UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
CURSO DE MESTRADO EM ENGENHARIA ELÉTRICA

DISSERTAÇÃO DE MESTRADO

SIMULAÇÃO GEOMÉTRICA E INTERATIVA DE USINAGEM NC USANDO CAMPOS DE DISTÂNCIA

ALLAN YOSHIO HASEGAWA

JOINVILLE, 2015

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM ENGENHARIA ELÉTRICA

ALLAN YOSHIO HASEGAWA

SIMULAÇÃO GEOMÉTRICA E INTERATIVA DE USINAGEM NC USANDO CAMPOS DE DISTÂNCIA

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Engenharia Elétrica.

Orientador:
Prof. Roberto Silvio Ubertino Rosso Jr., Ph.D.

JOINVILLE

2015

H346s Hasegawa, Allan Yoshio
Simulação geométrica e interativa de usinagem NC usando campos de distância / Allan Yoshio Hasegawa. – 2015.
171 p. : il. ; 21 cm

Orientador: Roberto Silvio Ubertino Rosso Junior
Bibliografia: 163-171
Dissertação (mestrado) – Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Engenharia Elétrica, Joinville, 2015.

1. Simulação de usinagem. 2. Representações geométricas. 3. Interativa. 4. Campos de distância. 5. Ray casting. I. Rosso Junior, Roberto Silvio Ubertino. II. Universidade do Estado de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica, III. Título.

CDD 621.3 – 23. ed.

ALLAN YOSHIO HASEGAWA

SIMULAÇÃO GEOMÉTRICA E INTERATIVA DE USINAGEM NC USANDO CAMPOS DE DISTÂNCIA

Dissertação apresentada ao Curso de Mestrado Profissional em Engenharia Elétrica como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração "Automação de Sistemas".

Banca Examinadora

Orientador:

Prof. Dr. Roberto Silvio Ubertino Rosso Junior
CCT/UDESC

Membros

Prof. Dr. Marcos de Sales Guerra Tsuzuki
Escola Politécnica/USP

Prof. Dr. André Tavares da Silva
CCT/UDESC

Joinville,SC, 18 de setembro de 2015.

AGRADECIMENTOS

Primeiramente eu gostaria de agradecer a minha família, pois sem ela eu não teria condições de me dedicar aos meus estudos.

Gostaria de agradecer a UDESC pelo financiamento deste trabalho com o Programa de Bolsas de Monitoria de Pós-Graduação (PROMOP).

Gostaria de agradecer o meu professor de engenharia de software Edson Murakami, que além de ensinar a teoria também passou a experiência dele na indústria. Por causa desse conhecimento foi possível manter o software desenvolvido neste trabalho organizado e de fácil manutenção, mesmo após ele ter tido um crescimento rápido.

Gostaria de agradecer o meu professor de estruturas de dados, linguagem de programação e processamento de imagens Alexandre Gonçalves Silva. O professor Alexandre me ensinou sobre o modelo de memória adotado pela linguagem C, e com isso como gerenciar memória de forma eficiente para organização de dados. Tal conhecimento foi usado extensivamente neste projeto.

Gostaria de agradecer o meu professor de programação paralela Maurício Aronne Pillon, que me deu uma fundação sólida sobre sistemas distribuídos e programação paralela. Essa fundação me ajudou a entender como uma CPU moderna funciona, e também foi essencial para que eu pudesse expandir meu entendimento de programação paralela e então desenvolvimento do software deste trabalho.

Gostaria de agradecer os meus professores de matemática Volnei Avilson Soeth e César Malluta, que me ensinaram toda a fundamentação matemática usada neste trabalho. Mas além disso, me ensinaram que apenas prestar atenção nas aulas e ler um monte não adianta nada para o entendimento de um tópico sem que a pessoa “coloque a mão na massa” e realmente tente resolver problemas.

Gostaria de agradecer os membros da banca, professor André Tavares da Silva e professor Marcos de Sales Guerra Tsuzuki, que descobriram diversos problemas neste trabalho. Várias melhorias no texto e na explicação deste trabalho foram feitas por causa dessas descobertas.

Gostaria de agradecer o professor Roberto Silvio Ubertino Rosso Jr. pela orientação deste trabalho.

Gostaria também de agradecer as comunidades do *forum* “ompf2.com” e do “Q&A site Computer Graphics” da rede “Stack Exchange”. Diversas discussões técnicas sobre problemas de implementação foram feitas, e elas me ajudaram a tomar decisões que melhoraram a qualidade deste trabalho.

"Simplicity is the highest goal,
achievable when you have overcome all difficulties.
After one has played a vast quantity of notes and more notes,
it is simplicity that emerges as the crowning reward of art."
- Frédéric Chopin

RESUMO

Neste trabalho foram estudadas as principais representações geométricas no contexto de simulações de usinagem e as estruturas de dados utilizadas por estas representações. Diversas técnicas de renderização interativa para modelos complexos foram exploradas. Após a fase exploratória, foi desenvolvido um protótipo de simulador usando Campos de Distância e *ray casting*. Testes foram executados, concluindo que Campos de Distância é uma representação adequada para simulações de usinagem, providenciando operações booleanas eficientes, consumo prático de memória e oferecendo uma visualização interativa de modelos complexos usando *ray casting*.

Palavras-chaves: simulação de usinagem, representações geométricas, interativa, campos de distância, *ray casting*.

ABSTRACT

In this work were studied the main geometric representations in the context of machining simulations and its data structures. Several interactive rendering methods for complex model were studied as well. After the exploratory phase, a prototype simulator using Distance Fields and ray cast was developed. Tests were performed, concluding that the Distance Fields representation is suitable for machining simulation as it offers efficient Boolean operations, practical memory requirements and methods for interactive rendering of complex models with ray casting.

Key-words: machining simulation, geometric representations, interactive, distance fields, ray casting.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2D	Segunda dimensão
3D	Terceira dimensão
AABB	*Axis-aligned bounding box*
ADF	*Adaptively distance fields*
AOS	*Arrays of structures*
API	*Application programming interface*
B-Rep	*Boundary representation*
BIH	*Bounding interval hierarchy*
BRDF	*Bidirectional reflectance distribution function*
BSP	*Binary space partition*
BVH	*Bounding volume hierarchy*
CAD	*Computer-aided design*
CAM	*Computer-aided manufacturing*
CNC	*Computer numerical control*
CSG	*Constructive solid geometry*
DDA	*Digital differential analyzer*
DF	*Distance fields*
DRT	*Distributed ray tracing*
FPS	*Frames per second*

GPGPU	*General-purpose computing on graphics processing units*
GPU	*Graphics processing unit*
LOD	*Level of detail*
LTE	*Light transport equation*
MC	*Marching cubes*
NC	*Numerically controled*
NPR	*Non-photorealistic rendering*
PDF	Função densidade de probabilidade
SAH	*Surface area heuristic*
SDF	*Signed distance fields*
SIMD	*Single instruction multiple data*
SOA	*Structures of arrays*
VM	*Virtual manufacturing*
WRT	*Whitted ray tracing*

SUMÁRIO

1 INTRODUÇÃO

Com o uso de tecnologias de manufatura virtual, ou *virtual manufacturing* (VM), é possível a produção de produtos de alta qualidade de maneira mais rápida, eficiente e com um custo reduzido. A ideia da manufatura virtual é a execução de todo o processo de desenvolvimento, simulação e fabricação do produto em um ambiente virtual (PORTO et al., 2002). Com a ajuda do computador é possível prever melhor os erros na produção antes deles acontecerem no mundo real.

Uma das principais áreas da VM é a simulação de usinagem de máquinas NC (*Numerical Control*), pois com uma solução gráfica computadorizada é possível verificar o processo de usinagem, otimizar o código de usinagem, visualizar o produto final, verificar colisões, avaliar os parâmetros da usinagem, e detectar erros no código de usinagem em um ambiente virtual. Essas características evitam desperdício de material, tempo de configuração da máquina e tempo gasto verificando o processo de usinagem na linha de produção (ZHANG et al., 2011).

Três características são desejáveis em um simulador de usinagem. A primeira se refere a precisão do modelo. Quanto maior a resolução da representação virtual, maior será a sua capacidade de mostrar para o usuário os detalhes do modelo. Isso implica que um número maior de erros de usinagem podem ser detectados durante a fase de *design*.

Outra característica importante na simulação de usinagem é a visualização interativa, assim permitindo que o engenheiro possa observar as mudanças de seu projeto de forma imediata e com isso acelerando a fase de projeto da usinagem.

A última característica importante em um simulador de usinagem é que o tempo de simulação seja menor que o tempo da usinagem real. Em um ambiente onde tempo é importante, a simulação pode não ser útil caso demore muito tempo para ser processada. A forma geométrica

da peça sendo usinada é calculada fazendo diversas operações booleanas de subtração entre o volume da peça e o volume da ferramenta. Assim, cada operação booleana simula a remoção do excesso de material da peça. Por causa disso, para redução do tempo de simulação de usinagem, é importante que um simulador de usinagem consiga efetuar operações booleanas de forma eficiente.

Durante este trabalho de pesquisa, e no *survey* de Zhang et al. (2011), nenhuma solução para simulação de usinagem que apresente essas três características foi encontrada. Diversos pacotes CAM (*Computer-aided Manufacturing*) oferecem uma simulação cinemática de usinagem onde apenas o caminho da ferramenta é mostrado graficamente. Porém, para melhor análise da usinagem é desejável uma simulação volumétrica para visualização do processo de remoção de material (KARUNAKARAN; SHRINGI, 2007).

No trabalho de Sullivan et al. (2012) é proposta uma técnica de alta precisão para simulação de usinagem chamada Composite Adaptively Sampled Distance Fields, ou Composite ADF (CADF). Ela é uma nova estrutura de dados para representação de formas geométricas para simulação volumétrica de usinagem. Essa estrutura permite operações booleanas entre a ferramenta e a peça de forma eficiente, principalmente para simulações de alta velocidade e com um baixo consumo de memória. A alta precisão é possível pois um modelo representado pela CADF é uma composição de primitivas volumétricas, como esferas e cilindros. Essas primitivas são representadas usando funções de distância implícitas (Seção 2.6.4). Durante a simulação as operações booleanas precisam apenas inserir novas primitivas no modelo, sem a necessidade de calcular superfícies. Assim, o custo computacional é reduzido (SULLIVAN et al., 2012).

A CADF combina o método de campos de distância usando funções implícitas (Seção 2.6) com a Octree (Seção 2.7), permitindo assim que a remoção de material devido a um movimento de ferramenta necessite verificar apenas uma fração do número total de células, possibilitando

um aumento considerável no desempenho em simulações de usinagem de alta precisão. A CADF consegue processar uma simulação com aproximadamente 634 mil movimentos de ferramenta em aproximadamente 42 minutos e consumindo apenas 63MB de memória. Nesse mesmo trabalho, Sullivan et al. (2012) investigam um simulador volumétrico comercial que usa B-Rep (Seção 2.2) para representação da peça sendo usinada e também outra tecnologia que usa *voxels* (Seção 2.5) em conjunto com *marching cubes* (MC)[1] (LORENSEN; CLINE, 1987; NEWMAN; YI, 2006). Foi verificado que o simulador comercial usando B-rep demora mais de 9 horas para processar uma simulação com $\sim$230 mil caminhos de ferramenta enquanto consome $\sim$700MB de memória. Já a simulação usando *voxels* demorou $\sim$55 minutos para processar uma simulação similar a do teste anterior, porém com uma tolerância 10 vezes maior que a CADF.

Enquanto a CADF apresenta duas características desejáveis em um simulador, que são a alta precisão e rapidez, ela não suporta visualização interativa. O problema é que a representação geométrica da peça é feita de forma implícita, ou seja, a sua superfície não é gerada durante a simulação. Sullivan et al. (2012) discutem métodos indiretos e diretos para visualização dos modelos representados usando CADF. Métodos indiretos envolvem a conversão dos modelos para outros formatos antes da visualização, como exemplo é a transformação do CADF em polígonos usando o algoritmo MC. Já o método direto faz uso de *ray tracing*, porém este pode demorar aproximadamente 4 segundos para sintetizar um quadro de resolução 800×600, o que impossibilita uma simulação interativa.

Outra solução atual para simulação de usinagem volumétrica é a OpenSCAM[2], que é um simulador geométrico *opensource* para máquinas CNC de três eixos. Seu desenvolvimento foi iniciado em 2011 e está sendo constantemente atualizado. Em seu núcleo a reconstrução da peça usinada é feita usando MC. Uma amostragem linear determina os pontos

[1] *Marching cubes* é um algoritmo que faz a extração de uma malha poligonal de uma isosuperfície representada por um campo escalar discreto tridimensional.

[2] OpenSCAM. Disponível em: <http://openscam.com>. Acesso em: 12/11/2014.

de interseção entre a superfície da peça e as arestas de uma célula 3D, e usando essas interseções, o algoritmo de MC recria as faces da superfície. Atualmente a OpenSCAM não suporta visualizar o processo de usinagem, apenas o produto final da simulação é apresentado ao usuário. A visualização desse modelo é feita usando rasterização com aceleração em hardware de forma interativa.

A principal desvantagem da OpenSCAM está na sua dificuldade em processar modelos complexos. Simulações com $\sim$15 mil movimentos de ferramenta e usando uma resolução de 1024^3 células 3D podem demorar horas para conclusão. Tal resolução ainda é alta o suficiente para que polígonos sejam visíveis, e nem sempre o modelo é apresentado ao usuário por causa da falta de memória gráfica. Mais detalhes sobre a OpenSCAM são apresentados na Seção 6.4 onde diversos testes foram realizados com ela.

Uma solução para um simulador de usinagem interativo e de alta precisão necessita resolver dois problemas. O primeiro é a representação geométrica do modelo. Uma simulação de usinagem requer operações booleanas eficientes entre os modelos da peça e da ferramenta. Além disso, a representação precisa consumir um limite prático de memória e oferecer algoritmos eficientes para a manipulação de seus dados.

O segundo problema que deve ser resolvido é o de visualização interativa. Os modelos geométricos gerados pela simulação de usinagem tendem a ser modelos complexos, cheios de detalhes, por isso soluções convencionais não são adequadas.

Ambos problemas foram abordados neste trabalho. Diversas técnicas de representação de modelos 3D foram estudadas e descritas no Capítulo 2, incluindo: geometria sólida construtiva, ou *Constructive Solid Geometry* (CSG), representação de fronteira, Z-Map, vetores, *voxels* binários e campos de distância. Foi concluído na Seção 2.8 que a representação de campos de distância apresenta um bom balanço entre precisão do modelo, custo das operações booleanas entre peça e ferramenta e suporte a algoritmos para visualização interativa.

Para resolver o problema da visualização interativa foram investigadas as duas principais técnicas de renderização[3]: rasterização[4] e *ray tracing*. O Capítulo 3 descreve os detalhes dessas duas técnicas e diversas variações de *ray tracing*. A rasterização é a técnica mais usada atualmente para visualização interativa. Já a *ray tracing* apresenta um custo inicial de processamento elevado, porém oferece soluções mais simples para renderização de modelos complexos.

Este trabalho propõe uma solução para simulação volumétrica de usinagem de alta precisão e interativa. A representação geométrica é feita usando campos de distância pois ela oferece operações booleanas eficientes, alta precisão, consumo de memória adequado para computadores atuais e fácil reconstrução da superfície. O Capítulo 4 descreve a implementação da manipulação geométrica, que envolve como o processo de simulação de remoção de material funciona em geral, a estrutura de dados usada para armazenar o modelo geométrico, a criação dos volumes da peça e da ferramenta, o algoritmo das operações booleanas e o cálculo do caminho da ferramenta. A proposta para visualização interativa é usar *ray casting*. Os detalhes da implementação estão no Capítulo 5 onde são explicadas técnicas para aceleração da renderização, reconstrução da superfície e sombreamento.

Com essa solução o usuário poderá visualizar todo o processo de remoção de material da peça, podendo acompanhar a operação em diversos ângulos e com uma alta precisão e qualidade de imagem. Assim até mesmo pequenos problemas no código da usinagem podem ser detectados durante a fase de simulação.

Os testes e resultados obtidos são descritos no Capítulo 6. Nesse capítulo foram testado diversos aspectos da manipulação geométrica, que inclui o tempo para criação dos volumes da ferramenta, o consumo de memória, o tempo das operações booleanas e o tempo total da si-

[3] Da palavra em inglês *rendering*. Neste trabalho, renderização define o processo de sintetização de uma imagem 2D ilustrando um modelo 3D.

[4] Técnica mais usada atualmente para renderização interativa de cenas 3D. Ela faz a projeção de polígonos em uma imagem constituída por *pixels*.

mulação. Além da manipulação geométrica, o Capítulo 6 também testa a técnica de *ray tracing* básico e não recursiva, que inclui o tempo para construção da cena e quantidade de raios por segundo que são processados. Após os testes dessas duas áreas é feita uma discussão de como o sistema se comporta como um todo, e onde estão os principais gargalos. O capítulo termina com uma comparação com uma solução *opensource* para simulação geométrica de usinagem, a OpenSCAM.

A conclusão do trabalho é feita no Capítulo 7, onde também são descritos possíveis trabalhos futuros.

1.1 Objetivos da Pesquisa

O objetivo geral do trabalho é desenvolver uma solução eficiente para simulações geométricas e interativas de usinagem. Os objetivos específicos são:

- Explorar as principais representações geométricas, analisando suas características no contexto de simulação de usinagem;
- Explorar técnicas de renderização interativa de modelos complexos;
- Estudar a compatibilidade das representações geométricas com as técnicas de renderização;
- Desenvolver uma representação geométrica adequada para simulação de usinagem;
- Desenvolver uma estrutura de dados capaz de efetuar operações booleanas entre modelos de forma eficiente e que disponibiliza os dados do modelo de forma compatível com a visualização interativa;
- Desenvolver uma representação de modelos com alta precisão usando limites aceitáveis de memória;
- Desenvolver um renderizador interativo para visualização do processo de simulação de usinagem.

1.2 Escopo

Uma simulação de usinagem pode ser dividida em duas partes: geométrica e física (ZHANG et al., 2011). Este trabalho tem como objetivo pesquisar apenas a simulação geométrica, que permite a visualização do modelo final, trajetória da ferramenta e verificações do código da usinagem.

Simuladores de usinagem profissionais apresentam uma quantidade elevada de funcionalidades, por isso é importante enfatizar que este trabalho tem como escopo estudar apenas a tecnologia para visualização interativa e manipulação geométrica. Portanto, características como interface com o usuário, interpretador de comandos de máquinas NC, suporte a arquivos de tecnologias como CAD (*Computer-aided Design*) ou CAM, e outras, não fazem parte do escopo desse trabalho.

2 REPRESENTAÇÃO GEOMÉTRICA PRA SIMULAÇÃO DE USINAGEM

O processo de usinagem com controle numérico, *Numerical Control* (NC), no mundo real contém uma quantidade de informações grande o suficiente para que seja inviável a sua simulação exata nos computadores atuais. Por causa disso, aproximações precisam ser feitas. As primeiras aproximações usadas na área de simulação de usinagem foram feitas com a representação em arame, ou *wireframe* (REQUICHA; ROSSIGNAC, 1992), pois ela é uma estrutura de dados simples e rápida. O problema da *wireframe* é que ela introduz ambiguidades na visualização de modelos 3D complexos, além de não gerar modelos geométricos sólidos (ZHANG et al., 2011).

Para solucionar o problema da ambiguidade foram iniciadas pesquisas em modelagem de sólidos, que é uma representação computacional sem ambiguidades de um objeto sólido físico (REQUICHA; ROSSIGNAC, 1992). Diversos modelos matemáticos, algoritmos e estruturas de dados que aproximam aspectos do mundo real foram desenvolvidos. Essas aproximações permitem a computação gráfica de simulações de usinagem, porém introduzem restrições e erros (HUGHES et al., 2013). Este capítulo apresenta e compara as principais soluções de representação de sólidos para simulação de usinagem, levantando as diferentes restrições e características de cada técnica.

A Seção 2.1 descreve a Geometria Sólida Construtiva, que é uma técnica para modelagem de sólidos complexos realizando operações booleanas com primitivas simples. A Seção 2.2 apresenta a representação de fronteiras, que é o método mais usado para aplicações interativas, principalmente para modelos 3D estáticos. A Seção 2.3 descreve a Z-map, uma das técnicas mais usadas para simulação de usinagem. Em seguida, a Seção 2.4 descreve o uso de vetores para representar materiais em excesso da peça. A Seção 2.5 mostra a representação geométrica

usando *voxels* binários. A última representação é a de campos de distância, Seção 2.6. Essas técnicas são geralmente otimizadas e organizadas usando particionamento espacial, que é discutido na Seção 2.7. Para finalizar, uma discussão é feita na Seção 2.8 sobre todas as representações abordadas neste capítulo.

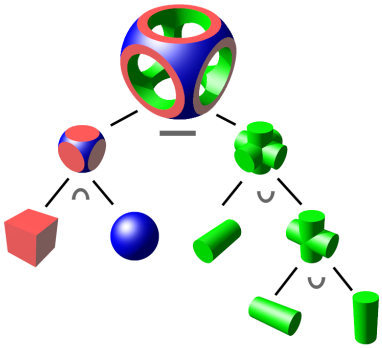
2.1 Geometria Sólida Construtiva

A Geometria Sólida Construtiva, ou *Constructive Solid Geometry* (CSG), é o nome dado ao conjunto de métodos que modelam sólidos complexos compondo diversos modelos simples. O modelo é representado por uma árvore binária de operações booleanas, onde as folhas armazenam as primitivas como esferas e cilindros, por exemplo, e os nós intermediários armazenam operadores de união, interseção e diferença ou transformações (MORTENSON, 2006).

A Figura 2.1 apresenta um exemplo de um modelo 3D representado por CSG. Nós e folhas são organizados em uma árvore binária. Folhas representam primitivas geométricas e nós intermediários as operações booleanas. Os nós descritos por $\bigcup$ representam a operação de união, $\bigcap$ a interseção e $-$ a subtração.

CSG é uma representação não pré-processada, ou seja, o objeto é representado sem especificar de forma explícita informações como faces, vértices e superfícies. Operações booleanas são eficientes e simples, pois precisam apenas adicionar primitivas em uma árvore (ZHANG et al., 2011). Além de eficiente, os modelos representados por CSG são de alta precisão e exatos, pois nenhuma aproximação é feita durante as operações booleanas (KARUNAKARAN et al., 2010). Entretanto, essa representação não pré-processada dificulta a visualização ou análise do modelo, assim, uma transformação para uma outra representação, como por exemplo a representação de fronteira que será apresentada na Seção 2.2, é necessária.

Figura 2.1: Exemplo de uma representação CSG



Fonte: Constructive solid geometry. Disponível em: <http://en.wikipedia.org/wiki/Constructive_solid_geometry>. Acesso em: 14/06/2014.

Uma simulação de usinagem usando CSG tem custo computacional $O(n^4)$, onde n é o número de movimentos da ferramenta (ZHANG et al., 2011).

2.2 Representação de Fronteira

A representação de fronteira, ou *Boundary Representation* (B-rep), define um sólido como uma coleção organizada de superfícies. A representação de um sólido é feita com a união de faces (superfícies) conectadas a arestas (curvas), onde essas arestas estão conectadas a vértices (pontos). Um método B-rep simples, que geralmente é chamado de B-Rep poliédrico, aproxima a superfície do sólido usando faces poligonais planas (MORTENSON, 2006). Interfaces de programação para computação gráfica interativa como a OpenGL[1] e a Direct3D[2] trabalham principalmente com triângulos para aproximar as superfícies de modelos 3D.

[1] OpenGL. Disponível em: <https://www.opengl.org/>. Acesso em: 23/02/2015.

[2] DirectX. Disponível em: <http://www.microsoft.com/en-us/download/details.aspx?id=35>. Acesso em: 23/02/2015.

Diferentemente da CSG, a B-rep é uma representação pré-processada. Ou seja, ela define faces, arestas e vértices do modelo de forma explícita. Assim, a visualização dela é feita de forma direta, eliminando a necessidade de conversão para outra representação (ZHANG et al., 2011). Uma das principais aplicações da representação de fronteiras é em programas 3D interativos como videogames, CAD e até mesmo simulações físicas.

Operações booleanas usando B-rep fazem a subdivisão dos polígonos nas áreas de colisão entre os modelos. Essas subdivisões fazem com que a complexidade da representação aumente com o número de operações booleanas. É estimado que a complexidade para uma simulação de usinagem usando n movimentos de ferramenta seja de $O(n^{1.5})$ (KARUNAKARAN et al., 2010).

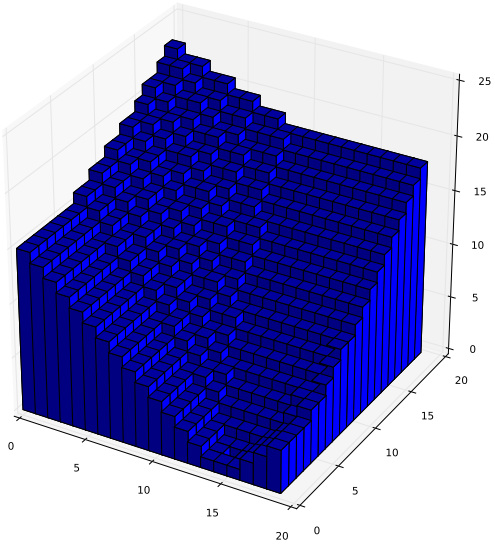
2.3 Z-Map

Z-Map (ANDERSON, 1978) é a representação de modelos 3D mais usada em simulação de usinagem (KARUNAKARAN et al., 2010). A Z-Map usa um histograma 3D para representação de modelos e detecção de colisão. A Figura 2.2 mostra um exemplo de Z-Map. O histograma 3D pode ser visto como uma matriz 2D, onde cada célula é uniformemente espaçada que armazena a altura da superfície diretamente acima dessa célula. Usando a informação da altura da superfície em cada célula é então possível reconstruí-la.

Operações booleanas são processadas por célula, onde elas atuam em apenas uma dimensão (altura) (ANDERSON, 1978). Com isso, a operação de remoção de material precisa apenas calcular o ponto de interseção (em apenas uma dimensão), e reduzir a altura da célula para esse ponto.

Z-Map é uma forma rápida e simples de representar modelos 3D para simulação de usinagem. O número de operações booleanas cresce

Figura 2.2: Exemplo de Z-Map



Fonte: Produção do próprio autor.

de forma linear com o número de movimentos da ferramenta. Entretanto, ela só consegue representar modelos sem *undercut*, ou seja, modelos 3D que não apresentam uma falta de material em alguma parte entre a superfície superior e inferior. Essa limitação existe pois apenas um valor de altura é armazenado por célula (ANDERSON, 1978). Além disso, aumentar a precisão da representação implica em um aumento drástico no custo computacional e de memória (ZHANG et al., 2011).

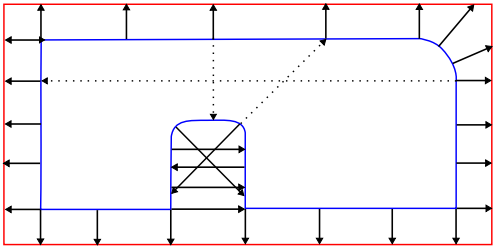
2.4 Método dos Vetores

O método dos vetores, ou *Vector Method* (CHAPPEL, 1983), representa o processo de usinagem colocando vetores em pontos da superfície da peça modelada. Esses vetores são criados normais à superfície e estendidos na direção positiva e negativa. Lembrando que a simulação de usinagem remove material em excesso da peça inicial até que ela fi-

que com a forma da peça desejada (a forma da peça modelada) então, vetores apontados na direção positiva são estendidos até a superfície da peça inicial ou até outra superfície da peça desejada. Vetores com a direção negativa são estendidos até a primeira interseção com outra superfície da peça.

Um exemplo desse método é ilustrado na Figura 2.3. Em azul é mostrada a forma desejada de uma peça após a simulação e em vermelho a forma inicial da peça. As linhas sólidas representam os vetores normais à superfície que são estendidos até a superfície da peça final ou inicial. As linhas pontilhadas representam os vetores negativos que são estendidos até interseção com a superfície desejada.

Figura 2.3: Exemplo da representação de Vetores



Fonte: Baseado em (CHAPPEL, 1983).

