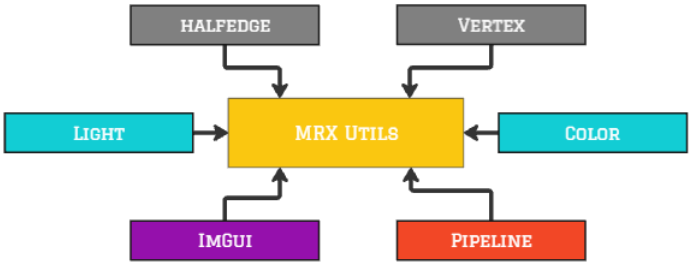


Figura 35 – Diagrama de dependências do módulo MRX Utils

Na figura 35 podemos ver as dependências de vários submódulos de diferentes módulos do sistema. Isso ocorre pois temos que desenhar vértices e linhas e superficies, para isso dependemos do submódulo Vertex, que contém a definição dos vértices e do submódulo HalfEdge, por causa das definições de tipo da estrutura. Além disso precisamos do submódulo ImGui para utilizarmos suas funções de desenho em tela para desenhar a memória de cor.

4.1.8 MRX Shapes

O módulo **MRX Shapes** foi separado do módulo **MRX Models**, ao qual originalmente pertencia. Essa separação foi motivada pelo fato de que as formas geométricas utilizadas na cena não constituem um modelo integrante da própria cena, mas sim um conjunto de algoritmos destinados a preencher a estrutura **Mesh** (4.1.5.4), localizada no módulo **MRX Models**.

Este é o menor módulo do sistema, pois possui apenas duas dependências, conforme a figura 36 nos mostra. No caso este módulo contém a modelagem procedural (A), em suma, são funções que geram as formas geométricas do projeto, sendo elas:

Cubo

Piramide (4 Lados)

Esfera

Cone

Um exemplo de forma geométrica gerada é apresentado na figura 37.

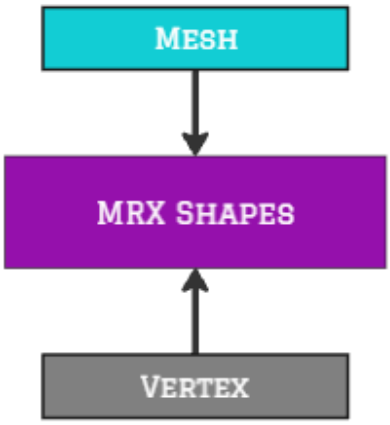


Figura 36 – Diagrama de depêndencias do módulo MRX Shapes

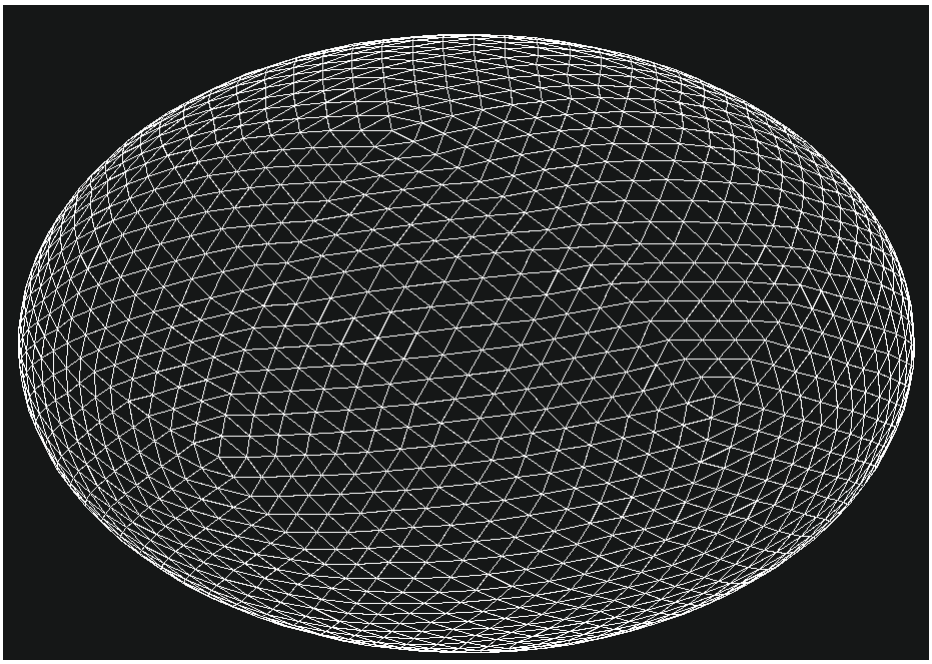


Figura 37 – Exemplo de esfera gerada pelo módulo MRX Shapes

4.1.9 MRX GUI

Por fim temos o módulo MRX GUI, responsável por tudo que comtempla a interface gráfica. A implementação da interface gráfica seguiu um padrão chamado MVC (*Model, View, Controller)* . Esta maneira de se estruturar aplicações gráficas já é considerada clássica quando o assunto são as interfaces gráficas.

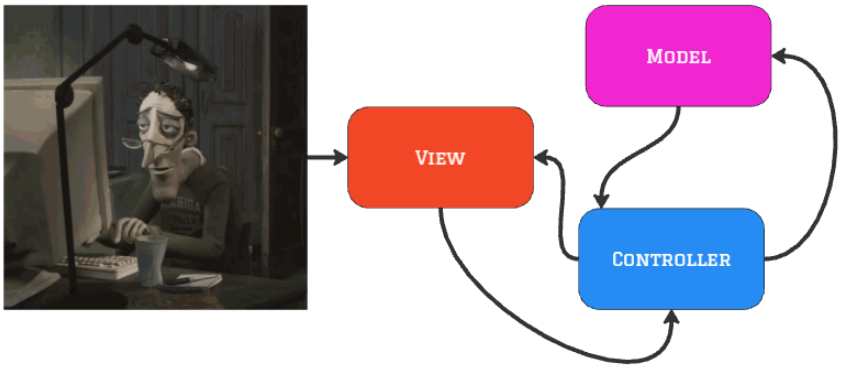


Figura 38 – Diagrama de funcionamento do MVC

A figura 38 ilustra o fluxo que o MVC segue. Nele primeiramente temos o usuário interagindo com o sistema gráfico em seu computador. Nessa interação os comandos são recebidos pela View (visão), que por sua vez processa o comando pedindo para que o Controller (controle) execute a tarefa, podendo ou não atualizar o Model (modelo). Ao término da execução das funções do controle a visão é atualizada.

Este modelo MVC foi escolhido como organização da interface gráfica, pois é simples de se implementar e proporciona um modelo de organização que é fácil de se manter, e simples de se compreender e extender.

Neste módulo temos um total de quatro submódulos pertencentes ao MRX GUI, cada um responsável por um aspecto da interface gráfica, sendo eles:

ImGui: Submódulo responsável por conter a biblioteca Dear ImGui (3.2.7.1).

ImGui-SDL2: Submódulo responsável por conter a camada de integração da biblioteca SDL2 com a biblioteca Dear ImGui (3.2.7.3).

Controller: Contém informações da interface gráfica, informações do modelo, além de métodos que modificam o próprio modelo da cena 3D.

View: É responsável por mostrar as imagens geradas pelo pipeline de visualização 3D, além de processar as entradas do usuário.

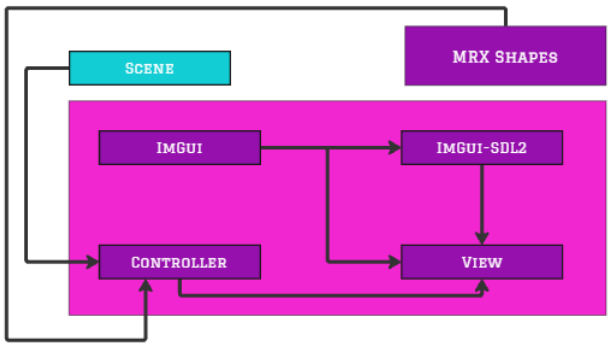


Figura 39 – Diagrama de dpêndencias do módulo MRX GUI

Na figura 39 podemos ver como se dão as dependências entre os submódulos, representados por caixas. E as setas partindo dos submódulos requisitados para os submódulos solicitantes conforme explicado na sessão (4.1.3).

4.1.9.1 Submódulo ImGui

Este submódulo consiste na biblioteca **Dear ImGui**, que é utilizada para incorporar componentes gráficos ao sistema, como botões, textos, caixas de texto, entre outros. A Dear ImGui é uma biblioteca amplamente reconhecida por seu desempenho e simplicidade, especialmente adequada para interfaces de usuário de desenvolvimento e depuração. Ela segue um modelo de GUI imediata, em que a interface é redesenhada a cada quadro, permitindo atualizações rápidas e interativas, ideais para aplicativos que exigem uma resposta visual dinâmica.

4.1.9.2 Submódulo ImGui-SDL2

O submódulo **ImGui-SDL2** atua como uma camada intermediária que conecta a biblioteca **Dear ImGui** ao **SDL2**. Tendo dependência direta com o submódulo ImGui (4.1.9.1), esse submódulo é essencial, pois permite a utilização das funcionalidades de manipulação de janelas e contextos gráficos oferecidas pelo SDL2, evitando a necessidade de recorrer diretamente às primitivas gráficas do sistema operacional. Sem essa camada, a implementação gráfica dependeria de APIs específicas de cada sistema, o que dificultaria a portabilidade do código e aumentaria a complexidade do desenvolvimento.

A utilização do **ImGui-SDL2** oferece uma abordagem mais modular e portátil, ao abstrair as operações de baixo nível e permitir que a interface gráfica funcione de forma consistente em diferentes plataformas. Existem, no entanto, outras alternativas para integrar o ImGui ao projeto, como **ImGui-OpenGL**, **ImGui-DirectX** e **ImGui-Vulkan**, que podem ser adequadas dependendo dos requisitos específicos do projeto e das bibliotecas gráficas preferidas. Ainda assim, o **ImGui-SDL2** é fortemente recomendado quando se busca uma solução robusta e multiplataforma, garantindo flexibilidade e facilitando a adaptação do código a novos ambientes.

A escolha do SDL2 como implementação para esta integração se deu apenas por familiaridade com a biblioteca, podendo ser alterado sem a necessidade de reescrever partes do sistema.

4.1.9.3 Submódulo Controller

O submódulo **Controller** é responsável por gerenciar as interações entre a interface gráfica e o modelo de dados da cena 3D. Ele atua como uma ponte, recebendo os comandos processados pela **View** e executando as ações necessárias para modificar o estado do **Model** (4.1.4) e da interface gráfica se necessário. Entre suas responsabilidades estão a atualização de informações visuais e de comportamento dos elementos da cena, conforme as entradas do usuário.

Além disso, o **Controller** mantém referências tanto à interface gráfica quanto ao modelo, permitindo uma comunicação eficiente entre essas partes. Possibilitando alterações na cena como: alterações na posição da câmera, ajustes na iluminação ou transformações em objetos. Esse submódulo assegura que as mudanças ocorram de maneira controlada e estruturada.

A arquitetura do **Controller** permite a separação clara entre lógica de controle e lógica de apresentação, o que facilita futuras expansões e a manutenção do código. Dessa forma, é possível introduzir novos recursos ou modificar a lógica de interação sem afetar diretamente a **View** ou o **Model**, preservando a modularidade e a organização do sistema.

O submódulo **Controller** depende do modelo de cenas 3D (4.1.5.5), permitindo que ele funcione como uma camada intermediária que coordena as ações a serem executadas. Essa estrutura impede que o usuário modifique o modelo diretamente, garantindo a aderência ao padrão arquitetural MVC, que preza pela separação de responsabilidades. Além disso, o **Controller** também depende do módulo **MRX Shapes** (4.1.8), o qual utiliza para adicionar à cena 3D formas geométricas selecionadas pelo usuário.

4.1.9.4 Submódulo View

O submódulo **View** é responsável pela apresentação visual da cena ao usuário e pelo processamento das entradas de interação. Ele atua como a camada de visualização do modelo MVC, exibindo o resultado das operações realizadas pelos outros submódulos e atualizando a interface com base nas mudanças do modelo de cenas 3D.

Para desempenhar suas funções o **View** possui dependências importantes. A primeira delas é o **ImGui**, utilizado para renderizar componentes da interface gráfica, como botões, menus e painéis informativos, que permitem ao usuário interagir com o sistema de forma intuitiva. A integração entre **ImGui** e SDL2 é feita pelo submódulo **ImGui-SDL2**, que facilita a manipulação de janelas e contextos gráficos, garantindo que a interface funcione de maneira consistente em diferentes plataformas.

Além disso, o **View** depende diretamente do **Controller**, que fornece as informações necessárias sobre o estado atual da cena 3D e instrui o **View** sobre as atualizações visuais a serem feitas. Essa dependência permite que o **View** exiba ao usuário as mudanças realizadas no modelo e responda de acordo com as interações, mantendo a integridade da arquitetura MVC. Dessa forma, o **View** se mantém focado na exibição e na captura de entradas, enquanto delega a lógica de controle ao **Controller**, garantindo a separação clara de responsabilidades no sistema.

4.2 Experimentações Propostas

As experimentações realizadas neste trabalho buscam explorar as variações encontradas na literatura sobre pipelines de visualização 3D, ressaltando a subjetividade inerente à implementação prática dos conceitos teóricos. As diferenças nos métodos e algoritmos descritos em diversas fontes literárias revelam que não há uma única abordagem correta para cada etapa do pipeline. Em vez disso, a implementação pode variar significativamente, influenciada por decisões específicas de design e interpretações dos autores.

Cada uma das variações testadas foi selecionada para investigar o impacto de pequenas mudanças em cálculos e processos, e como essas mudanças refletem na qualidade e eficiência da renderização final. Por exemplo, a normalização de vetores aplicada exclusivamente nas etapas de iluminação (4.2.3.1) e as diferentes abordagens para o cálculo dos vetores normais no somreamento *Phong* (4.2.2) são variações que, embora sutis, impactam diretamente a aparência visual da cena. Tais escolhas são encontradas na aplicação da teoria, revelando uma prática flexível e interpretativa.

Além disso, métodos alternativos como o cálculo do centroide geométrico (4.2.3.3) e a interpolação dos vetores normais no modelo Phong (4.2.3.2) impactam o resultado final obtido, avaliando o compromisso entre precisão e desempenho. A aplicação de coordenadas no SRT (2.5.1.4) e a filtragem de vértices antes do mapeamento para este espaço (4.2.3.4) também destacam como diferentes escolhas de coordenadas e transformações podem afetar o pipeline de visualização, influenciando tanto o processamento quanto a fidelidade visual.

Por fim, abordagens que alteram o teste de visibilidade utilizando a normal (4.2.3.5) demonstram a subjetividade presente na adaptação dos modelos teóricos. Tais variações permitem uma personalização da implementação que pode se alinhar com requisitos específicos, destacando a importância da flexibilidade na aplicação da teoria à prática. Ao final, os resultados obtidos a partir dessas experimentações oferecem uma compreensão mais profunda sobre as múltiplas interpretações possíveis do pipeline de visualização 3D e o efeito das escolhas de implementação na qualidade visual e no desempenho do sistema.

4.2.1 Métricas

Para avaliar a eficácia das variações implementadas nos *pipelines* de visualização 3D foram adotadas métricas que permitem quantificar o custo computacional e o desempenho do sistema. Essas métricas foram coletadas em cenários de teste (*benchmarks*) controlados, onde foram analisados aspectos como a taxa de quadros por segundo (FPS), tempos de execução de frames e a consistência do desempenho ao longo da execução. Esses dados fornecem dados valiosos para identificar gargalos e otimizar o *pipeline* de visualização.

4.2.1.1 Quadros por Segundo (FPS)

A métrica de quadros por segundo (FPS) é uma medida fundamental para avaliar a eficiência do sistema, indicando quantos quadros são renderizados por segundo durante a visualização da cena. Um valor mais alto de FPS está diretamente relacionado a uma maior fluidez e responsividade, características essenciais para aplicações interativas e em tempo real. Para calcular o FPS, utiliza-se a equação 45

$$\text{FPS} = \frac{\text{Total de } \textit{frames}}{\text{Tempo total de execução}} \tag{45}$$

Além do FPS foram analisadas variações dessa métrica em diferentes cenários, variando o número de objetos, a complexidade dos modelos e as condições de iluminação. Essa análise permite identificar quais abordagens são mais eficientes e quais consomem mais recursos, oferecendo uma base sólida para futuras otimizações.

4.2.1.2 Tempo de Execução por Frame

Outra métrica coletada foi o tempo de execução de cada frame. Essa métrica é armazenada em um vetor de tempos, permitindo a análise detalhada do desempenho ao longo da execução. A partir desses dados, são identificados o tempo de execução mínimo, que representa o caso de maior desempenho, o tempo de execução máximo, que indica o pior caso, e o tempo médio por frame, que fornece uma visão geral do desempenho. Esses valores ajudam a identificar inconsistências no desempenho, como picos de latência que podem impactar a experiência do usuário.

4.2.1.3 Consistência do Desempenho

Além das métricas de FPS e tempos de execução por frame, foi analisada a consistência do desempenho observando os piores 10% dos tempos de execução. Essa métrica é particularmente útil para identificar *frames* que demoram mais para serem processados, mesmo que a média de

FPS seja alta. Um sistema com boa consistência apresenta uma pequena variação entre o pior e o melhor caso, garantindo uma experiência suave e previsível.

4.2.2 Divergências Literárias

Em computação gráfica, existem algumas abordagens descritas na literatura para o cálculo do vetor normal médio em vértices compartilhados por múltiplas faces, especialmente nos modelos de iluminação de Gouraud (1971) e Phong (1973). Este trabalho foca em uma dessas divergências literárias, que é a maneira de calcular o vetor unitário médio num vértice. Essa questão é fundamental, pois o método escolhido pode impactar diretamente a aparência visual do objeto, especialmente em renderizações que fazem uso de técnicas de sombreamento suave.

Segundo Foley et al. (1995), o vetor unitário médio $\hat{N}$ em um vértice $\mathbf{V}$ compartilhado por diversas faces é obtido somando-se os vetores normais $\vec{N}_i$ das faces adjacentes ao vértice e, em seguida, normalizando a soma. Essa abordagem, apresentada na Equação 46, visa garantir que o vetor médio resultante mantenha a direção e magnitude unitária, proporcionando uma transição suave entre as faces adjacentes ao vértice

$$\hat{N} = \frac{\sum_{i=1}^{n} \vec{N}_i}{|\sum_{i=1}^{n} \vec{N}_i|} \tag{46}$$

Por outro lado, Conci, Azevedo e Leta (2003) sugerem uma abordagem alternativa, onde o vetor normal médio em um vértice é calculado como a média simples das componentes dos vetores normais das faces compartilhadas. Neste caso, como mostrado na Equação 47, o vetor médio não passa por uma normalização convencional após a soma das componentes, o que pode resultar em uma magnitude variável, dependendo das orientações dos vetores normais das faces adjacentes.

$$\hat{N} = \frac{\sum_{i=1}^{k} \vec{N}_i}{k} \tag{47}$$

Essa divergência entre as abordagens levanta uma questão importante sobre qual método gera um resultado visualmente mais fiel. O cálculo proposto por Foley et al. (1995), com a normalização final, tende a garantir que o vetor médio mantenha uma magnitude unitária, o que é essencial em contextos que exigem precisão na intensidade e direção da iluminação. Já a abordagem de Conci, Azevedo e Leta (2003) oferece uma solução mais simples, mas pode resultar em vetores com magnitude inconsistente, o que pode alterar ligeiramente a intensidade do sombreamento.

Outro ponto de incerteza na literatura é se os vetores normais das faces utilizadas no cálculo do vetor médio devem estar previamente normalizados. Essa questão pode impactar o

resultado final, uma vez que a soma de vetores não normalizados pode dar mais peso a faces com maior área, criando um vetor médio que privilegia as contribuições de certas faces sobre outras. No entanto, esta prática de normalização dos vetores antes da soma não é discutida explicitamente nos textos.

Este trabalho busca explorar essa diferença de cálculo e avaliar como cada método impacta a qualidade visual e a coerência das transições de iluminação nas renderizações. Ao testar as duas abordagens em cenários com diferentes modelos de iluminação, será possível observar quais efeitos visuais são enfatizados ou minimizados em cada caso, fornecendo uma base para análise prática dessa divergência literária.

4.2.3 Variações de implementação dos pipelines

4.2.3.1 Vetores Normalizados Somente na Iluminação

No pipeline de visualização 3D o processo convencional de se implementar envolve a normalização dos vetores em diversas etapas do pipeline. No entanto, uma abordagem alternativa a ser testada, diz respeito à normalização dos vetores apenas no momento do cálculo dos valores de iluminação, com o objetivo de reduzir o custo computacional associado às operações de normalização nas demais fases do pipeline.