O processo de remoção de material é feito testando o ponto de interseção entre o vetor e a geometria da ferramenta, reduzindo o comprimento do vetor quando a interseção acontece. O comprimento do vetor determina a quantidade de material em excesso ou a profundidade do corte, assim tornando a verificação da quantidade de material em excesso ou em deficiência em um processo trivial (ZHANG et al., 2011).

O trabalho de Chappel (1983), propôs o método “point-vector”, onde a representação da peça é formada apenas por pontos normais a superfície, como na Figura 2.3. A representação de vetores é melhorada no trabalho de Park et al. (2005), onde informações extras sobre partes

da superfície da peça são introduzidas. Por exemplo, a representação armazena informações sobre localização das paredes verticais, quinas e cantos arredondados. Essas informações então podem ser usadas na reconstrução da superfície para melhor aproximação do modelo usinado.

Assim como no Z-Map, a técnica de vetores também é rápida. O tempo da simulação apresenta uma complexidade crescente de forma linear para o número de movimentos da ferramenta. Porém, aumentar a precisão do modelo requer aumentar o número de vetores, que implica no aumento do número de cálculos de interseções, assim aumentando o custo computacional e de memória. A principal limitação do método é que ele não gera um modelo de forma direta, ou seja, a visualização ou análise requer uma transformação, como por exemplo para a B-rep. Essa transformação dificulta seu uso em simulações físicas (ZHANG et al., 2011).

2.5 Voxels Binários

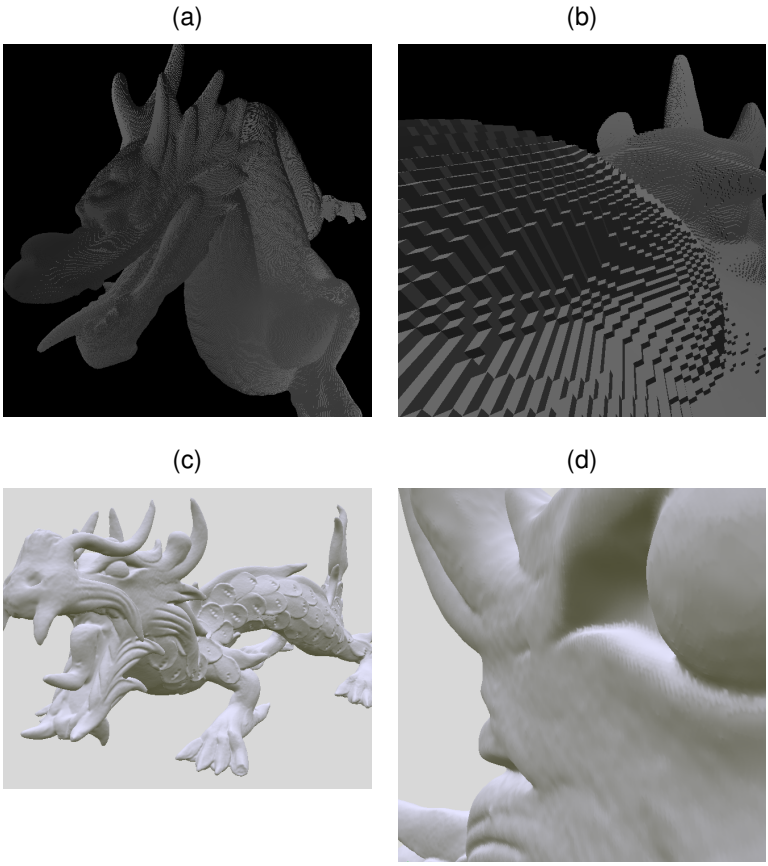
Um *voxel* representa um elemento[3] em uma matriz tridimensional. Por simplicidade, *voxels* são geralmente representados como cubos no espaço, porém podem assumir outras formas (SMITH, 1995). Uma analogia com o pixel pode ser feita, onde um pixel é o menor elemento em uma imagem 2D, enquanto um *voxel* é o menor elemento em um modelo 3D.

O *voxel* binário é uma forma simples para representação de modelos sólidos, onde o valor contido nele apenas indica se o *voxel* está (ou não) contido no modelo 3D. Os *voxels* contidos no modelo 3D são então usados para representar a geometria do objeto. Essa representação simplificada permite operações de transformações geométricas e booleanas de forma eficiente. Além disso, esses modelos armazenam informações volumétricas, permitindo assim a fácil computação de certas proprieda-

[3] Diversos tipos de elementos podem ser usados, como por exemplo: booleano (verdadeiro ou falso), binário (1 ou 0), número natural ou número real.

des físicas da peça sendo usinada, como a quantidade de material em excesso em uma região.

Figura 2.4: Exemplo de *voxels* binários e MC



Fonte: Produção do próprio autor.

Uma característica de modelos usando *voxels* binários é a geração de modelos quadriculados, pois a superfície é geralmente aproximada usando apenas cubos. Por causa disso, a representação de superfícies onduladas requer um número elevado de cubos para uma boa

aproximação, fazendo com que essa técnica exija um consumo elevado de memória (BRUNET; NAVAZO, 1990). A Figura 2.4 apresenta dois modelos similares representados com aproximadamente 26 milhões de *voxels* binários. As Figuras 2.4a e 2.4b mostram como o modelo *Dragon*[4] representado por *voxels* binários perde informações sobre a superfície, como a normal, mesmo quando é usado um número maior de *voxels* do que de triângulos do modelo original (26 milhões de *voxels* foram usados para representar o modelo original de aproximadamente 5.5 milhões de triângulos). Técnicas de *Marching Cubes* (MC) (LORENSEN; CLINE, 1987; NEWMAN; YI, 2006) podem ser usadas para melhorar a qualidade visual desses modelos, como mostram as Figuras 2.4c e 2.4d, porém elas apenas suavizam a superfície do modelo, sem melhorar a precisão dele.

2.6 Campos de Distância

Campos de Distância, ou *Distance Fields* (DF), é uma representação que armazena em cada ponto do seu campo a distância até o objeto mais próximo em um determinado domínio. Esses valores de distância podem ser usados para derivar outras propriedades, como a direção da superfície e se um ponto qualquer está dentro ou fora do modelo (JONES et al., 2006).

A DF é definida com relação a algum objeto geométrico T:

$$dist(T, \mathbf{p}) = \min_{\mathbf{q} \in T} \|\mathbf{q} - \mathbf{p}\| \tag{2.1}$$

$dist(T, \mathbf{p})$ é a menor distância entre $\mathbf{p}$ e T, onde T é um conjunto que contém todos os pontos representando um objeto geométrico e $\mathbf{p} \in \mathbb{R}^3$ pode ser um ponto qualquer. Logo, quando $\mathbf{p}$ é um ponto na superfície do volume de T, então $dist(T, \mathbf{p}) = 0$ (SHIRLEY; MARSCHNER, 2009).

A função de distância pode ser sinalizada para indicar se o ponto está dentro ou fora do modelo (FRISKEN; PERRY, 2006), assim, a função

[4] Modelo do Laboratório de Computação Gráfica da Universidade de Stanford. Disponível em: <http://graphics.stanford.edu/data/3Dscanrep>. Acesso em: 02/02/2014.

de distância sinalizada $dist_s$ é definida por:

$$dist_s(T, \mathbf{p}) = \begin{cases} -dist(T^{\complement}, \mathbf{p}) & \text{se } \mathbf{p} \in T, \\ +dist(T, \mathbf{p}) & \text{se } \mathbf{p} \in T^{\complement}. \end{cases} \tag{2.2}$$

onde $T^{\complement}$ é o complemento do conjunto T.

Campos de Distância Sinalizados, ou *Signed Distance Fields* (SDF), é o nome dado a representação que faz uso da função de distância sinalizada.

2.6.1 *Level Sets* e *Isosurfaces*

Em matemática é comum representar retas de forma paramétrica usando uma função $f(t)$, onde para cada valor real t é retornado um ponto contido na reta. Porém, retas podem ser representadas de forma que ao invés de passar t para uma função, é passado um valor (x, y) em $\mathbb{R}^2$ e a função retorna se esse ponto está, ou não, na reta. Essa função então está definindo a reta de forma implícita ao invés de paramétrica.

Usando uma função implícita $F : \mathbb{R}^2 \rightarrow \mathbb{R}$ e uma constante c qualquer, o conjunto *level set* para F em c pode ser definido como (HUGHES et al., 2013):

$$L_c = \{(x, y) : F(x, y) = c\}, \tag{2.3}$$

Quando $c = 0$, então o conjunto é chamado de *zero set* de F.

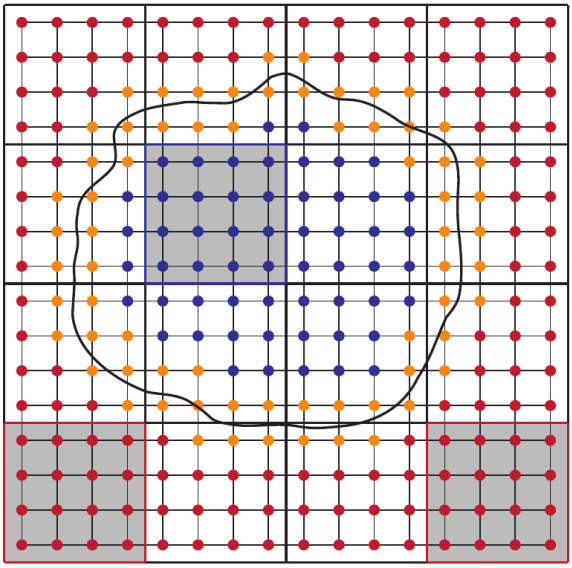
A definição de *level set* não se aplica apenas para retas, ela pode ser estendida para curvas e superfícies. No caso de uma superfície, o conjunto construído com uma função implícita $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ com valor $c = 0$ é chamado de *isosurfaces* (HUGHES et al., 2013).

Level sets são definidos usando funções de distância sinalizadas, e a *isosurface* é o conjunto de todos os pontos onde $dist(T, \mathbf{p}) = 0$.

2.6.2 *Narrow-band level set*

Diversos trabalhos precisam apenas das informações da *isosurface* próximas do *zero-set*, ou seja, caso a superfície seja representada por campos de distância, a função de distância não precisa levar em considerações valores distantes da superfície para fazer a sua reconstrução (HUGHES et al., 2013). Estruturas de dados *level sets* que levam em consideração apenas valores próximos a superfície são chamadas de *narrow-band level sets*. Essa característica é importante pois a simulação de usinagem geométrica busca aproximar apenas a superfície da peça usinada, com isso, valores de distâncias que não fazem parte da *narrow-band* podem ser descartados para melhor economia de espaço em memória e eficiência dos algoritmos de acesso.

Figura 2.5: *Narrow band level set* 2D



Fonte: Adaptada de (ZHU et al., 2012).

A Figura 2.5 apresenta um exemplo de uma *narrow-band level*

set. A linha preta ondulada representa a *isosurface* de um volume. Os pontos azuis representam *voxels* que estão dentro do volume, os pontos em vermelho indicam valores que estão fora e os pontos em amarelo mostram os valores pertencentes a *narrow-band*. Nessa representação, apenas os pontos amarelos armazenam os campos de distância sinalizados. Para economizar espaço, volumes de dados homogêneos são agrupados em um único valor. Por exemplo, na Figura 2.5 os blocos agrupam 4×4 *voxels*. Quando um desses blocos contém apenas valores completamente fora ou dentro da *narrow-band*, então o bloco pode ser descartado e substituído por um único valor indicando a homogeneidade do volume.

2.6.3 Operações Booleanas com Campos de Distância

Uma das principais características da DF é a sua eficiência e simplicidade para combinar modelos usando operações booleanas, como pode ser visto na Figura 2.6. Operações booleanas com objetos representados por DF podem ser definidas usando funções de $min(x, y)$ e $max(x, y)$ definidas por:

$$min(x,y) = \begin{cases} x & \text{se } x < y, \\ y & \text{caso contrário.} \end{cases}$$

$$max(x,y) = \begin{cases} x & \text{se } x > y, \\ y & \text{caso contrário.} \end{cases}$$

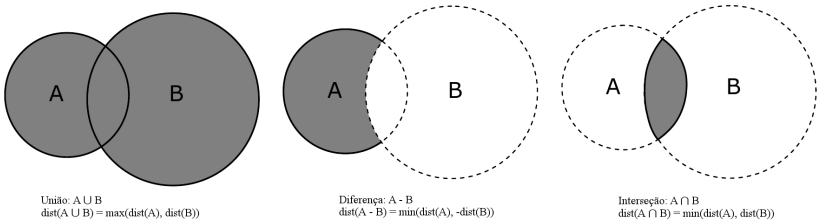
onde x e y são números reais.

Essa simplificação não resulta em distâncias estritamente euclidianas, porém oferecem uma aproximação razoável (FRISKEN; PERRY, 2006).

2.6.4 Representações da Função de Distância

Isosurfaces estão relacionadas a campos de distância pois elas são geralmente definidas usando funções de distância sinalizadas mape-

Figura 2.6: Operações booleanas com Campos de Distância



Fonte: Traduzido de (FRISKEN; PERRY, 2006)

adas à funções implícitas (BLOOMENTHAL, 2001). Por exemplo, a função de distância de uma esfera pode ser definida por:

$$dist(x, y, z) = \sqrt{x^2 + y^2 + z^2} - raio$$

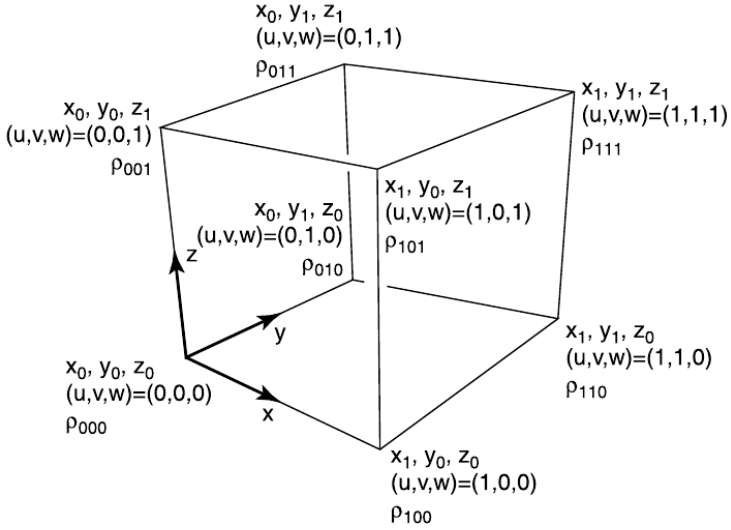
O uso de funções implícitas gera modelos de alta precisão, entretanto, esses modelos não são pré-processados. Isso implica que modelos criados usando operações booleanas precisam resolver uma ou mais funções implícitas para calcular a distância. Outro problema no uso de funções implícitas é que formas complexas são geralmente difíceis de serem especificadas e/ou estão associadas a um custo elevado para solucioná-las (FRISKEN; PERRY, 2006).

Por causa das dificuldades no uso de funções implícitas, DF são geralmente representadas por amostras discretas no espaço 3D, onde cada uma armazena a distância de sua posição até a superfície mais próxima. A representação mais simples organiza essas amostras em *voxels* uniformemente espaçados em uma matriz 3D (HAGAN; BRALEY, 2009). Cada *voxel* armazena uma amostra de distância. Uma das principais vantagem desse tipo de representação é que o número de operações booleanas não influencia na complexidade da representação, pois operações booleanas podem efetuar chamadas as funções *min* e *max*, como na Figura 2.6, e armazenar os resultados já processados. Entretanto, a qualidade dos modelos usando essa representação discreta geralmente

é inferior a representação usando funções implícitas, e a qualidade fica relacionada a quantidade de *voxels* da representação e ao algoritmo de reconstrução da superfície, que será discutido na Seção 2.6.5.

É comum organizar os *voxels* em células para facilitar certas operações que serão vistas neste capítulo, como reconstrução da superfície e cálculo da normal, por exemplo. Cada célula é uma octante que armazena oito amostras de distância em seus cantos $(\rho_{000}, \rho_{001}, \ldots, \rho_{111})$, como mostra a Figura 2.7. Células são alocadas de forma que cada uma de suas paredes podem ocupar o mesmo espaço de uma outra célula, assim *voxels* podem ser compartilhados por várias células.

Figura 2.7: Geometria de uma célula dos campos de distância.



Fonte: (SHIRLEY; MARSCHNER, 2009)

A organização dos *voxels* em uma grade, ou matriz 3D, oferece acesso rápido e constante aos seus valores. A desvantagem dessa organização uniforme é que ela não é adequada para modelos 3D com diferentes níveis de complexidades. Quando se usa um DF uniforme, re-

giões complexas acabam tendo de usar a mesma resolução que regiões simples. Assim, é necessário uma matriz de alta resolução para capturar pequenos detalhes, e com isso aumentando o consumo de memória drasticamente (JONES et al., 2006).

2.6.5 Interpolação Trilinear

A função da distância $dist(\mathbf{x})$, Eq. 2.1, que mapeia $dist : \mathbb{R}^3 \rightarrow \mathbb{R}$, trabalha no espaço contínuo. Assim, é necessário a reconstrução da superfície contínua caso uma representação dos campos de distância com amostras discretas seja usada. Essa reconstrução geralmente é feita usando a interpolação trilinear (HAGAN; BRALEY, 2009).

Usando a célula da Figura 2.7 e um ponto $\mathbf{p} = (x, y, z)$ dentro dela é então possível reconstruir a superfície usando interpolação trilinear (SHIRLEY; MARSCHNER, 2009; THEISEL, 2000):

$$\begin{aligned} dist(u,v,w) = \quad & +(1-u)(1-v)(1-w)\rho_{000} \\ & +(1-u)(1-v)(0+w)\rho_{001} \\ & +(1-u)(0+v)(1-w)\rho_{010} \\ & +(0+u)(1-v)(1-w)\rho_{100} \\ & +(0+u)(1-v)(0+w)\rho_{101} \\ & +(1-u)(0+v)(0+w)\rho_{011} \\ & +(0+u)(0+v)(1-w)\rho_{110} \\ & +(0+u)(0+v)(0+w)\rho_{111}, \end{aligned} \tag{2.4}$$

onde

$$\begin{aligned} u = \quad & \tfrac{x-x_0}{x_1-x_0}, \\ v = \quad & \tfrac{y-y_0}{y_1-y_0}, \\ w = \quad & \tfrac{z-z_0}{z_1-z_0}, \end{aligned}$$

e

$$\begin{aligned} 1-u = \quad & \tfrac{x_1-x}{x_1-x_0}, \\ 1-v = \quad & \tfrac{y_1-y}{y_1-y_0}, \\ 1-w = \quad & \tfrac{z_1-z}{z_1-z_0}. \end{aligned}$$

x_0 e x_1 delimitam o espaço ocupado pela célula no eixo X. As outras variáveis y_0, y_1, z_0 e z_1 representam limites em seus respectivos eixos.

Fazendo $u_0 = 1-u$ e $u_1 = u$, e usando definições similares para v_0, v_1, w_0, w_1, então a Equação 2.4 pode ser simplificada para (SHIRLEY; MARSCHNER, 2009):

$$dist = \sum_{i,j,k=0,1} u_i v_j w_k \rho_{ijk} \tag{2.5}$$

Um aspecto importante da reconstrução da superfície com interpolação trilinear é o uso de apenas interpolações lineares com os valores de distância. Por causa disso, certas características, como cantos não arredondados, não são reconstruídas de forma correta (FRISKEN et al., 2000), pois a interpolação trilinear tende a suavizar a superfície. Porém, é possível diminuir esse efeito aumentando o número de amostras de distâncias, como demonstrado no trabalho de Frisken et al. (2000).

2.6.6 Interseção entre DF e um Raio

Ray tracing pode ser usado para visualização de modelos representados por DF. Para isso, um raio $\mathbf{a} + t\mathbf{b}$ atravessa o volume verificando se há superfície nas células. Caso sim, a função da distância é resolvida pelo parâmetro t do raio na interseção com a superfície, ou seja:

$$dist(x_a + tx_b, y_a + ty_b, z_a + tz_b) = 0 \tag{2.6}$$

Ao aproximar $dist$ com uma interpolação trilinear, a Equação 2.6 é expandida em um polinômio cúbico em t. Esse polinômio pode gerar até três soluções, isso significa que um raio pode atravessar a superfície até três vezes. Soluções fora do intervalo da célula são descartadas. Quando há mais de uma solução, é escolhida a com o menor valor de t. Quando não há soluções, então o raio não faz interseção com a superfície nessa célula (SHIRLEY; MARSCHNER, 2009).

Para encontrar a interseção de um raio $\mathbf{p} = \mathbf{a} + t\mathbf{b}$ com a superfície, primeiramente é necessário converter ele nas coordenadas

(u_0, v_0, w_0) e (u_1, v_1, w_1) com as equações:

$$\begin{aligned} \mathbf{a}_0 &= (u_0^a, v_0^a, w_0^a) = (\tfrac{x_1 - x_a}{x_1 - x_0}, \tfrac{y_1 - y_a}{y_1 - y_0}, \tfrac{z_1 - z_a}{z_1 - z_0}), \\ \mathbf{b}_0 &= (u_0^b, v_0^b, w_0^b) = (\tfrac{x_b}{x_1 - x_0}, \tfrac{y_b}{y_1 - y_0}, \tfrac{z_b}{z_1 - z_0}) \end{aligned} \tag{2.7}$$

Note que $\mathbf{a}_0$ é uma localização enquanto $\mathbf{b}_0$ é uma direção. As equações são similares para $\mathbf{a}_1$ e $\mathbf{b}_1$:

$$\begin{aligned} \mathbf{a}_1 &= (u_1^a, v_1^a, w_1^a) = (\tfrac{x_a - x_0}{x_1 - x_0}, \tfrac{y_a - y_0}{y_1 - y_0}, \tfrac{z_a - z_0}{z_1 - z_0}), \\ \mathbf{b}_1 &= (u_1^b, v_1^b, w_1^b) = (\tfrac{-x_b}{x_1 - x_0}, \tfrac{-y_b}{y_1 - y_0}, \tfrac{-z_b}{z_1 - z_0}) \end{aligned} \tag{2.8}$$

A interseção com a superfície acontece quando $dist(\mathbf{p}) = 0$, onde a função $dist$ é dada por:

$$dist = \sum_{i,j,k=0,1} (u_i^a + t u_i^b)(v_j^a + t v_j^b)(w_k^a + t w_k^b)\rho_{ijk} \tag{2.9}$$

A Equação 2.9 pode ser expressa como um polinômio cúbico em t (SHIRLEY; MARSCHNER, 2009):

$$At^3 + Bt^2 + Ct + D = 0 \tag{2.10}$$

onde

$$\begin{aligned} A = &\quad \textstyle\sum_{i,j,k=0} u_i^b v_j^b w_k^b \rho_{ijk}, \\ B = &\quad \textstyle\sum_{i,j,k=0} (u_i^a v_j^b w_k^b + u_i^b v_j^a w_k^b + u_i^b v_j^b w_k^a)\rho_{ijk}, \\ C = &\quad \textstyle\sum_{i,j,k=0} (u_i^b v_j^a w_k^a + u_i^a v_j^b w_k^a + u_i^a v_j^a w_k^b)\rho_{ijk}, \\ D = &\quad \textstyle\sum_{i,j,k=0} u_i^a v_j^a w_k^a \rho_{ijk}. \end{aligned}$$

Note que A, B, C e D são coeficientes no polinômio cúbico da Equação 2.10. Já t é a sua variável. Lembrando que t é a variável da equação do raio $\mathbf{p} = \mathbf{a} + t\mathbf{b}$, logo, uma solução para o polinômio significa encontrar um valor de t onde o ponto $\mathbf{p}$ está na superfície ($dist(\mathbf{p}) = 0$). Quando se encontra uma solução em t, se encontra uma interseção entre o raio e a superfície.

2.6.7 Normal da Superfície

A normal, ou seja, o vetor perpendicular à superfície, é importante para DF pois ela é usada no cálculo de iluminação, detecção de

colisão e localização do ponto mais próximo que está na superfície (FRISKEN et al., 2000). Para campos de distância, a normal é igual ao gradiente da função de distância, ou seja, para um ponto (x, y, z) a normal é definida por (SHIRLEY; MARSCHNER, 2009):

$$\mathbf{N} = \nabla dist = \left(\frac{\partial dist}{\partial x}, \frac{\partial dist}{\partial y}, \frac{\partial dist}{\partial z} \right) \tag{2.11}$$

onde $\nabla dist$ é o gradiente da função de distância.

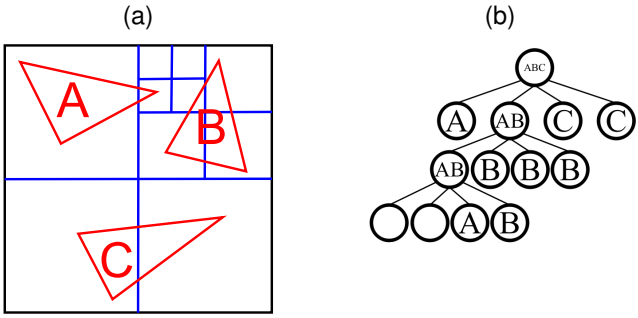
2.7 Organização da Superfície usando Particionamento Espacial

Durante a simulação de usinagem é importante que o acesso a superfície seja eficiente. Por exemplo, durante uma operação booleana apenas um subconjunto dos pontos que definem a peça precisam ser processados. Assim, melhor eficiência é alcançada quando apenas pontos que serão afetados durante uma operação booleana são processados, ignorando o resto.

Uma forma para acelerar o acesso às partes de interesse da superfície é fazendo o particionamento do espaço ocupado por essas partes. Esse particionamento espacial divide o espaço da superfície em dois ou mais subconjuntos. Cada um desses subconjuntos podem então continuar o processo de divisão de forma recursiva, criando assim uma estrutura hierárquica. Dessa forma, basta usar os subconjuntos que contém os pontos de interesse durante as operações booleanas.

Octree é uma técnica conhecida e bastante adotada em diversas áreas para realizar o processo de particionamento espacial. A Octree é uma estrutura de dados em forma de árvore onde cada nó possui oito filhos. Ela é útil para particionar espaços tridimensionais aplicando subdivisões recursivamente. Cada subdivisão divide o espaço em oito partes iguais. A Figura 2.8b apresenta uma Quadtree (HUNTER; STEIGLITZ, 1979) construída para armazenar os triângulos A, B e C da Figura 2.8a. A Quadtree funciona de maneira análoga a Octree, porém cada nó possui

Figura 2.8: Exemplo de Quadtree



Fonte: Produção do próprio autor.

apenas quatro filhos, assim sendo útil para particionar espaços bidimensionais (MEAGHER, 1982).

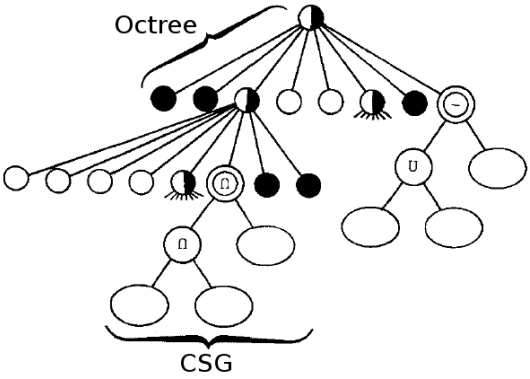
As próximas subseções apresentam como a Octree modifica as representações discutidas anteriormente neste capítulo. A Subseção 2.7.1 descreve a Graftree, que é uma combinação da CSG com a Octree. O particionamento espacial da B-rep é discutido com a Extended Octree, Subseção 2.7.2. *Voxels*, que representam elementos em uma matriz 3D, também podem ser organizados em uma Octree, como pode ser visto na Subseção 2.7.3. Campos de distância podem ser otimizados usando a Adaptively Distance Fields da Subseção 2.7.4, que é a união entre Octree e campos de distância. Por último é apresentada a *narrow-band level set* hierárquica na Seção 2.7.5.

2.7.1 Graftree

A Graftree (KAWASHIMA et al., 1991) reduz o custo computacional da CSG separando-a em diversas partições usando uma Octree (MEAGHER, 1982). A Octree faz a divisão espacial do modelo, indexando árvores CSG nas suas folhas. Cada árvore CSG representa um modelo local, assim, uma operação booleana que influencia apenas uma pequena

região do modelo 3D irá apenas ser adicionada nas árvores CSG que fazem interseção com essa operação booleana.