Ao limitar a normalização à fase de iluminação, busca-se simplificar o processamento, assumindo que as variações na magnitude dos vetores em etapas anteriores terão impacto mínimo na aparência final. Essa técnica explora o fato de que, somente nos cálculos de iluminação, a magnitude dos vetores normais é relevante e, portanto, devem ser normalizados. Esta experimentação visa verificar o equilíbrio entre otimização e qualidade visual, analisando se a normalização exclusiva na etapa de iluminação é uma prática viável e eficiente.

4.2.3.2 Alteração do Cálculo de Vetores no Phong

No modelo de iluminação Phong (1973), o cálculo dos vetores de iluminação e observação é essencial para a obtenção de um sombreamento suave e realista. Simplificando o sombreamento Phong, os vetores de iluminação $\vec{L}$ e observação $\vec{S}$ são calculados uma única vez no centroide da face, o que tende a causar o efeito facetado discutido na Seção 2.5.1.8.3. Outra simplificação viável é calcular ambos os vetores uma única vez no centroide do poliedro, o que poderá reduzir o número de operações e, potencialmente, o custo computacional. Essa alteração visa avaliar se o cálculo centralizado destes vetores impactará a precisão do sombreamento e a fidelidade visual da iluminação.

Além disso, será testada a interpolação dos vetores $\vec{L}$ e $\vec{S}$ com base em amostras calculadas nos vértices das faces, assim como acontece com o vetor normal. Ao explorar essa

segunda abordagem, este trabalho investigará o impacto no aspecto visual e no desempenho da renderização.

4.2.3.3 Cálculo do Centroide Geométrico

O centroide de um objeto 3D é empregado em diversas operações no pipeline de visualização, especialmente para definir o ponto de referência em cálculos de iluminação e transformações geométricas. Neste trabalho, consideraremos duas abordagens para calcular o centroide geométrico:

A primeira abordagem consiste em calcular uma média simples das coordenadas dos vértices do objeto. Esse método é direto e oferece um ponto central que leva em conta a posição relativa de todos os vértices, resultando em um centroide que representa a média espacial do objeto. O cálculo da média simples para o centroide C é dado por:

$$\begin{aligned} C_x &= \frac{1}{n}\sum_{i=1}^{n} x_i \\ C_y &= \frac{1}{n}\sum_{i=1}^{n} y_i \\ C_z &= \frac{1}{n}\sum_{i=1}^{n} z_i \end{aligned} \tag{48}$$

onde n é o número de vértices, e x_i, y_i, z_i representam as coordenadas dos vértices individuais.

A segunda abordagem calcula o centroide como sendo o ponto da caixa que envolve o objeto. Neste caso, o centroide para cada coordenada é dado por:

$$\begin{aligned} C_x &= \frac{\max(x) + \min(x)}{2} \\ C_y &= \frac{\max(y) + \min(y)}{2} \\ C_z &= \frac{\max(z) + \min(z)}{2} \end{aligned} \tag{49}$$

Este método é útil em casos onde o objeto possui simetria ou quando o cálculo rápido do ponto central é necessário, pois utiliza apenas os valores máximos e mínimos das coordenadas.

Ambas as abordagens serão testadas neste trabalho para avaliar como cada método afeta a precisão das operações subsequentes no pipeline, considerando aspectos como eficiência computacional e impacto na qualidade visual.

4.2.3.4 Utilização de Coordenadas no SRT para Etapas do Pipeline

No pipeline de visualização 3D, diferentes etapas tradicionalmente requerem o uso de sistemas de coordenadas específicos para realizar operações geométricas e de transformação. Neste trabalho parte dos processos foi implementada utilizando coordenadas no SRU, enquanto outras etapas fazem uso do SRT. Essa divisão é comumente observada na literatura, mas pode variar de acordo com a interpretação e implementação de cada autor.

O uso de coordenadas no SRT é especialmente vantajoso para operações que envolvem renderização final e mapeamento de pontos na tela, uma vez que este sistema facilita a conversão direta das posições para o espaço de visualização. Em contraste, o SRU é tipicamente empregado nas fases iniciais do pipeline, onde as operações geométricas são realizadas no espaço tridimensional original, permitindo maior precisão nos cálculos antes da projeção para o espaço da tela.

O objetivo deste teste é verificar a viabilidade de utilizar exclusivamente coordenadas no SRT ao longo de todas as etapas do pipeline. Se o SRT se mostrar adequado para todas as fases, isso poderá simplificar o fluxo de dados e resultar em uma implementação mais direta, reduzindo a subjetividade ao padronizar o uso de um único sistema de coordenadas. Essa abordagem visa avaliar as vantagens e limitações de uma estrutura simplificada, buscando um equilíbrio entre a eficiência do processo e a fidelidade visual.

4.2.3.5 Alteração do Teste de Visibilidade pela Normal

Na eliminação de faces ocultas pelo cálculo da normal (2.5.1.5.1), a visibilidade de cada face é determinada utilizando o sentido do vetor normal da face, que indica se a face está orientada para o observador ou para o lado oposto.

No *pipeline* A, esse processo inclui o uso de um vetor auxiliar, denotado por $\hat{O}$, para definir o sentido da normal da face. O vetor $\hat{O}$ é calculado como a diferença entre o centroide da face e o Ponto de Referência de Visualização (VRP), ajudando a definir a orientação correta da normal em relação ao observador.

Este teste propõe verificar se, no *pipeline* B, o uso do vetor $\hat{O}$ pode ser dispensado. Em vez disso, seria possível utilizar diretamente o vetor normal $\hat{n}$ da câmera, uma vez que durante o processo de transformações geométricas no pipeline acaba por converter o volume de visualização em um paralelepípedo canônico de visualização, o que pode simplificar a determinação de visibilidade. Se bem-sucedida, essa abordagem eliminaria a necessidade de um cálculo adicional, reduzindo o custo computacional e mantendo a precisão da ocultação.

5 Resultados e Discussões

Esta seção apresenta os resultados das análises das diversas experimentações propostas no Capítulo 4, que abrangiam divergências literárias e variações na implementação dos *pipelines*.

5.1 Interface gráfica

Como o objetivo principal era experimentar na prática as divergências entre os modelos de pipelines, métodos e processos, implementou-se uma interface gráfica simples para auxiliar nestas experimentações. A figura 40 exibe a interface do programa desenvolvido sem adição de objetos na cena. A interface é composta de quatro elementos principais: O primeiro é o menu superior, o visualizador hierárquico da cena, o inspetor de objetos e por fim a *viewport*.

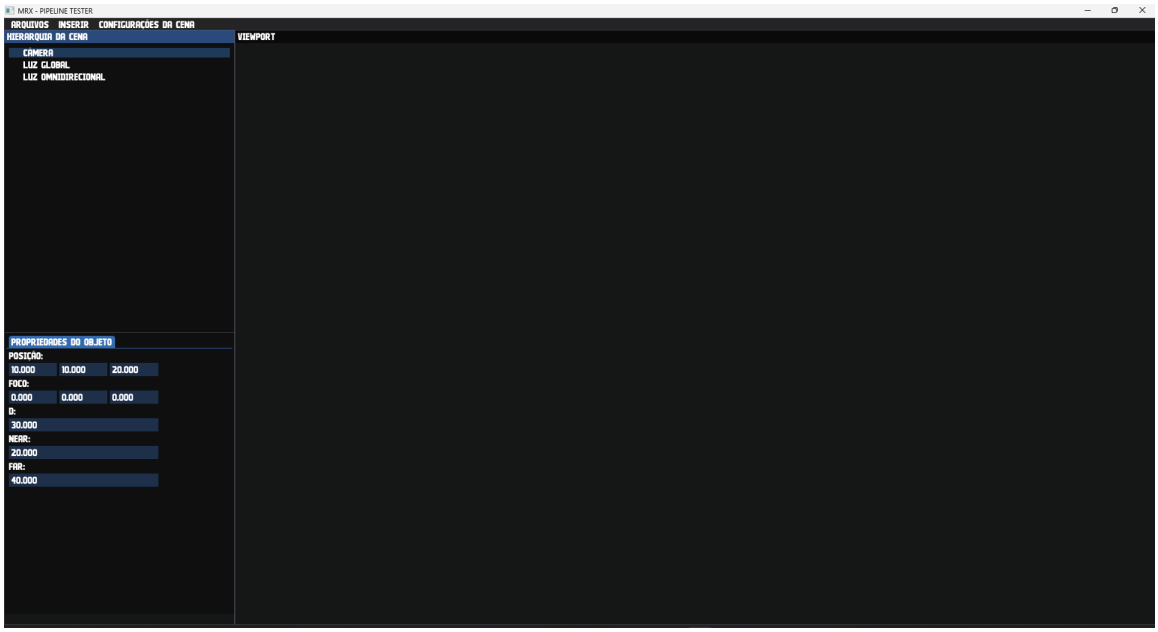


Figura 40 – Interface do programa sem objeto carregado na cena.

No menu superior encontram-se três opções principais: *Arquivos*, *Inserir* e *Configuração da cena*. No menu *Arquivos*, o usuário pode optar por *Novo* para iniciar uma nova cena, por *Abrir* para carregar um projeto previamente salvo ou por *Salvar* para registrar as alterações efetuadas na cena atual. Essa organização facilita o gerenciamento dos arquivos e simplifica o fluxo de trabalho.

O menu *Inserir* foi elaborado para simplificar a inclusão de novos objetos na cena. Por meio dele é possível adicionar diversos elementos geométricos como cubos, pirâmides, esferas, cilindros, torus e cones. Essa variedade permite a construção de cenas diversificadas, incentivando a experimentação com diferentes formas e estruturas que podem ser utilizadas para comparar os modelos de *pipeline* implementados.

Por fim, o menu *Configuração da cena* reúne os recursos voltados à personalização dos parâmetros que influenciam tanto a visualização quanto o desempenho da aplicação. Nele o usuário tem a possibilidade de ajustar o modelo de iluminação, selecionar o modelo de *pipeline*, definir os parâmetros de recorte e até mesmo executar *benchmarks* para analisar o comportamento da cena sob diferentes configurações. Dessa forma, a centralização das configurações contribui para uma compreensão mais clara dos efeitos de cada parâmetro e facilita a análise comparativa dos métodos utilizados.

Com essa estrutura a interface gráfica se mostra prática e funcional, permitindo ao usuário interagir de maneira intuitiva com os diversos recursos da aplicação e realizar experimentações detalhadas sobre os diferentes modelos e processos implementados.

O visualizador hierárquico da cena é um dos elementos principais da interface gráfica. Ele tem como objetivo exibir todos os elementos que compõem a cena atual de maneira organizada e acessível. Nesse painel os objetos estão dispostos em uma lista hierárquica, permitindo ao usuário identificar facilmente cada elemento presente na cena e sua respectiva relação com outros objetos.

Na Figura 41, observa-se um exemplo do visualizador hierárquico com diversos objetos inseridos na cena. Entre os itens listados é possível identificar elementos como a câmera, luzes (global e omnidirecional), além de vários objetos geométricos, como cubos, pirâmides, esferas e cilindros. Essa estrutura facilita a manipulação dos objetos, pois permite que o usuário trabalhe em projetos complexos, contendo múltiplos elementos, sem que a interface fique desordenada. Além disso, a hierarquia apresentada contribui para uma melhor compreensão da composição da cena, sendo uma ferramenta indispensável para o controle e a manipulação dos objetos inseridos.

O inspetor de objetos é outro elemento fundamental da interface gráfica, atuando em conjunto com o visualizador hierárquico para permitir a edição e configuração detalhada dos elementos presentes na cena. Sua principal função é exibir as propriedades do objeto selecionado no visualizador hierárquico e possibilitar sua modificação de forma prática e intuitiva.

Na Figura 42, é possível observar o inspetor de objetos configurado para um objeto

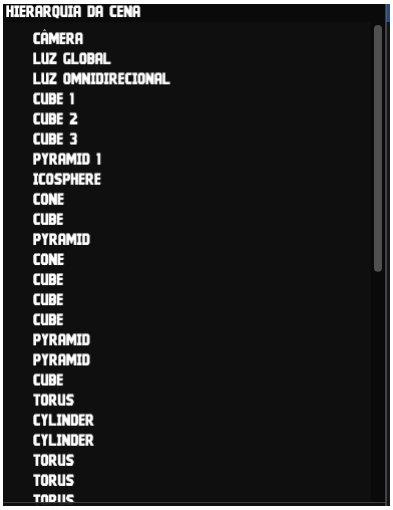


Figura 41 – Visualizador hierárquico da cena com diversos objetos inseridos.

geométrico, no caso *Cube 3*. Entre as propriedades exibidas destacam-se os coeficientes de material *Ka*, *Kd* e *Ks* que representam, respectivamente, os coeficientes de reflexão ambiente, difusa e especular 2.4.1.2. Além disso, é possível ajustar o brilho (*shininess*) do material, influenciando diretamente a aparência visual do objeto na cena.

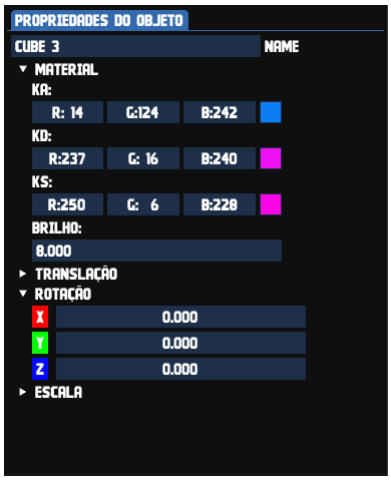


Figura 42 – Inspetor de objetos exibindo propriedades de um objeto geométrico.

O inspetor também oferece opções para transformações geométricas como translação, rotação e escala, com valores ajustáveis para cada eixo (X, Y e Z). Essa organização facilita a aplicação de transformações precisas, permitindo ao usuário alterar a posição, orientação e tamanho do objeto de forma controlada.

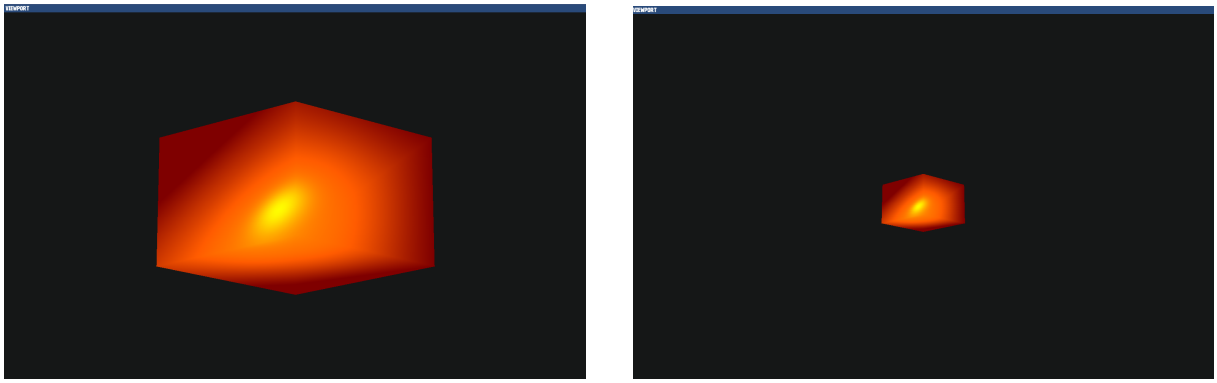
Vale ressaltar que o inspetor de objetos é dinâmico e adapta suas opções dependendo do tipo de objeto selecionado. Para câmeras, iluminação global ou lâmpadas omnidirecionais as propriedades exibidas incluem configurações específicas para esses elementos, como parâmetros de luz ou ajustes de projeção. Essa adaptabilidade torna o inspetor uma ferramenta versátil e essencial para o controle detalhado dos componentes da cena.

A combinação entre o visualizador hierárquico e o inspetor de objetos proporciona uma

experiência de uso organizada, permitindo que o usuário gerencie tanto a estrutura hierárquica quanto as propriedades individuais de cada elemento da cena com facilidade.

5.2 Diferenças entre os *Pipelines*

Conforme estabelecido na Seção 2.5.1, este trabalho tem como um de seus pilares a análise comparativa entre dois *pipelines* de renderização, buscando identificar como diferenças estruturais em suas implementações afetam a saída visual. A Figura 43 ilustra esse contraste por meio de renderizações lado a lado de um mesmo cubo, sob condições idênticas de iluminação, posição do observador e ponto focal.



(a) Renderização pelo *Pipeline* A (b) Renderização pelo *Pipeline* B

Figura 43 – Comparação visual entre os *pipelines*. Observa-se redução significativa do cubo no *Pipeline* B.

5.2.1 Divergências na Representação Espacial

A análise das renderizações revela uma discrepância fundamental: enquanto o *Pipeline A* exibe o cubo com dimensões ampliadas, o *Pipeline B* o apresenta em escala reduzida (Figura 43). Essa diferença persiste mesmo com parâmetros idênticos de plano *near* (20) e *far* (100), indicando que a variação está relacionada ao tratamento distinto da projeção geométrica.

A raiz da divergência reside na normalização do volume de visualização: o *Pipeline B* opera em um espaço normalizado, escalonando as coordenadas do objeto para um cubo unitário antes da projeção, enquanto o *Pipeline A* preserva as proporções originais. Para equalizar os resultados, como mostra a Figura 44, é necessário ajustar a distância do plano de projeção no *Pipeline B* para 100 unidades - contra 20 unidades no *Pipeline A*. Esse ajuste compensa a escala implícita da normalização, permitindo comparações equitativas.

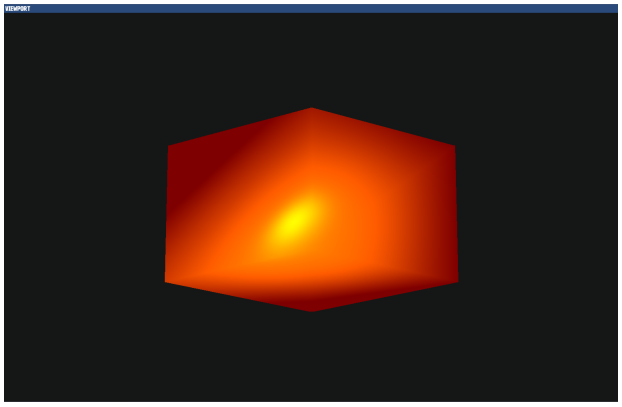


Figura 44 – Renderização pelo *Pipeline B* com distância de projeção ajustada para 100 unidades.

5.2.2 Implicações dos Parâmetros de Projeção

A escolha não calibrada de parâmetros introduz vieses computacionais, sem essa calibração, objetos idênticos podem ter custos computacionais distintos, dificultando a analise dos resultados. A Tabela 4 sintetiza as diferenças críticas entre os *pipelines*.

Tabela 4 – Parâmetros comparativos dos *pipelines*

Parâmetro	***Pipeline* A**	***Pipeline* B**
Sistema de coordenadas	Mão direita	Mão esquerda
Plano próximo (*near*)	20	20
Plano distante (*far*)	100	100
Distância de projeção	20	100

5.3 *Benchmarks*

Na ciência da computação, *benchmark* é o ato de executar um programa para avaliar o desempenho relativo de um algoritmo, geralmente através de testes controlados e métricas padronizadas. Neste trabalho, executou-se 10 iterações de cada *pipeline* (A e B) sob três modelos de iluminação (*Flat*, Gouraud e Phong), com o sistema operacional dedicado exclusivamente à aplicação para minimizar interferências externas. Para cada execução, registraram-se seis métricas-chave: (1) tempo total de renderização, (2) total de quadros, (3) FPS médio, (4) tempo médio por quadro, (5) menor tempo individual, (6) maior tempo individual, além dos tempos dos 10% piores quadros – critério essencial para avaliar a estabilidade em cenários críticos.

Cada combinação *pipeline*/modelo de iluminação foi submetida a 20 execuções completas, totalizando 120 testes (2 *pipelines* × 3 modelos de iluminação × 20 repetições). As métricas foram coletadas. E os dados brutos estão disponíveis no Apêndices B sendo encontrados nas tabelas 13 a 24, enquanto esta seção foca nas tendências consolidadas.