Figura 2.9: Exemplo de uma Graftree



Fonte: (KAWASHIMA et al., 1991).

A Figura 2.9 apresenta um exemplo usando Graftree, onde os nós são organizados com uma Octree e a superfície do modelo é representada usando CSG. Nós totalmente brancos representam regiões vazias do modelo. Nós totalmente pretos são regiões que estão dentro do modelo. Já os nós brancos e pretos significam regiões que estão parcialmente dentro do modelo. Os nós folhas com dois círculos representam regiões com superfícies, que são modeladas com CSG.

2.7.2 Extended Octree

O particionamento espacial da B-rep pode ser feito usando uma Extended Octree (BRUNET; NAVAZO, 1990), que faz a separação espacial dos polígonos do modelo. Nessa representação, os polígonos são armazenados nas folhas de uma Octree. Durante as operações booleanas o algoritmo só precisa atravessar as árvores dos dois modelos (ferramenta e peça) e verificar se houve interseção apenas nos nós que ocupam o mesmo espaço em ambas árvores.

2.7.3 Voxels binários armazenados em Octree

Voxels binários podem ser armazenados em uma grade, que é uma matriz 3D. Essa representação simples e de fácil implementação oferece acesso $O(1)$ aos *voxels*, fazendo com que ela seja adequada para execução de operações por *voxel*, como em transformações geométricas por exemplo. O problema de usar uma matriz 3D está no aumento drástico no consumo de memória para melhorar a precisão do modelo. Por exemplo, para dobrar a precisão de um modelo com 1024^3 *voxels* é necessário expandir o tamanho da matriz para 2048^3, ou seja, uma diferença de oito vezes do tamanho original. Além do custo em memória, esse aumento de precisão impacta no custo computacional das transformações geométricas e booleanas, pois ambas têm complexidades $O(n)$, onde n é o número de *voxels*.

Para aliviar os problemas da representação usando grade, são geralmente usadas estruturas de dados como a Octree (MEAGHER, 1982). Sua principal função é no agrupamento de nós com informações homogêneas. No contexto de *voxels* binários, todos os elementos pertencentes ao volume geométrico são homogêneos. Essa característica é importante pois em uma simulação de usinagem o modelo da peça e da ferramenta apresentam diversas regiões homogêneas, como o ar ao redor do modelo, ou regiões de seu interior sem superfície. Por causa disso essa estrutura de dados oferece uma redução drástica no consumo de memória em simulações de usinagem.

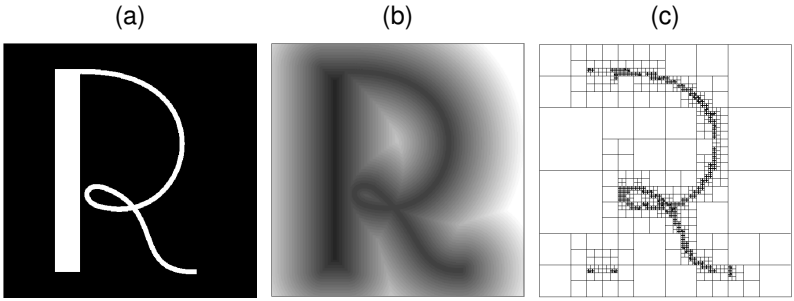
O custo do uso da Octree é de uma estrutura de dados mais complexa e tempo de acesso $O(\log n)$ (SAMET, 1984), ao contrário da grade que oferece acesso $O(1)$.

2.7.4 Adaptively Distance Fields

Campos de Distâncias Amostrados Adaptativamente, ou *Adaptively Sampled Distance Fields* (ADF) (FRISKEN et al., 2000), é uma técnica que se adapta melhor as diferentes complexidades de um modelo

3D do que a representação uniforme. ADF usa altas taxas de amostragem onde se encontra um nível elevado de variações de frequência nos campos de distância, e baixas taxas de amostragem em campos com baixa variância. A adaptatividade da técnica se baseia na subdivisão espacial usando uma Octree, onde as células da árvore são subdivididas de acordo com o nível de detalhes local no campo de distância (FRISKEN; PERRY, 2006). Usando a Figura 2.10a como exemplo, a transformação dela em campos de distância é mostrada na Figura 2.10b e a ADF na Figura 2.10c. Note que regiões complexas, como os contornos da letra "R", são subdivididos para melhorar a representação, enquanto regiões sem muita variância, como nas partes próximas as bordas da imagem, apresentam um número reduzido de subdivisões.

Figura 2.10: Campos de distância para um carácter "R"



Fonte: (FRISKEN et al., 2000)

A geração da ADF pode ser feita de duas formas: *bottom-up* e *top-down*. Primeiramente precisa ser construída uma função de distância. É desejável que essa função seja contínua, diferenciável e com um intervalo limitado, porém não é obrigatório. Por exemplo, uma função implícita de algum objeto pode ser usada diretamente, ou pode ser uma função que calcula a distância euclidiana do ponto até a superfície representada por polígonos mais próxima (FRISKEN et al., 2000).

O algoritmo de geração de ADF *bottom-up* começa com campos

de distância uniformemente amostrados com resolução finita. O processo de construção da Octree é feito de baixo pra cima, iniciando pelos *voxels* (as menores células) da árvore. Um grupo de oito células é homogêneo quando todas as células não possuem filhos e podem ser reconstruídas usando amostras da célula-pai. Uma margem de erro é usada para delimitar quando *voxels* são interpretados como homogêneos ou heterogêneos. A cada etapa do algoritmo, todas as células do nível atual da hierarquia verificam a sua homogeneidade. Caso um grupo de célula é identificado como homogêneo, então esse grupo é substituído por apenas uma célula. Após a verificação do nível atual, o próximo nível da hierarquia é verificado. O algoritmo termina quando se chega no nó raiz ou quando todas as células do nível atual são homogêneas (FRISKEN et al., 2000). A Octree resultante é a ADF.

O algoritmo de geração de ADF *top-down* faz o processo de construção da Octree começando pela raiz e terminando nas folhas. Primeiramente são calculadas as distâncias para o nó raiz, para então ser subdividido recursivamente de acordo com uma regra de subdivisão. O trabalho de Frisken et al. (2000) propôs um método onde a célula é subdividida quando a diferença entre o resultado da função de distância e da distância reconstruída é maior que uma margem de erro especificada. A diferença é calculada em 19 pontos distintos: centro da célula, centro das faces e arestas. A Figura 2.10c é um exemplo de uma ADF gerada com esse algoritmo.

O algoritmo *bottom-up* apresenta a desvantagem de consumir bastante memória, pois ele requer uma matriz 3D dos campos de distância antes de iniciar a construção da Octree. Já o algoritmo *top-down* precisa realizar buscas em nós vizinhos. Essas buscas são operações caras e podem influenciar no desempenho da aplicação.

ADF oferece uma forma compacta para representação de DF de alta qualidade e precisão (JONES et al., 2006), porém ao custo de uma estrutura de dados mais complexa, acesso aos *voxels* mais lento ($O(\log n)$), e um custo extra na computação dos campos de distância,

pois requer a criação de uma Octree.

No ADF de Frisken et al. (2000), cada nó da Octree armazena a sua própria célula do campo de distância, assim, nós vizinhos acabam não compartilhando os seus campos de distância. O trabalho de Bærentzen e Christensen (2002) apresenta uma separação da subdivisão espacial e da representação dos campos de distância. A importância dessa separação está na eliminação da redundância causada no método de Frisken et al. (2000). Nessa proposta, cada nó da Octree não mais armazena um octante com amostras de distância. Ao invés disso, a posição das amostras é usada para buscar seus valores em uma *hash table* 3D. A vantagem desta técnica é a redução no consumo de memória para representação da ADF, porém ao custo de um aumento na complexidade do acesso aos seus elementos, pois a *hash table* insere um acesso indireto a esses valores.

2.7.5 *Narrow-band Level Set* Hierárquica

Uma das características da representação de campos de distância é que *voxels* vizinhos tendem a conter valores de distância diferentes. Isso dificulta o agrupamento de espaços homogêneos como na representação de *voxels* binários da Seção 2.7.3. A ADF da Seção 2.7.4 é uma das soluções para este problema, porém uma alternativa é o uso de *narrow-band level set* hierárquica.

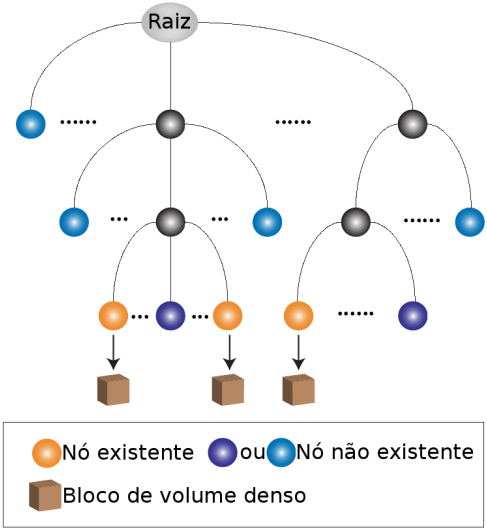
Narrow-band level set, como visto na Seção 2.6.2, considera apenas valores de distância próximos a superfície do volume, que são os valores pertencentes a *narrow-band*. Esses valores contidos na *narrow-band* tendem a ser *voxels* heterogêneos, porém, valores não pertencentes a *narrow-band* podem assumir apenas dois valores: fora do volume e dentro do volume. Por causa disso, *voxels* não pertencentes a *narrow-band* podem ser facilmente agrupados em regiões homogêneas.

Uma estrutura *narrow-band level set* hierárquica leva em consideração a heterogeneidade dos *voxels* pertencentes a *narrow-band* e a

homogeneidade das regiões não pertencentes a *narrow-band*. São usados blocos de volumes densos, que são grades, para as regiões pertencentes a *narrow-band*. Como é esperado *voxels* heterogêneos nessas regiões, então esses são colocados em uma grade pois ela oferece acesso rápido e direto. Regiões fora da *narrow-band* são agrupadas como nos *voxels* binários, porém nós podem assumir três estados: completamente dentro do volume, completamente fora do volume e *narrow-band*.

Um exemplo dessa estrutura é mostrado na Figura 2.11. Estruturas hierárquicas são naturalmente representadas por árvores. Nós folhas podem indicar três condições: a) nó completamente fora do volume; b) nó completamente dentro do volume; ou, c) bloco de dados denso contendo os campos de distância sinalizados. Note que informações sobre a superfície estão contidas apenas nos blocos densos.

Figura 2.11: *Narrow-band level set* hierárquica



Fonte: Traduzido de (ZHU et al., 2012).

A *narrow-band level set* hierárquica é útil para economia de me-

mória e aceleração dos algoritmos de operações booleanas, pois eles conseguem operar em regiões homogêneas assim como em *voxels* heterogêneos.

Quando comparada com a ADF, a *narrow-band level set* hierárquica é mais simples pois não requer nenhuma heurística para determinar quais *voxels* serão armazenados e quais regiões serão subdivididas. Basta apenas determinar se o *voxel* pertence ou não a *narrow-band*, e caso ele não pertença, então ele recebe um valor constante para indicar se está dentro ou fora do volume. A subdivisão vai acontecer caso a região apresente valores heterogêneos.

2.8 Discussão do Capítulo

A CSG é uma representação de modelos 3D com uma alta precisão. Além disso, ela oferece operações booleanas rápidas e um baixo consumo de memória. A CSG é comumente usada em aplicações CAD para interação com o usuário, pois ela oferece um método fácil para composição de peças mecânicas por meio de operações booleanas com primitivas (KARUNAKARAN et al., 2010). Entretanto, a representação computacional dos modelos 3D nessas aplicações CAD é feita usando outros métodos, como B-rep. Isso porque a CSG apresenta uma grande desvantagem, que é a sua representação não pré-processada. Essa característica faz com que a CSG seja inviável em aplicações onde é necessário efetuar diversas reconstruções da superfície, como em uma simulação de usinagem que mostra o processo de usinagem e não apenas o resultado final. Outras visualizações, como a de arame, também não são eficientes de serem geradas com a CSG (REQUICHA; ROSSIGNAC, 1992).

A Graftree é uma otimização para CSG. Ela inclui um custo extra para operações booleanas, pois durante tais operações o algoritmo precisa atravessar a árvore da Octree além da CSG. Esse processo faz a separação da árvore da CSG em pequenas partes, diminuindo a complexidade geral do algoritmo. Porém, a representação continua sendo não pré-processada, e regiões complexas do modelo 3D que implicam em um

número elevado de operações booleanas mantém o mesmo problema da CSG.

B-rep é a representação mais usada em aplicações de modelagem comerciais, incluindo AutoCAD e SolidEdge (KARUNAKARAN et al., 2010). B-rep representa o modelo 3D de forma aproximada, geralmente com uma malha de triângulos. Isso implica em uma menor precisão que a CSG na prática. A B-rep é considerada a representação mais adequada para visualizações interativas, pois ela tem uma longa história em videogames e aplicações CAD, além de suporte a aceleração gráfica em hardware. Porém, B-rep não é adequada para simulações de usinagem onde o número de operações booleanas é elevado. O problema está na subdivisão de polígonos durante cada operação booleana, fazendo com que a representação aumente em complexidade durante a simulação. Sullivan et al. (2012) testaram uma simulação de usinagem com ~ 250.000 caminhos de ferramenta com dois métodos: B-rep e *voxels*. A versão B-rep demorou mais de nove horas para completar, com um pico de uso de $\sim$ 700MB de memória. Já a versão usando *voxels* executou a operação em 55,27 minutos ocupando a mesma quantidade de memória.

Z-map foi uma representação de sólidos muito usada para simulação de usinagem (ZHANG et al., 2011). Porém ela apresenta uma grande limitação, que é não poder representar modelos com *undercut*.

O método de vetores é uma representação complexa. Primeiramente o algoritmo requer a geometria do modelo desejado (o resultado final da simulação de usinagem), que geralmente é exportado do CAD. Depois, é necessário um algoritmo eficiente para escolha dos pontos nessa superfície que vão conter os vetores. Como apenas vetores não são suficientes para uma boa representação das superfícies, outras técnicas foram propostas para agregar mais informações aos vetores (JERARD et al., 1989; PARK et al., 2005). Com isso, modelos não são representados de forma direta, dificultando a sua análise e visualização (ZHANG et al., 2011).

Voxels binários é uma representação simples, porém eficiente

para simulação de usinagem interativa. Operações booleanas são rápidas e a sua visualização interativa é possível. A principal desvantagem desse método é que *voxels* não conseguem representar superfícies suaves, como mostrou a Figura 2.4a. Várias técnicas foram propostas para melhorar a reconstrução da superfície, como *Marching Cubes* (MC) (LORENSEN; CLINE, 1987; NEWMAN; YI, 2006) e *Dual Countouring* (JU et al., 2002). Enquanto essas técnicas conseguem produzir superfícies mais suaves, elas não melhoraram a precisão do modelo, que continua sendo o tamanho do *voxel*.

Voxels binários apresentam um custo alto de memória para modelos complexos quando usados com grades. Entretanto, esses *voxels* representam apenas duas regiões: ar e volume da peça. Isso permite uma redução drástica no consumo de memória ao usar uma estrutura hierárquica de regiões homogêneas, como a Octree.

Campos de Distância é uma representação que oferece operações booleanas eficientes, que é um dos requisitos deste trabalho, pois essas operações booleanas são usadas para simulação da remoção de material.

Quando usada com funções implícitas, a DF oferece alta precisão na sua representação, porém acaba tendo os mesmos problemas de modelos não pré-processados da CSG. Mas ao usar uma representação discreta, a DF apresenta uma precisão maior que a de *voxels* binários ao custo de um consumo extra de memória, pois cada *voxel* geralmente armazena um ponto flutuante de oito *bits* para armazenar o valor de distância, ao invés de apenas um *bit*.

Diferentemente dos *voxels* binários, a DF com representação discreta dificilmente contém regiões homogêneas. Isso ocorre porque *voxels* vizinhos tendem a apresentar diferentes valores de distância. Por causa disso a DF não pode usar a mesma técnica da de *voxels* binários para economia de memória. Uma solução é a ADF, que oferece meios para uma economia drástica no consumo de memória. O problema com tal representação é que os algoritmos para criação da ADF têm um custo com-

putacional alto. A versão *bottom-up* requer a construção do modelo em grade antes da criação da ADF, assim apresentando um consumo alto de memória. Já a versão *top-down* faz diversos acessos aos nós-vizinhos em uma Octree durante a sua construção, que podem ser operações caras. Como este trabalho faz a criação de diversos volumes durante o processo de usinagem (um volume é criado para cada movimento da ferramenta, veja Capítulo 4), a ADF não se mostrou adequada por causa desse custo alto para sua criação.

Uma segunda opção para economia de memória com a DF é o uso de estruturas *narrow-band level sets* hierárquicas. Essas estruturas conseguem economia de memória pois regiões distantes da superfície são ignoradas, logo, essas regiões podem apresentar um valor homogêneo[5], e assim conseguem uma economia de memória similar a de *voxels* binários.

Além da economia de memória, estruturas *narrow-band level sets* hierárquicas também oferecem fácil e eficiente construção. A criação é feita de forma *top-down* e não precisa visitar nós vizinhos ou construção de uma grade antes de sua execução. Isso porque a subdivisão de uma região é feita apenas quando se encontra um valor heterogêneo.

Uma outra característica da DF é que ela oferece meios de visualização direta quando usada em conjunto com algoritmos de *ray tracing* e reconstrução da superfície com interpolação trilinear.

Diante dessas características, a DF com a *narrow-band level set* hierárquica foi a representação usada neste trabalho. As operações booleanas eficientes são essenciais para a rápida simulação da remoção de material. As estruturas *narrow-band level sets* hierárquicas permitem um consumo prático de memória, além da rápida construção de volumes representando movimentos das ferramentas. Além disso, a DF oferece meios para renderização direta usando *ray tracing*, que foi a característica usada para alcançar a visualização interativa (veja Capítulo 5).

[5] Por exemplo: todos os *voxels* distantes da superfície podem armazenar o valor $+\infty$.

3 TÉCNICAS DE RENDERIZAÇÃO PARA SIMULAÇÃO DE USINAGEM

A simulação de usinagem de alta precisão é um processo capaz de gerar modelos geométricos complexos, com um nível elevado de detalhes. Essa complexidade dificulta a visualização interativa desses modelos usando a rasterização, que é a técnica tradicional para renderização interativa.

Em modelos massivos, quando a rasterização é usada, algoritmos de simplificação da malha poligonal são geralmente usados (GOBBETTI et al., 2008; ALIAGA et al., 1998; BAXTER III et al., 2002; TIAN et al., 2010; PENG; CAO, 2012). Simplificações de malhas poligonais tradicionais geralmente fazem uso de etapas de pré-processamento demoradas (GOBBETTI et al., 2008; ALIAGA et al., 1998; BAXTER III et al., 2002; TIAN et al., 2010), como na solução de Cignoni et al. (2004) que processa de 15 a 30 mil triângulos por segundo usando 14 Athlon 2200+ CPUs. Soluções que requerem o uso de uma etapa de pré-processamento demorada não funcionam para cenas dinâmicas, onde o modelo geométrico pode ser alterado a cada quadro, como em simuladores de usinagem ou cenas com animações. Peng e Cao (2012) reduzem a etapa de pré-processamento fazendo uso de GPGPU, assim aumentando a complexidade do algoritmo, porém conseguindo de 6 a 22 quadros por segundo em uma cena com aproximadamente 332 milhões de triângulos. Porém, o trabalho de Peng e Cao (2012) não deixa claro o custo da etapa de pré-processamento, e apresenta resultados apenas em cenas estáticas.

Atualmente a rasterização é a técnica mais usada para computação gráfica interativa, por isso ela é estudada e discutida na Seção 3.2.

Além da rasterização, algoritmos de *ray tracing* também são usados para visualizações de modelos complexos (GOBBETTI et al., 2008; WALD et al., 2005; LAUTERBACH et al., 2008; DIETRICH et al., 2007).

Para cenas pequenas o algoritmo de rasterização apresenta melhor desempenho que *ray tracing* pois oferece melhor coerência de código e dados, permitindo assim a aceleração em hardware. Para cenas com modelos complexos, a rasterização não apresenta uma vantagem clara, pois as duas técnicas, em suas formas básicas, apresentam complexidade linear ao número de primitivas na cena, que não é adequada para modelos complexos. Assim, técnicas para remoção de primitivas não visíveis e estruturas espaciais são usadas para alcançar uma renderização com complexidade sub-linear. Por causa disso, ambas as técnicas de rasterização e *ray tracing* acabam adotando soluções similares para o problema de visualização de modelos complexos (GOBBETTI et al., 2008).

A área de *ray tracing* interativo teve um avanço considerável nos últimos anos, principalmente para cenas dinâmicas (BIKKER, 2012; WALD; HAVRAN, 2006; WALD et al., 2007; CRASSIN et al., 2011). Entretanto, esses trabalhos têm como foco principal cenas com um número reduzido de polígonos quando comparados com os modelos usados em simulações de usinagem. Este trabalho explora a possibilidade de usar *ray tracing* para renderizar cenas dinâmicas com modelos complexos. O objetivo final é uma solução que possa ser usada em simuladores de usinagem de alta precisão. A Seção 3.3 entra em detalhes sobre o funcionamento dos principais algoritmos de *ray tracing* e também das principais estruturas de dados aceleradoras que permitem que essa técnica alcance visualizações interativas.

As próximas seções descrevem os métodos de rasterização e *ray tracing* com mais detalhes. Porém, antes de descrevê-los, a Seção 3.1 revisa a equação do transporte de luz, que é a base para diversos algoritmos de *ray tracing* que serão abordados.

3.1 Equação da Renderização

A renderização de uma cena 3D pode ser entendida como a visualização do comportamento da luz visível em contato com as superfícies. No entanto, calcular o comportamento de cada fóton em uma cena é uma

tarefa computacionalmente inviável. Por isso, a computação gráfica usa aproximações e modelos simplificados, como a da óptica.

O trabalho de Kajiya (1986) introduziu a equação da renderização, que descreve o equilíbrio da distribuição da radiação em uma cena. Essa equação é normalmente usada para aproximar simulações do transporte de luz. A equação da renderização também é chamada de equação do transporte de luz, ou LTE, do inglês *light transport equation*, (SHIRLEY; MARSCHNER, 2009; PHARR; HUMPHREYS, 2010), e pode ser representada da seguinte forma:

$$L_s(x, k_o) = L_e(x, k_0) + \int_{x'} \rho(x, k_i, k_o) L_s(x', x - x') G(x, x') \mathrm{d}A' \tag{3.1}$$

onde:

$$G(x, x') = \frac{cos\Theta_i cos\Theta'}{\|x - x'\|^2} \upsilon(x, x'),$$

$$\upsilon(x, x') = \begin{cases} 1 & \text{se } x \text{ e } x' \text{ são} \\ & \text{mutuamente visíveis,} \\ 0 & \text{caso contrário.} \end{cases}$$

- $L_s(x, k)$: radiação na direção k com origem na posição x da superfície;
- L_e: radiação emitida de um ponto. L_e deixa de ser zero apenas quando x for uma fonte de luz;
- $\rho(k_i, k_o)$: BRDF[1]. Contribuição da radiação pela reflexibilidade da superfície
- $G(x, x')$: relacionamento geométrico entre duas superfícies;
- $\upsilon(x, x')$: função de visibilidade.

[1] BRDF, ou *Bidirectional Reflectance Distribution Function*, define como a luz é refletida em uma superfície.

A Equação 3.1 está formulada com algumas simplificações. O limite da integral, x', define as superfícies da cena. Assim, assume-se que a luz se propaga em linha reta, de forma instantânea e no vácuo.

A LTE calcula a radiação refletida em um ponto da superfície levando em consideração a radiação emitida da superfície, a BRDF e a distribuição da iluminação incidente (PHARR; HUMPHREYS, 2010). Ela simula o transporte da iluminação do mundo real, porém com algumas simplificações. A LTE assume que a luz é emitida, espalhada e absorvida apenas em superfícies, e desloca-se em linhas retas. Assim, efeitos como nuvens e fumaça, ou meios com um índice de refração variável, por exemplo o ar quente, são ignorados. Modelos quânticos também são ignorados, como comportamento de onda da luz. Assim, alguns fenômenos explicados pela teoria da eletrodinâmica quântica (QED) não são replicáveis pela LTE. Entretanto, a LTE pode ser estendida para suportar a velocidade da luz, difração, polarização e fluorescência (BIKKER, 2012).

3.2 Renderização com Rasterização

Atualmente, o algoritmo de rasterização é o mais usado para aplicações gráficas interativas. Ele é baseado nas técnicas de *scan conversion*[2] e *z-buffer* e define o processo de projeção de polígonos 3D em uma superfície plana 2D (BIKKER, 2012). Outra definição seria a de calcular todos os pixels em uma imagem para um polígono qualquer (SHIRLEY; MARSCHNER, 2009). Em sua essência, o comportamento do tempo de processamento da rasterização é linear em função ao número de triângulos, como na Figura 3.1.

[2] *Scan conversion* são técnicas que mapeiam uma função analítica e contínua em pixels.

Figura 3.1: Simplificação do algoritmo da rasterização

1: **para** $t \in$ {Lista de Triângulos} **faça**
2: calcule todos os pixels de t
3: **fim para**

Fonte: Produção do próprio autor.

Implementações comuns da rasterização impõem uma simplificação na representação de superfícies, pois a interpolação de superfícies complexas tende a ser computacionalmente cara. Cada superfície é transformada em uma malha de triângulos, possibilitando que o hardware se especialize no processamento individual desses triângulos. Uma desvantagem é que essa simplificação pode resultar em uma perda considerável na qualidade da imagem quando o número de triângulos usados é pequeno.

A rasterização processa um triângulo de cada vez, e cada triângulo não necessita de informações de outros triângulos, fazendo assim com que a rasterização seja uma solução eficiente para processamento paralelo. As APIs (*Application Programming Interface*) gráficas para aplicações interativas mais conhecidas são a Direct3D[3] e a OpenGL[4]. Implementações dessas APIs fazem uso da rasterização em seus núcleos e oferecem uma aceleração em hardware quando usadas com uma GPU, *Graphics Processing Unit*, moderna.

Uma das características da rasterização é a falta de suporte a informações globais, assim não implementa a recursividade e a integral da Equação 3.1. Isso faz com que vários efeitos importantes para o entendimento da cena não sejam calculados, como sombras e reflexos. Entretanto, o algoritmo pode ser estendido para aproximar efeitos que usam informações globais, como iluminações indiretas, em troca de custo computacional (BIKKER, 2012).

[3] DirectX. Disponível em: <http://msdn.microsoft.com/directx>. Acesso em: 11/11/14.
[4] OpenGL. Disponível em: <http://opengl.org>. Acesso em: 11/11/14.

3.3 Renderização com Ray Tracing

Renderização é um processo que tem como entrada um conjunto de objetos e produz uma lista de pixels como saída. Esse processo pode ser feito em duas maneiras gerais: *object-order rendering* e *image-order rendering* (SHIRLEY; MARSCHNER, 2009). *Object-order rendering* processa cada objeto em ordem, como mostrou a Figura 3.1 com a rasterização. Já a *image-order rendering* processa cada pixel em ordem, que é o caso do algoritmo de *ray tracing*.

O algoritmo de *ray tracing* básico funciona calculando cada pixel em ordem, com o objetivo de encontrar o objeto que é visto na posição do pixel na imagem. Primeiramente é calculado o raio de visão, que é uma reta que nasce na posição do observador e segue na direção que o pixel está olhando. Esse raio de visão busca interseção com o objeto mais próximo a câmera pois esse é o objeto que bloqueia a visão para o resto da cena. Uma vez que a interseção é encontrada, é feito o cálculo de sombreamento para determinar a cor do pixel (SHIRLEY; MARSCHNER, 2009). A Figura 3.2 mostra em pseudo-código o funcionamento do algoritmo de *ray tracing* básico.

Figura 3.2: Estrutura de um programa de *ray tracing* básico ou *ray cast*

```
1: para cada pixel faça
2:     calcule o raio de visão
3:     se raio colide com um objeto então
4:         calcule a normal com a superfície
5:         calcule a cor do pixel usando o modelo de sombreamento
6:     senão
7:         coloque a cor do pixel como a cor do plano de fundo
8:     fim se
9: fim para
```

Fonte: Traduzido de (SHIRLEY; MARSCHNER, 2009).

O algoritmo de *ray tracing* básico faz somente a travessia de raios primários. Ou seja, uma vez que a interseção com o objeto mais próximo

a câmera é encontrada, não é feito o lançamento de raios secundários de forma recursiva. É importante ressaltar que esse estilo de *ray tracing* tem diferentes nomes na literatura. Shirley e Marschner (2009) chamam essa técnica de *ray tracing*, enquanto técnicas que lançam raios secundários de forma recursiva são chamadas de *ray tracing* recursivo. Já Hughes et al. (2013) chamam a técnica que usa apenas raios primários de *ray cast* e a técnica que usa raios recursivos de *ray tracing*. Outros autores, como Bikker (2012), usam esses termos de forma intercambiável. Para deixar claro a distinção desses termos, este trabalho chama o algoritmo que lança apenas raios primários de *ray tracing* básico ou *ray cast*, já para algoritmos que lançam raios de forma recursiva o termo *ray tracing* recursivo será usado.

O teste de interseção usado nos algoritmos de *ray tracing* pode apresentar uma complexidade $O(n)$, onde n é o número de objetos na cena. Entretanto, é possível fazer uso de estruturas de dados espaciais para eliminar testes desnecessários, assim tornando a complexidade logarítmica. Por causa disso, o algoritmo do *ray tracing* básico pode apresentar complexidade $O(k \log n)$, onde k é o número de pixels da imagem, e n o número de objetos na cena (PHARR; HUMPHREYS, 2010). Porém, a criação dessa estrutura espacial precisa, no mínimo, processar todos os objetos, logo, essa criação apresenta complexidade linear.

Essa diferença de complexidade faz com que *ray tracing* trabalhe de forma diferente da rasterização. Como a rasterização processa triângulos com complexidade linear, então geralmente se usa etapas de pré-processamento para reduzir o número de triângulos antes deles serem processados pela rasterização (GOBBETTI et al., 2008). Já o algoritmo de *ray tracing* faz a eliminação desses objetos durante a renderização, porém requer a construção de uma estrutura de dados aceleradora. Alguns trabalhos (WALD et al., 2001; DIETRICH et al., 2007) na literatura sugerem que o algoritmo de *ray tracing* é mais adequado do que a rasterização para cenas complexas. Mas, como foi discutido no início deste capítulo, ambas as técnicas estão sendo usadas para visualização de ce-

nas complexa.

Uma das principais diferenças entre *ray tracing* e rasterização é que *ray tracers* são mais flexíveis pois não estão limitado a renderização apenas de malhas de polígonos. Qualquer primitiva geométrica que permita calcular intersecção com uma linha 3D pode ser usada em *ray tracers* (SHIRLEY; MARSCHNER, 2009), incluindo esferas, torus e superfícies de Bézier, por exemplo.

Ray tracers são geralmente usados de forma *offline*, onde o tempo de processamento rápido de cada quadro não é um requisito. Ao contrário da rasterização, o *ray tracing* não é acelerado em hardware em placas de vídeo usadas por consumidores finais. Entretanto, é possível a renderização interativa usando *ray tracers* por meio de técnicas de processamento paralelo (BENTHIN et al., 2011), computação distribuída (WALD et al., 2001; DIETRICH et al., 2007; BENTHIN, 2006), GPGPU[5] (HANNIEL; HALLER, 2011), ou CUDA (BIKKER, 2012).

3.3.1 Whitted Ray Tracing

Whitted Ray Tracing, ou WRT, define o algoritmo de *ray tracing* recursivo proposto por Whitted (1980). Ao adicionar recursividade no algoritmo de *ray tracing* básico, Whitted incrementa a qualidade da imagem com alguns efeitos de iluminação indireta nas superfícies, porém apenas para a transmissão especular em superfícies refletoras. Com esse avanço, três novos efeitos podem ser simulados: reflexão, refração e sombras.

Uma das principais características do trabalho de Whitted é a simplificação da integral da LTE. Ao invés de calcular a integral do hemisfério da LTE, WRT trata cada fonte de luz como um ponto no espaço e soma a radiação de cada um desses pontos.

[5] Este trabalho usa a mesma definição de GPGPU feita por (KIRK; HWU, 2010) que é a computação usando GLSL ou HLSL, que são *shaders* usados pela DirectX ou OpenGL, respectivamente.

O método WRT não gera imagens foto-realistas pois não implementa a LTE completa. Além disso, a imagem final apresenta *hard-shadows* e *aliasing*. *Hard-shadows* é um problema comum na computação gráfica onde um objeto pode estar totalmente em sombra, ou totalmente sem sombra. Essa característica não reflete a realidade onde existem sombras suaves e um período de transição entre regiões iluminadas e sombras. Já *aliasing* neste contexto é a criação de efeitos tipo escada em regiões da imagem que deveriam representar linhas retas ou superfícies suaves.

3.3.2 Ray Tracing Distribuído

O *ray tracing* recursivo WRT gera imagens limpas com *hard-shadows*, reflexos sempre nítidos e tudo parece em foco. O problema é que essa imagem não retrata a realidade. Esses efeitos são reduzidos usando o *ray tracing* distribuído (SHIRLEY; MARSCHNER, 2009), ou *distributed ray tracing*, técnica que foi introduzida por Cook et al. (1984).

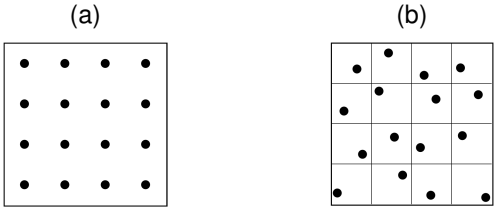
Ray tracing distribuído é uma adição simples ao WRT para suportar os seguintes efeitos: *antialiasing*, penumbras[6], profundidade de campo, reflexos mais realistas e *motion blur*. *Motion blur* é o efeito visual que tenta representar objetos em movimentos em uma imagem.

As imagens de saída de um *ray tracer* são malhas bidimensionais de pontos discretos. Já a radiação incidente no plano de projeção é uma função contínua. Essa diferença piora a qualidade da imagem, assim gerando os defeitos chamados de *aliasing*. A solução desse problema, *antialising*, pode ser conseguida usando a teoria de amostragem (PHARR; HUMPHREYS, 2010). A forma mais simples de adicionar *antialiasing* em *ray tracers* é lançando mais raios por pixel. A Figura 3.3a mostra a amostragem regular, onde os raios são distribuídos de forma uniforme. O problema com essa abordagem é que padrões regulares podem gerar interferências, como o padrão de Moiré. A Figura 3.3b apresenta uma solução mais adequada, onde cada amostragem é distribuída

[6] Também conhecido como *soft-shadows* em inglês.

de forma aleatória, porém limitada a um certo espaço, assim impedindo que raios sejam lançados muitos próximos um dos outros. Essa solução é chamada de amostragem estratificada (SHIRLEY; MARSCHNER, 2009). Outras técnicas existem, como *low-discrepancy sampling*, *Poisson disk pattern* e o algoritmo *best-candidate* de Mitchell, porém essas tendem a aumentar o custo computacional na geração da amostragem. A amostragem adaptativa também pode ser usada para tentar gerar imagens de alta qualidade aumentando o número de amostras em regiões mais complexas da imagem (PHARR; HUMPHREYS, 2010).

Figura 3.3: Amostragem de *pixels* para *antialising* em *ray tracers*



Fonte: Adaptada de (SHIRLEY; MARSCHNER, 2009).

Fontes de luz no WRT não apresentavam área, sendo apenas um ponto no espaço ou apenas uma direção. Isso implica que um ponto em uma superfície pode estar complemente na sombra, ou completamente fora da sombra, ou seja, não há valores intermediários. Por causa disso, o método WRT gera *hard-shadows*. O *ray tracing* distribuído resolve esse problema dando área para as fontes de luz. Para testar se uma superfície está ou não em sombra, um raio é lançado em direção a um ponto aleatório na luz. Essa simples modificação no algoritmo irá criar penumbras porque cada ponto na luz tem apenas uma fração da intensidade original (SHIRLEY; MARSCHNER, 2009).

Ray tracers tradicionais criam uma imagem com todos os objetos em um foco perfeito, não simulando assim a realidade. Isso acontece porque a técnica usada assume que a luz incidente no olho do observador

se concentra em um único ponto. O *Ray tracing* distribuído simula profundidade de campo introduzindo uma lente de tamanho maior que zero em frente ao olho do observador (COOK et al., 1984).

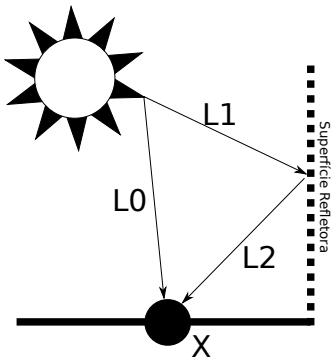
Nem todas as superfícies refletem a luz como um espelho perfeito. Algumas apresentam reflexos borrados, um meio termo entre reflexões especulares e difusas. Para simular esse efeito de reflexão, o *ray tracing* distribuído calcula de forma aleatória o raio do reflexo. Quanto maior o ângulo entre esse novo raio e o equivalente a de um espelho perfeito, maior será o borramento. Uma variável pode controlar os limites dos valores aleatórios, assim controlando a intensidade do borramento (SHIRLEY; MARSCHNER, 2009).

Quando um objeto está em movimento no momento que uma câmera tira uma foto, ele aparece borrado na imagem final. Isso acontece por que o tempo de exposição da câmera a cena não é igual a zero. Em *ray tracers* esse efeito pode ser simulado criando uma cena com movimentos e incluindo informação de tempo em cada raio. Dado um tempo, o raio pode testar intersecção contra um objeto nesse tempo, e assim criar o efeito de movimento, *motion blur* (SHIRLEY; MARSCHNER, 2009).

3.3.3 Método de Pathtracing

WRT (Seção 3.3.1) e *ray tracing* distribuído (Seção 3.3.2) são técnicas que solucionam o problema da iluminação direta e parcialmente o da indireta. Elas simulam a iluminação indireta em reflexos especulares perfeitos, porém, outras situações, como da Figura 3.4, não são simuladas (SUFFERN, 2007). A iluminação indireta é importante para a qualidade final da imagem pois muitas superfícies no mundo real recebem a maior parte de sua iluminação por meio de superfícies refletoras. O problema é que qualquer superfície pode refletir a iluminação de qualquer outra superfície, assim resultando em um algoritmo com complexidade $\mathcal{O}(n^2)$, onde n é o número de superfícies. Esse problema da iluminação indireta é normalmente conhecido como iluminação global (SHIRLEY;

Figura 3.4: Exemplo de iluminação direta (L0) e indireta (L2) no ponto x



Fonte: Adaptada de (SUFFERN, 2007).

MARSCHNER, 2009).

O algoritmo *pathtracing* é uma solução probabilística para calcular a iluminação direta e indireta. Ele lança raios da câmera e os segue até que uma fonte de luz seja encontrada, ou uma distância máxima seja alcançada ou até que os raios saiam da cena. Cada vez que um raio intersecciona com um objeto, uma função probabilística gera a nova direção do raio levando em consideração a BRDF da superfície (SUFFERN, 2007).

Pathtracers suportam qualquer tipo de BRDF, incluindo superfícies transparentes, tornando essa solução flexível e capaz de simular diversos tipos de transporte de iluminação entre superfícies (SUFFERN, 2007). A principal desvantagem da técnica é que ela pode gerar imagens com ruídos. Isso por que uma superfície é iluminada apenas quando um raio encontra uma fonte de luz (SHIRLEY; MARSCHNER, 2009). Por exemplo, alguns caminhos são bloqueados por superfícies, enquanto outros nunca encontram uma fonte de luz ou a luz está simplesmente muito longe para influenciar no resultado. Esses problemas não apenas custam tempo de processamento, como introduzem erros e ruídos na imagem final.

Uma variação de *pathtracing* é o algoritmo de *lighttracing*, onde os raios fazem o caminho da fonte de luz até a câmera. A combinação dessas duas técnicas é o *bidirectional pathtracing*, onde um caminho é construído com origem na fonte de luz, e outro com origem na câmera. Após algumas intersecções os dois caminhos se conectam, assim criando um caminho completo (BIKKER, 2012). *Bidirectional pathtracing* apresenta bons resultados em cenas onde as superfícies são, na maior parte, iluminadas de forma indireta. Entretanto, em cenas com a maior parte iluminada de forma direta, *bidirectional pathtracing* deixa de ser eficiente, podendo até ser computacionalmente mais caro que o algoritmo de *pathtracing* (LAFORTUNE, 1996).

Matematicamente, *pathtracing* resolve a integral da LTE usando métodos de Monte Carlo. Esses métodos referem-se a uma série de métodos estatísticos para encontrar soluções para problemas como calcular o valor esperado de uma função, ou integrar funções que não são possíveis de serem resolvidas de forma analítica, como a LTE (Equação 3.1).

Métodos de Monte Carlo calculam um valor estimado de uma integral com base em amostras aleatórias. A incerteza do resultado depende da variância, que é inversamente proporcional ao número de amostras (LAFORTUNE, 1996). Se o número de amostras for suficientemente grande, Monte Carlo pode calcular a resposta certa de acordo com a Lei dos Grandes Números (BIKKER, 2012).

Uma das vantagens de integrar usando Monte Carlo é a sua simplicidade, pois apenas duas operações são necessárias: amostragem e estimação. Amostragem está relacionada com a forma em que as variáveis aleatórias são escolhidas, enquanto a estimação é referente a função estimadora de Monte Carlo. Métodos de redução de variância referem-se a técnicas para construção de estimadores. Quanto mais eficiente for o estimador, mais baixa será a variância usando o mesmo número de amostras (VEACH, 1997).

O erro de um estimador de Monte Carlo padrão é proporcional a $1/\sqrt{N}$, onde N é o número de amostras (LAFORTUNE, 1996). Essa

proporção não é eficiente para computação gráfica interativa, onde cenas com fontes de luz pequenas podem fazer com que um raio necessite de muitas reflexões para encontrar uma fonte de luz. E mesmo quando encontrar, a influência dessa luz para a imagem final pode ser desprezível, pois cada reflexão reduz a energia da luz. Por isso, esforço precisa ser feito para encontrar raios que encontram a luz em menos etapas para melhorar a eficiência (BIKKER, 2012).

Métodos de Monte Carlo buscam reduzir a variância sem aumentar o número de amostras. Assim, os principais métodos de Monte Carlo são importantes para algoritmos de iluminação global: amostragem estratificada, amostragem de importância e múltiplos estimadores. Apenas uma breve descrição dos principais métodos será apresentada a seguir. A tese de Lafortune (1996) apresenta toda a formalização matemática desses métodos, e outros.

- Amostragem estratificada: Quanto mais os pontos amostrais se agrupam, menor será a eficiência do método. A amostragem estratificada tenta distribuir as amostras da forma mais uniforme possível. O método divide o domínio da integral em diversos subdomínios e estima integrais parciais em cada um desses subdomínios. Assim, há uma garantia que cada subdomínio terá pelo menos uma amostra. A variância da amostragem estratificada é sempre menor ou igual a variância do estimador normal. Como esse método não precisa ter conhecimento da integral, e o custo computacional extra é insignificante, geralmente acaba sendo uma boa otimização. Entretanto, quando o número de amostras é elevado, os pontos amostrais começam a se agrupar novamente, reduzindo assim a eficiência do método (LAFORTUNE, 1996).

- Amostragem de importância: faz uso da função densidade de probabilidade (PDF) para gerar mais amostras nas áreas onde a função da integral tem maior influência no resultado. O maior desafio é encontrar uma função PDF que aproxima a função da integral (LA-

FORTUNE, 1996).

- Múltiplos estimadores: múltiplos estimadores podem ser usados quando a construção de uma única PDF é difícil de ser feita. Assim, cada PDF aproxima uma seção da função da integral (LAFORTUNE, 1996).

Pathtracing pode ser categorizado com viés ou sem viés, ou *biased* e *non-biased* respectivamente. O termo viés vem da estatística e refere-se a amostra viesada. Um *pathtracing* com viés tende a convergir ao resultado final mais rapidamente, mas pode apresentar erros sistemáticos na aproximação da radiação. Assim, a versão com viés é a melhor escolha para sistemas interativos, enquanto a sem viés geralmente é usada como referência.

3.3.4 Outras Técnicas de Ray Tracing

Uma especialização do *pathtracing* é a técnica *Metropolis Light Transport*, ou MLT (VEACH; GUIBAS, 1997). Ela funciona implementando uma nova função de amostragem de importância em um *path tracer* bidirecional. Essa nova função gera amostras levando em consideração a contribuição do caminho para a imagem. Em cenas complexas onde as luzes são na grande maioria indiretas, o algoritmo MLT irá encontrar os caminhos mais relevantes, ou seja, os caminhos que mais contribuem para a iluminação da imagem final, e irá fazer pequenos ajustes nesses caminhos para calcular o resultado final. Assim, um maior esforço computacional é gasto nos caminhos mais importantes.

Photon-Mapping é outra técnica para resolver o problema de iluminação global. Ela funciona em duas etapas. A primeira etapa constrói um *photon map* emitindo fótons da fonte de luz em direção ao modelo e armazenando informações nas superfícies tocadas por esses fótons. Essas informações são então usadas na segunda etapa, otimizando os métodos de amostragem de caminhos. A segunda etapa é a própria renderização da cena. Ela é executada usando um *ray tracer* distribuído onde

os raios são lançados do olho em direção a cena. *Photon-mapping* também pode gerar um *photon map* separado para o cálculo de cáusticas, que é um efeito difícil de ser conseguido em *path tracers* por causa da quantidade de ruídos (JENSEN, 1996). A primeira etapa pode ser definida como um algoritmo de pré-processamento, e dependendo da qualidade pode ser uma técnica computacionalmente cara. Assim, a técnica não é adequada para cenas com superfícies e fontes de luzes dinâmicas.

3.4 Estruturas de Dados Aceleradoras para Ray Tracing

O *ray tracing* precisa conhecer os pontos das superfícies que interseccionam com os caminhos da luz para calcular a iluminação da cena. Uma implementação ingênua vai encontrar esses pontos com uma complexidade computacional $O(n)$, onde n é o número de superfícies, para cada caminho de luz. A complexidade do cálculo de interseções apresenta a função assintótica dominante em um *ray tracer*. Esse problema foi reportado no trabalho de Whitted (1980), onde em cenas complexas o *ray tracing* pode gastar mais de 95% de seu tempo no processamento de interseções. Logo, é importante acelerar o algoritmo de interseção entre raios e cena, e para isso diversas técnicas chamadas de estruturas aceleradoras foram propostas. Essas estruturas apresentam formas diferenciadas para organização dos objetos geométricos na cena.

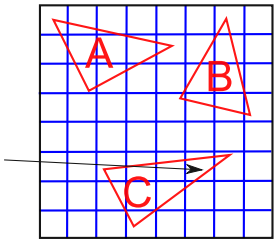
A Seção 3.4.1 apresenta a organização dos objetos da cena em uma grade, ou matriz 3D. Uma classe bastante usada na literatura é a de subdivisões espaciais, como a Octree e a KD-tree, que são descritas na Seção 3.4.2. A organização dos objetos em volumes hierárquicos também são importantes, e por isso é feita uma discussão sobre essa técnica na Seção 3.4.3.

3.4.1 Organização em Grade

Uma das formas mais simples para acelerar o teste de interseção é organizando as superfícies em uma grade, ou *grid*. A grade é uma ma-

triz 3D com elementos uniformemente espaçados e de mesmo tamanho. A Figura 3.5 mostra como os triângulos A, B e C são distribuídos em uma grade. O teste de interseção é acelerado pois apenas os triângulos contidos nas células em que o raio atravessa são verificados. Neste exemplo, 6 células e o triângulo C são testados por interseção, ignorando todos os outros triângulos. A sequência de células da grade a serem verificadas é tradicionalmente calculada usando o algoritmo 3DDDA, ou 3D *digital differential analyzer* (AMANATIDES; WOO, 1987).

Figura 3.5: Raio atravessando uma grade



Fonte: Produção do próprio autor.

As principais vantagens dessa estrutura são a sua simplicidade, construção rápida e acesso aos seus elementos com complexidade constante. Entretanto, a grade apresenta um consumo elevado de memória além de não oferecer nenhum mecanismo para acelerar a travessia de espaços homogêneos, como células vazias por exemplo.

3.4.2 Subdivisões Espaciais

A classe de subdivisões espaciais faz o particionamento do espaço onde os objetos da cena residem. Um raio fazendo a travessia da cena precisa apenas testar interseção com os objetos dentro das partições em que o raio atravessa, assim melhorando a travessia de cada raio, porém adicionando o tempo de criação dessa estrutura. Uma das principais desvantagem do método é a presença do mesmo objeto em mais de uma partição, assim aumentando o consumo de memória e também

do número de testes de interseção para o mesmo objeto (WÄCHTER; KELLER, 2006).

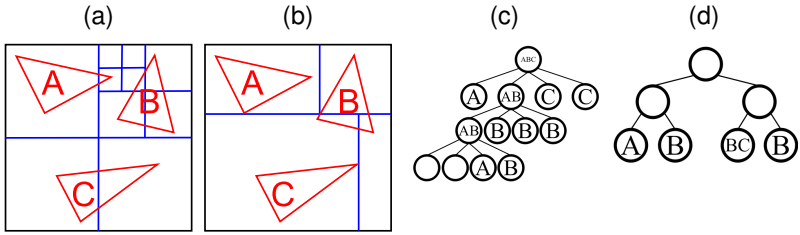
Alguns exemplos de estruturas de dados desta classe são a Octree e a KD-tree:

- Octree: Estrutura de dados descrita na Seção 2.7. A Figura 3.6c apresenta a Quadtree, que é o análogo 2D da Octree, construída para armazenar os triângulos A, B e C da Figura 3.6a. A Octree consegue melhorar a eficiência da travessia evitando testes de interseções desnecessários. Por exemplo, caso um raio atravesse a Quadtree como na Figura 3.7a então o teste de interseção precisa verificar apenas interseção entre o raio e o triângulo C. Na árvore da Figura 3.7b, apenas os três nós escurecidos e um triângulo são verificados durante o teste de interseção. Todos os outros triângulos que não estão contidos nas células da Quadtree em que o raio atravessa podem ser ignorados. Essa otimização é importante pois ela faz a redução da complexidade do teste de interseção de $O(n)$ para $O(\log n)$. O trabalho de Crassin et al. (2011) descreve um método eficiente com complexidade $O(n)$ para a construção da Octree em tempo-real, permitindo assim cenas dinâmicas.

- KD-tree: Estrutura de dados em forma de árvore para particionamento binário de um espaço k-dimensional. KD-tree é uma especialização de uma classe maior chamada *Binary Space Partition*, ou BSP. As BSPs subdividem o espaço usando planos arbitrários, enquanto a KD-tree usa somente planos alinhados aos eixos cartesianos (WÄCHTER; KELLER, 2006). Um exemplo de KD-Tree pode ser visto na Figura 3.6d, que é a representação em árvore da Figura 3.6b. Note que a KD-Tree pode fazer subdivisões onde os filhos possuem tamanhos diferentes, e cada divisão gera apenas dois filhos. O uso de subdivisões usando planos nos eixos alinhados reduz a complexidade da construção da árvore. Ao subdividir um volume, a posição do plano é importante, pois ela tem um im-

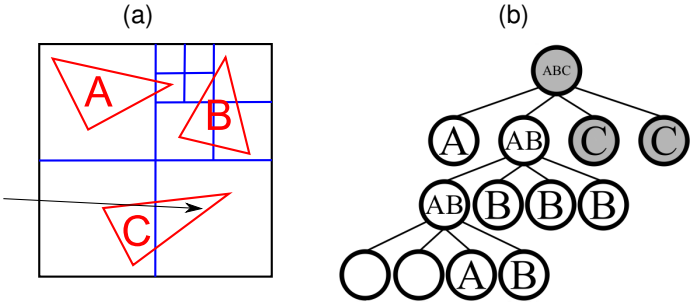
pacto na eficiência da travessia do raio e no consumo de memória. A SAH, ou *Surface Area Heuristic*, é uma estratégia bem conhecida para escolha da posição do plano da divisão introduzida por MacDonald e Booth (1990). A construção de uma KD-tree pode ser feita em $O(n \log n)$ (WALD; HAVRAN, 2006).

Figura 3.6: Exemplo de Quadtree e KD-Tree



Fonte: Produção do próprio autor.

Figura 3.7: Raio atravessando uma Quadtree



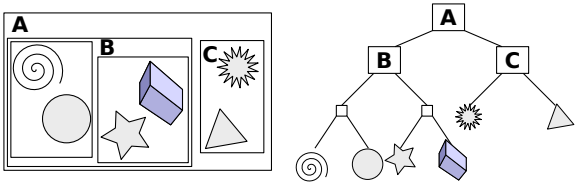
Fonte: Produção do próprio autor.

3.4.3 Bounding Volume Hierarchy

A *Bounding Volume Hierarchy*, ou BVH, é uma estrutura de dados em forma de árvore para objetos geométricos introduzida pelo trabalho (RUBIN; WHITTED, 1980). Geralmente os objetos geométricos são colocados dentro de volumes que apresentam cálculos de interseção eficientes, como ortoedros (também conhecidos como caixas) ou esferas (KAY; KAJIYA, 1986). Esses volumes formam os nós-folhas da árvore. Outras caixas ou esferas podem agrupar uma ou mais folhas, assim criando um volume maior. Esses volumes podem ser agrupados por outros volumes maiores, recursivamente. O nó raiz será um volume que engloba todos os outros volumes.

Um exemplo dessa organização pode ser visto na Figura 3.8. O nó A é o nó raiz, que é um volume que engloba todos os outros nós. Os nós B e C são nós filhos do nó A. Esses nós agrupam outros volumes que estão completamente contidos neles. Os nós folhas armazenam os objetos geométricos da cena. Os volumes de cada nó da BVH estão representados por caixas, que podem ser usadas para otimizar a travessia do raio na cena. O teste de interseção entre raio e caixa é eficiente, assim, por exemplo, um raio pode testar interseção com a caixa do nó C, e caso não haja interseção se elimina a necessidade de testar interseção com os objetos contidos em C (que podem conter um teste de interseção caro), pois é garantido que o volume do nó pai será sempre maior que de seus filhos em uma BVH.

Figura 3.8: Exemplo de Bounding Volume Hierarchy



Fonte: Schreiberx. Disponível em: <http://en.wikipedia.org/wiki/Bounding_volume_hierarchy>. Acesso em: 20/05/2015.

Cada objeto geométrico tem apenas uma instância na hierarquia, permitindo assim melhor previsibilidade no cálculo de consumo de memória e garante que apenas um teste de interseção será feito durante a travessia do raio. Entretanto, regiões de nós distintos em uma BVH podem compartilhar um mesmo espaço, e isso pode resultar em uma travessia ineficiente com interseção de primitivas na ordem errada ou travessia do mesmo espaço diversas vezes (WALD, 2004).

A BVH também pode usar SAH para melhores divisões, assim como a KD-tree. O número de divisões por nó pode variar, podendo ter apenas duas, um máximo de quatro, ou um número arbitrário de divisões (BIKKER, 2012).

Uma variação da BVH é a BIH, *Bounding Interval Hierarchy*, que é uma estrutura de dados proposta por Wächter e Keller (2006). Ao invés de armazenar caixas delimitadoras para cada nó, ela armazena apenas dois planos paralelos a um dos eixos cartesianos. Esses planos representam intervalos onde primitivas estão presentes. A principal vantagem do método é um custo de memória previsível enquanto mantem uma travessia tão eficiente quanto a da KD-tree.

3.5 Discussão do capítulo

As técnicas de *ray tracing* são flexíveis e capazes de renderizar qualquer representação de superfícies que suporte teste de interseção contra um raio. Já a rasterização com a OpenGL e DirectX geralmente são usadas para visualizar B-Reps simples, planares, e constituídas de triângulos.