5.4 Comparação de desempenho entre os *Pipelines*

Para comparar o desempenho dos *pipelines* A e B (2.5.1) nos métodos de iluminação (*Constante*, *Gouraud* e *Phong* - 2.5.1.8), utilizamos a Análise de Variância (ANOVA) com um nível de confiança de 95%, avaliando diferenças entre grupos considerando a variável de interesse tempo total de cada benchmark. O estudo incluiu 20 medições do tempo total de cada benchmark para cada combinação de *pipeline* e método de iluminação, totalizando 120 testes, conforme detalhado nas tabelas de estatísticas básicas por método (Tabelas 13, 14, 17, 18, 21 e 22). As condições necessárias para a análise, como distribuição normal dos dados e variâncias equivalentes entre grupos, foram confirmadas. Os resultados do teste ANOVA podem ser consultados no Anexo C.2.4, nas Tabelas 25 e 26.

5.4.1 Resultados ANOVA

Os resultados demonstraram padrões estatisticamente significativos. Primeiro, o método de iluminação mostrou-se o fator mais impactante (F=2317.16, p<0.0001), com o *Phong* demandando 58.4% mais tempo que o *Gouraud* e 62.1% mais que o *Flat* no Pipeline B (Tabelas 21 e 22). Segundo, a Pipeline Adair apresentou superioridade consistente, especialmente em cenários complexos, com redução média de 36.8% no tempo total para *Phong* comparado à Smith.

Tabela 5 – Teste t para comparação entre Pipelines no método *Flat*

Parâmetro	**Pipeline A**	**Pipeline B**
Média (s)	33.60	42.07
Variância	3.46	0.34
Estatística t	-19.41	
p-valor	<0.001	

Tabela 6 – Teste t para comparação entre Pipelines no método *Gouraud*

Parâmetro	**Pipeline A**	**Pipeline B**
Média (s)	34.52	53.05
Variância	2.81	0.67
Estatística t	-44.44	
p-valor	<0.001	

Para detalhar essas diferenças, realizamos testes estatísticos específicos (*testes t*) entre as pipelines em cada método de iluminação (Tabelas 5, 6 e 7). No método *Flat*, a diferença média foi de 8.46 segundos (t=-19.41, p<0.001), com a Pipeline B sendo 20.1% mais rápida. No entanto, essa vantagem inverte-se drasticamente nos métodos complexos: no *Gouraud*, a Pipeline A foi

Tabela 7 – Teste t para comparação entre Pipelines no método *Phong*

Parâmetro	**Pipeline A**	**Pipeline B**
Média (s)	43.25	68.45
Variância	0.57	0.88
Estatística t	-93.74	
p-valor	<0.001	

34.8% mais eficiente (diferença de 18.53 segundos, t=-44.44, p<0.001), e no *Phong*, a diferença aumentou para 25.19 segundos (t=-93.74, p<0.001), consolidando a superioridade da Pipeline A em cenários realistas.

A Tabela 8 sintetiza os resultados da ANOVA de dois fatores com repetições. Destacam-se:

Tabela 8 – Interpretação dos fatores na performance das pipelines

Fator Analisado	**Grau de Influência**	**Impacto Prático**
Método de Iluminação	Extremamente Alto	Define a base de tempo de execução
Pipeline	Altíssimo	*Pipeline A* é melhor em cenas complexas
Combinação Pipeline + Método	Muito Alto	A diferença aumenta com a complexidade

A análise da interação entre pipelines e métodos de renderização revelou que as otimizações do *pipeline A* amplifica seus benefícios em operações matemáticas complexas. Embora mantenha vantagem de 20.1% mesmo no *Flat* (Tabelas 13 e 14), sua variabilidade é 3.2 vezes maior nesse cenário (σ=1.81s vs 0.57s do *pipeline B*), indicando maior inconsistência em tarefas simples. O ápice do desempenho ocorreu no *Phong*, com o *pipeline A* processando quadros em 0.123s contra 0.189s do *pipeline B* - diferença que equivaleria a mais ou menos 3 quadros adicionais por segundo.

Paradoxalmente, o *pipeline B* demonstrou comportamento oposto: no modelo de iluminação constante, seu pico de latência foi 8.7% menor (0.68s vs 0.72s), sugerindo um *pipeline* mais previsível. Essa dicotomia indica que a seleção ideal depende do perfil de uso: aplicações críticas com modelos de iluminação realista podem se beneficiar da velocidade do *pipeline A*, enquanto sistemas com restrições de hardware podem preferir a consistência do *pipeline B* em cenas básicas. A relação inversa entre ganho de performance e estabilidade temporal reforça a necessidade de análise contextualizada para cada cenário de renderização.

5.5 Análise Estatística dos 10% Piores Frames

A análise dos 10% piores frames revelou padrões críticos sobre a estabilidade temporal dos *pipelines*, complementando os resultados da ANOVA. Para cada método de renderização,

foram comparadas as métricas de tendência central (média) e variabilidade (desvio padrão e coeficiente de variação) entre os *pipelines* A e B, conforme detalhado nas Tabelas 9, 10 e 11.

Tabela 9 – Análise dos 10% Piores Frames e Resultados Gerais - Modelo de Iluminação Constante

Categoria	**Pipeline**	**Média (s)**	**Desvio Padrão**	**Coef. Variação (%)**
10% Piores	*Pipeline A*	0,1088	0,0047	4,35
	Pipeline B	0,1297	0,0047	3,62
Geral	*Pipeline A*	0,0934	0,0126	13,45
	Pipeline B	0,1198	0,0021	1,75

Tabela 10 – Análise dos 10% Piores Frames e Resultados Gerais - Modelo de Iluminação Gouraud

Categoria	**Pipeline**	**Média (s)**	**Desvio Padrão**	**Coef. Variação (%)**
10% Piores	*Pipeline A*	0,1073	0,0050	4,62
	Pipeline B	0,1385	0,0031	2,25
Geral	*Pipeline A*	0,0957	0,0128	13,41
	Pipeline B	0,1468	0,0154	10,48

Tabela 11 – Análise dos 10% Piores Frames e Resultados Gerais - Modelo de Iluminação Phong

Categoria	**Pipeline**	**Média (s)**	**Desvio Padrão**	**Coef. Variação (%)**
10% Piores	*Pipeline A*	0,1473	0,0021	1,42
	Pipeline B	0,2315	0,0032	1,39
Geral	*Pipeline A*	0,1230	0,0021	1,73
	Pipeline B	0,1891	0,0199	10,51

O **coeficiente de variação (CV)** é uma medida de consistência: ele indica quanto os tempos de renderização variam em relação à média. Quanto menor o CV, mais previsível é o desempenho, mesmo em cenários críticos.

No modelo de iluminação **Phong** (Tabela 11), o *Pipeline A* apresentou média 36,4% menor nos 10% piores frames (0,1473s vs. 0,2315s), mantendo CV equivalente ao *Pipeline B* (1,42% vs. 1,39%). Contudo, na média geral, o *Pipeline B* mostrou CV 6 vezes maior (10,51% vs. 1,73%), revelando uma instabilidade cronica neste caso. Essa combinação faz do *Pipeline A* a escolha ideal para aplicações que exigem tanto velocidade quanto consistência.

Para o modelo **Gouraud** (Tabela 10), o *Pipeline A* reduziu a média dos piores tempos em 22,5% (0,1073s vs. 0,1385s), porém com CV 2,1 vezes maior (4,62% vs. 2,25%). Na média geral, o *Pipeline B* apresentou variabilidade 31% menor (10,48% vs. 13,41%), sugerindo que otimizações agressivas no *Pipeline A* introduzem flutuações mesmo em cenários normais.

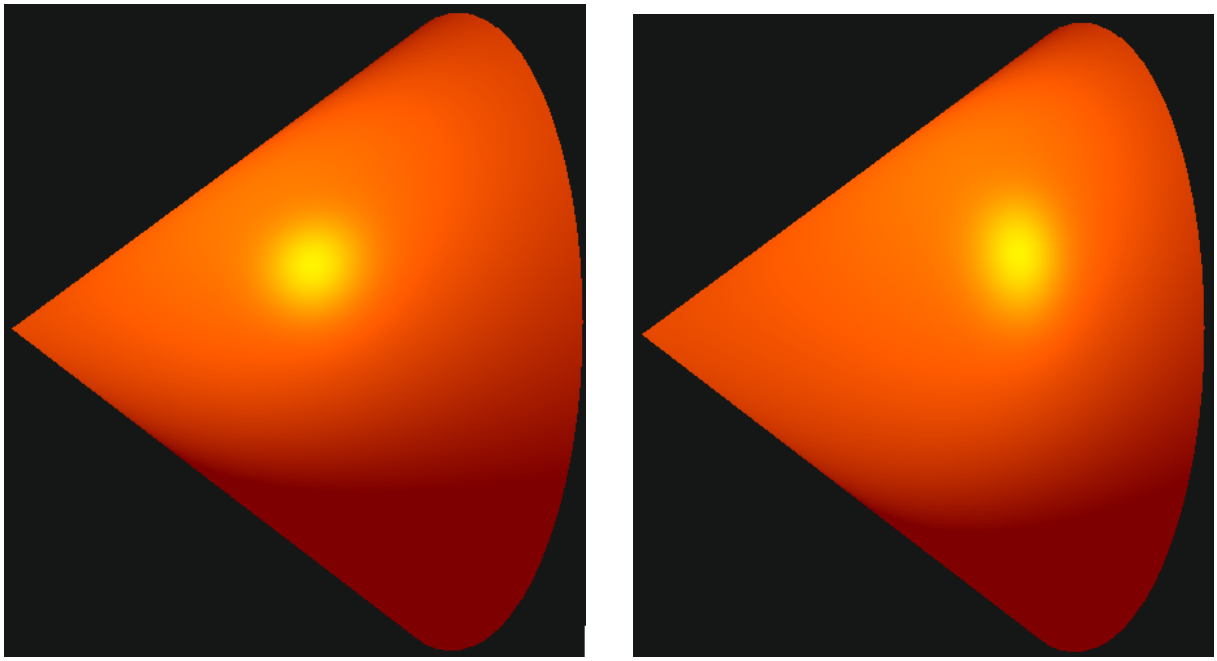
Os resultados mais intrigantes vieram do modelo **Constante** (Tabela 9). O *Pipeline A* evitou picos extremos (média de 0,1088s vs. 0,1297s nos piores frames), mas com CV 20%

maior (4,35% vs. 3,62%). Na média geral, porém, o *Pipeline B* mostrou estabilidade 7,7 vezes superior (CV de 1,75% vs. 13,45%), comprovando sua adequação para sistemas que priorizam consistência sobre velocidade bruta.

Essa análise revela um equilíbrio delicado: enquanto o *Pipeline A* oferece desempenho superior em cenários críticos, o *Pipeline B* garante previsibilidade em operações cotidianas. A escolha ideal dependerá do perfil de risco da aplicação - se tolerância a picos esporádicos em troca de velocidade ou estabilidade a qualquer custo.

5.6 Divergências literárias

Conforme mencionado na sessão 4.2.2, as divergências na computação gráfica quanto ao cálculo do vetor médio unitário em vértices revelam nuances teóricas e práticas. A formulação de Conci, Azevedo e Leta (2003) na Equação 47 propõe uma média aritmética simples das normais adjacentes, enquanto Foley et al. (1995) na Equação 46 adota a soma vetorial seguida de normalização. Essa distinção transcende a mera notação: a abordagem de Foley preserva rigorosamente a natureza vetorial das normais ao realizar a normalização *após* a soma, garantindo que o resultado final seja efetivamente um vetor unitário.



(a) Método proposto por Foley et al. (1995)

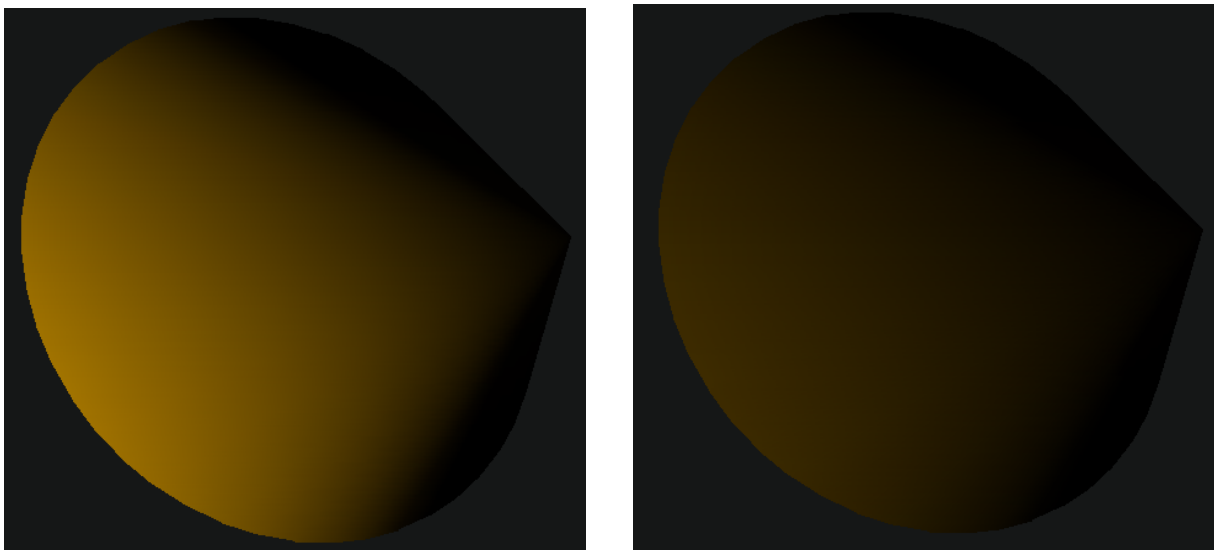
(b) Método proposto por Conci, Azevedo e Leta (2003)

Figura 45 – Diferença entre algoritmos para determinar o vetor normal médio de um vértice

A Figura 45 ilustra um cone renderizado com o modelo de iluminação Phong, que exige o cálculo do vetor normal médio nos vértices para determinar a cor e, posteriormente, interpolá-la ao longo da superfície (conforme descrito na Seção 2.5.1.8.3). Observa-se uma

diferença significativa no efeito especular (2.4.2.5): além de um deslocamento em direção à base do cone, sua forma foi alterada. No método clássico proposto por Foley et al. (1995), o reflexo assume um formato aproximadamente circular, enquanto a abordagem de Conci, Azevedo e Leta (2003) resulta em um efeito ovalado.

Adicionalmente, nota-se uma mudança sutil, mas perceptível, na região onde a luz não incide diretamente. O gradiente da reflexão especular torna-se menos suave, evidenciando uma transição mais abrupta. Esse comportamento pode impactar a percepção do material e da continuidade da iluminação, sobretudo em superfícies curvas.



(a) Método de Foley no modelo Gouraud (b) Método de Azevedo no modelo Gouraud

Figura 46 – Comparação dos métodos de cálculo no modelo de iluminação Gouraud

A Figura 46 apresenta as diferenças entre os métodos de Foley e Azevedo quando aplicados ao modelo de iluminação Gouraud. O impacto mais evidente aparece na suavidade do gradiente de iluminação. Enquanto o método de Foley preserva uma transição mais uniforme na superfície iluminada, o método de Azevedo exibe uma transição mais abrupta, resultando em um amarelo menos saturado e uma percepção reduzida da suavidade na área especular.

Essas diferenças se devem à natureza não unitarizada da média proposta por Azevedo, que influencia diretamente os cálculos de interpolação no modelo Gouraud. Como o modelo baseia-se na média ponderada de vetores normais, qualquer variação na magnitude do vetor impacta proporcionalmente a distribuição da iluminação especular e difusa.

A simplificação proposta por Azevedo, embora computacionalmente eficiente, introduz uma ambiguidade geométrica. A média não normalizada dos vetores normais altera implicitamente a magnitude do vetor resultante, o que pode comprometer a precisão em modelos de iluminação baseados na orientação da superfície, como os modelos de Gouraud e Phong. Nesses casos, a não unitariedade do vetor normal pode distorcer os cálculos de difusão e especularidade, resultando em artefatos visuais indesejados. Por outro lado, o método de Foley preserva a

coerência matemática ao atuar em espaços projetivos, onde a normalização posterior à soma garante a invariância rotacional e mantém a integridade da iluminação.

Essa dualidade reflete a tensão entre eficiência computacional e rigor matemático. Enquanto a estratégia de Azevedo otimiza o desempenho, sendo vantajosa em cenários onde se quer otimizar os calculos de iluminação, a abordagem de Foley prioriza a precisão física da simulação luminosa. No modelo de iluminação Phong, as alterações propostas por Azevedo podem ser consideradas toleráveis, dado que o processo de cálculo não é baseado em uma simulação física estrita do mundo real, mas sim em uma aproximação visual que preserva a percepção geral da cena. No entanto, no modelo Gouraud, a diferença na suavidade das transições e no gradiente de iluminação é mais evidente, podendo impactar significativamente a percepção da continuidade luminosa e da uniformidade do material. Dessa forma, a escolha entre ambos os métodos depende do equilíbrio entre fidelidade visual e custo computacional, considerando os requisitos específicos de cada aplicação.

5.7 Variação na implementação dos Pipelines

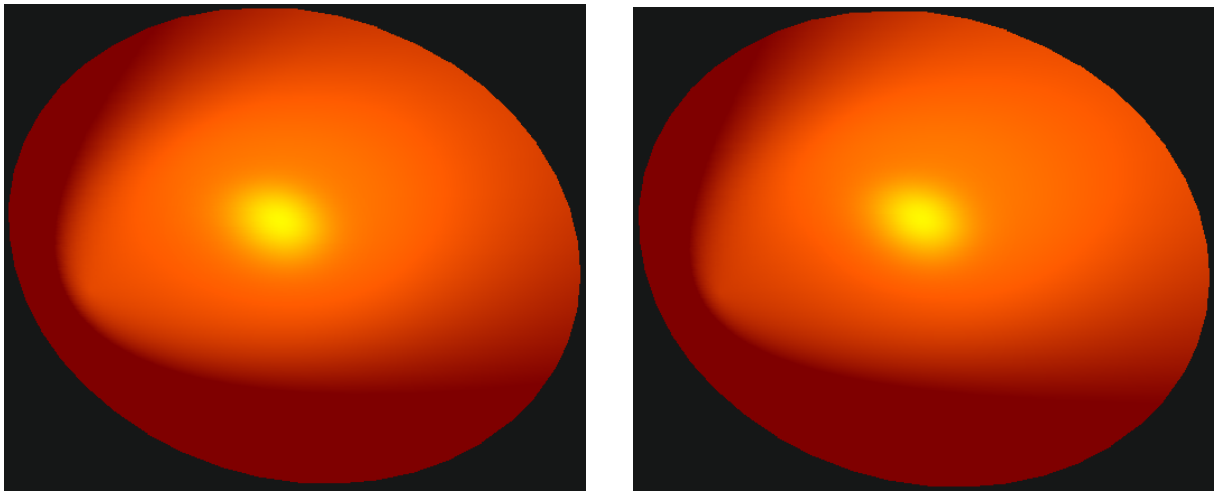
Conforme proposto na sessão 4.2.3, existem algumas tomadas de decisões na hora da implementação que não estão bem documentadas na teoria clássica da computação gráfica, cabendo ao programador tomar estas decisões, gerando por vezes resultados sutilmente diferentes como no caso da iluminação Phong descrita na sessão anterior. E apesar destas decisões não terem por vezes impactos significativos no resultado final da imagem, é de suma importância que estas divergências estejam bem documentadas para que os programadores possam tomar as decisões sabendo dos impactos na imagem gerada que essa decisão terá.

5.7.1 Cálculo do centroide geométrico

A começar por um caso semelhante ao descrito na Seção 5.6, onde existem duas fórmulas conhecidas na teoria clássica da computação gráfica para se determinar o centroide geométrico. A primeira alternativa é calcular a média simples das coordenadas dos vértices do objeto, utilizando a fórmula 48, descrita na Seção 4.2.3.3. A segunda maneira é utilizando a estratégia do box envolvente do objeto, empregando a fórmula 49, também descrita na mesma seção.

Em formas geométricas simples, como polígonos regulares ou sólidos com simetria bem definida, o cálculo exato do centroide geométrico pode ser realizado analiticamente, considerando as propriedades das figuras. No entanto, para formas mais complexas, como malhas tridimensionais com milhares de vértices e topologias irregulares, determinar o centroide geométrico exato torna-se computacionalmente inviável devido à elevada complexidade algorítmica. Isso ocorre porque seria necessário integrar sobre a superfície ou volume do objeto, o que exige um modelo matemático detalhado da geometria.

Por essa razão, aproximações como a média das coordenadas dos vértices são amplamente utilizadas na computação gráfica. Essa estratégia oferece um equilíbrio aceitável entre precisão e eficiência computacional, especialmente em aplicações interativas, como renderização em tempo real e simulação física. A fórmula baseada no box envolvente também é uma aproximação válida e frequentemente utilizada, particularmente em casos onde se deseja um cálculo rápido que seja independente da densidade de vértices do objeto.