Rasterização é o método mais usado para visualizações interativas para cenas simples. Entretanto, *ray casting* apresenta uma complexidade diferente da rasterização, e alguns trabalhos na literatura indicam que *ray casting* oferece melhor escalabilidade referente a complexidade do modelo 3D, porém ambas as técnicas são usadas atualmente para visualização interativa de modelos complexos. Para alcançar complexidade

sub-linear, a rasterização faz a redução do número de triângulos antes do processo de renderização, enquanto *ray casting* oferece complexidade sub-linear durante a renderização caso seja usada com estruturas de dados espaciais. Assim, a etapa de pré-processamento da rasterização geralmente é constituída de simplificação da malha de triângulos e/ou de técnicas para remoção de triângulos não visíveis. Já o pré-processamento usado em *ray casting* geralmente é a construção da estrutura de dados espacial.

Entre as técnicas de *ray tracing* há a troca de simplicidade e desempenho pela qualidade da imagem. Por exemplo, *ray casting* lança apenas um raio por pixel, e sem nenhuma recursão, porém isso faz com que essa seja a técnica com pior qualidade da imagem final. WRT também lança um raio por pixel, porém o algoritmo recursivo requer um número maior de testes de interseção. Com isso o custo computacional aumenta, mas efeitos como reflexo, refração e sombras são possíveis. Já o *ray tracing* distribuído e o *pathtracing* lançam vários raios por pixel, assim aumentando ainda mais o custo computacional, porém aumentando a qualidade da imagem final.

Além das diferentes técnicas de *ray tracing*, diferentes estruturas de dados podem ser usadas para acelerar a travessia dos raios. Grades são adequadas para cenas interativas pois elas oferecem acesso direto a superfícies e rápida construção. Já a Octree, KD-tree e BVH são técnicas que ocupam menos memória, apresentam algoritmos de travessia de raios mais eficiente que grades, porém a construção dessas árvores têm um custo computacional elevado.

Neste trabalho a visualização interativa dos modelos complexos gerados pela simulação de usinagem é feita usando *ray casting*. O uso de *ray casting* ao invés da rasterização se dá por causa da sua simplicidade, pois ela permite a visualização direta de campos de distância e a etapa de pré-processamento é apenas a construção da estrutura de dados aceleradora, diferente da rasterização que precisa de algoritmos complexos de simplificação de malha e remoção de triângulos não visíveis. Além disso,

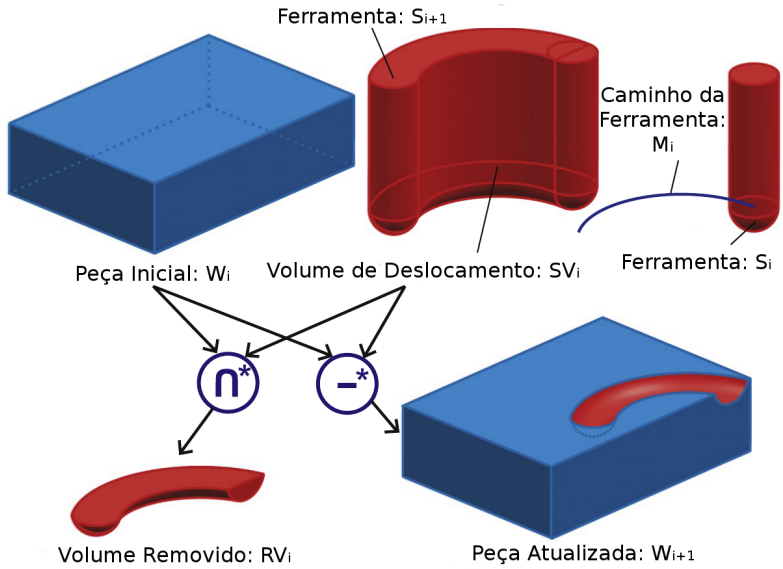
a principal vantagem da rasterização é sua aceleração em hardware, porém para fazer uso dela é necessário a sincronização do modelo geométrico com a GPU. Essa sincronização aumenta a complexidade pois ela pode ser uma etapa demorada. Além disso, a GPU apresenta uma restrição maior na quantidade de memória disponível para o modelo 3D, e assim aumenta a complexidade da implementação.

Campos de distância é a representação para os modelos 3D usados durante a usinagem, como discutido na Seção 2.8. A sua visualização direta com *ray casting* é possível pois existe o teste de interseção entre um raio e essa representação, como visto na Seção 2.6.6. Por causa disso, os modelos 3D não precisam ser transformados em uma malha poligonal antes de sua visualização, que é uma etapa custosa e necessária caso tenha sido usada a rasterização.

4 MANIPULAÇÃO GEOMÉTRICA

A simulação de usinagem geométrica tem como finalidade computar o modelo geométrico da peça durante, e após o término da usinagem. Esse modelo é construído removendo materiais com operações booleanas regularizadas de subtração entre a peça e o volume de deslocamento da ferramenta. A Figura 4.1 mostra as etapas desse processo. Primeiramente é definida a peça inicial, W_i, onde i é o índice da iteração do processo. O volume de deslocamento da ferramenta, SV_i, é construído usando a geometria da ferramenta, S_i, e o seu caminho percorrido, M_i. A peça atualizada, W_{i+1}, é então calculada fazendo uma operação booleana de subtração entre SV_i e W_i. Esse processo se repete até o fim dos movimentos da ferramenta.

Figura 4.1: Processo de Simulação de Usinagem



Fonte: Traduzido de (SULLIVAN et al., 2012).

Um sistema gráfico para verificação e simulação de usinagem NC contém dois componentes principais responsáveis pela manipulação geométrica: a) Computação da geometria do volume de deslocamento da ferramenta para cada um de seus movimentos; b) Implementação da operação booleana de subtração do volume de deslocamento com a peça inicial (DU et al., 2005; WANG; WANG, 1986). Este capítulo apresenta a proposta de implementação desses componentes.

Com base nas discussões da Seção 2.8 este trabalho representa a geometria da peça usando campos de distância sinalizados (SDF). A sua principal vantagem está na sua eficiência para realização de operações booleanas, que é um aspecto importante em uma simulação de usinagem. Além disso, ela oferece precisão superior a de *voxels* binários.

Os volumes da peça e da ferramenta são armazenados em uma estrutura de dados para *narrow-band level sets*. A escolha da estrutura de dados é fundamental pois ela afeta todo o sistema, desde a criação das geometrias até as operações booleanas e visualização do modelo. A Seção 4.1 apresenta como o modelo geométrico é representado em memória. Geralmente, a literatura escolhe a Octree para organizar essa representação em memória, como visto na Seção 2.7, porém, a estrutura de dados VDB (MUSETH, 2013) foi usada neste trabalho pois ela oferece acesso $O(1)$ em média, além de manter a maioria dos benefícios da Octree. A Seção 4.2 descreve como as operações booleanas são feitas usando SDF e como a escolha da estrutura de dados pode acelerar o algoritmo.

Diversos volumes foram usados como primitivas neste trabalho, como esfera, cilindro e caixas alinhada aos eixos (AABB). Esses volumes são inicialmente representados usando uma função de distância sinalizada e contínua. A Seção 4.3 descreve as funções de distâncias desenvolvidas neste trabalho.

A representação geométrica usando SDF e funções implícitas resultam em modelos não pré-processados, e com isso dificultando a visualização interativa. Por causa disso é usada uma representação discreta

para modelagem geométrica. A Seção 4.4 discute diversas técnicas para amostragem do volume original para transformação da função de distância contínua em uma representação discreta.

O Código-G, que é uma linguagem usada para especificar os movimentos da ferramenta e outras características da usinagem, é descrito na Seção 4.5. O Código-G é apresentado pois é importante conhecer como ele funciona para então entender como os volumes de deslocamento da ferramenta são criados.

Uma visão geral da arquitetura desenvolvida neste trabalho é apresentada na Seção 4.6, que explica como os componentes anteriores estão organizados. Por último é feita uma discussão do capítulo na Seção 4.7.

4.1 VDB para Simulação de Usinagem

Uma das etapas da simulação de usinagem é a criação de volumes de deslocamento da ferramenta. Geralmente a criação desses modelos requer acesso aos valores da estrutura de dados em intervalos regulares. Portanto, acesso rápido a esses valores, e de forma coerente, é desejável. Além disso, as operações booleanas entre campos de distância são feitas efetuando operações simples (*max* e *min*) por elemento. Ou seja, o tempo de acesso aos elementos das representações domina o tempo de execução das operações booleanas.

A estrutura de dados que armazena o modelo geométrico é o componente mais importante do sistema, pois ela impacta diretamente no desempenho da criação dos volumes de deslocamento, operações booleanas e na visualização do modelo. Este trabalho tem como foco principal a simulação geométrica de usinagem, assim, apenas a superfície do modelo da peça sendo usinada tem importância. Por causa disso, uma estrutura de dados eficiente para representação de superfícies usando campos de distância sinalizados (SDF) foi usada, essa chamada de *narrow-band level set* hierárquica, como visto na Seção 2.7.5. Essa estrutura pode ser

usada usando Octree, porém foi usada a VDB neste trabalho.

VDB (MUSETH, 2013) é uma estrutura de dados hierárquica e um conjunto de ferramentas para manipulação eficiente de dados volumétricos e dinâmicos discretizados em grades tridimensionais. A principal aplicação da VDB é na representação de *narrow-band level sets* hierárquicos em diversas áreas que requerem transformações dependente do tempo, como por exemplo na simulação de fluídos e efeitos especiais.

VDB é uma estrutura de dados hierárquica em forma de árvore, porém com profundidade pré-definida. Ou seja, essa propriedade é definida antes da criação da árvore e permanece constante durante seu uso. VDB assemelha-se com a B+tree, que funciona de forma similar a árvore de busca binária, porém seus nós podem ter mais de dois filhos. O fator de ramificação da VDB é variado, porém limitado a potências de dois. Essas características implicam que a árvore da VDB é de pouca profundidade e larga, e com isso acaba reduzindo o custo para travessia da árvore do nó raiz até o nó folha. Ao contrário da Octree que é limitada a apenas oito filhos por nó, assim tendendo a construir árvores profundas. A VDB também pode armazenar valores em nós intermediários, por isso funciona como uma estrutura de dados hierárquica, assim como a Octree.

Nós folhas da VDB armazenam blocos densos de dados. Esses blocos são grades com um número fixo de *voxels*. Por exemplo, neste trabalho os nós folhas são grades com $8 \times 8 \times 8$ *voxels*. Essa característica combinada com a pouca profundidade da VDB permite acesso rápido aos nós vizinhos de um *voxel*, pois, caso o vizinho pertence a mesma grade, o acesso é direto. Porém, mesmo no pior caso, por causa da pouca profundidade da árvore, o acesso será mais eficiente que o pior caso em uma Octree. Com isso a VDB oferece melhor coerência espacial de seus valores, permitindo melhor gerencia da memória *cache* e do *prefetcher* presente em processadores modernos. Ao contrário da Octree onde nós vizinhos podem estar armazenados em locais distantes em memória, e assim um acesso a um *voxel* vizinho tem chances maiores de invalidar a memória *cache*.

Além da profundidade pré-definida, a VDB também usa árvores com uma configuração pré-definida. Essa configuração determina quantas subdivisões um nó pode ter. A notação $(2^n_l)^3$, onde l é o nível da árvore e n é a dimensão do nó, é usada para indicar o número de divisões para um nó. Assim, a profundidade do nó é o que determina quantos filhos ele pode ter. Isso significa que, por exemplo, uma VDB com configuração $[6, 4, 3]$ irá ter nós com $(2^6)^3$ filhos cada no primeiro nível ($l = 0$), nós com $(2^4)^3$ filhos cada no segundo nível ($l = 1$) e nós com $(2^3)^3$ valores cada no último nível ($l = 2$), totalizando uma resolução de 8192^3 valores.

O nó raiz é uma exceção e não apresenta um número de divisões constante. A sua implementação é feita com uma *hash table* e introduz uma propriedade importante para VDB que é poder instanciar sub-árvores pré-definidas, como no exemplo anterior, em qualquer lugar no espaço, fazendo com que a VDB não tenha limites[1] na indexação de seus elementos. Após diversos testes empíricos, a configuração $[3, 5, 3, 3]$ foi usada neste trabalho pois foi a que apresentou melhor desempenho para visualização interativa.

O tempo de acesso para inserção, remoção e modificação dos valores nos nós folhas e intermediários na VDB é $O(1)$, porém, o nó raiz, que contém uma *hash table*, domina a função assintótica tendo uma complexidade de $O(\log n)$, onde n é o número de filhos do nó raiz. Na prática, a *hash table* gerencia poucas entradas e as chaves são bem espalhadas, fazendo com que o algoritmo apresente acesso aleatório aos valores da árvore em média em tempo constante (MUSETH, 2013).

Por causa do rápido acesso aos valores, porém ainda mantendo todas as vantagens da Octree, a VDB foi usada neste trabalho. O tempo constante para acessar valores de forma aleatória é usado pelo algoritmo de reconstrução da superfície que será discutido na Seção 5.4.

[1] A implementação atual é limitada pelos 32bits da representação de inteiros.

4.2 Operações Booleanas

Operações booleanas em *narrow-band level sets* são efetuadas executando operações binárias de max e min (veja Tabela 4.1) em cada elemento de dois modelos geométricos.

Tabela 4.1: Operações booleanas entre *narrow-band level sets* A e B

Operação	Representação Simbólica	Distância Combinada
Interseção	$dist(A \bigcap B)$	$min(dist(A), dist(B))$
União	$dist(A \bigcup B)$	$max(dist(A), dist(B))$
Subtração	$dist(A - B)$	$min(dist(A), -dist(B))$

Fonte: Adaptado de (SULLIVAN et al., 2012).

Este trabalho faz uso da implementação das operações booleanas da biblioteca OpenVDB[2], porém a Figura 4.2 apresenta um pseudocódigo independente de implementação da operação de união. Outras operações, como interseção e subtração, são implementadas da mesma forma, porém substituindo a distância combinada usando a Tabela 4.1.

O algoritmo pode ser acelerado usando a hierarquia da VDB. Caso um nó de A seja homogêneo, chamado de n_a^h, então se verifica se ele intersecciona algum nó de B que esteja no mesmo nível. Caso negativo, então basta copiar n_a^h para a árvore de saída do algoritmo, chamada de C. Caso exista um nó em B que não seja homogêneo e que intersecciona n_a^h, então é copiado esse nó de B para C, e é feita as operações em cada elemento, porém usando sempre o mesmo valor contido em n_a^h. O último caso é quando ambos nós de A e B são homogêneos, então não há nada que possa ser feito para otimizar, e a execução procede como no algoritmo da Figura 4.2.

[2] OpenVDB. Disponível em: <http://www.openvdb.org/>. Acesso em: 27/10/2014.

Figura 4.2: União de dois *narrow band level sets* A e B

Requisitos: $A \leftarrow$ modelo VDB com profundidade 4
Requisitos: $B \leftarrow$ modelo VDB com profundidade 4
1: $raiz_A \leftarrow$ raiz de A
2: $raiz_B \leftarrow$ raiz de B
3: **para** $n_a^1 \in raiz_A$ e $n_b^1 \in raiz_B$ **faça**
4: {n_a^l, onde l representa o nível do nó}
5: **se** n_a^1 intersecciona n_b^1 **então**
6: **para** $n_a^2 \in n_a^1$ e $n_b^2 \in n_b^1$ **faça**
7: **se** n_a^2 intersecciona n_b^2 **então**
8: **para** $n_a^3 \in n_a^2$ e $n_b^3 \in n_b^2$ **faça**
9: {n_a^3 são valores nas folhas de A}
10: $C \leftarrow C \bigcup \{max(n_a^3, n_b^3) | n_a^3$ intersecciona $n_b^3\}$
11: **fim para**
12: **fim se**
13: **fim para**
14: **fim se**
15: **fim para**
16: **retorne** C

Fonte: Baseado na implementação da OpenVDB.

4.3 Funções de Distâncias Implícitas para Fresas e Peça Inicial

Durante a simulação de usinagem os volumes da peça inicial e da ferramenta precisam ser criados. Para realizar as operações booleanas, esses volumes precisam estar organizados em uma VDB, como visto na Seção 4.1. Porém, a VDB precisa saber a distância entre a localização de seus voxels e a superfície mais próxima. Essa distância é calculada usando funções implícitas, que apresentam uma alta precisão e eficiência.

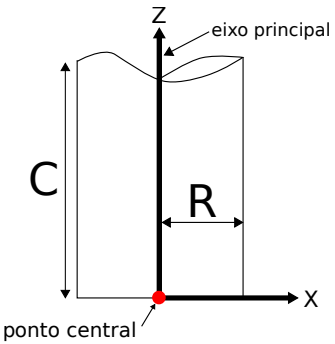
Esta seção descreve funções implícitas de fresas, que são ferramentas de usinagem, nas Seções 4.3.1, 4.3.2 e 4.3.3. A peça inicial é definida por uma função de distância de uma caixa alinhada aos eixos, como é mostrado na Seção 4.3.4. As funções implícitas das fresas são usadas para gerar o volume de deslocamento da ferramenta. A matemá-

tica dessa operação está descrita na Seção 4.3.5.

4.3.1 Fresas como Função de Distâncias

Funções de distâncias podem representar a geometria de fresas com uma alta precisão e eficiência. Isso se dá pelo uso de funções implícitas. Uma ferramenta de topo plano, ou *flat endmill*, pode ser representada pela função implícita de um cilindro, como mostra a Figura 4.3. Nesta imagem, C representa o comprimento da ferramenta, e R o raio do cilindro.

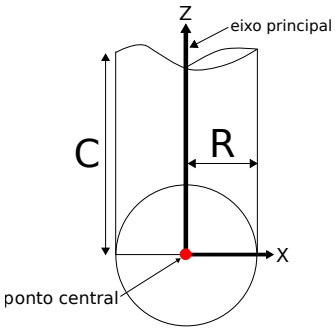
Figura 4.3: Representação da Ferramenta de Topo Plano



Fonte: Baseado em (YAU et al., 2005).

Usando a operação de união, como visto na Seção 4.2, é possível a representação de fresas mais complexas, como a de topo esférico, ou *ball endmill*. Essa ferramenta é uma composição de duas funções implícitas, a função de um cilindro com a de uma esfera, como pode ser visto na Figura 4.4. Nesta representação, o raio do cilindro é compartilhado com o raio da esfera.

Figura 4.4: Representação da Ferramenta de Topo Esférico



Fonte: Baseado em (YAU et al., 2005).

4.3.2 Função Implícita da Esfera

A função de distância sinalizada de uma esfera representada pelo seu centro $\mathbf{c}$, raio r e um ponto $\mathbf{p}$ qualquer é dada por:

$$dist_{esfera}(\mathbf{p}) = \sqrt{(\mathbf{c}-\mathbf{p})\cdot(\mathbf{c}-\mathbf{p})} - r \tag{4.1}$$

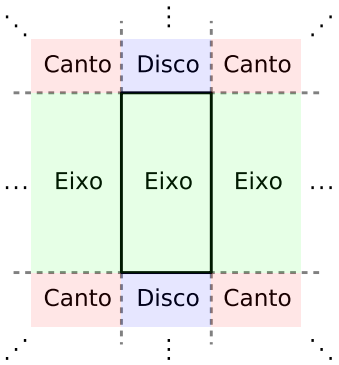
4.3.3 Função Implícita do Cilindro

O trabalho de Accary e Galin (2004) apresenta uma forma de resolver o problema da distância de um ponto $\mathbf{p}$ qualquer a um cilindro dividindo-o em partes. A Figura 4.5 apresenta os três casos em que um ponto $\mathbf{p}$ qualquer pode estar em relação a um cilindro. Classificando esse ponto em um dos três casos permite identificar qual função de distância pode ser usada para saber a distância do ponto até o cilindro.

Dado um cilindro definido pelo seu raio r e seu comprimento l, e um ponto $\mathbf{p}$ classificado a um dos casos descritos anteriormente, o cálculo da distância sinalizada é então feito seguindo as seguintes regras:

- Eixo: Distância entre $\mathbf{p}$ e o eixo principal central do cilindro, menos seu raio. Note que a distância é o tamanho do segmento de reta

Figura 4.5: Classificação dos pontos para um Cilindro



Fonte: Produção do próprio autor.

perpendicular ao eixo principal que inicia em $\mathbf{p}$ e termina no eixo principal;

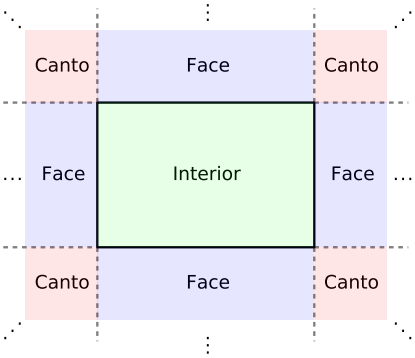
- Disco: Distância entre $\mathbf{p}$ e sua projeção no disco;
- Canto: Menor distância entre $\mathbf{p}$ e o círculo ao redor do disco do cilindro.

4.3.4 Função Implícita da Caixa Delimitadora Alinhada aos Eixos

A função de distância sinalizada de uma caixa delimitadora alinhada aos eixos (AABB) retorna a menor distância entre um ponto e suas seis faces. Uma forma de resolver esse problema é decompondo-o em três casos. A Figura 4.6 apresenta os casos que um ponto pode estar relativo a AABB. Cada um desses casos é mapeado a uma função de distância sinalizada.

Dada uma AABB e um ponto $\mathbf{p}$ classificado a um dos casos descritos anteriormente, o cálculo da distância sinalizada é então feito seguindo as seguintes regras:

Figura 4.6: Classificação dos pontos para AABB



Fonte: Produção do próprio autor.

- Interior: Menor distância entre $\mathbf{p}$ e uma das faces em linha reta. Calcula a distância com o sinal negativo;
- Face: Idem anterior, porém com o sinal positivo;
- Canto: Menor distância entre $\mathbf{p}$ e uma das arestas da AABB. Sinal será sempre positivo.

A classificação do ponto $\mathbf{p}$ é feita usando o algoritmo da Figura 4.7.

Figura 4.7: Classificação dos pontos para AABB

Requisitos: $\mathbf{l}, \mathbf{h} =$ Ponto inferior, e superior, de uma AABB
Requisitos: $\mathbf{p} =$ Ponto usado para calular a distância
1: **se** $\mathbf{p} \geq \mathbf{l}$ e $\mathbf{p} \leq \mathbf{h}$ **então**
2: **retorne** ponto está no interior
3: **fim se**
4: **se** Ao menos duas dimensões de $\mathbf{p}$ estão dentro da AABB **então**
5: **retorne** ponto está diretamente em frente de uma face
6: **fim se**
7: **retorne** ponto está em um dos cantos

Fonte: Produção do próprio autor.

4.3.5 Volumes de Deslocamento

Um volume de deslocamento define o volume gerado pela movimentação de um objeto arbitrário ao longo de um caminho arbitrário com possíveis rotações arbitrárias (ABDEL-MALEK et al., 2006). O volume de deslocamento de um conjunto arbitrário de pontos S ao longo de um caminho M em um espaço Euclidiano de d dimensões é geralmente formulado como uma união infinita expressa formalmente como (SULLIVAN et al., 2012):

$$sweep(S, M) = \bigcup_{q \in M} S^q \tag{4.2}$$

onde S^q denota o conjunto S posicionado de acordo com uma configuração q do deslocamento $M(t)$, sendo $t \in [0, 1]$ o parâmetro de tempo da movimentação. $M(t)$ define o deslocamento com transformações de corpos rígidos. Essas transformações podem ser representadas em matrizes com coordenadas homogêneas:

$$M(t) = \begin{bmatrix} R(t) & T(t) \\ 0 \dots 0 & 1 \end{bmatrix} \tag{4.3}$$

onde $R(t)$ é uma matriz de rotação ortogonal tridimensional 3×3 dependente do tempo t, e $T(t)$ é um vetor de translação tridimensional também dependente a t.

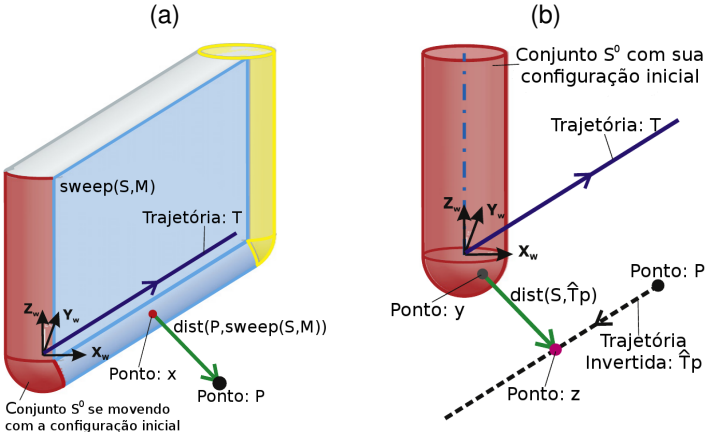
Para qualquer conjunto S simétrico movendo-se ao longo do caminho $M(t)$, como visto na Figura 4.8a, a distância de um ponto $\mathbf{P}$ até a superfície do volume de deslocamento $sweep(S, M(t))$ é definida por (SULLIVAN et al., 2012):

$$dist(\mathbf{P}, sweep(S, M(t))) = \inf_{q \in \partial sweep(S, M(t))} \parallel \mathbf{q} - \mathbf{P} \parallel \tag{4.4}$$

onde $\partial sweep(S, M(t))$ representa os pontos contidos na superfície do volume de deslocamento. Assim, encontrar os campos de distância do volume de deslocamento requer calcular a superfície do volume, que pode ser uma tarefa difícil. Por isso este trabalho faz uso de uma abordagem diferente, onde ao invés de movimentar o volume da peça e manter o

ponto $\mathbf{P}$ parado, se move o ponto e mantêm a peça parada. Uma analogia seria ao invés de mover a ferramenta enquanto a peça sendo usinada permanece estática, é movida a peça enquanto a ferramenta permanece parada.

Figura 4.8: Distância de um volume de deslocamento



Fonte: (SULLIVAN et al., 2012).

Essa nova abordagem está apresentada na Figura 4.8b, onde ao invés de deslocar a ferramenta em uma trajetória T, como na Figura 4.8a, é feito o deslocamento do ponto $\mathbf{P}$ em uma trajetória inversa definida por:

$$\hat{T}_P = \{\mathbf{P}^b \mid b \in \hat{M}\} \tag{4.5}$$

onde $\hat{M}$ é o deslocamento inverso, ou seja, $\hat{M}(t)$ é a matriz inversa de $M(t)$ para todos os valores de t (ERDIM; ILIEŞ, 2008). Assim, para o mesmo problema descrito pela Equação 4.4, considere a curva da trajetória inversa $\hat{T}_P$ do ponto $\mathbf{P}$ e um conjunto S. A função de distância mínima de um ponto $\mathbf{y}$ na superfície de S até um ponto $\mathbf{z}$ na trajetória inversa $\hat{T}_P$ é então definida como (SULLIVAN et al., 2012):

$$dist(S, \hat{T}_P) = \min_{\mathbf{y} \in \partial S, \mathbf{z} \in \hat{T}_P} \| \mathbf{y} - \mathbf{z} \| \tag{4.6}$$

Ao usar essa abordagem da trajetória inversa, como mostra a Figura 4.8b, a mesma distância pode ser encontrada sem precisar calcular a superfície do volume de deslocamento, ou seja:

$$dist(\mathbf{P}, sweep(S, M(t))) = dist(S, \hat{T}_P) \tag{4.7}$$

Essa identidade pode então ser usada para simplificar o cálculo do volume de deslocamento, pois ao invés de buscar a distância de um ponto até a superfície do volume de deslocamento, basta encontrar a menor distância do ponto em movimento ($\hat{M}(t)$) com a superfície do volume estático ($M(0)$).

A Seção 4.3.5.1 descreve como usar essa abordagem para encontrar a distância entre um ponto e o volume de deslocamento de uma esfera com movimentação linear de forma analítica. Já a Seção 4.3.5.2 apresenta uma solução para distância entre um ponto e um volume de deslocamento de um cilindro com movimentação linear usando métodos numéricos iterativos.

4.3.5.1 Esfera com Movimentação Linear

Usando a propriedade da Equação 4.7 é possível simplificar o problema da distância entre um ponto arbitrário e um volume de deslocamento se movendo linearmente. Assim, basta encontrar a menor distância entre um segmento de reta e o volume em sua posição original.

Essa simplificação é importante pois ela permite encontrar a menor distância entre um segmento de reta e uma esfera de forma analítica e eficiente. Dada uma esfera com o centro $\mathbf{c}$ e raio r, se deslocando com $T(t)$, onde T é um vetor de translação e $t \in [0, 1]$, então a menor distân-

cia entre um ponto $\mathbf{p}$ e a esfera é dada com a seguinte equação:

$$
\begin{aligned}
t_0 &= (\hat{T} \cdot (\mathbf{c} - \mathbf{p}))/(\hat{T} \cdot \hat{T}), \\
t_1 &= \begin{cases} 0 & \text{caso } t_0 \leq 0, \\ 1 & \text{caso } t_0 \geq 1, \\ t_0 & \text{caso contrário.}, \end{cases} \\
dist(S_e, \mathbf{p}) &= \sqrt{(\mathbf{p} + \hat{T}t_1) \cdot (\mathbf{p} + \hat{T}t_1)} - r
\end{aligned}
\tag{4.8}
$$

onde S_e é o volume de deslocamento da esfera se movendo com a transformação $T(t)$, e $\hat{T}$ é a transformada inversa de $T(1)$. A matriz $M(t)$ usada na Seção 4.3.5 foi simplificada para o vetor de translação $T(t)$ pois a esfera está se deslocando de forma linear. Assim, a transformada inversa $\hat{T}$ é apenas a negação dos valores de $T(1)$.

4.3.5.2 Cilindro com Movimentação Linear

A função de distância entre um ponto e um volume de deslocamento de um cilindro se movimentando de forma linear também pode ser simplificada usando a Equação 4.7. Essa simplificação transforma a função de distância de forma que ela retorna a menor distância entre um segmento de reta e um cilindro.

Dado um cilindro S_c com sua posição descrita pelo vetor de translação $T(t)$ onde $t \in [0, 1]$, e um ponto $\mathbf{p}$ qualquer, a menor distância entre os dois é definida por:

$$
dist(S_c, \mathbf{p}) = \min_{t \in [0,1]} (dist(S_0, \mathbf{p} + \hat{T}(t)))
\tag{4.9}
$$

onde S_0 é o cilindro S_c na posição $T(0)$, e $\hat{T}(t)$ é a transformada inversa de $T(t)$.

A Equação 4.9 descreve um sistema de minimização com uma variável $t \in [0, 1]$. O algoritmo de Brent (1973), que é um método numérico para solucionar equações lineares, foi usado neste trabalho para encontrar o valor de t que retorna a menor distância.

4.4 Representação Discreta da Função de Distância

A Seção 4.3 apresentou funções de distância implícitas e contínuas para as superfícies dos volumes das fresas e da peça inicial. A Seção 4.3.5 mostrou funções de distância para os volumes de deslocamento das ferramentas. Essas funções são então usadas para fazer a criação dos volumes que serão organizados em uma VDB e usados durantes a simulação de usinagem. Uma característica da VDB é que ela armazena *voxels* nos nós folhas, assim, é necessário a transformação das funções de distância contínuas em representações discretas.

Representações discretas de funções de distância contínuas são geralmente obtidas tirando amostras da função em intervalos regulares (SHIRLEY; MARSCHNER, 2009). O problema com essa abordagem é que diversas amostras distantes da superfície são avaliadas. Como apenas campos de distâncias próximos à superfície são armazenados em um *narrow-band level set*, essas amostras distantes introduzem custos computacionais desnecessários.

Esta seção apresenta três soluções para seleção das amostras a serem usadas para criação dos *narrow-band level sets*. A Seção 4.4.1 apresenta uma otimização para amostragens regulares. A Seção 4.4.2 descreve uma solução que busca avaliar funções de distância pertencentes apenas a *narrow-band*. Ambas abordagens anteriores apresentam algoritmos que não levam em consideração as características da forma geométrica, porém essas podem ser úteis para redução no número de amostras, como na solução da Seção 4.4.3, onde é apresentada uma técnica para geração de amostras para cilindros.

4.4.1 Amostragem Regular

A criação de *level sets* usando amostragens regulares da função de distância é trivial. Uma solução simples é apresentada no algoritmo da Figura 4.9. Essa implementação delimita a área do volume com uma AABB, e para cada *voxel* contido nessa caixa é feita uma amostra da

função de distância. O problema com essa abordagem é que diversas amostras distantes da superfície são realizadas, porém não são usadas na representação de *narrow-band level sets*, assim desperdiçando recursos computacionais.

Figura 4.9: Amostragem Regular Simples

Requisitos: $V \leftarrow$ Volume
Requisitos: $dist \leftarrow$ Função de distância sinalizada de V
1: $min, max \leftarrow$ limites da caixa delimitadora de V
2: **para** $i \in (min_x, min_x + 1, \ldots, max_x)$ **faça**
3: **para** $j \in (min_y, min_y + 1, \ldots, max_y)$ **faça**
4: **para** $k \in (min_z, min_z + 1, \ldots, max_z)$ **faça**
5: faça amostragem de $dist(i, j, k)$
6: **fim para**
7: **fim para**
8: **fim para**

Fonte: Produção do próprio autor.

Como *narrow-band level sets* só levam em consideração amostras próximas a superfície, então é possível otimizar a versão simples da amostragem regular pulando regiões de desinteresse, ou seja, regiões distantes da superfície. A função de distância retorna a distância entre a posição de amostra até a superfície mais próxima. Usando essa propriedade é possível determinar quantas amostras podem ser ignoradas em uma direção com a garantia de elas não fazem parte da *narrow-band*. Assim, como o resultado da função de distância é usado para determinar o número de amostras ignoradas em uma direção, essa versão otimizada do algoritmo de amostragem regular consegue ignorar várias amostras distantes da superfície, enquanto a amostragem próxima a superfície se comporta igual a versão não otimizada da amostragem regular. O algoritmo da Figura 4.10 apresenta a implementação da amostragem regular otimizada.

Durante este trabalho não foi encontrado nenhuma desvantagem no uso da versão da amostragem regular otimizada para a versão sim-

ples, pois apenas amostras que não são usadas na *narrow-band level set* são ignoradas. Assim, a amostragem regular otimizada foi usada para a criação de todos os volumes neste trabalho, com exceção dos cilindros, pois esses apresentam uma versão específica para eles, como mostrará a Seção 4.4.3.

Figura 4.10: Amostragem Regular Otimizada

Requisitos: $V \leftarrow$ Volume
Requisitos: $H \leftarrow$ Tamanho da *narrow-band*
Requisitos: $dist \leftarrow$ Função de distância sinalizada de V
1: $min, max \leftarrow$ limites da caixa delimitadora de V
2: $min \leftarrow min - H$
3: $max \leftarrow max + H$
4: **para** $i \in (min_x, min_x + 1, \ldots, max_x)$ **faça**
5: **para** $j \in (min_y, min_y + 1, \ldots, max_y)$ **faça**
6: $k \leftarrow min_z$
7: **enquanto** $k \leq max_z$ **faça**
8: $m \leftarrow 1$
9: faça amostragem de $dist(i, j, k)$
10: $d \leftarrow dist(i, j, k)$
11: **se** $\sqrt{d^2} \geq H$ **então**
12: $m \leftarrow m + \sqrt{d^2}$
13: **fim se**
14: $k \leftarrow k + m$
15: **fim enquanto**
16: **fim para**
17: **fim para**

Fonte: Baseado na implementação da OpenVDB.

4.4.2 Amostragem *Narrow-band*

As soluções de amostragens regulares apresentadas na Seção 4.4.1, incluindo a amostragem regular simples e amostragem regular otimizada, tiram amostras da função de distância em regiões que não são de interesse, ou seja, fora da *narrow-band*. Assim, a fim de reduzir o nú-

mero de amostras, foi desenvolvida uma solução que reduz ainda mais o número de amostras feitas fora na *narrow-band*.

A solução de amostragem *narrow-band* é dividida em duas etapas: a primeira etapa busca um ponto próximo a superfície, enquanto a segunda etapa avalia esse ponto e todos os outros conectados e que estão próximos à superfície. O algoritmo da Figura 4.11 apresenta o pseudocódigo desta solução. A primeira etapa usa o gradiente da função de distância para saber a direção e distância até a superfície mais próxima. Assim que é encontrado um *voxel* na *narrow-band*, então a segunda etapa é executada, que avalia todos os *voxels* vizinhos que pertencem a *narrow-band* de forma recursiva. Note que este este algoritmo não funciona caso o volume apresente superfícies desligadas.

A vantagem desse método é que o número de amostras para criação de um volume é reduzido em comparação com a amostragem regular otimizada. Entretanto, esse algoritmo precisa manter uma estrutura de dados auxiliar para verificar quais *voxels* já foram visitados. Foi testado armazenar os *voxels* já visitados em uma *hashmap* 3D e em uma grade da OpenVDB, porém ambas soluções apresentaram um tempo de processamento maior que a versão da amostragem regular otimizada em todos os casos de testes. O problema é que ao introduzir a necessidade de uma estrutura de dados auxiliar, também são inseridas mais chamadas a memória, que já é um dos principais gargalos do sistema, como será discutido na Seção 6.3. Assim, a amostragem *narrow-band* não foi usada neste trabalho.

4.4.3 Amostragem para Cilindros

As duas soluções anteriores avaliam técnicas de amostragem para volumes em geral, entretanto, é possível adotar técnicas especializadas para certas formas geométricas. No caso de um cilindro, é possível usar o fato de que a distância de um ponto até a superfície não muda caso esse ponto seja transladado em paralelo ao eixo principal do cilindro. É fá-

Figura 4.11: Amostragem *narrow-band*

Requisitos: $V \leftarrow$ Volume
Requisitos: $H \leftarrow$ Tamanho da *narrow-band*
Requisitos: $dist \leftarrow$ Função de distância sinalizada de V
$min, max \leftarrow$ limites da caixa delimitadora de V
$min \leftarrow min - H$
$i, j, k \leftarrow (min_x, min_y, min_z)$
{Primeira etapa busca a superfície}
repita
    $\mathbf{g} \leftarrow \nabla dist(i, j, k)$
    {$\nabla dist(i, j, k)$ é o gradiente da função de distância}
    $i, j, k \leftarrow (i + \mathbf{g}_x, j + \mathbf{g}_y, k + \mathbf{g}_z)$
até $dist(i, j, k) \leq H$
{Segunda etapa avalia todas as amostras próximas a superfície}
$a\_visitar \leftarrow (i, j, k)$
$visitados \leftarrow \varnothing$
enquanto $a\_visitar$ não estiver vazio **faça**
    $(x, y, z) \leftarrow$ um elemento de $a\_visitar$
    remove (x, y, z) de $a\_visitar$
    se $(x, y, z) \notin visitados$ **então**
        $visitados \leftarrow visitados \bigcup (x, y, z)$
        faça amostragem de $dist(x, y, z)$
        $d \leftarrow dist(x, y, z)$
        se $\sqrt{d^2} \geq H$ **então**
            $a\_visitar \leftarrow a\_visitar \bigcup$ todos os vizinhos de (x, y, z)
        fim se
    fim se
fim enquanto

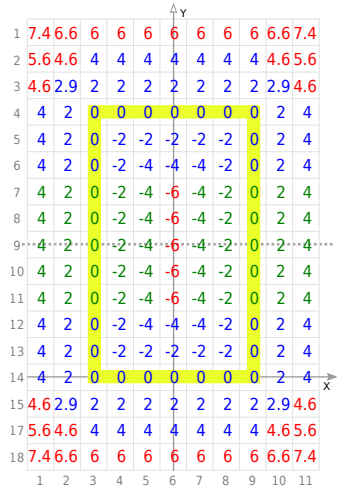
Fonte: Produção do próprio autor.

cil de visualizar essa propriedade quando se usa um cilindro de tamanho infinito e uma reta paralela a esse cilindro. Todos os pontos pertencentes a essa reta irão conter a mesma distância até o cilindro. Porém, para um cilindro não infinito, essa propriedade só é válida para pontos onde a distância até os discos do cilindro seja maior que da sua superfície lateral.

A Figura 4.12 mostra valores de distância para um cilindro. A superfície do cilindro é representada pela cor amarela. Os números re-

presentam valores de distância até a superfície. Este exemplo usa uma *narrow-band* com distâncias de até quatro unidades. Assim, valores em vermelho representam pontos fora da *narrow-band*. Valores em azul e verde representam valores pertencentes a *narrow-band*. A otimização descrita acima é representada pela cor verde, que mostra como todos os valores verdes apresentam os mesmos valores quando vistos de forma paralela ao eixo principal do cilindro. Note como pontos próximos aos discos do cilindro são apresentados em azul, pois não fazem parte da otimização discutida anteriormente.

Figura 4.12: Exemplo de distâncias de um cilindro.



Fonte: Produção do próprio autor.

Uma segunda otimização usa a característica da simetria do cilindro. A Figura 4.12 mostra uma linha horizontal tracejada que indica uma possível redução do problema pela metade, pois basta calcular apenas uma das metades das amostras e refletir o resultado na outra parte.

Usando o exemplo da Figura 4.12, essas duas otimizações permitem que seja feita apenas a amostragem das linhas 1-7. Após isso, a

segunda otimização é usada para refletir os resultados, assim construindo o volume do cilindro com um número reduzido de amostras comparado com as técnicas vistas nas Seções 4.4.1 e 4.4.2. Essa otimização reduz drasticamente o tempo de processamento para construção de cilindros altos, onde o comprimento do seu eixo principal é maior que seu raio, como a Seção 6.1 mostrará. Para cilindros baixos o algoritmo se reduz para a versão de amostragem regular otimizada descrita na Seção 4.4.1. Por causa disso, esta versão otimizada para cilindros é usada para criação de todos os cilindros neste trabalho pois não apresenta desvantagens.

4.5 Código-G

Em máquinas NC convencionais, o caminho da ferramenta é controlado por um conjunto de instruções que representam as movimentações da ferramenta. Esse conjunto de instruções é normalmente definido usando Código-G (SULLIVAN et al., 2012).

Um dos principais problemas com o Código-G é que apesar de existir uma norma, ela deixa vários pontos abertos para implementações específicas. Por causa disso não existe uma especificação universalmente compatível (HEUSINGER et al., 2006; ROSSO JR, 2005). Para tentar maximizar a compatibilidade e adoção, este trabalho adota a especificação do Código-G do sistema aberto LinuxCNC[3].

LinuxCNC é um sistema de software para o controle computadorizado de máquinas-ferramentas, como por exemplo, fresadoras e tornos. O sistema está disponível de forma *open source* com as licenças *GNU General Public License*[4] e *Lesser GNU General Public License*[5]. LinuxCNC implementa a sua própria especificação, baseada na linguagem RS274/NGC (KRAMER et al., 2010), do Código-G. Essa oferece

[3] LinuxCNC - About. Disponível em: <http://www.linuxcnc.org/index.php/about>. Acesso em: 19/09/2014.

[4] GNU General Public License. Disponível em: <http://www.gnu.org/copyleft/gpl.html>. Acesso em: 19/09/2014.

[5] Lesser GNU General Public License. Disponível em: <http://www.gnu.org/licenses/lgpl.html>. Acesso em: 19/09/2014.

um Código-G independente das variações que diferentes fabricantes impõem.

O Código-G é baseado em linhas de códigos. Cada linha é chamada de bloco, e pode incluir vários comandos para execução de diversas ações. Um bloco consiste de uma ou mais palavras. Uma palavra consiste de uma letra seguida por um número, ou expressão que pode ser avaliada como número. Uma palavra pode executar um comando ou providenciar um argumento para um comando.

A Figura 4.13 apresenta a gramática do Código-G do LinuxCNC implementada neste trabalho na *Extended Backus-Naur Form* (EBNF) (International Organization for Standardization, 1996).

Figura 4.13: EBNF do Código-G

```
c_word = "/" , { char };
t_word = "T" | "t", { number };
m_word = "M" | "m", { number };
g_word = "G" | "g", { number };
parameter_name = "<", { char - ">" }, ">";
parameter_word = "#", parameter_name | expr;
group = "(", expr, ")";
factor = parameter_word | number | group;
term = factor, { ("*", factor) | "/", factor) };
expr = term, { ("+", term) | "-", term) };
parameter_assign = (parameter_word, "=", expr);
word = parameter_assign | c_word | t_word | m_word | g_word | . . . ;
block = ("/", { char }) | [ word ];
```

Fonte: Produção do próprio autor.

Diversos comandos são suportados pelo Código-G, como troca de ferramenta, mudança no modo de movimentação (em linha reta ou em arco, por exemplo), ajuste da velocidade de avanço, etc. A movimentação da ferramenta acontece apenas no final de cada bloco, e apenas quando

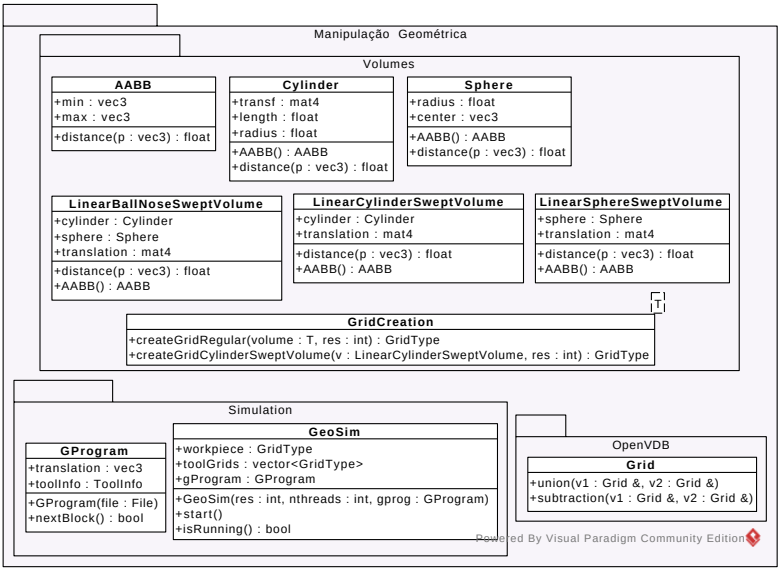
esse bloco apresenta comandos de movimentos. Assim, o simulador faz a leitura do programa a cada bloco, e no final de cada bloco com comando de movimento, ele cria o volume de deslocamento da ferramenta que reflete as informações do bloco. Após a criação desse volume é feita a operação booleana com a peça para simular a remoção do material. Em seguida é avaliada a próxima linha do Código-G e se repete esse processo até o final do programa. Assim, caso uma usinagem com Código-G contenha mil blocos com movimentos de ferramentas, então serão criados mil volumes de deslocamento e serão executadas mil operações booleanas entre esses volumes e a peça.

4.6 Arquitetura do Módulo da Manipulação Geométrica

A Figura 4.14 apresenta uma visão geral da arquitetura do módulo da manipulação geométrica. O diagrama de classe da figura é uma simplificação com os principais componentes implementados neste trabalho.

O pacote “Volumes” agrupa classes representando os volumes usados neste trabalho. As classes “AABB”, “Cylinder” e “Sphere” implementam as funções implícitas discutidas na Seção 4.3. Já as classes “LinearCylinderSweptVolume” e “LinearSphereSweptVolume” são usadas para calcular o volume de deslocamento linear de um cilindro e de uma esfera, como discutido na Seção 4.3.5. A classe “LinearBallNoseSweptVolume” é a união dos outros dois volumes de deslocamento anteriores. Note que todas essas classes apresentam uma função de distância e de AABB com a mesma declaração. Com isso, o método genérico “GridCreation::createGridRegular” pode ser reutilizado para criação de qualquer um desses volumes. Esse método implementa a criação de *narrow-band level sets* como visto na Seção 4.4.1. A implementação otimizada para cilindros, como descrita na Seção 4.4.3, é feita no método “GridCreation::createGridCylinderSweptvolume”. Ambos os métodos da classe “GridCreation” recebem como parâmetro o volume

Figura 4.14: Diagrama de Classes do Módulo da Manipulação Geométrica



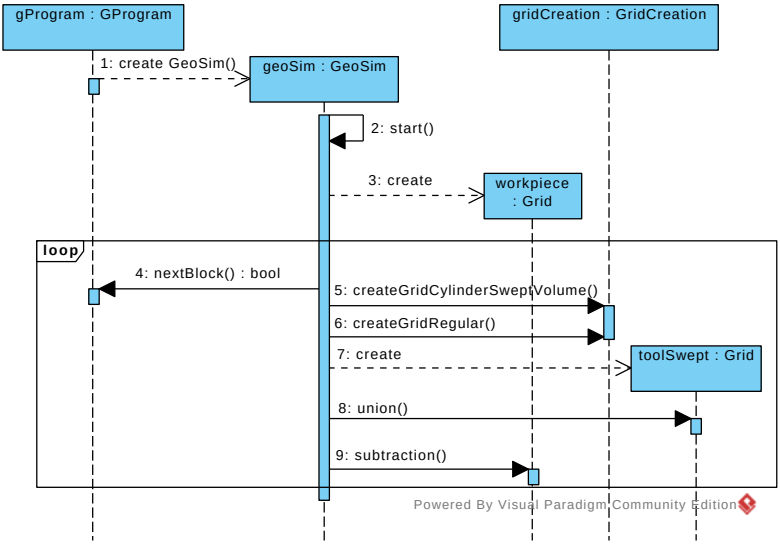
Fonte: Produção do próprio autor.

a ser criado e a resolução da grade.

A implementação da grade é feita pela biblioteca OpenVDB, como visto na Seção 4.1, que está representada pela classe “Grid”. Essa classe está dentro do pacote “OpenVDB” para indicar que foi usada uma biblioteca externa. Essa classe apresenta apenas dois métodos, que são as operações de união e subtração descritas na Seção 4.2. Essa visão da classe é uma visão simplificada da implementação real, porém útil para visualização geral do sistema. As operações booleanas são feitas usando duas grades (“v1” e “v2”) e armazenando o resultado da operação na primeira grade (“v1”).

O pacote “Simulation” agrupa as classes que controlam a simulação. “GProgram” implementa um *parser* para o Código-G, como visto na Seção 4.5, além de armazenar o estado atual do programa do Código-

Figura 4.15: Diagrama de Sequência do Módulo da Manipulação Geométrica



Fonte: Produção do próprio autor.

G. O programa armazena informações como as ferramentas carregadas, localização da ferramenta e se a usinagem terminou. O método "GProgram::nextBlock" é usado para fazer o *parsing* do próximo bloco de comando, e retorna verdadeiro caso o programa ainda não tenha terminado.

A classe "GeoSim" é responsável pelo gerenciamento de toda simulação. Ela mantém uma instância de um programa do Código-G, que é usada para verificar o estado da usinagem e os movimentos da ferramenta em cada bloco de comando. Além disso, a classe também armazena a grade da peça sendo usinada e outras grades que serão discutidas mais adiante nesta seção.

A Figura 4.15 apresenta o diagrama de sequência do módulo de manipulação geométrica. Após a instanciação da "GProgram" e da

"GeoSim", o método "GeoSim::start" é invocado. Esse método faz a criação da grade da peça inicial e entra em um *loop* até que a simulação termine. Durante o *loop* é analisado o próximo bloco de comando do Código-G, que retorna informações sobre o caminho percorrido pela ferramenta. Esse caminho é usado para criar o volume de deslocamento da ferramenta. Caso a ferramenta seja composta por mais de um tipo de volume, então é feita a união entre os volumes. Com o volume da ferramenta pronto, é feita a operação de subtração entre ele e o volume da peça.

A construção dos volumes de deslocamento da ferramenta é a tarefa mais custosa durante o *loop* da simulação. Por causa disso esses volumes são criados de forma paralela e armazenados temporariamente na variável "GeoSim::toolGrids". Enquanto a criação é feita de forma paralela, as operações booleanas entre esses volumes da ferramenta e da peça é feita de forma sequencial. Uma *thread* é lançada no inicio da execução do método "GeoSim::start" para realizar as operações de subtração do volume da ferramenta na peça em sequencia. Essa *thread* trabalha em modo produtor-consumidor, onde o produtor é a criação dos volumes, e o consumidor é a operação de subtração.

4.7 Discussão do Capítulo

Este capítulo apresentou o processo de simulação geométrica, que inicia-se com a criação do modelo da peça. Essa peça inicial é primeiramente definida por uma função de distância contínua de uma caixa alinhada aos eixos cartesianos (Seção 4.3.4), para então ser transformada em uma representação discreta usando uma amostragem regular otimizada (Seção 4.4.1). Os valores da amostragem regular otimizada são então armazenados na estrutura de dados OpenVDB (Seção 4.1).

A simulação de usinagem progride criando volumes de deslocamentos da ferramenta (Seção 4.3.5) e usando-os com operações booleanas de subtração (Seção 4.2) contra a peça inicial para simular a remoção de material. O processo de criação desses volumes de deslocamento é igual ao processo de criação da peça inicial, porém, volumes de desloca-

mento de cilindros podem ser otimizados usando uma técnica de amostragem específica para essa geometria (Seção 4.4.3). Uma técnica de amostragem da *narrow-band* (Seção 4.4.2) foi desenvolvida, entretanto, enquanto ela reduz o número de amostras para criação dos volumes, ela requer o uso de uma estrutura auxiliar para determinar valores já visitados. Por causa disso a técnica de amostragem da *narrow-band* acaba sendo mais custosa que a de amostragem regular otimizada, e por isso não é usada neste trabalho. Cada volume de deslocamento representa um bloco de comando do Código-G (Seção 4.5), assim, a simulação repete esse passo até processar todos os blocos de comandos do código da usinagem.

5 RAY TRACING PARA SIMULAÇÃO DE USINAGEM

A visualização interativa da simulação de usinagem é realizada usando *ray casting*, como visto na Seção 3.3. O processo de renderização de um quadro segue o seguinte fluxo:

- Geração das AABB dos volumes dos nó folhas contendo dados da superfície (Seção 4.1 para ver como o modelo geométrico é representado em memória);
- Criação da BVH com as AABB calculadas no passo anterior;
- Criação dos raios que serão lançados em direção a cena;
- Travessia dos raios pela cena e identificação das interseções com a superfície do modelo;
- Cálculo da normal para cada interseção encontrada;
- Sombreamento da superfície usando a técnica de Gooch;
- Envio dos pixels calculados para a GPU para visualização.

Antes de entrar em detalhes sobre as etapas descritas acima, este capítulo apresenta uma discussão sobre a decisão de usar CPU ao invés de GPU para algoritmos de *ray tracing* na Seção 5.1. Tal decisão é importante pois influência a arquitetura do projeto.

A Seção 5.2 descreve a técnica de pacotes de raios, que permite que raios coerentes façam a travessia da cena com quase o mesmo custo computacional de um único raio. Essa técnica é introduzida no começo do capítulo pois o uso de pacotes de raios invalida os métodos tradicionais de travessia da cena, que são abordados na Seção 5.3. Além da travessia, a Seção 5.3 também comenta sobre a estrutura de dados aceleradora usada com *ray casting*.

O teste de interseção entre o raio e a superfície é feito em conjunto com o algoritmo para reconstrução da superfície usando interpolação trilinear. Diversas técnicas foram estudadas, implementadas e descritas na Seção 5.4. Entretanto, apenas o cálculo da interseção não é suficiente para reconstrução da cena. O sombreamento é essencial para que haja uma percepção de profundidade na cena. O cálculo do sombreamento usa informações sobre a direção da superfície para estimar a intensidade da iluminação em certas regiões do modelo. A direção da superfície é então encontrada usando o cálculo da normal, que é apresentado na Seção 5.5.1.

Uma das últimas etapa da renderização de um quadro é a de sombreamento. Este trabalho enfatiza a transmissão efetiva da informação sobre a superfície para o usuário mais do que na renderização de cenas foto realísticas. Por isso foi implementado o sistema de sombreamento de modelos técnicos proposto por Gooch et al. (1998), e apresentado na Seção 5.5.

A técnica de pacotes de raios pode ser vista como uma forma de paralelizar o algoritmo de *ray casting* usando computação vetorial, mas além desse estilo, este trabalho também usa programação paralela *multi-core*. A distribuição da carga entre os núcleos da CPU é baseada nos pixels que cada núcleo processa. Assim, a Seção 5.6 descreve como é feita a escolha dos pixels que serão estimados e computados por cada processo.