(a) Calculo do centroide pela média (b) Calculo do centroide pelo box envolvente

Figura 47 – Comparação dos métodos de calculo do centroide

Na Figura 47, observa-se uma diferença sutil entre os métodos de cálculo apresentados nas Figuras 47a e 47b. Diferentemente do caso discutido na Seção 5.6, onde alterações nos métodos resultaram em impactos perceptíveis nos efeitos difuso e especular, aqui a mudança no cálculo do centroide afeta apenas de forma leve o posicionamento desses efeitos, para além de tornar o objeto da Figura 47b levemente mais fosco em comparação ao da Figura 47a. Essa diferença é quase imperceptível na prática e não compromete a qualidade visual em modelos de iluminação Constante ou Gouraud.

Essa constatação reforça a viabilidade do uso de aproximações, como o cálculo pela média ou pelo box envolvente, em aplicações que demandam alta eficiência computacional. Embora sejam métodos simplificados, ambos apresentam precisão suficiente para a maioria das situações em computação gráfica, especialmente quando o centroide é utilizado para efeitos visuais ou transformações geométricas. Assim, a escolha do método depende mais de restrições práticas, como desempenho e simplicidade de implementação, do que de exigências rigorosas de precisão. No entanto, em casos onde a exatidão é crítica, métodos mais sofisticados podem ser necessários, ainda que impliquem maior custo computacional.

5.7.2 Teste de Visibilidade pela Normal

Na Seção 4.2.3.5, foi proposto um experimento para verificar a possibilidade de modificar o vetor utilizado no teste de visibilidade no *Pipeline B*. Essa proposta surgiu da hipótese de que seria viável utilizar diretamente o vetor $\hat{n}$, obtido pela Equação 6 descrita na sessão 2.5.1.1.4, eliminando a necessidade do vetor auxiliar $\hat{O}$.

Os experimentos realizados confirmaram que essa abordagem foi bem-sucedida. A visibilidade das faces pôde ser determinada diretamente pelo vetor $\hat{n}$ da câmera, sem comprometer a precisão da ocultação. Além disso, verificou-se que a normalização de $\hat{n}$ não foi necessária, resultando em uma redução adicional no custo computacional do processo.

No entanto, a alteração na metodologia impactou a lógica da avaliação da visibilidade, exigindo ajustes na interpretação dos resultados, incluindo a inversão da condição do teste $\cos\theta > 0$. Apesar dessa modificação, a nova abordagem mostrou-se eficiente, simplificando os cálculos no *pipeline* B. A figura 48 demonstra uma esfera corretamente rasterizada.

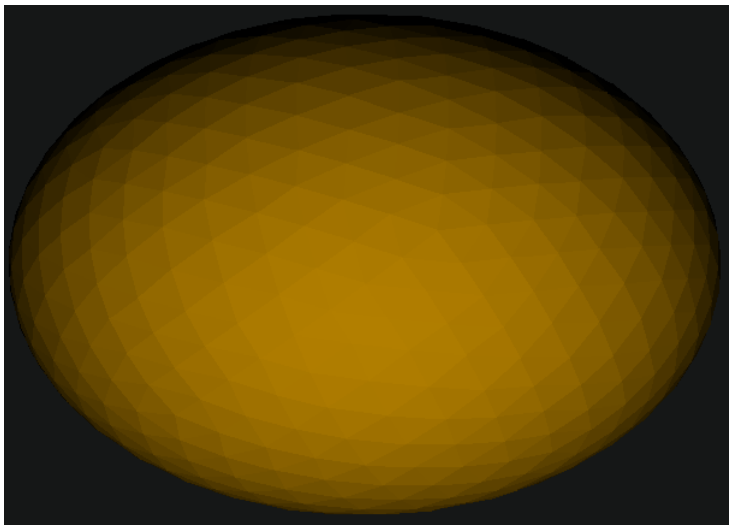
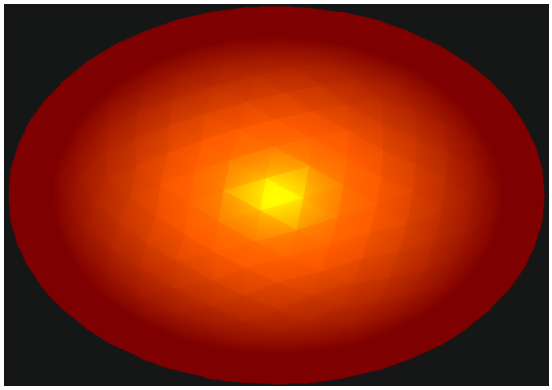


Figura 48 – Resultado da alteração do teste de visibilidade

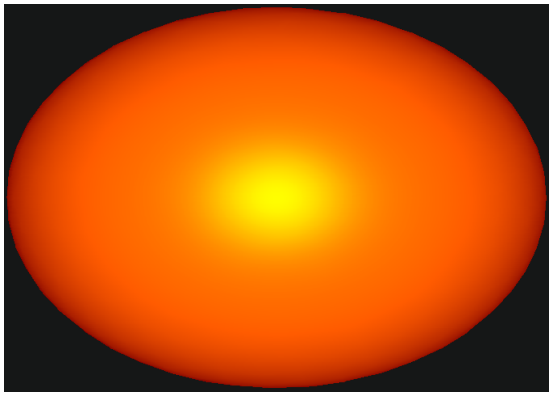
5.7.3 Alteração do Cálculo de Vetores no Phong

Dentre os modelos de iluminação implementados, o modelo de *Phong* é, sem dúvida, o que mais se aproxima da realidade. No entanto, também é o mais complexo e computacionalmente custoso. Conforme discutido na Seção 2.5.1.8.3, os cálculos dos vetores $\hat{L}$ e $\hat{S}$ utilizam o centroide da face como referência. Contudo, conforme mencionado na Seção 4.2.3.2, essa abordagem pode resultar em um efeito facetado no objeto, como ilustrado na Figura 49a.

Para mitigar esse problema, foi implementada a proposta de Blinn (1977) que modifica o cálculo dos vetores $\hat{L}$ e $\hat{S}$, passando a utilizar o centroide do objeto. Como resultado, o efeito de facetas observado na Figura 49a foi eliminado, conforme demonstrado na Figura 49b.



(a) Renderização com sombreamento *Phong* baseada no centroide da face. O efeito de facetas é perceptível.



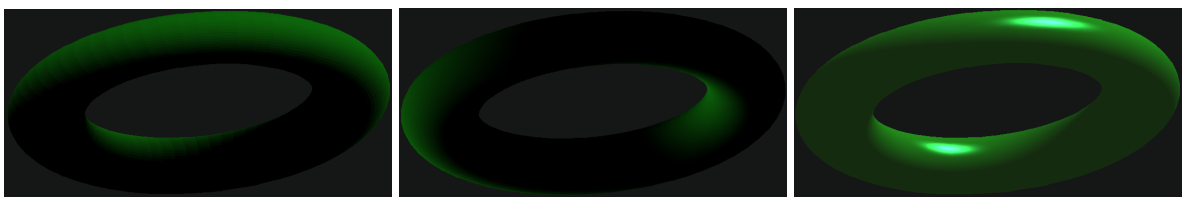
(b) Renderização com sombreamento *Phong* baseada no centroide do objeto. O efeito de facetas é eliminado.

Figura 49 – Comparação entre os diferentes cálculos dos vetores $\hat{L}$ e $\hat{S}$ no modelo de *Phong Shading*.

Isso demonstra que a simplificação proposta por Blinn (1977), condiz com posicionar a luz e observador no infinito, levando-nos a calcular os vetores $\vec{L}$ e $\vec{S}$ apenas uma única vez, considerando como ponto iluminado o centroide do objeto.

5.7.4 Vetores Normalizados Somente na Iluminação

Na Seção 4.2.3.1 foi proposta uma alteração na forma como a normalização dos vetores é aplicada ao longo do *pipeline* de visualização 3D. Durante as etapas convencionais do *pipeline* (A ou B), diversos vetores são calculados e normalizados imediatamente após sua determinação. No entanto, a hipótese investigada neste experimento considera a possibilidade de adiar esse processo, realizando a normalização apenas na fase de iluminação. Essa abordagem parte do princípio de que a magnitude dos vetores não influencia diretamente a ocultação de faces ou a coerência geométrica da cena, sendo relevante apenas nos cálculos de iluminação. A figura 50 demonstra os vetores sendo normalizados ao longo do *pipeline*.

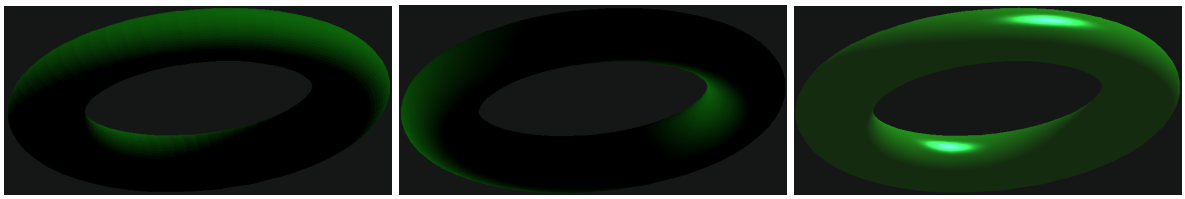


(a) Constante (b) Gouraud (c) Phong

Figura 50 – Vetores normalizados ao longo do *pipeline*

Ao aplicar essa modificação, a normalização deixou de ser realizada em etapas intermediárias, como na determinação do vetor que define a orientação da face em relação à câmera, utilizado no teste de visibilidade. Além disso, a normalização do vetor normal médio dos vértices, empregada nos modelos de iluminação de Gouraud e Phong, foi postergada para o momento espe-

cífico do cálculo de iluminação. Essa estratégia visa reduzir o custo computacional, minimizando operações desnecessárias sem comprometer a precisão visual do modelo renderizado.



(a) Constante (b) Gouraud (c) Phong

Figura 51 – Vetores somente na iluminação

Conforme a Figura 51 demonstra, na análise dos resultados obtidos, podemos constatar que a ausência de normalizações intermediárias não influencia significativamente a qualidade da imagem final, ou seja, a proposta de otimização mantém a qualidade e fidelidade visual, enquanto melhora o desempenho do *pipeline*.

A validação numérica da equivalência visual foi realizada mediante subtração pixel a pixel entre as renderizações convencionais e otimizadas, seguida da aplicação de testes estatísticos na matriz de diferenças resultante. A comparação direta das imagens, efetuada através da diferença absoluta entre cada par de pixels correspondentes, demonstrou identidade completa entre as representações ($\Delta = 0.00$ em todas as métricas analisadas).

A média e mediana das diferenças registraram valor zero na escala de 0 a 255, com desvio padrão igualmente nulo, indicando ausência total de variação nos valores comparados. A soma cumulativa das diferenças atingiu 0.00 em todos os 540 mil pixels analisados, abrangendo as três dimensões do espaço cromático RGB. Cada canal de cor - vermelho, verde e azul - apresentou distribuição estatística idêntica entre as versões comparadas, com diferenças máximas e mínimas rigorosamente nulas em todos os componentes.

A inexistência de discrepâncias mensuráveis, mesmo em nível subpixel, sustenta conclusivamente a tese de que as operações de normalização intermediárias constituem etapa redundante para a representação visual final nos modelos testados. Essa equivalência numérica absoluta valida a eficácia da otimização proposta, demonstrando que o adiamento da normalização para a fase de iluminação não compromete a integridade visual ou matemática do processo de renderização.

5.7.5 Utilização de Coordenadas no SRT para Etapas do Pipeline

Na Seção 4.2.3.4 foi comentado que, durante as etapas do pipeline de visualização 3D, é comum o uso de diferentes sistemas de coordenadas – SRU (Sistema de Referência do Universo), SRC (Sistema de Referência da Câmera) e SRT (Sistema de Referência da Tela). Embora o trabalho utilize tanto as coordenadas do SRU quanto as do SRT em determinadas etapas do pipeline, o teste proposto visa verificar se é possível adotar exclusivamente o SRT ao longo de

todo o pipeline, com o objetivo de reduzir o fluxo de dados e simplificar o processamento. Para essa experimentação, utilizamos o *pipeline A* como base.

A primeira etapa consistiu em identificar os pontos do pipeline onde são empregados sistemas de coordenadas distintos do SRT. Observou-se que as etapas de eliminação de superfícies não visíveis e de aplicação dos modelos de iluminação utilizam o SRU. Na eliminação de superfícies não visíveis, por exemplo, o elemento do SRU utilizado é o vetor $\hat{O}$, que vai do observador até o centroide da face ou do objeto. Considerando que a posição da câmera deve permanecer no SRU (devido à projeção, que impede seu mapeamento para a tela), optou-se por alterar o cálculo do centroide, fazendo-o utilizar as coordenadas do SRT (excetuando-se o eixo z). Essa mudança também demandou a atualização do cálculo da normal da face, uma vez que ela participa do teste $\hat{N} \cdot \hat{O} > 0$, onde $\hat{N}$ representa a normal da face.

Na etapa de iluminação, que requer o centroide da face ou do objeto para os cálculos dos vetores de reflexão difusa e especular – além do uso da normal da face –, os mesmos elementos já haviam sido adaptados na fase de ocultação de faces. Dessa forma, a alteração para o uso exclusivo das coordenadas do SRT foi estendida para a iluminação.

Ao testar essas modificações, constatou-se que o método de ocultação de faces ajustado, operou corretamente, conforme ilustrado na Figura 52.

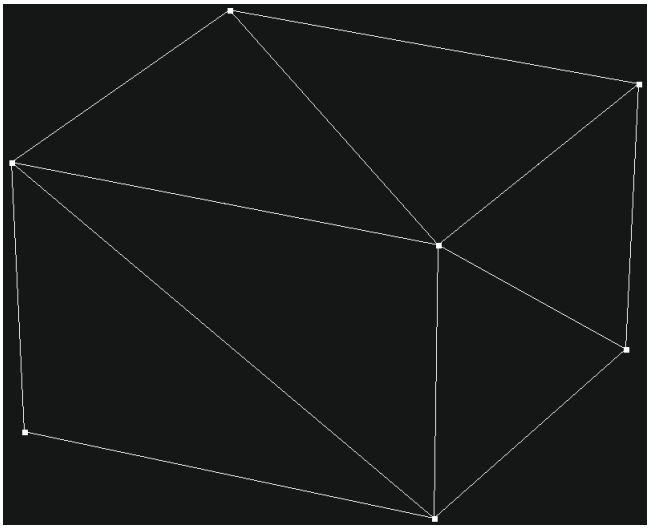
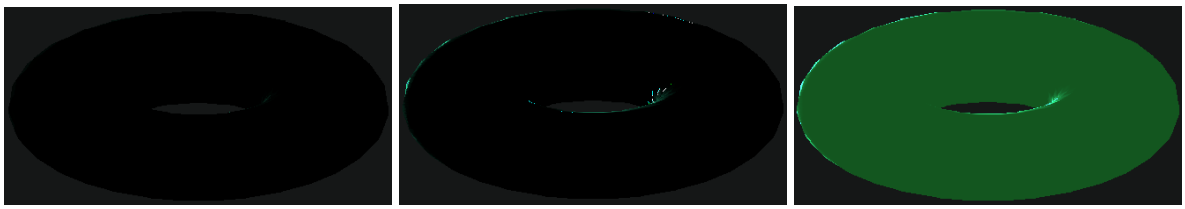


Figura 52 – Ocultação de faces utilizando coordenadas do SRT.

Na etapa de iluminação, os resultados foram significativamente diferentes dos obtidos com o método convencional. Para efeito de comparação, utilizou-se o mesmo objeto renderizado na Figura 51 da sessão anterior. A Figura 53 apresenta as renderizações para os modelos de iluminação Constant, Gouraud e Phong, utilizando exclusivamente coordenadas de tela em todo o pipeline.

Como se observa na Figura 53, os resultados diferem consideravelmente dos obtidos anteriormente. Essa divergência se deve, principalmente, à forma como os vetores normais e os centroides são calculados com as coordenadas de tela. Como a posição do pixel passa a influenciar



(a) Constant (b) Gouraud (c) Phong

Figura 53 – Resultados dos três modelos de iluminação utilizando exclusivamente coordenadas do SRT.

os cálculos de iluminação, qualquer variação na orientação altera tanto o vetor normal quanto o centroide, dando a impressão de que a fonte de luz se desloca pela cena, mesmo permanecendo estacionária.

Em conclusão, a experimentação revelou que a utilização exclusiva das coordenadas do SRT simplifica o fluxo de dados e reduz o custo computacional. Contudo, essa abordagem altera significativamente os cálculos de iluminação, resultando em diferenças visuais perceptíveis. Assim, embora seja uma estratégia viável para aplicações onde a otimização do desempenho seja prioritária, é fundamental considerar o impacto na qualidade visual, principalmente em cenários que exigem precisão na iluminação.

6
Conclusão

O presente trabalho cumpriu com êxito os objetivos gerais definidos na sessão 1.1.1. Durante o desenvolvimento, foram implementados dois modelos distintos de pipeline (2.5.1), o que possibilitou a identificação e documentação das particularidades de cada abordagem – tanto por meio do código quanto de descrições textuais. Além disso, realizou-se uma análise estatística comparativa (vide 5.4), que evidenciou a forte influência combinada do método de iluminação e do pipeline escolhido no desempenho do sistema, especialmente em termos de tempo de execução.

O estudo estatístico demonstrou que, embora o *pipeline A* apresente um tempo de execução mais rápido, sua estabilidade não é garantida, conforme evidenciado pelos 10% piores frames, com exceção do modelo de iluminação Phong. Em contrapartida, o *pipeline B*, mesmo com um tempo médio superior, se destaca pela maior estabilidade. Dessa forma, a escolha entre os pipelines deve considerar as prioridades do sistema: aplicações que valorizam uma velocidade média maior podem se beneficiar do *pipeline A*, enquanto sistemas que exigem estabilidade frente a picos de processamento podem optar pelo *pipeline B*. Vale ressaltar, entretanto, que ambos os pipelines representam simplificações do modelo originalmente proposto por Smith (1983).

Adicionalmente, foram formuladas e testadas diversas hipóteses experimentais (4.2). Dentre elas, a hipótese de utilizar exclusivamente as coordenadas do SRT para todas as etapas do pipeline não apresentou resultados satisfatórios, impactando de maneira significativa a qualidade visual das imagens quando comparada à abordagem tradicional.

Outros aspectos relevantes discutidos neste trabalho incluem a identificação de divergências entre os pipelines A e B, a começar pela orientação dos sistemas de coordenadas – sendo o pipeline A orientado pela regra da mão direita e o pipeline B pela regra da mão esquerda – o que influencia diretamente o cálculo do vetor $\hat{n}$ na transformação do SRU para o SRC (2.5.1.1.4). Essa discrepância também exigiu ajustes nos parâmetros dos planos próximo e distante, bem como na distância do plano de projeção, para garantir uma representação espacial coerente.

O estudo abordou ainda o método de determinação do vetor unitário médio do vértice, utilizado nos modelos de iluminação Phong e Gouraud. Embora Foley et al. (1995) tenha proposto um método para sua obtenção, a simplificação sugerida por Conci, Azevedo e Leta (2003) mostrou-se válida, mesmo gerando diferenças perceptíveis – sobretudo no modelo de Gouraud – pois o modelo de iluminação Phong Phong (1973) é, por natureza, uma aproximação destinada a incrementar o realismo da cena e não uma representação física exata. Da mesma forma, a comparação entre dois métodos para o cálculo do centroide indicou que ambas as abordagens produzem resultados similares, deixando a escolha a critério do programador, uma vez que o custo computacional é comparável.

Outra questão investigada foi o teste de visibilidade de superfícies baseado na normal, que envolve o produto escalar entre os vetores $\hat{O}$ e $\hat{N}$. No *pipeline B*, a alteração para utilizar o vetor $\hat{n}$ da câmera (2.5.1.1.4) – sem a necessidade de normalização – demonstrou-se funcional e eficaz, evidenciando uma possível otimização neste estágio.

Adicionalmente, foram realizados experimentos com o modelo de iluminação Phong, nos quais o cálculo dos vetores $\hat{L}$ e $\hat{S}$ foi modificado para ser efetuado a partir do centroide do objeto, em vez do centroide da face. Essa mudança eliminou o efeito de facetamento, resultando em renderizações mais suaves, o que comprova que, por vezes, simplificações podem não só otimizar o desempenho, mas também melhorar a qualidade visual. Ainda, testou-se a hipótese de normalizar os vetores utilizados no pipeline apenas no momento dos cálculos de iluminação, uma vez que a magnitude é relevante somente nesta etapa. Os resultados confirmaram que essa estratégia otimiza os cálculos sem comprometer o resultado final.