Assim que todos os pixels são calculados, eles são enviados para a GPU usando OpenGL para visualização, e o processo é repetido para encontrar o próximo quadro.

Uma visão geral da arquitetura desenvolvida neste trabalho pode ser vista na Seção 5.7, que explica como os componentes anteriores estão organizados. Por último é feita uma discussão do capítulo na Seção 5.8.

5.1 Diferença entre Ray Tracing com GPGPU ou CPU

Ray tracing é um problema facilmente paralelizável pois cada raio lançado a cena pode ser processado individualmente. Diversas tecnologias para programação paralela foram encontradas na literatura para solução deste problema, porém as principais são GPGPU (*General-Purpose computing on Graphics Processing Units*) (BIKKER, 2012; LAINE; KARRAS, 2010; HANNIEL; HALLER, 2011; AILA; LAINE, 2009; ROMISCH, 2009), programação CPU *multi-core*, e/ou SIMD (*Single Instruction Multiple Data*) (WALD et al., 2014; WALD et al., 2001; BENTHIN, 2006; WALD et al., 2006; BOULOS et al., 2007; OVERBECK et al., 2008; JAMRIŠKA, 2010; KNOLL et al., 2006; MARMITT et al., 2004), ou ambas combinadas (ZHU et al., 2012; CRASSIN et al., 2011).

As implementações estado da arte para GPGPU e CPU são competitivas, resultando em resultados similares (WALD et al., 2014). Entretanto, ambas apresentam modelos de programação distintos.

CPUs apresentam diversas opções de arquiteturas, então, CPU neste trabalho se refere a arquiteturas convencionais modernas disponíveis a consumidores. Um exemplo nessa categoria seria a arquitetura Haswell[1] que oferece SMT (*Simultaneous Multithreading*) e instruções AVX-2. Essa plataforma convencional apresenta um processador conectado a uma única memória central. SMT na Haswell permite que duas *threads* sejam executadas em cada núcleo simultaneamente, onde cada uma delas pode compartilhar memória com todas as outras. Além disso, cada *thread* pode executar diferentes listas de instruções independentemente. O processamento SIMD é feito com as instruções AVX-2, que permitem operações com 256 *bits* de pontos flutuantes com uma única instrução.

Assim como nas CPUs, as GPUs também apresentam variações nas arquiteturas, por isso apenas as convencionais são levadas em consi-

[1] Intel Haswell. Disponível em: <http://ark.intel.com/products/codename/42174/Haswell?wapkw=haswell>. Acesso em: 03/11/2014.

deração, como a GCN[2] da AMD. GPUs são processadores *multithreaded* construídos com o objetivo de processar um número elevado de pixels de forma eficiente. Seu foco principal está na quantidade de computação, ao invés de latência. Além disso, GPUs apresentam mecanismos sofisticados para gerenciar filas de tarefas e para esconder computações SIMD usando sistemas em hardware (GASTER et al., 2011). Comparadas com CPUs, as GPUs apresentam uma maior largura de banda para acesso a memória global disponível a todas as *threads*.

GPUs são geralmente hardware dedicados com processadores e memórias separados da CPU e da memória principal do computador. Isso resulta em uma programação heterogênea. A principal vantagem no uso de GPUs para *ray tracing* é que a renderização das cenas pode ser feita sem usar recursos da CPU, deixando-a livre para outras tarefas, como gerência da cena e dos modelos, por exemplo. Entretanto, essa separação da memória introduz problemas e limitações. A memória global da GPU tende a ser mais rápida que a memória compartilhada da CPU, e essa diferença resulta em um custo maior. Por isso, memória em GPU é um recurso mais limitado que em CPU. Além da limitação do volume da memória, existe também o problema de sincronização entre CPU e GPU. Há um custo elevado na troca de informações entre CPU e GPU, assim, é possível que em algumas aplicações o tempo para enviar e receber dados da GPU seja superior ao de processar esses dados.

GPUs são adequadas para processar algoritmos simples, paralelos e que exigem bastante poder computacional com o mínimo de sincronização entre as *threads* e com poucos acessos a memória global. Isso faz com que GPUs não sejam eficientes para operações sequenciais, como criação da árvore da estrutura de dados da Seção 4.1 e operações booleanas da Seção 4.2. Assim, a manipulação geométrica, desde a criação dos volumes até as operações booleanas, são feitas usando a CPU.

[2] GCN Architecture. Disponível em: <http://www.amd.com/en-gb/innovations/software-technologies/gcn>. Acesso em: 15/11/2014.

Uma das características do uso de *voxels* é o seu alto consumo de memória, principalmente quando se trabalha com modelos de alta precisão. A quantidade de memória usada por tais modelos extrapola a quantia oferecida pelas GPUs convencionais disponíveis hoje em dia. Além disso, quando um modelo é atualizado na CPU ele precisa ser sincronizado com a GPU, dificultando a visualização interativa pois o tempo da sincronia é elevado. Por causa disso, foi decidido usar uma implementação inteiramente em CPU, incluindo a manipulação geométrica e visualização interativa da cena, neste trabalho.

Uma implementação inteiramente em CPU tem vantagens e desvantagens. Em geral uma CPU apresenta uma capacidade de memória principal maior, permitindo assim trabalhar com modelos de alta precisão de forma eficiente. Ela também permite que o módulo de manipulação geométrica e de visualização interativa compartilhem o mesmo espaço de memória. Com isso, modificações no modelo geométrico são visualizadas instantaneamente pois não há necessidade de sincronização. A principal desvantagem ao usar uma CPU convencional é que o módulo de visualização interativa concorre por recursos de computação, e de largura de banda da memória, com o módulo da manipulação geométrica. Isso faz com que um módulo acabe interferindo no desempenho de outro.

5.2 Pacotes de Raios

Diversas técnicas foram publicadas para acelerar o *ray tracing* como estruturas de dados mais eficientes, algoritmos de interseção mais rápidos, programação paralela, computações aproximadas, etc. Entre elas está a técnica de usar pacotes de raios, publicada por Wald et al. (2001), que foi essencial para o desenvolvimento dos primeiros *ray tracers* em tempo real.

Ray tracing requer um custo computacional alto, e seu baixo desempenho geralmente está associado a pouca largura de banda e alta latência da memória principal. A principal contribuição dos pacotes de raios é a computação coerente dos raios. A ideia principal dessa técnica

é agrupar raios coerentes (que seguem caminhos similares na cena) em um pacote (grupo) e fazer todas as computações com esse pacote, ao invés de tratar cada raio de forma individual.

Computação de pacotes de raios em CPU geralmente é otimizada usando SIMD. Assim, cada teste de interseção, por exemplo, com todos os raios do grupo é quase tão eficiente quanto um teste com um único raio. Wald et al. (2001) propuseram criar pacotes com o número de raios relacionado a largura do registrador SIMD. Por exemplo, CPUs com suporte a instruções AVX-2 apresentam largura SIMD de 256 *bits*, assim, usando uma precisão de ponto flutuante de 32 *bits*, é possível realizar oito operações de ponto flutuantes com uma instrução SIMD. Logo, CPUs com largura SIMD de 256 *bits* operam com pacotes de oito raios.

Para facilitar a computação vetorial os raios são armazenados em *structures of arrays* (SOA), ao invés do método tradicional que é *arrays of structures* (AOS). A Figura 5.1 apresenta como um pacote com oito raios é representado em memória. Para acessar a direção no eixo x do terceiro raio, por exemplo, basta acessar o elemento *Rays::dir_x[2]*. A Figura 5.2 mostra a função para o teste de interseção entre um pacote de raios e uma AABB. As funções que operam com tipos de dados $float8$, ou $float[8]$, como a div e a sub, estão operando em oito valores simultaneamente. Ou seja, a mesma instrução é aplicada em múltiplos dados, assim, o uso de pacotes de dados e SIMD é uma forma de paralelismo de dados.

Neste trabalho, a principal vantagem no uso de pacotes de raios está na redução do número de acessos a estrutura de dados do modelo geométrico. Isto porque o algoritmo precisa buscar na estrutura os valores dos campos de distância para realizar o teste de interseção com a superfície. Fazendo a travessia em pacotes, os valores dos campos de distância podem ser compartilhados entre os oito raios do pacote, assim reduzindo drasticamente a quantidade de acesso a estrutura de dados, e com isso usando menos largura de banda da memória principal.

Figura 5.1: Pacote com oito raios usando SOA

```
1  struct Rays {
2    float org_x[8];
3    float org_y[8];
4    float org_z[8];
5    float dir_x[8];
6    float dir_y[8];
7    float dir_z[8];
8    float tfar[8];
9    float tnear[8];
10   int   hitid[8];
11 } __attribute__ ((aligned(16)));
```

Fonte: Produção do próprio autor.

5.3 Estrutura de Dados Aceleradora

O cenário da simulação é composto por uma câmera, fontes de luzes e o modelo geométrico da peça. A visualização deste modelo é feita usando *ray casting*, logo, é necessário uma forma eficiente para encontrar as interseções entre os raios e a superfície. A Seção 4.1 mostrou como o modelo da peça é representado em memória, e como apenas nós folhas contêm informações sobre a superfície. Assim, regiões do modelo que estão distantes da superfície não precisam ser verificadas por interseção. Zhu et al. (2012) propuseram dois níveis de travessia do raio na cena. Um nível usando uma BVH, onde o raio pode atravessar regiões homogêneas e esparsas de forma eficiente, e o segundo nível é a travessia dos raios nos volumes densos próximos a superfície. Uma variação dessa técnica de dois níveis foi adotada neste trabalho, onde a diferença está na forma que a travessia dos volumes densos é feita.

A solução de Zhu et al. (2012) usa raios individuais para travessia de volumes densos, porém tal solução não é adequada para pacotes de raios. Assim, o segundo nível da travessia adota uma solução específica para travessia de volumes densos usando pacotes de raios.

A construção e travessia da BVH são descritas na Seção 5.3.1. Já a Seção 5.3.2 mostra a travessia dos volumes densos.

Figura 5.2: Teste de interseção entre pacotes de raios e AABB

```
1   bool8 intersect(const AABB& box, const Rays& rays, float8& tnear,
        float8& tfar) {
2     float8 lx1 = div(sub(box.min_x, rays.org_x), rays.dir_x);
3     float8 ly1 = div(sub(box.min_y, rays.org_y), rays.dir_y);
4     float8 lz1 = div(sub(box.min_z, rays.org_z), rays.dir_z);
5     float8 lx2 = div(sub(box.max_x, rays.org_x), rays.dir_x);
6     float8 ly2 = div(sub(box.max_y, rays.org_y), rays.dir_y);
7     float8 lz2 = div(sub(box.max_z, rays.org_z), rays.dir_z);
8     float8 lxmax = max(lx1, lx2);
9     float8 lymax = max(ly1, ly2);
10    float8 lzmax = max(lz1, lz2);
11    float8 lxmin = min(lx1, lx2);
12    float8 lymin = min(ly1, ly2);
13    float8 lzmin = min(lz1, lz2);
14    float8 lmax = min(lxmax, lymax, lzmax);
15    float8 lmin = max(lxmin, lymin, lzmin);
16    if (lmax >= 0 && lmax > lmin) {
17      tnear = lmin;
18      far = lmax;
19      return true;
20    }
21    return false;
22  }
```

Fonte: Baseado em (GEIMER; MÜLLER, 2003).

5.3.1 Travessia da BVH

Existem diversas alternativas para resolver o primeiro nível da travessia da cena, como Octree, KD-tree, BVH e outras (a Seção 3.4.2 contem uma discussão sobre elas). Entre elas, a estrutura BVH é a que apresenta pouco consumo de memória, fácil e rápida construção, além de baixo nível hierárquico permitindo travessias eficientes (WALD et al., 2014). Assim, este trabalho faz o primeiro nível da aceleração da travessia usando BVH.

Lembrando da representação do modelo geométrico descrito na Seção 4.1, apenas os nós folhas dos volumes densos contêm os campos de distância sinalizados da superfície. Todos os outros nós da estrutura podem ser descartados para a reconstrução da superfície. Cada nó folha armazena $8 \times 8 \times 8$ *voxels* organizados em formato de grade, assim, seu

bounding box volume é facilmente calculado pois é apenas uma caixa. Essas caixas são então usadas para construção da BVH, assim, quando um raio faz a interseção com uma dessas caixas é garantido que ela contêm um volume denso da superfície.

É possível economizar espaço de memória armazenando apenas a posição de origem da caixa, pois essa posição é suficiente para determinar qual volume denso está sendo referenciado. O tamanho da caixa também não precisa ser armazenado pois é o valor constante de $8 \times 8 \times 8$ para todas os volumes densos. Uma vez que se tem a origem da caixa, determinar a posição de cada *voxels* do volume denso é uma tarefa simples pois basta somar o deslocamento a posição de origem da caixa. O acesso aos *voxels* usando suas posições é feito em tempo constante ($O(1)$), como visto na Seção 4.1. Isso torna a BVH uma estrutura auxiliar com baixo consumo de memória e poder computacional para sua criação.

A implementação da Embree (WALD et al., 2014) foi usada para construção e travessia dos raios da BVH. Embree[3] é uma coleção de algoritmos de alto desempenho para *ray tracing*. Esses algoritmos são otimizados para processadores que suportam instruções SSE, AVX ou AVX-2, ou processadores Xeon Phi.

Foi usado o algoritmo de construção chamado *Binned SAH BVH* da Embree, que resulta em uma estrutura onde o tempo médio da travessia é reduzido ao custo de um tempo maior para a criação da estrutura. Tal algoritmo constrói uma BVH otimizada para travessia com técnicas para subdivisão espacial usando *surface area heuristics* (SAH) (WALD, 2007).

A travessia da BVH é feita usando pacotes de raios com a Embree, onde a interseção de um pacote é testada para cada volume da BVH em seu caminho. Quando um pacote de raio intersecciona um volume denso, então o segundo nível da travessia é executado. Caso o segundo nível não encontre uma interseção, o primeiro nível pode continuar

[3] Embree. Disponível em: <https://embree.github.io/>. Acesso em: 23/05/2014.

a travessia até que o raio saia da cena.

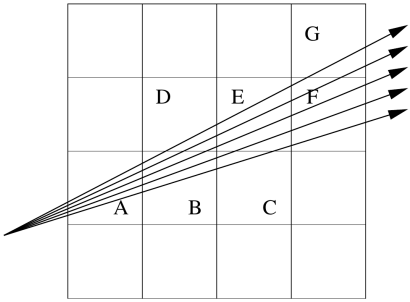
5.3.2 Travessia da Grade

O segundo nível faz a travessia dos pacotes de raios nos nós folhas da estrutura do modelo geométrico. Os *voxels* presentes nesses nós estão organizados de forma contínua e uniforme em memória (em formato de grade). A técnica tradicional para travessia de raios em grade é a 3DDDA, *digital differential analyzer* (AMANATIDES; WOO, 1987). O problema é que esse algoritmo seleciona apenas uma célula a cada iteração, fazendo com que ele não seja adequado para uso com pacotes de raios. Por exemplo, a Figura 5.3 mostra cinco raios atravessando uma grade. A primeira divergência acontece na célula B onde alguns raios vão acessar a célula C enquanto outros vão para a D. Uma possível solução para esse problema seria criar pequenos pacotes, dividindo os raios que vão para C dos que vão para D. O problema é que é possível que esses pequenos pacotes venham a compartilhar células no futuro, como no exemplo onde alguns raios que entraram em C passam por E, assim como outros raios divergentes que entraram em D mas acabam na mesma célula E. Caso seja mantida a separação dos pacotes pequenos, então é perdida a coerência em alguns raios. Porém, juntar novamente os raios após uma divergência é uma operação cara.

O trabalho de Wald et al. (2006) propôs uma técnica para travessia de grades baseada em fatias para solucionar os problemas do 3DDDA. A ideia principal do algoritmo é fazer a travessia em fatias ao invés de célula a célula. Usando a Figura 5.3 como exemplo, e usando fatias verticais, a travessia por fatia iria primeiramente verificar a célula A, fazendo o teste de interseção entre todos os raios e essa célula. A segunda fatia iria ser para as células D e B, seguida de E e C, para então terminar com a fatia das células G e F. O custo seria de 7 interseções com 5 raios, ao invés de 27 visitas a células caso os raios fossem atravessados individualmente. Note que são feitos 7×5 testes de interseção, que é maior que o número de visitas nas células, mas na prática, esse

custo extra é compensado pelo uso coerente dos raios. Ou seja, a informação da célula é acessada apenas uma vez por pacote, ao invés de uma vez por raio, reduzindo assim o custo de acesso dessa informação da memória.

Figura 5.3: Cinco raios coerentes atravessando uma grade



Fonte: (WALD et al., 2006).

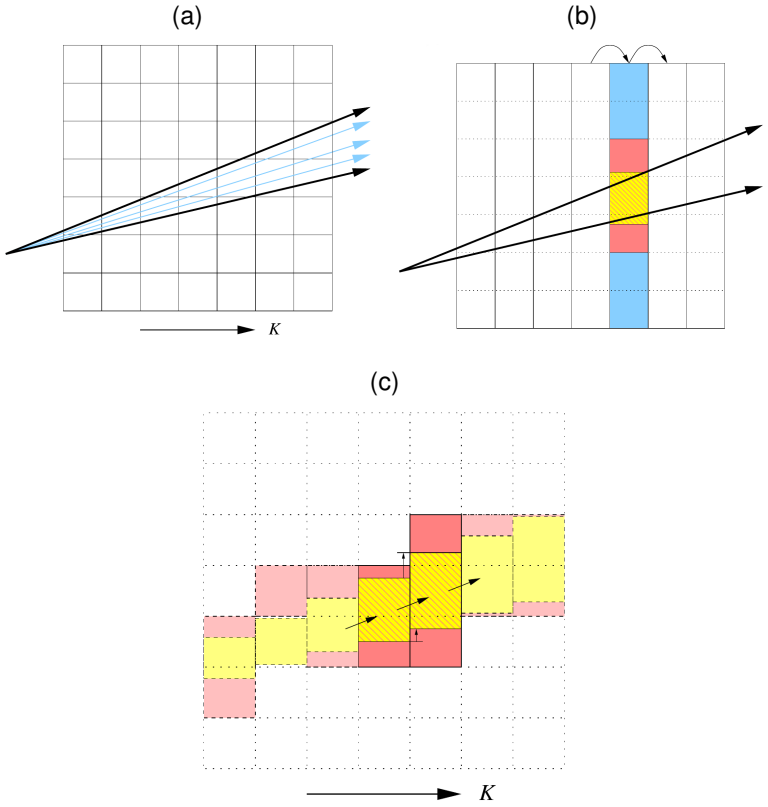
A Figura 5.4 mostra a técnica de travessia em fatias. Inicialmente é calculado o *frustum*[4] do pacote de raios usando os elementos mais extremos, ou seja, é definido o volume mínimo no qual todos os raios do pacote estão contidos. Na Figura 5.4a o *frustum* é delineado pelos raios da cor preta. Uma vez que esse volume é calculado, ele é usado para identificar quais células serão verificadas em uma determinada fatia. A Figura 5.4b ilustra as células que serão verificadas em amarelo e vermelho, enquanto as que serão descartadas estão marcadas em azul. A Figura 5.4c mostra todas as células, de todas as fatias, que serão acessadas usando essa técnica.

5.4 Reconstrução da Superfície

O modelo geométrico da peça é um *narrow-band level set* (Seção 4.1) representado de forma discreta. Entretanto, a função de interseção

[4] Pirâmide cortada por dois planos paralelos que engloba todos os raios do pacote.

Figura 5.4: Travessia da grade com pacotes de raios



Fonte: (WALD et al., 2006).

requer uma superfície contínua. Assim, a reconstrução da superfície é uma etapa essencial para visualização da peça, pois ela transforma os campos de distância em uma superfície contínua.

As duas técnicas mais comum para reconstrução da superfície são a *Marching Cubes* (MC) (LORENSEN; CLINE, 1987) e interpolação trilinear. MC transforma a representação discreta do modelo em uma malha triangular, que é uma operação custosa. Já a interpolação trilinear pode ser feita durante o teste de interseção de maneira eficiente e com

uma precisão superior a de MC, por isso essa foi a escolha neste trabalho.

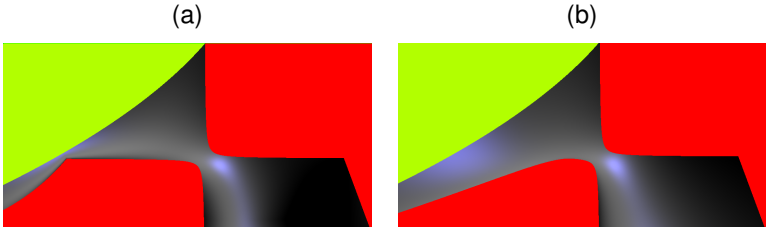
A reconstrução da superfície é feita para cada raio durante o teste de interseção, que é a solução da Equação 2.9. Tal equação pode ser expandida no polinômio da Equação 2.10. O trabalho de Schwarze (1990) apresenta uma solução analítica para esse polinômio cúbico. Enquanto essa solução é teoricamente a mais precisa, ela apresenta diversos problemas quando aplicada na prática. O algoritmo de Schwarze (1990) faz uso de operações custosas de raiz quadrada e diversas chamadas a função de cosseno. Além disso, essa solução apresenta problemas numéricos, causando assim erros de estabilidade numérica, que são agravados ao usar pontos flutuantes de precisão simples (MARMITT et al., 2004).

Um método rápido para cálculo de interseção é o de interpolação linear. Nesse método, primeiramente é calculada a distância no ponto em que o raio entra na célula e a distância no ponto em que o raio sai da célula. A interseção ocorre quando os sinais das distâncias são diferentes. A localização da interseção é aproximada por meio de interpolação linear entre os valores das distâncias previamente calculados. Porém, essa solução não consegue detectar todas as interseções. Além disso, a localização da interseção pode ser calculada de forma errada. A Figura 5.5a mostra a imprecisão do método, onde em vermelho são os raios sem colisão, em amarelo os raios que colidem com o interior do objeto e a parte sombreada mostra regiões onde a interseção foi encontrada com sucesso. Esse problema aparece porque esse método não resolve o polinômio cúbico da Equação 2.10 de forma correta.

O trabalho de Neubauer et al. (2002) propôs uma solução para melhor aproximar a localização das interseções executando diversas interpolações lineares, convergindo o resultado a cada iteração. Nesse método de Neubauer, o problema de má detecção de interseções permanece.

O trabalho de Marmitt et al. (2004) apresenta um novo método para detecção de interseções entre o raio e a superfície representada

Figura 5.5: Comparação entre: a) interpolação linear; e b) método de Marmitt



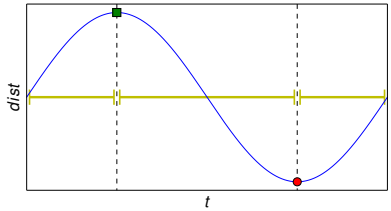
Fonte: Produção do próprio autor.

por DF. O método de Marmitt é dividido em duas etapas: primeiramente é encontrado o intervalo onde a primeira interseção acontece, para só depois aproximar o resultado com o método de Neubauer.

Analisando o polinômio cúbico da Equação 2.10 é possível concluir que a função de distância pode ter até três soluções. Essas soluções indicam o valor de t quando a distância do ponto $\mathbf{p} = \mathbf{a} + t\mathbf{b}$ até a superfície é zero. Para evitar que a solução errada seja escolhida, se divide o polinômio cúbico em três intervalos delimitados pelos pontos extremos, como mostra a Figura 5.6 onde o marcador quadrado apresenta o ponto extremo máximo, o circular apresenta o ponto extremo mínimo e as linhas horizontais delimitam os intervalos. Note que esses intervalos estão limitados à célula. Esses pontos extremos são as soluções da primeira derivada do polinômio cúbico. O uso dos pontos extremos para as divisões da solução garante que cada intervalo contém apenas uma solução.

Com as divisões dos pontos extremos do polinômio calculadas, o teste de interseção analisa cada um desses três intervalos em ordem, começando pelo mais próximo. Há interseção quando o sinal da função de distância difere para os valores do início e do final do intervalo. Caso não haja interseção, então o próximo intervalo é verificado. Caso haja interseção, então a próxima etapa do algoritmo é executada. Caso os

Figura 5.6: Intervalos em um polinômio cúbico



Fonte: Produção do próprio autor.

três intervalos tenham sido verificados e não houve interseção, então é possível afirmar que não há interseção nos limites da célula.

A próxima etapa do algoritmo procura a localização da interseção com o método de Neubauer. Agora o algoritmo de Neubauer vai convergir para a resposta correta, pois ele vai trabalhar em um intervalo que é garantido existir uma, e apenas uma, solução. A Figura 5.7 apresenta o pseudo-código para o algoritmo de Marmitt.

O método de Marmitt resolve os problemas dos métodos anteriores, como pode ser visto na Figura 5.5b. Quando comparado com a solução analítica de Schwarze, o método de Marmitt obtém os mesmos resultados porém de forma mais eficiente. Ao usar processamento SIMD, o método de Marmitt é cerca de 60% mais rápido que a solução analítica (MARMITT et al., 2004). Por isso foi a solução adotada neste trabalho.

5.5 Sombreamento

Uma das partes essenciais para renderização de superfícies é o seu sombreamento[5], pois sem isso não seria possível representar profundidade e detalhes da superfície para o usuário. Sombreamento com *ray tracing* é bem flexível, e capaz de construir desde imagens foto realistas até visuais que parecem desenhos animados. Uma alternativa é a técnica

[5] Do inglês *shading*.

Figura 5.7: Pseudo-código para o Método de Marmitt

Requisitos: $t_0 = t_{in}; t_1 = t_{out}$
  $f_0 \leftarrow dist(t_0); f_1 \leftarrow dist(t_1)$
  // Encontre pontos extremos resolvendo:
  // $dist'(t) = 3At^2 + 2Bt + C$
  se f' tem soluções **então**
    $e_0 \leftarrow$ menor solução de $dist'$
    se $e_0 \in [t_0, t_1]$ **então**
      se $sign(dist(e_0)) = sign(dist(f_0))$ **então**
        // Avance o raio para o segundo segmento
        $t_0 \leftarrow e_0; f_0 \leftarrow dist(e_0)$
      senão
        $t_1 \leftarrow e_0; f_1 \leftarrow dist(e_0)$
      fim se
    fim se
    $e_1 \leftarrow$ segunda solução de $dist'$
    se $e_1 \in [t_0, t_1]$ **então**
      se $sign(dist(e_1)) = sign(dist(f_0))$ **então**
        // Avance o raio para o terceiro segmento
        $t_0 \leftarrow e_1; f_0 \leftarrow dist(e_1)$
      senão
        $t_1 \leftarrow e_1; f_1 \leftarrow dist(e_1)$
      fim se
    fim se
  fim se
  se $sign(f_0) = sign(f_1)$ **então**
    retorne "Sem Interseção"
  fim se
  para $i = 1..N$ **faça**
    $t \leftarrow t_0 + (t_1 + t_0)\frac{-f_0}{f_1 - f_0}$
    se $sign(dist(R(t))) = sign(f_0)$ **então**
      $t_0 \leftarrow t; f_0 \leftarrow dist(R(t))$
    senão
      $t_1 \leftarrow t; f_1 \leftarrow dist(R(t))$
    fim se
  fim para
  retorne $t_0 + (t_1 + t_0)\frac{-f_0}{f_1 - f_0}$

Fonte: Traduzido de (MARMITT et al., 2004)

de *non-photorealist rendering* (NPR) de Gooch et al. (1998) para melhor visualização de desenhos técnicos. Tal solução usa tanto a luminosidade como a mudança de tom para indicar a orientação da superfície, enquanto valores extremos de luminosidade (altos e baixos) são reservados para mostrar os cantos dos objetos. O sombreamento de Gooch é desejável

em aplicações onde a comunicação da forma geométrica é mais valiosa do que realismo, que é o caso deste projeto.