Por fim, o trabalho atingiu os objetivos específicos descritos em 1.1.2, documentando detalhadamente as decisões e funções implementadas, de modo a oferecer uma base sólida para futuras pesquisas e desenvolvimentos na área. Em síntese, este estudo não só demonstrou a viabilidade de diversas simplificações e otimizações no pipeline de visualização 3D, mas também evidenciou as implicações dessas escolhas no desempenho e na qualidade visual dos sistemas. Assim, os resultados apresentados fornecem subsídios importantes para a seleção do pipeline mais adequado conforme as necessidades específicas de cada aplicação, contribuindo para o avanço do conhecimento nesta área.

7

Trabalhos Futuros

Embora o presente trabalho tenha alcançado seus objetivos, ele ainda está longe de ser considerado completo. Existem diversas direções que podem ser exploradas em futuras pesquisas, seja para expandir o escopo, aprimorar as experimentações realizadas ou implementar novas funcionalidades. Esta seção documenta algumas dessas possíveis melhorias, as quais não foram abordadas neste projeto devido a limitações de escopo, tempo ou recursos.

Um aspecto fundamental a ser aprimorado diz respeito à otimização dos pipelines. Apesar de terem sido implementados com sucesso, os algoritmos atuais podem ser otimizados para melhorar tanto a performance quanto a estabilidade. No projeto, toda a execução ocorre em uma única thread, sem a aplicação de técnicas de paralelização. Como a computação gráfica se beneficia intensamente da execução paralela, futuras versões podem explorar a paralelização a nível de processador – o que poderia potencialmente triplicar ou quadruplicar o desempenho – bem como a utilização de aceleração via GPU, aproveitando seu hardware especializado em cálculos matriciais de alta demanda.

No nível da aplicação, melhorias podem ser implementadas no gerenciamento de memória e na refatoração do código. Durante a implementação, cada função foi desenvolvida com uma única responsabilidade, o que, embora torne o código modular, aumentou o número de chamadas de função durante a execução. Uma abordagem que reduza essa sobrecarga, por meio de uma melhor organização dos dados e da minimização de chamadas repetitivas, poderá resultar em ganhos significativos de desempenho.

Outros pontos de atenção dizem respeito à resolução de bugs e à melhoria da usabilidade. Por exemplo, pequenos problemas, como a movimentação indesejada da câmera quando o usuário interage com outras partes da interface gráfica, ou as operações geométricas dos objetos que ocorrem fora da origem — dificultando sua manipulação — devem ser corrigidos em versões futuras. Comentários e marcações **@todo** presentes no código já indicam áreas que necessitam de ajustes e refatorações, servindo de guia para futuros mantenedores.

Além dessas melhorias, o projeto pode se beneficiar da implementação de técnicas avançadas de renderização e otimização. A incorporação de *Level-of-Detail* (LoD) permitiria reduzir a complexidade dos modelos renderizados em função da distância do observador, otimizando a performance sem comprometer significativamente a qualidade visual. De forma similar, a utilização de *Triangle Strips* pode diminuir a quantidade de dados processados durante a renderização de superfícies, melhorando a eficiência do pipeline gráfico.

Outras otimizações técnicas podem ser consideradas para enriquecer tanto a experiência visual quanto a performance do sistema. A implementação de efeitos de *fog* (névoa) pode ser explorada para melhorar a imersão em cenas complexas e reduzir a carga de renderização, atenuando a visibilidade de elementos distantes. Da mesma forma, a implementação de sombras realistas – utilizando técnicas como *shadow mapping* ou *shadow volumes* – pode aumentar o realismo da cena, embora exija um processamento adicional que deverá ser otimizado para manter o equilíbrio entre qualidade visual e desempenho.

Ademais, a otimização das malhas utilizadas no projeto é outro caminho promissor. A aplicação de técnicas de redução de polígonos, reestruturação e organização dos dados geométricos pode resultar em modelos mais leves e eficientes, sem sacrificar a fidelidade visual. A integração de algoritmos que automatizem a simplificação de malhas ou que otimizem a forma como os dados são armazenados e acessados contribuirá significativamente para o desempenho do sistema.

Em suma, a integração de estratégias de paralelização, otimização de gerenciamento de memória, correção de bugs e a adoção de técnicas avançadas como LoD, Triangle Strips, efeitos de fog, implementação de sombras e otimizações de malhas representam caminhos promissores para o aprimoramento do projeto. Essas melhorias não apenas potencializariam o desempenho e a estabilidade do sistema, mas também ampliariam sua aplicabilidade em cenários de computação gráfica de alta demanda, constituindo uma base sólida para trabalhos futuros.

Referências

AKENINE-MOOLLER, T. et al. *Real-Time Rendering*. 4th edition. ed. [S.l.]: CRC Press, 2018. ISBN 9781138627000; 1138627003. Citado 15 vezes nas páginas 23, 24, 26, 29, 30, 31, 37, 41, 46, 49, 64, 66, 67, 70 e 124.

BLINN, J. F. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 11, n. 2, p. 192–198, jul. 1977. ISSN 0097-8930. Disponível em: <https://doi.org/10.1145/965141.563893>. Citado 2 vezes nas páginas 112 e 113.

BOULOS, P.; CAMARGO, *Geometria Analítica: Um Tratamento Vetorial*. 3. ed. São Paulo: Makron Books do Brasil Editora LTDA, 2004. Citado 6 vezes nas páginas 150, 151, 152, 154, 155 e 157.

CONCI, A.; AZEVEDO, E.; LETA, F. *Computação Gráfica - Teoria e Prática*. [S.l.: s.n.], 2003. v. 1. ISBN 10: 85-352-2329-0. Citado 5 vezes nas páginas 19, 96, 108, 109 e 118.

DORTA, B. *Computação Gráfica - Bruno Dorta*. [S.l.], 2024. Acessado em: 7 de outubro de 2024. Disponível em: <https://www.brunodorta.com.br/cg/>. Citado na página 42.

ESPINOSA, I. C. d. O. N.; BISCOLLA, L. M. d. C. C. O.; FILHO, P. B. *Álgebra linear para computação*. 1. ed. [S.l.]: LTC, 2007. Citado 2 vezes nas páginas 158 e 159.

FOLEY, J. D. et al. *Introduction to Computer Graphics*. [S.l.]: Addison-Wesley Professional, 1993. ISBN 9780201609219; 0201609215. Citado 2 vezes nas páginas 38 e 66.

FOLEY, J. D. et al. *Computer Graphics: Principles and Practice*. 2nd. ed. Reading, MA: Addison-Wesley, 1995. Citado 24 vezes nas páginas 19, 22, 23, 26, 29, 30, 31, 33, 36, 37, 40, 47, 60, 62, 63, 64, 65, 66, 67, 70, 96, 108, 109 e 118.

GOURAUD, H. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20, n. 6, p. 623–629, 1971. Citado 5 vezes nas páginas 19, 26, 70, 71 e 96.

HEARN, D.; BAKER, M. P. *Computer Graphics, C Version*. 2nd. ed. New Jersey: Prentice Hall, 1997. Citado 18 vezes nas páginas 22, 23, 24, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 40, 46, 47, 66 e 71.

LATTNER, C.; ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, p. 75–86, 2004. Citado na página 77.

LLVM Project. *Clang 13 Documentation*. [S.l.], 2021. Acessado em: 26 de outubro de 2024. Disponível em: <https://clang.llvm.org/docs/>. Citado na página 77.

MEYERS, S. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. [S.l.]: O'Reilly Media, 2017. Citado na página 76.

Microsoft. *Microsoft Visual C++ Documentation*. [S.l.], 2022. Acessado em: 26 de outubro de 2024. Disponível em: <https://docs.microsoft.com/en-us/cpp/>. Citado na página 77.

PEREIRA, J. M. *Pipeline de Visualização Tridimensiona*. [S.l.], 2013. Disponível em: <https://disciplinas.ist.utl.pt/~leic-cg.daemon/textos/livro/Pipeline%20Visualizacao.pdf>. Citado na página 45.

PHONG, B. T. *Illumination of Computer-Generated Images*. [S.l.], 1973. Citado 7 vezes nas páginas 19, 26, 70, 73, 96, 97 e 118.

PRUNIER, J.-C. *Understanding Left-Handed vs. Right-Handed Coordinate Systems*. [S.l.], 2025. Acessado em: 28 de Março de 2025. Disponível em: <https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/coordinate-systems.html>. Citado na página 41.

Santa Catarina, A. *Aulas de Computação Gráfica 2024*. [S.l.], 2024. Disponível em: <https://www.inf.unioeste.br/~adair/CG/Notas%20Aula/Aulas%20de%20CG%202024.pdf>. Citado 22 vezes nas páginas 30, 34, 35, 38, 39, 40, 42, 43, 48, 49, 51, 52, 53, 54, 56, 57, 58, 59, 60, 61, 63 e 64.

SMITH, A. R. The viewing transformation. 1983. Disponível em: <http://alvyray.com/Memos/CG/Pixar/view84.pdf>. Citado 8 vezes nas páginas 18, 27, 36, 38, 44, 45, 59 e 117.

STEIMBRUCH, A.; WINTERLE, P. *Geometria Analítica*. 1. ed. [S.l.]: Pearson Universidades, 1987. Citado 3 vezes nas páginas 153, 154 e 155.

STROUSTRUP, B. *The C++ Programming Language*. 4th. ed. [S.l.]: Addison-Wesley Professional, 2013. Citado na página 76.

VELHO, L.; GOMES, J. *Sistemas Gráficos 3D*. [S.l.]: IMPA, 2007. ISBN 9788524401672. Citado na página 27.

Apêndices

APÊNDICE A – Geraçao Procedural de Formas Geometricas Simples

A geração procedural é uma abordagem eficiente para criar formas geométricas primitivas, como esferas, cilindros e cones. Esse método permite que algoritmos procedurais gerem essas geometrias de maneira eficiente e parametrizável, o que facilita sua inclusão em simulações, renderizações e outras aplicações gráficas (AKENINE-MOOLLER et al., 2018). Um aspecto central desse método e a chamada **aproximação poligonal**, na qual o objeto e discretizado em faces planas, formando uma malha que aproxima superfícies curvas. Para objetos convexos, essa malha cria uma superfície fechada que representa o formato desejado.

A aproximação poligonal é especialmente útil ao representar superfícies curvas com faces planas, pois simplifica a modelagem de geometrias complexas. Cada forma geométrica e composta por polígonos que se ajustam para aproximar as curvas do objeto, proporcionando um equilíbrio entre a simplicidade da representação e a precisão visual. A densidade da malha poligonal influencia diretamente na qualidade da aproximação. Abaixo, discutimos como esferas, cilindros e cones podem ser gerados com esse método.

A.1 Esferas

Uma técnica comum para a criação de esferas e a subdivisão de um polígono de vinte lados (icosaedro), onde as faces triangulares são subdivididas recursivamente e os novos vértices são projetados na superfície da esfera. O código abaixo implementa essa abordagem:

```
1  /**
2   * @brief Metodo para criacao de um icosaedro
3   *
4   * @param radius Distancia dos vertices da origem
5   * @param subdivisions Numero de subdivisoes em cima da malha mais
        basica. O numero de faces quadruplica a cada subdivisao.
6   * @param shift Deslocamento do icosaedro
7   * @return models::Mesh* Ponteiro para a malha do icosaedro
8   */
9  models::Mesh *shapes::icosphere(float radius, int subdivisions, core:
       :Vector3 shift)
10 {
11   std::vector<core::Vertex *> vertices;
12   std::vector<std::vector<int>> faces;
```

```
13
14  // Constantes do Icosaedro
15  const float t = (1.0f + std::sqrtf(5.0)) / 2.0f;
16
17  // Definir os 12 vertices iniciais de um Icosaedro
18  vertices.push_back(new core::Vertex(-1 + shift.x, t + shift.y, 0 +
     shift.z, 1.0f, nullptr, "v0"));
19  vertices.push_back(new core::Vertex(1 + shift.x, t + shift.y, 0 +
     shift.z, 1.0f, nullptr, "v1"));
20  vertices.push_back(new core::Vertex(-1 + shift.x, -t + shift.y, 0 +
      shift.z, 1.0f, nullptr, "v2"));
21  vertices.push_back(new core::Vertex(1 + shift.x, -t + shift.y, 0 +
     shift.z, 1.0f, nullptr, "v3"));
22
23  vertices.push_back(new core::Vertex(0 + shift.x, -1 + shift.y, t +
     shift.z, 1.0f, nullptr, "v4"));
24  vertices.push_back(new core::Vertex(0 + shift.x, 1 + shift.y, t +
     shift.z, 1.0f, nullptr, "v5"));
25  vertices.push_back(new core::Vertex(0 + shift.x, -1 + shift.y, -t +
      shift.z, 1.0f, nullptr, "v6"));
26  vertices.push_back(new core::Vertex(0 + shift.x, 1 + shift.y, -t +
     shift.z, 1.0f, nullptr, "v7"));
27
28  vertices.push_back(new core::Vertex(t + shift.x, 0 + shift.y, -1 +
     shift.z, 1.0f, nullptr, "v8"));
29  vertices.push_back(new core::Vertex(t + shift.x, 0 + shift.y, 1 +
     shift.z, 1.0f, nullptr, "v9"));
30  vertices.push_back(new core::Vertex(-t + shift.x, 0 + shift.y, -1 +
      shift.z, 1.0f, nullptr, "v10"));
31  vertices.push_back(new core::Vertex(-t + shift.x, 0 + shift.y, 1 +
     shift.z, 1.0f, nullptr, "v11"));
32
33  // Normalizar os vertices
34  for (auto &vertex : vertices)
35  {
36    vertex->setVector(vertex->normalize());
37  }
38
39  // Definir as 20 faces triangulares do Icosaedro
40  faces = {
41      {0, 11, 5},
42      {0, 5, 1},
```

```
      {0, 1, 7},
      {0, 7, 10},
      {0, 10, 11},
      {1, 5, 9},
      {5, 11, 4},
      {11, 10, 2},
      {10, 7, 6},
      {7, 1, 8},
      {3, 9, 4},
      {3, 4, 2},
      {3, 2, 6},
      {3, 6, 8},
      {3, 8, 9},
      {4, 9, 5},
      {2, 4, 11},
      {6, 2, 10},
      {8, 6, 7},
      {9, 8, 1}};

  // Subdivisao recursiva para aproximar uma esfera
  std::unordered_map<std::pair<int, int>, int, hash_pair> cache;
  for (int i = 0; i < subdivisions; i++)
  {
    std::vector<std::vector<int>> newFaces;
    for (auto &face : faces)
    {
      int v1 = face[0];
      int v2 = face[1];
      int v3 = face[2];

      // Encontrar pontos medios e criar novos triangulos
      int a = getMidPoint(v1, v2, cache, vertices);
      int b = getMidPoint(v2, v3, cache, vertices);
      int c = getMidPoint(v3, v1, cache, vertices);

      newFaces.push_back({v1, a, c});
      newFaces.push_back({v2, b, a});
      newFaces.push_back({v3, c, b});
      newFaces.push_back({a, b, c});
    }
    faces = newFaces;
  }
```

```
85
86    // Ajustar os vertices para o raio desejado
87    for (auto &vertex : vertices)
88    {
89      vertex->setX(vertex->getX() * radius);
90      vertex->setY(vertex->getY() * radius);
91      vertex->setZ(vertex->getZ() * radius);
92    }
93
94    // Criar a malha da Icosphere
95    models::Mesh *icosphere = new models::Mesh(vertices, faces, "
       icosphere");
96
97    return icosphere;
98  }
99
100 // Estrutura auxiliar para o cache de subdivisoes
101 struct hash_pair
102 {
103   template <class T1, class T2>
104   std::size_t operator()(const std::pair<T1, T2> &p) const
105   {
106     auto hash1 = std::hash<T1>{}(p.first);
107     auto hash2 = std::hash<T2>{}(p.second);
108     return hash1 ^ (hash2 << 1); // Combinar os dois hashes
109   }
110 };
111
112 /**
113  * @brief Calcula o ponto medio entre dois vertices e gerencia o
       cache.
114  *
115  * Esta funcao calcula as coordenadas do ponto medio entre dois
       vertices,
116  * dados seus indices no vetor de vertices. Ela utiliza um cache para
        evitar
117  * o recalculo de pontos medios ja computados, otimizando o processo
       de
118  * subdivisao de uma malha.
119  *
120  * @param v1 Indice do primeiro vertice.
121  * @param v2 Indice do segundo vertice.
```

```
122 * @param cache Referencia para um mapa que armazena os pontos medios
        calculados.
123 *              A chave do mapa e um par ordenado dos indices dos
    vertices
124 *              (usando std::minmax para garantir que a ordem nao
    importe), e o
125 *              valor e o indice do novo vertice na lista `vertices`.
126 * @param vertices Referencia para um vetor que armazena todos os
    vertices da malha.
127 *
128 * @return O indice do vertice do ponto medio no vetor `vertices`.
129 *         Se o ponto medio j\ tiver sido calculado, retorna o indice
130 *         armazenado no cache. Caso contrario, calcula o ponto medio
    ,
131 *         adiciona o novo vertice ao vetor `vertices` e atualiza o
    cache
132 *         antes de retornar o indice.
133 *
134 * @remark A funcao normaliza o vetor do ponto medio antes de
    adiciona-lo
135 *         ao vetor de vertices, garantindo que o novo vertice fique
    na
136 *         superficie de uma esfera unitaria (ou seja, a distancia da
        origem
137 *         e 1).
138 *
139 * @remark O cache e crucial para o desempenho, pois evita calculos
140 *         redundantes do mesmo ponto medio, o que seria
    computacionalmente
141 *         custoso, especialmente com um grande numero de subdivisoes
    .
142 *
143 * @code
144 * // Exemplo de uso:
145 * std::vector<core::Vertex*> vertices;
146 * std::unordered_map<std::pair<int, int>, int, hash_pair> cache;
147 * // ... (adicionar vertices iniciais ao vetor vertices) ...
148 * int indexMidPoint = getMidPoint(0, 1, cache, vertices);
149 * core::Vertex* midVertex = vertices[indexMidPoint];
150 * // ... (usar o vertice midVertex) ...
151 * @endcode
152 */
```

```
153 int getMidPoint(int v1, int v2, std::unordered_map<std::pair<int, int
      >, int, hash_pair> &cache, std::vector<core::Vertex *> &vertices)
154 {
155   auto key = std::minmax(v1, v2);
156   if (cache.find(key) != cache.end())
157   {
158     return cache[key];
159   }
160
161   // Obter coordenadas dos dois vertices
162   core::Vertex *vertex1 = vertices[v1];
163   core::Vertex *vertex2 = vertices[v2];
164
165   // Encontrar ponto medio entre v1 e v2
166   core::Vertex *midVertex = new core::Vertex(
167       (vertex1->getX() + vertex2->getX()) / 2.0f,
168       (vertex1->getY() + vertex2->getY()) / 2.0f,
169       (vertex1->getZ() + vertex2->getZ()) / 2.0f,
170       1.0f, nullptr, "v" + std::to_string(vertices.size()));
171   midVertex->setVector(midVertex->normalize());
172
173   vertices.push_back(midVertex);
174   cache[key] = vertices.size() - 1;
175
176   return vertices.size() - 1;
177 }
```

Listing A.1 – Geraçao de Esfera por Subdivisao de Icosaedro

A.2 Cilindros

A geraçã de cilindros utiliza equações cilíndricas para definir altura, raio e o numero de subdivisões nas bases circulares:

```
1 /**
2  * @brief Cria um cilindro com topo e base circulares
3  *
4  * @param radius Raio da base/topo
5  * @param height Altura do cilindro
6  * @param segments Numero de segmentos (resolucao)
7  * @param shift Deslocamento do cilindro
8  * @return models::Mesh* Malha do cilindro
```

```
 */
models::Mesh *shapes::cylinder(float radius, float height, int
    segments, core::Vector3 shift)
{
  std::vector<core::Vertex *> vertices;
  std::vector<std::vector<int>> faces;

  // Vertices da base
  for (int i = 0; i < segments; i++)
  {
    float theta = 2.0f * PI * i / segments;
    float x = radius * cos(theta) + shift.x;
    float z = radius * sin(theta) + shift.z;
    core::Vertex *v = new core::Vertex(x, shift.y, z, 1.0f, nullptr,
     "v_base_" + std::to_string(i));
    vertices.push_back(v);
  }

  // Vertices do topo
  for (int i = 0; i < segments; i++)
  {
    float theta = 2.0f * PI * i / segments;
    float x = radius * cos(theta) + shift.x;
    float z = radius * sin(theta) + shift.z;
    core::Vertex *v = new core::Vertex(x, shift.y + height, z, 1.0f,
     nullptr, "v_top_" + std::to_string(i));
    vertices.push_back(v);
  }

  // Centro da base e topo
  core::Vertex *baseCenter = new core::Vertex(shift.x, shift.y, shift
   .z, 1.0f, nullptr, "base_center");
  vertices.push_back(baseCenter);
  int baseCenterIdx = vertices.size() - 1;

  core::Vertex *topCenter = new core::Vertex(shift.x, shift.y +
   height, shift.z, 1.0f, nullptr, "top_center");
  vertices.push_back(topCenter);
  int topCenterIdx = vertices.size() - 1;

  // Faces da base e topo (triangulos)
  for (int i = 0; i < segments; i++)
```

```
46    {
47      int next = (i + 1) % segments;
48      // Base
49      faces.push_back({i, next, baseCenterIdx});
50      // Topo
51      faces.push_back({segments + i, topCenterIdx, segments + next});
52    }
53
54    // Faces laterais (quadrados divididos em triangulos)
55    for (int i = 0; i < segments; i++)
56    {
57      int next = (i + 1) % segments;
58      int currentBase = i;
59      int currentTop = segments + i;
60      int nextBase = next;
61      int nextTop = segments + next;
62
63      faces.push_back({currentBase, currentTop, nextTop});
64      faces.push_back({currentBase, nextTop, nextBase});
65    }
66
67    return new models::Mesh(vertices, faces, "cylinder");
68  }
```

Listing A.2 – Geraçao de Cilindro com Bases Circulares

A.3 Cones

A geração de cones segue uma logica similar aos cilindros, com adaptações para o vértice único:

```
1  /**
2   * @brief Metodo para criacao de um cilindro
3   *
4   * @param radius Raio do cilindro
5   * @param height Altura do cilindro
6   * @param segments Numero de segmentos do cilindro
7   * @param shift Deslocamento do cilindro
8   * @return models::Mesh* Ponteiro para a malha do cilindro
9   *
10  */
```

```
11 models::Mesh *shapes::cone(float radius, float height, int segments,
       core::Vector3 shift)
12 {
13   std::vector<core::Vertex *> vertexes;
14   std::vector<std::vector<int>> faces;
15
16   for (int i = 0; i < segments; i++)
17   {
18     float theta = static_cast<float>(i) / segments;
19     float ratio = 2.0f * PI * theta;
20     float x = radius * std::cosf(ratio);
21     float y = radius * std::sinf(ratio);
22
23     core::Vertex *vertex = new core::Vertex(x + shift.x, y, 0.0f +
       shift.y, 1.0f + shift.z, nullptr, "v" + std::to_string(i));
24     vertexes.push_back(vertex);
25   }
26
27   // Adiciona o vertice central da base
28   core::Vertex *center = new core::Vertex(0.0f + shift.x, 0.0f +
       shift.y, 0.0f + shift.z, 1.0f, nullptr, "center");
29   vertexes.push_back(center);
30   int centerIndex = vertexes.size() - 1;
31
32   // Cria a face da base (sentido anti-horario)
33   for (int i = 0; i < segments; i++)
34   {
35     int next = (i + 1) % segments;
36     faces.push_back({i, centerIndex, next});
37   }
38
39   // Adiciona o vertice central do topo
40   core::Vertex *top = new core::Vertex(0.0f + shift.x, 0.0f + shift.y
       , height + shift.z, 1.0f, nullptr, "top");
41   vertexes.push_back(top);
42   int topIndex = vertexes.size() - 1;
43
44   // Conecta o topo com os vertices da base (sentido anti-horario)
45   for (int i = 0; i < segments; i++)
46   {
47     int next = (i + 1) % segments;
48     faces.push_back({i, next, topIndex});
```

```
49    }
50
51    // Cria a malha do cilindro
52    models::Mesh *cone = new models::Mesh(vertexes, faces, "cone");
53
54    return cone;
55  }
```

Listing A.3 – Geraçao de Cone por Aproximaçao Poligonal

A.4 Torus

O torus, comumente conhecido como forma de rosquinha, e uma superfície quádrica gerada pela revolução de um circulo em torno de um eixo externo ao seu plano. Sua geração procedural utiliza coordenadas paramétricas com dois níveis de subdivisão:

```
1  /**
2   * @brief Cria um torus (rosquinha)
3   * @param majorRadius Raio principal (do centro ao tubo)
4   * @param minorRadius Raio do tubo
5   * @param majorSegments Segmentos ao redor do centro
6   * @param minorSegments Segmentos ao redor do tubo
7   * @param shift Deslocamento do torus
8   * @return models::Mesh* Malha do torus
9   * @note Requer ajustes no sistema de recorte 3D
10  */
11 models::Mesh *shapes::torus(float majorRadius, float minorRadius,
12                              int majorSegments, int minorSegments,
13                              core::Vector3 shift) {
14   std::vector<core::Vertex *> vertices;
15   std::vector<std::vector<int>> faces;
16
17   // Geracao dos vertices usando coordenadas toroidais
18   for (int i = 0; i < majorSegments; i++) {
19     float majorAngle = 2.0f * PI * i / majorSegments;
20     float cosMajor = cos(majorAngle);
21     float sinMajor = sin(majorAngle);
22
23     for (int j = 0; j < minorSegments; j++) {
24       float minorAngle = 2.0f * PI * j / minorSegments;
25
```

```
26        // Calculo das coordenadas 3D
27        float x = (majorRadius + minorRadius * cos(minorAngle)) *
     cosMajor + shift.x;
28        float y = (majorRadius + minorRadius * cos(minorAngle)) *
     sinMajor + shift.y;
29        float z = minorRadius * sin(minorAngle) + shift.z;
30
31        vertices.push_back(new core::Vertex(x, y, z, 1.0f, nullptr,
32                           "v_" + std::to_string(i) + "_" + std::
     to_string(j)));
33      }
34    }
35
36    // Conexao das faces em malha quadrangular
37    for (int i = 0; i < majorSegments; i++) {
38      for (int j = 0; j < minorSegments; j++) {
39        int nextI = (i + 1) % majorSegments;
40        int nextJ = (j + 1) % minorSegments;
41
42        // indices dos vertices do quadrilatero
43        int v0 = i * minorSegments + j;
44        int v1 = i * minorSegments + nextJ;
45        int v2 = nextI * minorSegments + nextJ;
46        int v3 = nextI * minorSegments + j;
47
48        // Divisao do quadrilatero em dois triangulos
49        faces.push_back({v0, v1, v2});
50        faces.push_back({v0, v2, v3});
51      }
52    }
53
54    return new models::Mesh(vertices, faces, "torus");
55  }
```

Listing A.4 – Geraçao de Torus por Coordenadas Parametricas

Esta implementação permite gerar toroides com diferentes proporções, desde formas padrão ate configurações mais exóticas como toroides esmagados ou alongados, ajustando a relação entre os raios principal e secundário.

A.5 Pirâmide

A pirâmide de base quadrada e gerada através da definição explicita de seus vértices fundamentais e conexão das faces triangulares:

```
/**
 * @brief Cria uma piramide de base quadrada
 * @param base Tamanho da aresta da base
 * @param height Altura do vertice superior
 * @param shift Deslocamento espacial
 * @return models::Mesh* Malha da piramide
 */
models::Mesh *shapes::pyramid(float base, float height, core::Vector3
    shift) {
  // Configuracao dos vertices fundamentais
  float halfBase = base / 2.0f;
  core::Vertex *v0 = new core::Vertex( halfBase + shift.x,  halfBase
   + shift.y, 0.0f + shift.z, 1.0f, nullptr, "A");
  core::Vertex *v1 = new core::Vertex(-halfBase + shift.x,  halfBase
   + shift.y, 0.0f + shift.z, 1.0f, nullptr, "B");
  core::Vertex *v2 = new core::Vertex(-halfBase + shift.x, -halfBase
   + shift.y, 0.0f + shift.z, 1.0f, nullptr, "C");
  core::Vertex *v3 = new core::Vertex( halfBase + shift.x, -halfBase
   + shift.y, 0.0f + shift.z, 1.0f, nullptr, "D");
  core::Vertex *v4 = new core::Vertex(0.0f + shift.x, 0.0f + shift.y,
    height + shift.z, 1.0f, nullptr, "E");

  // Definicao das faces triangulares
  std::vector<std::vector<int>> faces = {
    {0, 3, 2, 1},    // Face da base
    {0, 1, 4},       // Face frontal
    {1, 2, 4},       // Face esquerda
    {2, 3, 4},       // Face traseira
    {3, 0, 4}        // Face direita
  };

  return new models::Mesh({v0, v1, v2, v3, v4}, faces, "pyramid");
}
```

Listing A.5 – Geraçao de Piramide de Base Quadrada

A.6 Cubo

O cubo unitário e gerado através da especificação de seus oito vértices fundamentais e conexão das faces quadrangulares divididas em triângulos:

```
/**
 * @brief Gera cubo unitario centrado na origem
 * @param shift Deslocamento espacial
 * @return models::Mesh* Malha do cubo
 */
models::Mesh *shapes::cube(core::Vector3 shift) {
  // Definicao dos oito vertices do cubo
  std::vector<core::Vertex*> vertices = {
    new core::Vertex(-1.0f + shift.x, -1.0f + shift.y, -1.0f + shift.z, 1.0f, nullptr, "v0"),
    new core::Vertex( 1.0f + shift.x, -1.0f + shift.y, -1.0f + shift.z, 1.0f, nullptr, "v1"),
    new core::Vertex( 1.0f + shift.x, -1.0f + shift.y,  1.0f + shift.z, 1.0f, nullptr, "v2"),
    new core::Vertex(-1.0f + shift.x, -1.0f + shift.y,  1.0f + shift.z, 1.0f, nullptr, "v3"),
    new core::Vertex(-1.0f + shift.x,  1.0f + shift.y, -1.0f + shift.z, 1.0f, nullptr, "v4"),
    new core::Vertex( 1.0f + shift.x,  1.0f + shift.y, -1.0f + shift.z, 1.0f, nullptr, "v5"),
    new core::Vertex( 1.0f + shift.x,  1.0f + shift.y,  1.0f + shift.z, 1.0f, nullptr, "v6"),
    new core::Vertex(-1.0f + shift.x,  1.0f + shift.y,  1.0f + shift.z, 1.0f, nullptr, "v7")
  };

  // Conexao das 12 faces triangulares (2 triangulos por face do cubo)
  std::vector<std::vector<int>> faces = {
    // Face inferior
    {0, 2, 1}, {0, 3, 2},
    // Face superior
    {4, 5, 6}, {4, 6, 7},
    // Face frontal
    {0, 1, 5}, {0, 5, 4},
    // Face traseira
    {2, 3, 7}, {2, 7, 6},
```

```
29      // Face esquerda
30      {3, 0, 4}, {3, 4, 7},
31      // Face direita
32      {1, 2, 6}, {1, 6, 5}
33    };
34
35    return new models::Mesh(vertices, faces, "cube");
36  }
```

Listing A.6 – Geraçao de Cubo Unitario

A.7 Considerações de Implementação

A geração procedural oferece vantagens significativas em flexibilidade e controle paramétrico, porem requer atenção a aspectos específicos de cada forma geométrica. A Tabela 12 sintetiza os principais parâmetros de controle:

Tabela 12 – Parâmetros de geração procedural por forma geométrica

Forma	**Parâmetros**	**Efeito**
Esfera	Subdivisões	Densidade da malha poligonal
Cilindro	Segmentos	Resolução das bases circulares
Cone	Segmentos	Suavidade da superfície cônica
Torus	Major/Minor Segments	Resolução axial e tubular
Piramide	Base/Altura	Proporções geométricas
Cubo	-	Tamanho fixo unitário

Principais estratégias de otimização implementadas:

- **Controle de Complexidade**: Adaptação dinâmica do Level of Detail (LOD) via parâmetros de subdivisão (Listings A.1 e A.4)
- **Reuso de Vertices**: Cache de pontos médios na icosphere (Seção A.1)
- **Topologia Eficiente**: Uso de malhas trianguladas para cubos (Listing A.6) e pirâmides (Listing A.5)
- **Parametrização Inteligente**: Equações paramétricas para torus (Equaçao 4.1) e formas de revolução

O balanceamento entre qualidade visual e desempenho é crítico, especialmente considerando que:

- A complexidade de vértices varia quadraticamente com subdivisões em esferas
- Torus possuem complexidade $O(m \times n)$ para m segmentos principais e n tubulares
- Formas poliédricas (cubo, pirâmide) mantêm complexidade constante

A escolha de algoritmos adequados impacta diretamente na eficiência:

- Subdivisão recursiva para suavização de superfícies
- Amostragem angular paramétrica para formas de revolução
- Modelagem explicita para poliedros regulares

A implementação demostra que diferentes estrategias são necessárias para diferentes categorias de formas, sempre buscando manter:

- Interface de parametrização consistente
- Controle preciso do nível de detalhe
- Eficiência na alocação de recursos

APÊNDICE B – Tabelas do Capítulo 5 - Resultados e Discussões

Neste anexo, estão apresentadas as tabelas correspondentes aos resultados obtidos nos *benchmarks* realizados entre os pipelines A e B (2.5.1), executados 10 vezes cada. Essas tabelas contêm informações detalhadas sobre as métricas analisadas, como tempo total, FPS médio, tempo médio por quadro, menor tempo e maior tempo, entre outras.

As informações aqui dispostas servem como complemento às discussões apresentadas no Capítulo 5, permitindo uma análise mais aprofundada e detalhada dos dados. Devido ao volume e ao nível de detalhamento, as tabelas foram alocadas neste anexo para evitar interrupções na fluidez do texto principal.

APÊNDICE C – Tabelas do Capítulo 5 - Resultados e Discussões

Neste anexo, estão apresentadas as tabelas correspondentes aos *benchmarks* realizados para cada modelo de iluminação: Flat, Gouraud e Phong. Essas tabelas contêm informações detalhadas sobre as métricas analisadas, como tempo total, FPS médio, tempo médio por quadro, menor tempo e maior tempo.

As tabelas foram agrupadas por modelo de iluminação para facilitar a comparação e a análise dos resultados. Cada modelo foi submetido a 20 execuções, e os dados correspondentes são apresentados a seguir.

C.1 Organização dos Dados

Os dados estão organizados da seguinte forma:

- **Flat:** Tabelas contendo os *benchmarks* para o modelo de iluminação constante.
- **Gouraud:** Tabelas contendo os *benchmarks* para o modelo de iluminação Gouraud.
- **Phong:** Tabelas contendo os *benchmarks* para o modelo de iluminação Phong.

C.2 Tabelas de benchmark

A seguir, encontram-se as tabelas divididas por pipeline:

C.2.1 Tabelas Sombreamento Flat (Constante)

Tabelas das estatísticas dos 20 *benchmarks* executados para o modelos de iluminação constante, e média de tempo dos 10% piores quadros para cada *benchmark*.

C.2.2 Tabelas Sombreamento Gouraud

Tabelas das estatísticas dos 20 *benchmarks* executados para o modelos de iluminação Gouraud, e média de tempo dos 10% piores quadros para cada *benchmark*.

Tabela 13 – Estatísticas Modelo de Iluminação Constante - Pipeline A

Exec.	Tempo Total	Frames	FPS Médio	Média Tempo/Quadro	Menor Tempo	Maior Tempo
1	28.1566	709	25.1451	0.0406	0.0062	0.5960
2	32.5450	709	21.7852	0.0865	0.0084	0.6711
3	32.9016	709	21.5491	0.0934	0.0081	0.7900
4	33.4126	709	21.2195	0.0945	0.0089	0.7293
5	32.9867	709	21.4935	0.0947	0.0081	0.7931
6	33.1473	709	21.3894	0.0943	0.0082	0.7612
7	34.1031	709	20.7899	0.0958	0.0075	0.7369
8	33.8996	709	20.9147	0.0970	0.0083	0.7824
9	33.1959	709	21.3581	0.0955	0.0091	0.6978
10	34.6139	709	20.4831	0.0967	0.0080	0.7740
11	33.7912	709	20.9818	0.0974	0.0074	0.6971
12	39.1768	709	18.0974	0.1040	0.0083	0.7943
13	33.3061	709	21.2874	0.1030	0.0079	0.5764
14	34.1040	709	20.7893	0.0959	0.0084	0.6123
15	33.4936	709	21.1683	0.0962	0.0076	0.6779
16	33.2017	709	21.3543	0.0950	0.0084	0.7710
17	33.9626	709	20.8759	0.0958	0.0082	0.7712
18	34.1586	709	20.7561	0.0971	0.0075	0.8293
19	34.1039	709	20.7894	0.0972	0.0078	0.6655
20	33.7978	709	20.9777	0.0967	0.0085	0.7038

Nota: O tempo e das colunas *Tempo total*; *Media Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 14 – Estatísticas Modelo de Iluminação Constante - Pipeline B

Exec.	Tempo Total	Frames	FPS Médio	Média Tempo/Quadro	Menor Tempo	Maior Tempo
1	43.7579	709	16.2028	0.1244	0.0216	0.7210
2	43.2879	709	16.3787	0.1266	0.0210	0.7723
3	42.4693	709	16.6944	0.1219	0.0202	0.7513
4	41.4098	709	17.1216	0.1192	0.0204	0.6986
5	41.7612	709	16.9775	0.1182	0.0197	0.6814
6	42.0140	709	16.8753	0.1191	0.0237	0.6800
7	42.2847	709	16.7673	0.1197	0.0203	0.6480
8	41.9392	709	16.9054	0.1197	0.0193	0.6745
9	41.5478	709	17.0647	0.1189	0.0195	0.8731
10	41.7013	709	17.0019	0.1183	0.0197	0.6863
11	41.6307	709	17.0307	0.1183	0.0203	0.6070
12	41.7518	709	16.9813	0.1184	0.0205	0.6012
13	41.7209	709	16.9939	0.1185	0.0203	0.6022
14	42.6147	709	16.6375	0.1198	0.0208	0.6299
15	41.8729	709	16.9322	0.1200	0.0196	0.5966
16	41.7605	709	16.9778	0.1187	0.0205	0.5912
17	41.8908	709	16.9250	0.1188	0.0201	0.5998
18	41.8101	709	16.9576	0.1189	0.0210	0.6105
19	41.9902	709	16.8849	0.1190	0.0202	0.5944
20	42.1094	709	16.8371	0.1194	0.0208	0.5899

Nota: O tempo e das colunas *Tempo total*; *Média Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 15 – Media - 10% piores frames - Iluminação Constante - Pipeline A

Execuções	**Media**	**Desvio Padrão**
1	0.0996	0.0723
2	0.1072	0.0773
3	0.1074	0.0894
4	0.1076	0.0828
5	0.1071	0.0903
6	0.1063	0.0867
7	0.1100	0.0825
8	0.1097	0.0883
9	0.1064	0.0799
10	0.1137	0.0884
11	0.1087	0.0796
12	0.1261	0.0874
13	0.1067	0.0661
14	0.1076	0.0700
15	0.1076	0.0771
16	0.1081	0.0879
17	0.1108	0.0873
18	0.1107	0.0949
19	0.1082	0.0759
20	0.1070	0.0799

Tabela 16 – Média - 10% piores frames - Iluminação Constante - Pipeline B

Execuções	**Media**	**Desvio Padrão**
1	0.1451	0.1157
2	0.1351	0.1139
3	0.1375	0.1378
4	0.1280	0.1048
5	0.1287	0.1081
6	0.1306	0.1113
7	0.1326	0.1070
8	0.1296	0.1103
9	0.1294	0.1176
10	0.1281	0.1032
11	0.1257	0.0955
12	0.1264	0.0962
13	0.1261	0.0961
14	0.1294	0.0992
15	0.1269	0.0966
16	0.1256	0.0952
17	0.1260	0.0974
18	0.1267	0.0972
19	0.1265	0.0968
20	0.1304	0.1059

Tabela 17 – Estatísticas Modelo de Iluminação Gouraud - Pipeline A

Exec.	Tempo Total	Frames	FPS Médio	Media Tempo/Quadro	Menor Tempo	Maior Tempo
1	30.2139	709	23.4330	0.0426	0.0066	0.5727
2	32.1508	709	22.0524	0.0887	0.0070	0.5503
3	32.6830	709	21.6932	0.0922	0.0071	0.5514
4	32.8556	709	21.5793	0.0931	0.0075	0.5383
5	33.0572	709	21.4476	0.0937	0.0070	0.5447
6	33.3035	709	21.2890	0.0943	0.0066	0.5486
7	33.2896	709	21.2979	0.0947	0.0072	0.5503
8	35.8654	709	19.7684	0.0982	0.0073	0.5280
9	35.0076	709	20.2527	0.1007	0.0091	0.5744
10	35.0405	709	20.2337	0.0996	0.0077	0.6335
11	35.2583	709	20.1088	0.1000	0.0079	0.6354
12	35.7055	709	19.8569	0.1010	0.0093	0.7057
13	35.2742	709	20.0997	0.1010	0.0092	0.6859
14	35.7637	709	19.8245	0.1011	0.0092	0.7010
15	36.5140	709	19.4172	0.1029	0.0080	0.7320
16	35.6511	709	19.8872	0.1027	0.0077	0.7176
17	35.4161	709	20.0192	0.1014	0.0076	0.8398
18	36.5902	709	19.3767	0.1025	0.0091	0.7385
19	35.3891	709	20.0344	0.1026	0.0080	0.7781
20	35.2709	709	20.1015	0.1006	0.0077	0.7452

Nota: O tempo e das colunas *Tempo total*; *Média Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 18 – Estatísticas Modelo de Iluminação Gouraud - Pipeline B

Exec.	Tempo Total	Frames	FPS Médio	Média Tempo/Quadro	Menor Tempo	Maior Tempo
1	56.1099	709	12.6181	0.0800	0.0177	0.6496
2	53.5717	709	13.2346	0.1555	0.0166	0.6177
3	52.6460	709	13.4673	0.1506	0.0181	0.6235
4	52.1054	709	13.6070	0.1486	0.0183	0.6738
5	53.0419	709	13.3668	0.1492	0.0175	0.6911
6	53.3235	709	13.2962	0.1508	0.0174	0.6062
7	52.5269	709	13.4979	0.1502	0.0174	0.6509
8	52.5496	709	13.4920	0.1491	0.0174	0.6752
9	52.5360	709	13.4955	0.1491	0.0165	0.6615
10	52.6150	709	13.4752	0.1492	0.0165	0.6464
11	52.4357	709	13.5213	0.1491	0.0171	0.6742
12	52.8409	709	13.4176	0.1493	0.0179	0.5993
13	52.6515	709	13.4659	0.1496	0.0175	0.6034
14	53.1738	709	13.3336	0.1501	0.0175	0.6039
15	52.9296	709	13.3952	0.1505	0.0174	0.6120
16	53.1059	709	13.3507	0.1504	0.0175	0.6010
17	53.0290	709	13.3700	0.1505	0.0177	0.6028
18	53.4421	709	13.2667	0.1510	0.0166	0.5980
19	53.4330	709	13.2689	0.1516	0.0175	0.6162
20	52.9978	709	13.3779	0.1509	0.0169	0.6056

Nota: O tempo é das colunas *Tempo total*; *Média Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 19 – Media - 10% piores frames - Iluminação Gouraud - Pipeline A

Execuções	**Media**	**Desvio Padrão**
1	0.0997	0.0656
2	0.0996	0.0619
3	0.1008	0.0616
4	0.1006	0.0599
5	0.1017	0.0609
6	0.1031	0.0606
7	0.1021	0.0610
8	0.1144	0.0631
9	0.1088	0.0636
10	0.1084	0.0702
11	0.1078	0.0700
12	0.1100	0.0786
13	0.1087	0.0755
14	0.1109	0.0774
15	0.1131	0.0801
16	0.1099	0.0792
17	0.1116	0.0929
18	0.1153	0.0813
19	0.1096	0.0862
20	0.1100	0.0819

Tabela 20 – Media - 10% piores frames - Iluminação Gouraud - Pipeline B

Execuções	**Média**	**Desvio Padrão**
1	0.1513	0.0952
2	0.1404	0.0909
3	0.1373	0.0899
4	0.1369	0.0927
5	0.1404	0.0955
6	0.1379	0.0851
7	0.1379	0.0900
8	0.1372	0.0924
9	0.1370	0.0902
10	0.1381	0.0907
11	0.1373	0.0918
12	0.1370	0.0855
13	0.1363	0.0845
14	0.1384	0.0880
15	0.1377	0.0864
16	0.1378	0.0848
17	0.1369	0.0842
18	0.1380	0.0836
19	0.1394	0.0875
20	0.1371	0.0856

C.2.3 Tabelas modelo de iluminação Phong

Tabelas das estatísticas dos 20 *benchmarks* executados para o modelos de iluminação phong, e média de tempo dos 10% piores quadros para cada *benchmark*.

Tabela 21 – Estatísticas Modelo de Iluminaçao Phong - Pipeline A

Exec.	Tempo Total	Frames	FPS Médio	Média Tempo/Quadro	Menor Tempo	Maior Tempo
1	41.2096	709	17.1805	0.1262	0.0079	0.5748
2	41.9474	709	16.9021	0.1170	0.0071	0.5590
3	42.6369	709	16.6288	0.1200	0.0074	0.5366
4	42.8439	709	16.5484	0.1212	0.0051	0.5262
5	42.8618	709	16.5416	0.1216	0.0079	0.5325
6	43.0844	709	16.4561	0.1219	0.0078	0.5364
7	43.2393	709	16.3971	0.1224	0.0080	0.5375
8	43.1349	709	16.4368	0.1225	0.0073	0.5394
9	43.1003	709	16.4500	0.1223	0.0072	0.5317
10	43.0230	709	16.4796	0.1221	0.0071	0.5190
11	43.3540	709	16.3537	0.1225	0.0076	0.5327
12	43.2639	709	16.3878	0.1230	0.0071	0.6287
13	43.4855	709	16.3043	0.1232	0.0078	0.6293
14	43.5757	709	16.2705	0.1237	0.0079	0.6630
15	44.1207	709	16.0695	0.1246	0.0088	0.6708
16	44.0161	709	16.1077	0.1252	0.0084	0.6542
17	43.9970	709	16.1147	0.1250	0.0083	0.6425
18	44.1908	709	16.0441	0.1252	0.0090	0.6219
19	44.1709	709	16.0513	0.1255	0.0090	0.6547
20	43.8103	709	16.1834	0.1249	0.0095	0.6330

Nota: O tempo é das colunas *Tempo total*; *Média Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 22 – Estatísticas Modelo de Iluminação Phong - Pipeline B

Exec.	Tempo Total	Frames	FPS Médio	Media Tempo/Quadro	Menor Tempo	Maior Tempo
1	72.2218	709	9.8031	0.1027	0.0254	0.7111
2	69.1998	709	10.2457	0.2002	0.0238	0.6896
3	67.9940	709	10.4274	0.1943	0.0239	0.6992
4	68.5453	709	10.3435	0.1933	0.0243	0.6803
5	68.4684	709	10.3551	0.1940	0.0247	0.6770
6	68.0079	709	10.4253	0.1932	0.0240	0.6725
7	68.4728	709	10.3545	0.1932	0.0255	0.7228
8	68.3325	709	10.3757	0.1938	0.0241	0.6790
9	68.1719	709	10.4002	0.1933	0.0238	0.6646
10	67.8215	709	10.4539	0.1926	0.0242	0.6720
11	67.9549	709	10.4334	0.1923	0.0249	0.6602
12	68.1052	709	10.4104	0.1927	0.0238	0.6759
13	68.0442	709	10.4197	0.1929	0.0247	0.6816
14	68.1785	709	10.3992	0.1930	0.0237	0.6832
15	68.2222	709	10.3925	0.1933	0.0239	0.6786
16	68.3330	709	10.3757	0.1934	0.0242	0.6784
17	68.0556	709	10.4180	0.1931	0.0236	0.6734
18	68.3098	709	10.3792	0.1931	0.0241	0.6756
19	68.3731	709	10.3696	0.1936	0.0248	0.6719
20	68.1066	709	10.4102	0.1933	0.0242	0.6748

Nota: O tempo é das colunas *Tempo total*; *Média Tempo/Quadros*; *Menor/Maior Tempo* são dados em segundos.

Tabela 23 – Media - 10% piores frames - Iluminação Phong - Pipeline A

Execuções	**Media**	**Desvio Padrão**
1	0.1519	0.0830
2	0.1429	0.0758
3	0.1447	0.0748
4	0.1450	0.0725
5	0.1447	0.0732
6	0.1462	0.0746
7	0.1465	0.0738
8	0.1464	0.0733
9	0.1462	0.0721
10	0.1455	0.0725
11	0.1470	0.0738
12	0.1485	0.0814
13	0.1481	0.0804
14	0.1496	0.0843
15	0.1497	0.0847
16	0.1490	0.0824
17	0.1485	0.0820
18	0.1482	0.0800
19	0.1493	0.0822
20	0.1470	0.0815

Tabela 24 – Media - 10% piores frames - Iluminação Phong - Pipeline B

Execuções	**Média**	**Desvio Padrão**
1	0.2433	0.1417
2	0.2335	0.1363
3	0.2304	0.1366
4	0.2316	0.1351
5	0.2316	0.1343
6	0.2299	0.1321
7	0.2369	0.1466
8	0.2313	0.1364
9	0.2300	0.1318
10	0.2289	0.1314
11	0.2285	0.1303
12	0.2300	0.1334
13	0.2308	0.1369
14	0.2310	0.1341
15	0.2318	0.1378
16	0.2308	0.1334
17	0.2295	0.1318
18	0.2301	0.1321
19	0.2314	0.1342
20	0.2295	0.1316

C.2.4 Anova - Pipelines x Método de Iluminação

Resultados da ANOVA de fator duplo com repetição, onde os fatores analisados são os modelos de iluminação e modelos de *pipeline*.

Tabela 25 – Resultados detalhados da ANOVA de dois fatores com repetição.

Fonte	**SQ**	**gl**	**QM**	**F**	**valor-p**	**F critico**
Método	6740.2014	2	3370.1007	2317.1591	4.8×10^{-93}	3.0759
Pipeline	9080.7835	1	9080.7835	6243.6178	2.1×10^{-101}	3.9243
Interação	1418.8484	2	709.4242	487.7744	1.32×10^{-56}	3.0759
Erro	165.8028	114	1.4544			

Tabela 26 – Estatísticas descritivas por método e pipeline.

Método	**Pipeline**	**Contagem**	**Soma**	**Média**	**Variância**	**Total**	**Media Total**
CONSTANTE	A (Adair)	20	672.0586	33.6029	3.4614	40	37.8346
	B (Smith)	20	841.3251	42.0663	0.3396		
GOURAUD	A (Adair)	20	690.3002	34.5150	2.8148	40	43.7841
	B (Smith)	20	1061.0652	53.0533	0.6663		
PHONG	A (Adair)	20	865.0664	43.2533	0.5678	40	55.8500
	B (Smith)	20	1368.9190	68.4460	0.8766		

C.2.5 Tabelas Estatísticas Comparativas

Estas tabelas apresentam as medias das 10 execuções das Tabelas 13, 14, 17, 18, 21 e 22

Tabela 27 – Estatísticas comparativas - Método FLAT

Iluminação	**Pipeline**	**Métrica**	**Valores**		
			Média	Desvio Padrão	IC (95%)
CONSTANTE	Smith	Tempo Total (s)	42.0663	0.5680	0.2658
		FPS Méedio	16.8574	0.2226	0.1042
		Tempo/Quadro (s)	0.1198	0.0021	0.001
	Adair	Tempo Total (s)	33.6029	1.8134	0.8487
		FPS Médio	21.1603	1.1684	0.5468
		Tempo/Quadro (s)	0.0934	0.01256	0.0059

Tabela 28 – Estatísticas comparativas - Método GOURAUD

Iluminação	**Pipeline**	**Métrica**	**Valores**		
			Média	Desvio Padrão	IC (95%)
GOURAUD	Smith	Tempo Total (s)	53.0533	0.7956	0.3723
		FPS Médio	13.3659	0.1962	0.0918
		Tempo/Quadro (ms)	0.1468	0.0154	0.0072
	Adair	Tempo Total (s)	34.5150	1.6353	0.7653
		FPS Médio	20.5887	1.0215	0.4781
		Tempo/Quadro (ms)	0.0957	0.0128	0.0060

Tabela 29 – Estatísticas comparativas - Método PHONG

Iluminação	**Pipeline**	**Métrica**	**Valores**		
			Média	Desvio Padrão	IC (95%)
PHONG	Smith	Tempo Total (s)	68.4459	0.9125	0.4271
		FPS Médio	10.3596	0.1348	0.0631
		Tempo/Quadro (s)	0.1891	0.0199	0.0093
	Adair	Tempo Total (s)	43.2533	0.7345	0.3437
		FPS Médio	16.3954	0.2800	0.1310
		Tempo/Quadro (s)	0.1230	0.0021	0.0010

Anexos

ANEXO A – Fundamentos de Geometria Analítica e Álgebra Linear

A.1 Vetores

A.1.1 Noção intuitiva de vetores

Exitem grandezas, chamadas escalares, que são caracterizadas por um número (e a unidade correspondente):. 50cm^2 de área, 4m de comprimento, 7kg de massa. Outras no entanto requerem mais do que isso. Por exemplo, para caracterizarmos uma força ou uma velocidade precisamos dar a direção, a intensidade (ou módulo) e o sentido:

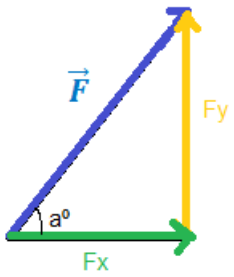


Figura 54 – Exemplo de vetores

Tais grandezas são chamadas vetoriais. Na figura 54 acima as flechas nos dão a ideia exata das grandezas mencionadas. Onde o vetor $\vec{F}$ é a soma dos vetores $\vec{F}x$ e $\vec{F}y$.

A.1.2 Definição Formal de Vetores

Um vetor pode ser definido como o par ordenado de dois pontos (a, b) no espaço, onde a é o ponto de origem e b é o ponto de extremidade do segmento orientado. Quando os dois pontos coincidem, ou seja, (a, a), o vetor resultante é chamado de vetor nulo, pois não possui magnitude nem direção. Observe que, se $a \neq b$, o vetor (a, b) é diferente do vetor (b, a), pois a ordem dos pontos determina a direção do vetor (BOULOS; CAMARGO, 2004).

A.1.3 Operações com Vetores

A.1.3.1 Soma

A adição de dois vetores resulta em um novo vetor, denominado vetor soma. Para dois vetores, $\vec{A}$ e $\vec{B}$, a soma pode ser realizada utilizando a regra do paralelogramo ou a regra do triângulo (BOULOS; CAMARGO, 2004).

Na **regra do paralelogramo**, os vetores $\vec{A}$ e $\vec{B}$ são dispostos a partir de uma mesma origem. Em seguida, desenha-se um paralelogramo onde $\vec{A}$ e $\vec{B}$ são lados adjacentes. O vetor resultante $\vec{C} = \vec{A} + \vec{B}$ é representado pela diagonal do paralelogramo que parte da origem comum dos vetores.

Na **regra do triângulo**, o vetor $\vec{B}$ é posicionado a partir da extremidade do vetor $\vec{A}$. O vetor resultante $\vec{C} = \vec{A} + \vec{B}$ conecta a origem de $\vec{A}$ à extremidade de $\vec{B}$, formando um triângulo.

A soma de dois vetores $\vec{A}$ e $\vec{B}$ pode ser expressa pela equação 50:

$$\vec{C} = \vec{A} + \vec{B} = (A_x + B_x, A_y + B_y) \tag{50}$$

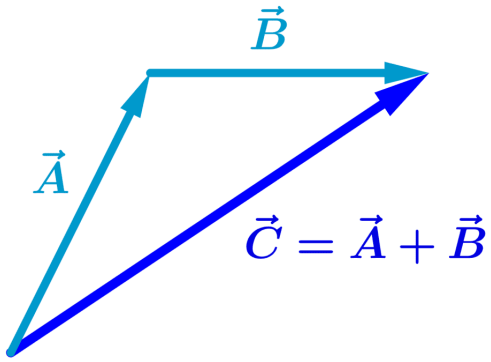


Figura 55 – Representação gráfica da soma de dois vetores $\vec{A}$ e $\vec{B}$ resultando em $\vec{C} = \vec{A} + \vec{B}$

A figura 55 ilustra a soma de dois vetores usando a regra do paralelogramo. O vetor $\vec{C}$, representado pela diagonal do paralelogramo, é o vetor resultante que corresponde à soma de $\vec{A}$ e $\vec{B}$. Esse método gráfico permite visualizar a direção e o sentido do vetor resultante, sendo uma abordagem útil para operações vetoriais em espaços bidimensionais.

A.1.3.2 Subtração

A subtração de dois vetores resulta em um novo vetor, onde o segundo vetor é subtraído do primeiro. Para dois vetores, $\vec{A}$ e $\vec{B}$, a operação de subtração pode ser representada visualmente invertendo a direção de $\vec{B}$ e, em seguida, somando-o a $\vec{A}$, como mostrado na figura 56. Essa operação é semelhante à soma de vetores, diferenciando-se apenas pelo sinal utilizado.

A subtração entre dois vetores $\vec{A}$ e $\vec{B}$ pode ser expressa pela equação 51:

$$\vec{C} = \vec{A} - \vec{B} = (A_x - B_x, A_y - B_y) \tag{51}$$

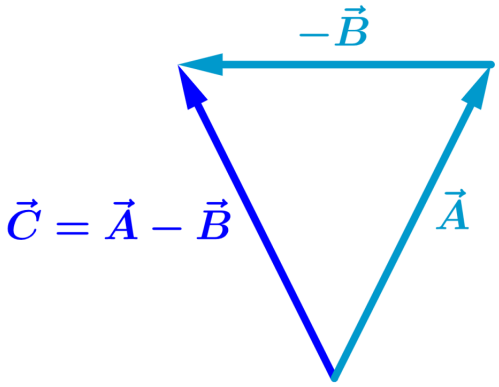


Figura 56 – Representação gráfica da subtração de dois vetores $\vec{A}$ e $\vec{B}$, resultando em $\vec{C} = \vec{A} - \vec{B}$

Na figura 56, a subtração é realizada invertendo a direção do vetor $\vec{B}$ para $-\vec{B}$ e, em seguida, aplicando a regra do triângulo ou do paralelogramo para obter o vetor resultante $\vec{C}$. Esse método visualiza a subtração como uma soma do vetor $\vec{A}$ com o vetor inverso de $\vec{B}$, o que permite compreender a direção e o sentido do vetor resultante de maneira intuitiva.

A.1.3.3 Multiplicação

Igualmente da mesma maneira que no anexo A.1.3.2 a multiplicação de vetores segue a mesma lógica da adição (A.1.3.1), diferenciando-se apenas no operador utilizado.

$$\vec{C} = \vec{A} \cdot \vec{B} = (A_x \cdot B_x, A_y \cdot B_y)$$

Equação 52 – Multiplicação de dois vetores $\vec{A}$ e $\vec{B}$

A.1.3.4 Multiplicação de número real por vetor

A operação de multiplicação de um número real α por um vetor $\vec{v}$, representada por $\alpha\vec{v}$, define uma nova escala e direção para o vetor (BOULOS; CAMARGO, 2004). Essa operação é caracterizada pelas seguintes propriedades:

Se $\alpha = 0$ ou $\vec{v} = \vec{0}$, então $\alpha\vec{v} = \vec{0}$ (por definição).

Se $\alpha \neq 0$ e $\vec{v} \neq \vec{0}$, o vetor resultante $\alpha\vec{v}$ possui as seguintes características:

a) $\alpha\vec{v}$ é paralelo ao vetor $\vec{v}$, ou seja, $\alpha\vec{v} \parallel \vec{v}$.

b) $\alpha\vec{v}$ tem o mesmo sentido que $\vec{v}$ se $\alpha > 0$ e sentido oposto se $\alpha < 0$.

c) A norma do vetor $\alpha\vec{v}$ é dada por $||\alpha\vec{v}|| = |\alpha|\,||\vec{v}||$, indicando que o vetor resultante é escalado pela magnitude de α.

A figura 57 ilustra o efeito da multiplicação de um número real positivo no vetor $\vec{A}$, ampliando sua magnitude em duas vezes, resultando no vetor $2\vec{A}$, que é paralelo e mantém o mesmo sentido de $\vec{A}$.

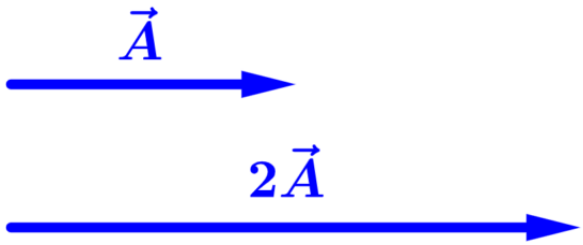


Figura 57 – Representação gráfica da multiplicação de um vetor $\vec{A}$ por um número real, resultando em $2\vec{A}$

A.1.4 Produto Escalar

chama-se produto escalar (ou produto interno usual) de dois vetores $\vec{u} = x_1\vec{i} + y_1\vec{j} + z_1\vec{k}$ e $\vec{v} = x_2\vec{i} + y_2\vec{j} + z_2\vec{k}$ e se representa por $\vec{u}$ e $\vec{v}$ ao número real

$$\vec{u} \cdot \vec{v} = x_1x_2 + y_1y_2 + z_1z_2 \tag{53}$$

A.1.5 Modulo vetorial

O módulo de um vetor $\vec{u}$ é representado por $|\vec{u}|$ e corresponde, do ponto de vista geométrico, ao seu comprimento (STEIMBRUCH; WINTERLE, 1987). Sendo sempre um número real positivo.

$$|\vec{u}| = \sqrt{\vec{u} \cdot \vec{u}} \tag{54}$$

$$|\vec{u}| = \sqrt{x^2 + y^2 + z^2} \tag{55}$$

A.1.6 Angulo entre dois vetores

Considerando os vetores não nulos $\vec{u}$ e $\vec{v}$. Tomemos um ponto $O \in E^3$, e sejam $P, Q \in E^3$ tais que $\vec{u} = \vec{OP}, \vec{v} = \vec{OQ}$. Seja θ a medido em radianos (graus) do angulo POQ satisfazendo $0 \leq \theta \leq \pi$ ($0° \leq \theta \leq 180°$).

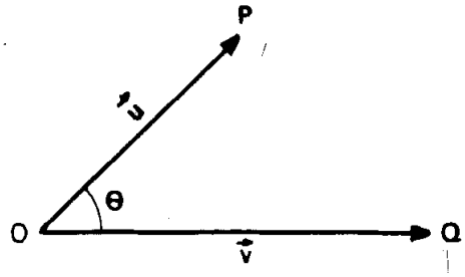


Figura 58 – Produto escalar
(BOULOS; CAMARGO, 2004)

O número θ se chama medida em radianos (graus) do ângulo entre $\vec{u}$ e $\vec{v}$. Procuremos agora uma expressão que nos forneça $\vec{\theta}$ em termos de $\vec{u}$ e $\vec{v}$. Para isso, fixemos uma base ortonormal $(\vec{i}, \vec{j}, \vec{k})$ e sejam $\vec{u} = (x_1, y_1, z_1)$ e $\vec{v} = (x_2, y_2, z_2)$. Sendo a base ortonormal de qualquer vetor poder ser calculada utilizando a Equação 54 (BOULOS; CAMARGO, 2004)

Aplicando a lei dos cossenos ao triângulo **POQ** resulta em:

$$(1)\ |\vec{QP}|^2 = |\vec{u}|^2 + |\vec{v}|^2 - 2 \cdot |\vec{u}||\vec{v}| \cos\theta$$

Logo simplificando temos que:

$$(2)\ \vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}| \cos\theta$$

Observações:

a) Se $\vec{u} \cdot \vec{v} > 0$, de acordo com a formula (2), $\cos\theta$ deve ser um número positivo, isto é, $\cos\theta > 0$, o que implica que $0° \leq \theta < 90°$ (STEIMBRUCH; WINTERLE, 1987), neste caso θ é um angulo agudo ou nulo, demonstrado na Figura 58

b) Se $\vec{u} \cdot \vec{v} < 0$, de acordo com a formula (2), $\cos\theta$ deve ser um número negativo, isto é $\cos\theta < 0$, o que implica que $90° < \theta \leq 180°$, neste caso é um angulo obtuso ou raso (STEIMBRUCH; WINTERLE, 1987), demonstrado na figura abaixo:

A.2 Bases vetoriais

Chama-se base V^3 a qualquer tripa ordenada $E = (\vec{e_1}, \vec{e_2}, \vec{e_3})$ linearmente independe de vetores de V^3. Se $(\vec{e_1}, \vec{e_2}, \vec{e_3})$ é uma base V^3, isto é para todo $\vec{v} \in V^3$, existem escalares a_1, a_2, a_3, tais que $\vec{v} = a_1\vec{e_1} + a_2\vec{e_2} + a_3\vec{e_3}$.

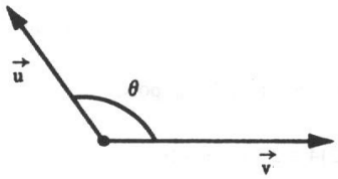


Figura 59 – Angulo obtuso
(STEIMBRUCH; WINTERLE, 1987)

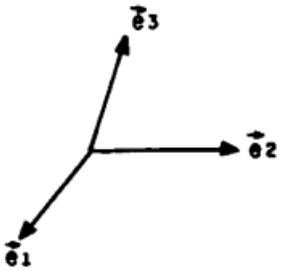


Figura 60 – Base de um $E \in V^3$
(BOULOS; CAMARGO, 2004)

Sabemos também que essa tripla (a_1, a_2, a_3) de escalares é única. A conclusão é que, escolhida uma base de $E \in V^3$, fica associada univocamente a cada vetor $\vec{v}$ uma tripla de escalares (a_1, a_2, a_3). Essa tripla é denominada *tripla de coordenadas do vetor $\vec{v}$ em relação a base E*. Observe que é importante a ordem dos escalares; trata-se de uma tripla ordenada (BOULOS; CAMARGO, 2004).

Se, por abuso de linguagem, dissermos que as coordenadas estão nessa ordem $\vec{v} = a_1\vec{e_1} + a_2\vec{e_2} + a_3\vec{e_3}$.

A notação utilizada para indicar que a_1, a_2, a_3 são as coordenadas (nessa ordem!) do vetor $\vec{v}$ em relação à base E é

$$\vec{v} = (a_1, a_2, a_3)_E \tag{56}$$

e se não houver perigo de dúvida quanto à base escolhida, omite-se o índice "E":

$$\vec{v} = (a_1, a_2, a_3) \tag{57}$$

Em outros termos , (1) e (2) são simplesmente "abreviaturas"de $\vec{v} = a_1\vec{e_1} + a_2\vec{e_2} + a_3\vec{e_3}$ (BOULOS; CAMARGO, 2004).

A.3 Mudança de base

A escolha de uma base conveniente ajuda muitas vezes a resolver um problema, tornando-o mais simples. Acontece que os vetores dados podem já estar referidos a uma certa base, digamos $E = (\vec{e_1}, \vec{e_1}, \vec{e_1})$. Introduzindo-se a base conveniente que supostamente vai ajudar-nos, seja ela $F = (\vec{f_1}, \vec{f_1}, \vec{f_1})$, precisamos saber a relação entre as duas, para que trabalhando coma solução em termos de F, possamos no final passar para a base inicial E.

Podemos expressar de modo único cada elemento de D em termos da base E, conforme já sabemos. Escrevamos então:

$$\begin{aligned} \vec{f}1 &= a_{11}\vec{e_1} + a_{21}\vec{e_2} + a_{31}\vec{e_3} \\ \vec{f}2 &= a_{21}\vec{e_1} + a_{22}\vec{e_2} + a_{32}\vec{e_3} \\ \vec{f}3 &= a_{31}\vec{e_1} + a_{23}\vec{e_2} + a_{33}\vec{e_3} \end{aligned} \tag{58}$$

onde os a_{ij} são números reais.

O Próximo passo agora é resolver o seguinte problema. É dado

$$\vec{v} = x_1\vec{e_1} + x_2\vec{e_2} + x_3\vec{e_3} = (x_1, x_2, x_3)_E \tag{59}$$

onde agora o índice E é necessário, pois podemos escrever também

$$\vec{v} = y_1\vec{f_1} + y_2\vec{f_2} + y_3\vec{f_3} = (y_1, y_2, y_3)_F \tag{60}$$

Queremos saber qual é a relação entre as coordenadas x_1, x_2, x_3 de $\vec{v}$ em relação à base de E, e as coordenadas y_1, y_2, y_3 do mesmo vetor $\vec{v}$ em relação à base F. A idéia é muito simples para resolver isto. Usando (1) em (3), teremos $\vec{v}$ em função dos elementos de E. Em seguinda é só comparar com (2).

Substituindo $\vec{f_1}, \vec{f_2}, \vec{f_3}$ dados por (1) na relação (3) resulta

$$\begin{aligned} \vec{v} &= y_1(a_{11}\vec{e_1} + a_{21}\vec{e_2} + a_{31}\vec{e_3}) + y_2(a_{12}\vec{e_1} + a_{22}\vec{e_2} + a_{32}\vec{e_3}) + y_3(a_{31}\vec{e_1} + a_{32}\vec{e_2} + a_{33}\vec{e_3}) \\ &= (y_1a_{11} + y_2a_{12} + y_3a_{13})\vec{e_1} + (y_1a_{21} + y_2a_{22} + y_3a_{23})\vec{e_2} + (y_1a_{31} + y_2a_{32} + y_3a_{33})\vec{e_3} \end{aligned}$$

Comparando com (2), e usando o fato que de que um vetor se expressa de modo único como combinação linear dos elementos de uma base, vem

$$\begin{aligned} x_1 &= a_{11}y_1 + a_{12}y_2 + a_{13}y_3 \\ x_2 &= a_{21}y_1 + a_{22}y_2 + a_{23}y_3 \\ x_3 &= a_{31}y_1 + a_{32}y_2 + a33y_3 \end{aligned} \tag{61}$$

Agora basta resolver o sistema. Para simplificar os índices e termos vamos utilizar a notação do anexo A.3.1.

$$\begin{pmatrix} x_1 \\ x_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{5}$$

Vamos dar um nome à matriz 3×3 acima.

Dada as bases $E = (\vec{e_1}, \vec{e_2}, \vec{e_3})$ e $F = (\vec{f_1}, \vec{f_2}, \vec{f_3})$, podemos escrever:

$$\begin{aligned} \vec{f_1} &= a_{11}\vec{e_1} + a_{21}\vec{e_2} + a_{31}\vec{e_3} \\ \vec{f_2} &= a_{12}\vec{e_1} + a_{22}\vec{e_2} + a_{32}\vec{e_3} \\ \vec{f_3} &= a_{13}\vec{e_1} + a_{23}\vec{e_2} + a_{33}\vec{e_3}] \end{aligned} \tag{62}$$

À Matriz

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

dá-se o nome de *matriz de mudança de base E para a base F*. Indica-se, para ressumir assim:

$$\mathbf{E} \xrightarrow{\mathbf{M}} \mathbf{F}$$

Observe que os elementos a_{11}, a_{12}, a_{13} que aparecem na 1º igualdade em (6), devem ficar na 1ª coluna da matriz **M**. Da mesma forma, os elemento da 2ª igualdade, devem ficar na 2ª coluna da matriz **M**. Os da 3ª igualdade na 3ª coluna de **M** (BOULOS; CAMARGO, 2004).

A.3.1 Matrizes

Na álgebra linear, uma matriz é um quadro retangular compostos por números. Um matriz costuma se presentada por uma letra maiúscula, tal como A, e tem um determinado número de

linhas (m) e de colunas (n). Neste caso, representa-se por $\mathbf{A}_{m\times n}$. A notação a_{ij} indica o elemento da matriz A que está na linha i e na coluna j. Assim a Matriz A como notação $A = (a_{ij})$, com $i = 1, 2, ..., m$ e $j = 1, 2, ..., n$ e podemos representar esta matriz da seguinte maneira

$$A = (a_{ij}) = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

A matriz representada tem **m** linhas e **n** colunas. Observe que a utilização de battas para indicar uma matriz não é conveniente, pois elas já são utilizadas para indicar o determinante da matriz, e esses dois conceitos são distintos: *matriz* é uma tabela de números reais, e o *determinante da matriz* é um número real. (ESPINOSA; BISCOLLA; FILHO, 2007)

Observações:

a) Se **m = n**, chamamos de **matriz quadrada de ordem m**. Quando **m ≠ n**, dizemos que é uma **matriz retangular**. (ESPINOSA; BISCOLLA; FILHO, 2007)

b) Dizemos que A é uma **matriz linha** se $m = 1$, $A = (a_{11}, a_{12}, ..., a_{1n})$. (ESPINOSA; BISCOLLA; FILHO, 2007)

c) Dizemos que A é uma **matriz coluna** se $n = 1$,

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

(ESPINOSA; BISCOLLA; FILHO, 2007)

d) Chamamos A de **matriz identidade** se A for quadrada e $A = (a_{ij})$ com

$$a_{ij} = \begin{cases} 1, & \text{se } i = j \\ 0, & \text{se } i \neq j \end{cases}$$

isto é,

$$I_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Utilizaremos a notação I_n para indicar a matriz identidade de ordem n. (ESPINOSA; BISCOLLA; FILHO, 2007)

A.3.2 Operações com matrizes

A.3.2.1 Adição

Sejam $A = (a_{ij})$ e $B = (b_{ij})$ matrizes $\mathbf{m} \times \mathbf{n}$, a **matriz soma** $A + B$ é por (ESPINOSA; BISCOLLA; FILHO, 2007):

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}}_{A} + \underbrace{\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{pmatrix}}_{A+B}$$

Equação 63 – Adição matricial

A.3.2.2 Multiplicação por escalar

Dados uma matriz $A = (a_{ij})$, $\mathbf{m} \times \mathbf{n}$, e um número real α, temos uma nova matriz $\alpha \cdot A$ que também é uma matriz $\mathbf{m} \times \mathbf{n}$ (ESPINOSA; BISCOLLA; FILHO, 2007).

$$\alpha \cdot A = \alpha \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} & \cdots & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & \cdots & \alpha a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha a_{m1} & \alpha a_{m2} & \cdots & \alpha a_{mn} \end{pmatrix}$$

Equação 64 – Multiplicação por matricial por escalar

A.3.2.3 Multiplicação de matrizes

Sejam $A = (a_{ij})$ $\mathbf{m} \times \mathbf{n}$ e $B = (b_{jk})$ matriz $\mathbf{n} \times \mathbf{p}$, a **matriz produto** $A \cdot B$ ou simplesmente AB é a matriz $\mathbf{m} \times \mathbf{p}$, que tem como termo geral

$$c_{ij} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk} = a_{ik} \cdot b_{1k} + \ldots + a_{in} \cdot b_{nk}$$

Isto é, devemos fazer o produto de cada linha da matriz A pelas colunas de B, obtendo assim as linhas da nova matriz (ESPINOSA; BISCOLLA; FILHO, 2007).

A.3.3 Concatenação de Matrizes

A concatenação de matrizes é uma operação fundamental na computação gráfica, especialmente para transformações de objetos em espaços tridimensionais. Em vez de aplicar transformações sucessivas a um objeto (como translação, rotação e escala), podemos combiná-las em uma única matriz composta, o que simplifica e acelera os cálculos durante a renderização.

Para representar transformações 3D, usamos matrizes 4x4, que permitem incluir translações e rotações em um único sistema matricial. Quando aplicamos várias transformações a um objeto, podemos concatená-las multiplicando as respectivas matrizes de transformação. A ordem das operações é crucial, pois a multiplicação de matrizes não é comutativa (ou seja, $AB \neq BA$).

Por exemplo, considere três transformações básicas: uma matriz de escala S, uma matriz de rotação R e uma matriz de translação T. Para aplicar essas transformações a um objeto, começando pela escala, depois a rotação e, por último, a translação, a matriz composta M seria calculada como:

$$M = T \cdot R \cdot S \tag{65}$$

Ao aplicar M a um ponto homogêneo $\vec{P}$, realizamos todas as transformações de uma vez:

$$\vec{P'} = M \cdot \vec{P} = (T \cdot R \cdot S) \cdot \vec{P} \tag{66}$$

Essa abordagem reduz a necessidade de realizar múltiplas operações para cada ponto, uma vez que todas as transformações já estão consolidadas em uma única matriz.

A.3.3.1 Transformações Geométricas com Matrizes 4x4

As transformações geométricas em computação gráfica 3D, como translação, rotação e escala, podem ser representadas por matrizes 4x4 no espaço homogêneo. Utilizando matrizes homogêneas, é possível combinar várias operações de transformação em uma única matriz, o que facilita o processamento e a manipulação de objetos no espaço tridimensional.

Abaixo, são apresentados exemplos de matrizes 4x4 para as operações de translação, rotação em torno dos eixos x, y e z, e escala.

A.3.3.1.1 Matriz de Translação T:

A matriz de translação permite deslocar um objeto ao longo dos eixos x, y, e z por uma distância definida por (t_x, t_y, t_z).

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{67}$$

A.3.3.1.2 Matriz de Rotação em torno do eixo x (R_x, ângulo α):

A rotação em torno do eixo x por um ângulo α altera a orientação do objeto no plano yz, mantendo as coordenadas em x inalteradas.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{68}$$

A.3.3.1.3 Matriz de Rotação em torno do eixo y (R_y, ângulo β):

A rotação em torno do eixo y por um ângulo β modifica as coordenadas no plano xz, enquanto o valor em y permanece constante.

$$R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{69}$$

A.3.3.1.4 Matriz de Rotação em torno do eixo z (R_z, ângulo θ):

A rotação em torno do eixo z por um ângulo θ afeta as coordenadas no plano xy, mantendo o valor em z inalterado.

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{70}$$

A.3.3.1.5 Matriz de Escala S:

A matriz de escala permite alterar o tamanho de um objeto ao longo dos eixos x, y e z, com fatores de escala (s_x, s_y, s_z) para cada eixo respectivamente.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{71}$$

Essas matrizes de transformações geométricas são fundamentais para a manipulação de objetos em gráficos 3D, pois permitem alterar a posição de um vértice, ou realizar transformações de base no mapeamento de coordenadas sistema do universo para o sistema da câmera.

A.3.3.2 Concatenando as Matrizes

Para concatenar as transformações, multiplicamos as matrizes na ordem desejada. Supondo que queremos escalar, depois rotacionar e, por fim, transladar o objeto, calculamos a matriz composta M como:

$$M = T \cdot R \cdot S = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{72}$$

Essa matriz composta M pode ser usada para aplicar todas as transformações desejadas a qualquer ponto $\vec{P}$ da cena, economizando operações e melhorando a eficiência computacional do sistema gráfico. Dessa forma, a concatenação de matrizes permite combinar múltiplas transformações em uma única operação de multiplicação, essencial para a performance em gráficos 3D.

A.4 Estrutura do Arquivo JSON

O arquivo armazena informações da cena no formato JSON com a seguinte estrutura:

```
{
    "scene": {
        "camera": {
            "d": float,          // Distancia focal
```

```
            "far": float,        // Plano de corte distante
            "near": float,       // Plano de corte proximo
            "position": {        // Posicao da camera
                "x": float,
                "y": float,
                "z": float
            },
            "target": {         // Ponto de mira da camera
                "x": float,
                "y": float,
                "z": float
            },
            "up": {             // Vetor up da camera
                "x": float,
                "y": float,
                "z": float
            }
        },
        "illumination": int,    // Modelo de iluminacao (0-2)
        "pipeline": int,        // Pipeline de renderizacao (0-1)
        "max_viewport": {       // Viewport maximo
            "x": float,
            "y": float
        },
        "min_viewport": {      // Viewport minimo
            "x": float,
            "y": float
        },
        "max_window": {         // Janela de projecao maxima
            "x": float,
            "y": float
        },
        "min_window": {         // Janela de projecao minima
            "x": float,
            "y": float
        },
        "objects": [           // Lista de objetos na cena
            {
                "id": string,
                "name": string,
                "num_faces": int,
                "index_vertices": [ // Faces do objeto
```

```
                    [int, int, int], // Indice da lista de vertices
    que compem a face
                    ...
                ],
                "vertices": [       // Vertices do objeto
                    {
                        "id": string,
                        "x": float,
                        "y": float,
                        "z": float,
                        "w": float
                    },
                    ...
                ],
                "material": {       // Propriedades do material
                    "ambient": [float, float, float],
                    "diffuse": [float, float, float],
                    "specular": [float, float, float],
                    "shininess": float
                }
            }
        ]
    }
}
```

Listing A.1 – Exemplo de JSON

A descrição de alguns campos que contem valores fixos pode ser encontrada abaixo.

pipeline:

0: Pipeline A (padrão)

1: Pipeline B

illumination:

0: Modelo Constante (padrão)

1: Modelo Gouraud

2: Modelo Phong

material: Componentes RGBA para:

ambient: Cor ambiente

diffuse: Cor difusa

specular: Cor especular

shininess: Expoente de brilho