Sombreamento de Gooch faz a mistura da interpolação entre dois tons para definir a intensidade da iluminação da superfície. Os tons azul e amarelo são usados por causa da transição entre uma cor fria para uma cor quente independe da cor difusa do objeto. Essas cores são definidas no espaço RGB como:

$$\begin{aligned} k_{azul} &= (0, 0, b), \\ k_{amarelo} &= (y, y, 0) \end{aligned} \tag{5.1}$$

onde $b \in [0, 1]$ e $y \in [0, 1]$.

Essas cores tendem a ser complementares, ou seja, quando somadas elas se cancelam e criam uma cor em tom de cinza. Além disso, elas apresentam um contraste elevado quando colocadas uma no lado da outra. Para desenhos técnicos, isso facilita a percepção da variação da orientação da superfície.

A relação desses tons com a cor difusa da superfície é dada por:

$$\begin{aligned} k_{fria} &= k_{azul} + \alpha k_d, \\ k_{quente} &= k_{amarelo} + \beta k_d \end{aligned} \tag{5.2}$$

Assim, o sombreamento de Gooch é controlado por quatro parâmetros: b, y, α e β. Gooch et al. (1998) demonstra bons resultados usando $b = 0.4, y = 0.4, \alpha = 0.2, \beta = 0.6$, que são os valores adotados neste trabalho.

O sombreamento da superfície usando a técnica de Gooch é feito com a interpolação entre as duas cores definidas na Equação 5.2 com o ângulo de incidência da luz em relação a superfície, ou seja:

$$I = \left(\frac{1 + \hat{\mathbf{l}} \cdot \hat{\mathbf{n}}}{2} \right) k_{quente} + \left(1 - \frac{1 + \hat{\mathbf{l}} \cdot \hat{\mathbf{n}}}{2} \right) k_{fria} \tag{5.3}$$

onde $\hat{\mathbf{l}}$ é a direção da luz e $\hat{\mathbf{n}}$ a normal da superfície, que será apresentada na Seção 5.5.1.

5.5.1 Cálculo da Normal

A Equação 2.11 apresenta a função matemática para o cálculo da direção da superfície, que é uma função gradiente. Na prática essa função pode ser calculada de forma eficiente com apenas quatro amostras da função de distância, como mostra o pseudo-código da Figura 5.8. O código é baseado na definição de derivadas com limites. Por exemplo, a derivada parcial da função de distância em x pode ser interpretada como:

$$\frac{\partial dist}{\partial x} = \lim_{\triangle h \to 0} \frac{dist(x + \triangle h, y, z) - dist(x, y, z)}{\triangle h} \tag{5.4}$$

Figura 5.8: Cálculo da normal da superfície

Requisitos: $\mathbf{P} \leftarrow$ Ponto na superfície
Requisitos: $h \leftarrow$ Deslocamento da amostra
Requisitos: $dist \leftarrow$ Função de distância sinalizada
1: $(px, py, pz) \leftarrow \mathbf{P}$
2: $d \leftarrow dist(px, py, pz)$
3: $dx \leftarrow dist(px + h, py, pz)$
4: $dy \leftarrow dist(px, py + h, pz)$
5: $dz \leftarrow dist(px, py, pz + h)$
6: $\mathbf{g} \leftarrow (dx - d, dy - d, dz - d)$
7: **retorne** $normalize(\mathbf{g})$

Fonte: Produção do próprio autor.

É possível aumentar o número de amostras da função gradiente e suavizar a normal. Note que o cálculo da normal está relacionado diretamente com o algoritmo de sombreamento. Isso significa que aumentar o número de amostras e o valor de h do algoritmo da Figura 5.8 resulta em uma superfície mais suave, porém com perda de informação. Por isso, foi adotada a solução com apenas quatro amostras e $h = 0.01$. O valor do h foi encontrado de forma empírica após realizar diversos testes. Caso o valor seja muito pequeno, se encontra problemas com a precisão de nú-

meros flutuantes e a função gradiente acaba retornando um vetor nulo. Já um h grande perde muitas informações sobre a superfície.

5.6 Amostragem dos Pixels

A Seção 3.3 apresentou diversos métodos para *ray tracing*. Uma das características que os diferenciam é a amostragem dos pixels. Este trabalho usa a amostragem proposta por Whitted (1980), que é apenas uma amostra por pixel feita de forma uniforme. Ou seja, é lançado um raio na cena para cada pixel. A vantagem do método está na sua fácil implementação e menor consumo computacional para renderização de um quadro. Entretanto, a técnica é conhecida pela sua geração de *aliasing* e do padrão de Moiré devido ao espaçamento uniforme das amostras. *Aliasing* foi encontrado nas bordas das peças usinadas, porém o padrão de Moiré não foi detectado em nenhum dos testes, independentemente da posição e orientação da câmera.

A computação das amostras dos pixels são aceleradas com dois métodos de paralelização: *multi-core* e SIMD. O método SIMD já foi comentado na Seção 5.2, onde pacotes de raios são processados ao invés de raios individuais. Esses pacotes são criados usando raios primários de pixels vizinhos, assim a coerência deles é garantida.

A paralelização *multi-core* distribui a computação entre os núcleos de uma CPU convencional. Um modelo baseado em tarefas foi usado, onde cada tarefa trabalha de forma completamente independente e sem qualquer tipo de sincronismo. Cada tarefa é mapeada a um quadrado contendo $n \times n$ pixels, onde esses pixels são processados de forma sequencial em um único processo.

Diferentes quadrados de $n \times n$ pixels apresentam diferentes custos computacionais. Por exemplo, um quadrado onde todos os pixels não fazem interseção com o modelo vai ser processado em tempo menor que outro onde todos os pixels fazem interseção. Para garantir que a carga é balanceada entre os núcleos foi usado particionamento recursivo das ta-

refas. Ou seja, caso um quadrado com 512×512 pixels esteja demorando muito para ser processado, ele pode ser dividido em quatro quadrados com 256×256 pixels cada, dando a oportunidade para que esses quatro quadrados sejam processados em paralelo.

Cada vez que o quadrado é subdividido, novas tarefas são agendadas para serem executadas. Esse processo de divisão e agendamento tem um custo, ou seja, um *overhead*. Uma forma de controlar um bom balanço entre subdivisões e *overhead* é definindo um limite mínimo nas subdivisões. Esse limite é chamado de *grain size* (REINDERS, 2007), ou granularidade do algoritmo. Neste trabalho foi encontrado, após realizar diversos testes empíricos, que o melhor limite para maximar a taxa de quadros por segundo (FPS) é de um quadrado de tamanho 32×32. Valores acima deste tendem a piorar a FPS em cenas complexas pois a CPU acaba ficando muito tempo ociosa. No entanto, valores abaixo aumentam o *overhead* do agendador de tarefas de forma considerável.

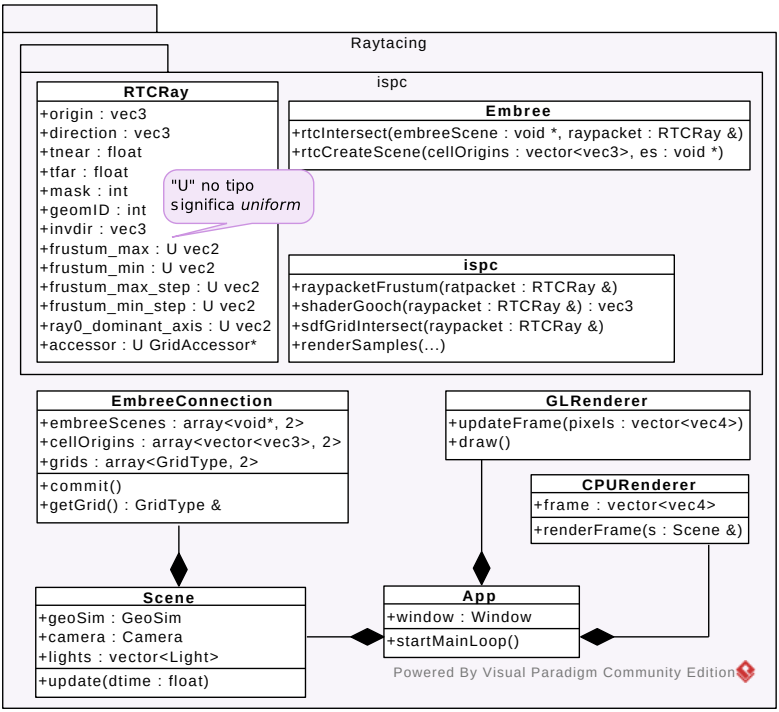
5.7 Arquitetura do Módulo de Ray Casting

A Figura 5.9 mostra o diagrama de classe do módulo de *ray casting*. O pacote "ispc" agrupa as classes representando os códigos que são executados em unidades SIMD. O ispc[6] é um compilador para uma linguagem de programação baseada em C para geração de códigos SIMD. Esse compilador é usado em conjunto com a biblioteca Embree (Seção 5.3.1) para maximizar a eficiência da travessia de raios na CPU.

A classe "RTCRay" armazena um pacote de raios. O ispc converte variáveis normais, como a "origin" e "direction" em uma representação do tipo SOA, como visto na Seção 5.2. Variáveis do tipo "uniform" não são convertidas em SOA, isso indica que as variáveis que descrevem o *frustum* do pacote de raios são armazenadas na classe normalmente, ou seja, existe apenas um *frustum* em cada pacote de raios. Cada pa-

[6] Intel SPMD Program Compiler. Disponível em: <https://ispc.github.io>. Acesso em: 23/05/2015.

Figura 5.9: Diagrama de Classes do Módulo de Ray Casting



Fonte: Produção do próprio autor.

cote de raios também armazena um ponteiro para um “GridAccessor”, que oferece recursos para acessar os voxels do volume da peça de forma eficiente. Como cada pacote de raios atravessa as mesmas células da grade, então não há necessidade de usar um “GridAccessor” por raio.

A classe “Embree” está representando, de forma simples, as principais funções da biblioteca Embree. O método “rtcCreateScene” faz a construção da BVH, como visto na Seção 5.3. A função recebe como parâmetro a origem dos nós folhas e um ponteiro indicando onde armazenar a cena. O método “rtcIntersect” é usado para atravessar um pacote de raios na BVH.

A classe “ispc” contém um conjunto de métodos necessários para a travessia do pacote de raios na grade dos nós folhas, como visto na Seção 5.3.2, e do sombreamento, como discutido na Seção 5.5.

A “App” é a classe principal da aplicação. Ela gerencia a janela da aplicação, a entrada de comandos e interação com o usuário, a cena e dois renderizadores. A cena é onde ficam informações como câmera e luzes, além da “GeoSim”, que é a classe que gerencia a simulação de usinagem, como discutida na Seção 4.6.

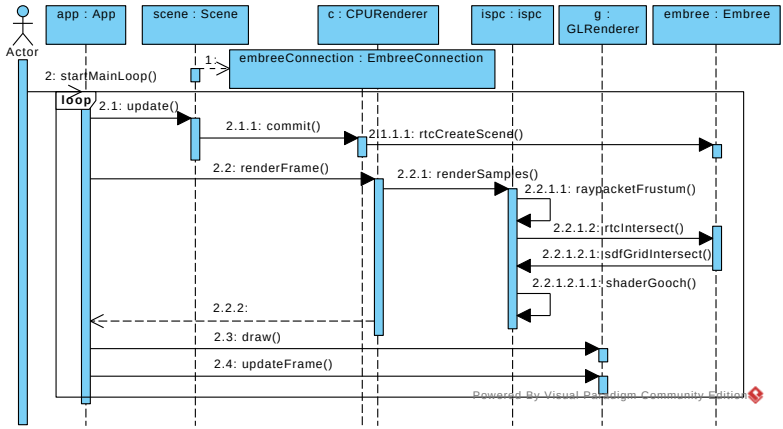
A classe “GLRenderer” faz a conexão com a OpenGL, que é usada para renderizar um quadro com uma textura 2D na tela. A função “updateFrame” envia uma textura 2D para a GPU, enquanto a função “draw” é usada para atualizar o quadro na janela da aplicação.

A classe “CPURenderer” faz a renderização de um quadro usando *ray casting*. A função “renderFrame” faz a divisão dos pixels (Seção 5.6), criação e lançamento dos pacotes de raios em direção a cena.

A classe “EmbreeConnection” faz a conexão entre a grade da peça que está armazenada na classe “GeoSim” com a cena que será usada para renderização. Essa classe armazena duas cenas da Embree, que são árvores BVH, e duas grades da VDB. A criação da BVH é um processo custoso, que pode demorar mais de um quadro para ser concluído. Por isso, a construção da BVH é feita em uma *thread* secundária para não bloquear a *thread* que faz a renderização da cena. Assim que a construção da BVH é concluída, então a BVH nova é colocada no lugar da BVH antiga. Duas grades também são mantidas para permitir que operações booleanas sejam feitas em paralelo com a renderização da cena, onde uma grade é usada para operações booleanas e a outra usada para renderização. Isso é necessário pois operações booleanas podem realocar e invalidar espaços de memória, causando problemas caso isso aconteça enquanto outra *thread* esteja fazendo a travessia dos raios nessas áreas. O método “commit” faz a criação da BVH que será usada para travessia dos raios, como descrito na Seção 5.3, e a atualização da grade que será usada para travessia dos raios.

A Figura 5.10 apresenta o diagrama de sequência para o módulo de *ray casting*. O *loop* principal da aplicação inicia com uma atualização da cena com o método “Scene::update”, que faz a atualização da câmera e invoca o método “EmbreeConnection::commit” quando necessário.

Figura 5.10: Diagrama de Sequência do Módulo de Ray Casting



Fonte: Produção do próprio autor.

Em seguida é feita a renderização de um quadro usando *ray casting*. Cada pacote de raio calcula o seu *frustum* com o método “ispc::raypacketFustrum”, como discutido na Seção 5.3.2. Em seguida o pacote é atravessado na BVH com “Embree::rtcIntersect”, e caso haja interseção com algum nó folha da VDB, o método “ispc::sdfGridIntersect” é usado para fazer a travessia do pacote na grade e verificar se houve ou não interseção com a superfície. Caso haja interseção, o sombreamento de Gooch, como visto na Seção 5.5, é calculado no método “ispc::shaderGooch”.

Assim que o quadro é renderizado, o quadro anterior é mostrado na tela com “GLRenderer::draw”, e o novo quadro é enviado para a GPU com “GLRenderer::updateFrame”. Esse *loop* é repetido até que o usuário termine a aplicação.

5.8 Discussão do Capítulo

Este capítulo apresentou a técnica de *ray casting* para renderização da simulação usada neste trabalho. Inicialmente é feita uma discussão entre o uso de GPGPU e CPU (Seção 5.1). O algoritmo de *ray casting* é trivial de ser paralelizado, assim sendo adequado para GPUs. Entretanto, este trabalho faz uso de modelos geométricos dinâmicos onde operações booleanas são feitas de forma sequencial, que são mais adequadas para CPU. Assim, com o modelo sendo gerado na CPU há a necessidade de sincronia dele com a GPU, que acaba sendo uma operação custosa. Por causa disso foi usado o algoritmo de *ray casting* para CPU para eliminar essa sincronia.

A técnica de pacotes de raios (Seção 5.2) foi usada para acelerar a travessia dos raios. A travessia é feita em duas etapas. Primeiramente os raios atravessam uma BVH (Seção 5.3.1) criada pela biblioteca Embree. Caso o pacote de raio faça interseção com um volume denso contendo uma grade representando os campos de distância da superfície, então a técnica de travessia de grades é feita (Seção 5.3.2).

Durante a travessia da grade é feita a reconstrução da superfície (Seção 5.4) para verificar se há interseção entre os raios e a superfície. Essa reconstrução transforma a representação discreta da superfície em uma representação contínua. A técnica de Marmitt et al. (2004) foi usada pois ela funciona de forma eficiente com pacotes de raios enquanto oferece resultados corretos.

Diferentes pacotes de raios são lançados em direção a cena de forma paralela para acelerar o algoritmo de *ray casting*, como a Seção 5.6 mostrou. Além disso, este trabalho fez uso de um *ray tracer* não recursivo. A principal vantagem dele é sua eficiência, pois usa apenas uma amostra, e apenas uma travessia da cena, por pixel. A desvantagem é que a técnica não gera imagens foto-realistas pois não simula efeitos reais como iluminação global, penumbras, *motion blur*, reflexos e outros. Entretanto, este trabalho aplica o sombreamento de Gooch et al. (1998)

(Seção 5.5), que não requer técnicas de *ray tracing* avançadas. Essa técnica de sombreamento não busca imagens foto-realistas, ao invés, ela realça as formas geométricas da peça.

6 TESTES E RESULTADOS

Diversos testes foram realizados para verificar a eficiência e usabilidade do projeto desenvolvido. Duas simulações foram usadas para realizar os testes, a *bear* e a *cameo*. Ambas são distribuídas com o projeto *opensource* OpenSCAM[1] no formato código-G como discutido na Seção 4.5. A simulação *bear* apresenta pouco mais de 15 mil blocos de comandos e usa uma fresa de topo plano, como na Figura 4.3, com tamanho aproximado de $1,5mm \times 18mm$. A sua peça inicial é um volume com as medidas $80mm \times 80mm \times 20mm$. Já a simulação *cameo* contém em torno de 35 mil blocos de comandos e usa uma fresa de topo esférico, como na Figura 4.4, com $1mm$ de raio e $4mm$ de comprimento. A sua peça inicial possui um volume de tamanho $167,2mm \times 207,9mm \times 6mm$.

Todos os testes foram realizados em um computador pessoal facilmente acessível a consumidores. O hardware é composto por uma CPU Intel i7 4770k @ 3.5GHz, RAM 16GB DDR3 2400MHz, GPU AMD RadeonHD 7990 e HDD 7200RPM. A configuração em software é Debian Testing, Linux 3.16.0, GCC 4.9.1, ispc 1.7.1dev[2], TBB 4.2[3], embree git28/11/2014[4] e o *driver opensource* radeon v7.5.

O software desenvolvido é composto por dois grandes módulos: módulo responsável pela manipulação geométrica (Seção 4.6) e o módulo de *ray casting* (Seção 5.7). A Seção 6.1 apresenta os testes relacionados a manipulação geométrica. Essa seção analisa a criação dos volumes de deslocamento, consumo de memória dos modelos da peça e as operações booleanas.

A Seção 6.2 discute os testes relacionados ao algoritmo de *ray*

[1] OpenSCAM. Disponível em: <openscam.com>. Acesso em: 27/12/2014.

[2] Intel SPMD Program Compiler. Disponível em: <https://ispc.github.io/>. Acesso em: 22/10/2014.

[3] Intel Threading Building Blocks. Disponível em: <https://www.threadingbuildingblocks.org/>. Acesso em: 22/10/2014.

[4] Embree. Disponível em: <https://embree.github.io/>. Acesso em: 22/10/2014.

casting, como construção da cena, travessia dos raios, testes de interseção e sombreamento.

A Seção 6.3, identifica e discute os principais gargalos do sistema.

Por último, na Seção 6.4 é feita uma comparação do software desenvolvido neste trabalho com o projeto *opensource* OpenSCAM.

6.1 Análise do Módulo de Manipulação Geométrica

Esta seção apresenta os resultados relacionados ao módulo responsável pela manipulação geométrica.

6.1.1 Criação dos Volumes de Deslocamento

As Tabelas 6.1 e 6.2 apresentam a quantidade de amostras para a criação de todos os volumes de deslocamentos da ferramenta durante a simulação dos testes *bear* e *cameo*. Lembrando que cada amostra é uma invocação da função de distância para um certo objeto geométrico. Essas tabelas foram criadas para para que trabalhos futuros que otimizam as funções de amostragem possam usá-las para comparação. Esses números serão analisados nas Tabelas 6.3 e 6.4.

A primeira coluna, “Res.” (Resolução), das Tabelas 6.1 e 6.2 descreve a resolução de grade usada no modelo geométrico. Por exemplo, uma resolução de 64 significa que o modelo ocupa aproximadamente $64 \times 64 \times 64$ *voxels*.

Como visto na Seção 4.3.5.2, quando um volume de deslocamento é gerado para um cilindro, cada amostra depende de uma função minimizadora usando a função de distância do cilindro como função objetora. A segunda coluna (“Cil.”-Cilindro), e terceira coluna, (“VD Cil.”-Volume de Deslocamento com Cilindro), das Tabelas 6.1 e 6.2 mostram a quantidade de amostras da função de distância do cilindro, e da distância do volume de deslocamento de um cilindro, respectivamente. As próxi-

mas duas colunas dessas tabelas, (“Cil.Oti.”-Cilindro Otimizado) e (“VD Cil.Oti.”-Volume de Deslocamento com Cilindro Otimizado), apresentam quantas amostras foram necessárias calcular quando usado o algoritmo otimizado para criação de volumes de deslocamentos de cilindros, como discutidos na Seção 4.4.3. A última coluna, “Esfera”, apresenta a quantidade de amostras da função de distância da esfera. Note que a Tabela 6.1 não mostra nenhuma amostra nesta coluna, pois a simulação *bear* usa uma ferramenta de topo plano, diferente da simulação *cameo* que usa uma ferramenta de topo esférica, como pode ser visto na Tabela 6.2.

Analisando as Tabelas 6.1 e 6.2 se concluí que o número de amostras aumenta proporcionalmente a resolução da grade. Lembrando que a resolução aumenta o número de voxels de forma exponencial.

Tabela 6.1: Número de amostras para criação dos volumes com o modelo *bear*

Res.	Cil.	VD Cil.	Cil.Oti.	VD Cil.Oti.	Esfera
64	276.245.889	28.361.578	108.392.582	11.088.982	-
128	969.666.758	100.219.573	320.133.205	32.741.590	-
256	4.059.283.466	420.312.747	1.202.445.911	122.494.809	-
512	17.991.869.936	1.861.483.384	6.282.044.544	641.460.503	-
1024	80.683.042.641	8.336.207.891	33.569.382.062	3.431.707.212	-
2048	361.513.025.767	37.302.724.352	170.113.027.497	17.394.774.482	-

Tabela 6.2: Número de amostras para criação dos volumes com o modelo *cameo*

Res.	Cil.	VD Cil.	Cil.Oti.	VD Cil.Oti.	Esfera
64	77.988.881	7.976.859	77.988.881	7.976.859	7.976.859
128	124.055.193	12.845.017	124.055.193	12.845.017	12.845.017
256	269.560.493	28.161.535	269.560.493	28.161.535	28.161.535
512	808.757.841	84.751.456	808.757.841	84.751.456	84.751.456
1024	3.159.797.636	330.586.473	3.159.797.636	330.586.473	330.586.473
2048	13.606.563.796	1.420.667.595	13.606.563.796	1.420.667.595	1.420.667.595

As seguintes tabelas, 6.3 e 6.4, apresentam algumas estatísticas para a simulação *bear* e *cameo*, respectivamente, usando dados das Tabelas 6.1 e 6.2.

A segunda coluna (“# Op.Bool.”-Número de Operações Boolea-

nas) das Tabelas 6.3 e 6.4 mostra o número de operações booleanas efetuadas para sua respectiva simulação. A terceira coluna ("Tol.(mm)"-Tolerância em Milímetros) apresenta a tolerância da representação geométrica. Por exemplo, para uma resolução de 2048^3 *voxels* o modelo geométrico da simulação *bear* é capaz de representar detalhes de até $\sim 0,04$ milímetros.

A quarta coluna das Tabelas 6.3 e 6.4 apresenta a relação entre o número de amostras da função de distância de cilindro com o de seu volume de deslocamento. Essa relação apresenta quantas amostras o algoritmo de Brent precisa realizar para encontrar a menor distância entre um segmento de reta e um cilindro, como visto na Seção 4.3.5.2. Para ambas as simulações, esses resultados demonstram que para cada amostra do volume de deslocamento, a função minimizadora faz aproximadamente $9,5$ chamadas a função objetora em média.

A quinta coluna ("Amostras/Vol."-Amostras por Volume) das Tabelas 6.3 e 6.4 apresenta a relação entre o número de amostras realizadas com o algoritmo de amostragem regular otimizado, Seção 4.4.1, com o algoritmo de amostragem específico para cilindros, Seção 4.4.3. Na simulação *bear* a função de amostragem específica para cilindros consegue ser entre ~ 2 até ~ 3 vezes mais eficiente que a versão de amostragem regular otimizada. Na simulação *cameo* não há nenhum ganho de desempenho na versão específica para cilindro, pois a diferença do comprimento do eixo principal do cilindro com seu raio não é grande o suficiente para que a otimização tenha efeito. A última coluna apresenta a média de amostras calculadas por volume.

Tabela 6.3: Estatísticas para criação dos volumes com o modelo *bear*

Res.	# Op.Bool.	Tol.(mm)	Cil./VD	Cil.Oti./Cil.	Amostras/Vol.
64	15152	1,250	9,740	0,392	731.849
128		0,625	9,675	0,330	2.160.876
256		0,312	9,658	0,296	8.084.399
512		0,156	9,665	0,349	42.335.038
1024		0,078	9,679	0,416	226.485.428
2048		0,039	9,691	0,471	1.148.018.379

Tabela 6.4: Estatísticas para criação dos volumes com o modelo *cameo*

Res.	# Op.Bool.	Tol.(mm)	Cil./VD	Cil.Oti./Cil.	Amostras/Vol.
64	35279	3,248	9,777	1,000	452.216
128		1,624	9,658	1,000	728.196
256		0,812	9,572	1,000	1.596.504
512		0,406	9,543	1,000	4.804.640
1024		0,203	9,558	1,000	18.741.261
2048		0,102	9,578	1,000	80.538.995

6.1.2 Consumo de Memória

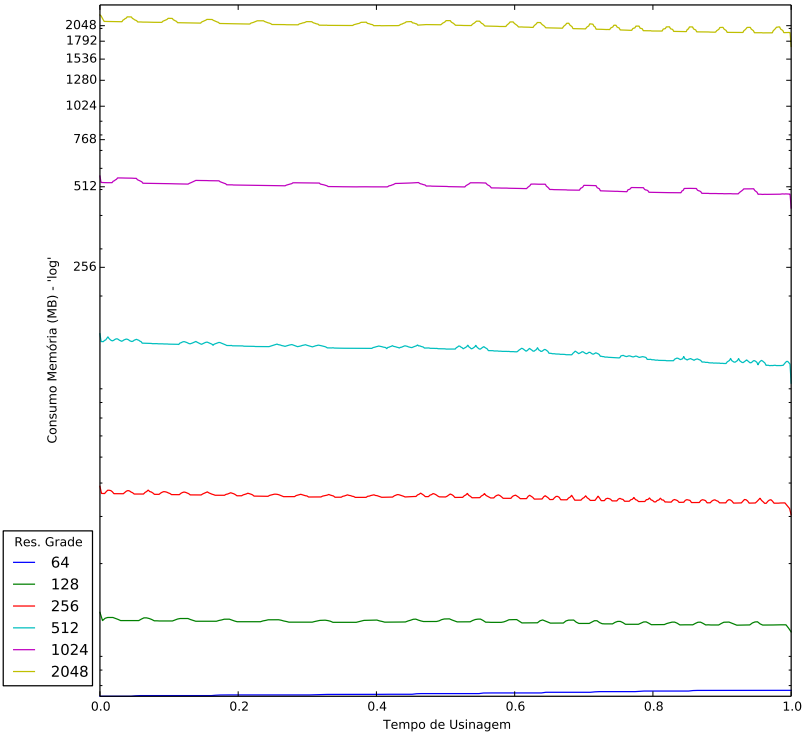
Um aspecto importante na simulação de usinagem é o seu consumo de memória. As Figuras 6.1 e 6.2 mostram a quantidade de memória usada no decorrer da execução da simulação para diferentes níveis de resolução. Os valores apresentados nessas imagens foram coletados observando o campo "*size*" do pseudo-arquivo "*proc/self/statm*", que é atualizado pelo *kernel* do Linux.

Em ambas as simulações foi possível perceber que não há grandes variações no consumo de memória no decorrer da simulação, e que a quantidade de memória requerida aumenta de forma exponencial em relação a resolução. Note que o gráfico está em uma escala logarítmica no eixo vertical.

6.1.3 Operações Booleanas

As Figuras 6.3 e 6.4 apresentam o tempo médio para execução de uma operação booleana nas simulações *bear* e *cameo*, respectivamente. Cada barra dos gráficos apresentam o tempo médio das operações booleanas com uma linha no topo da barra indicando o desvio padrão. O desvio padrão para todas as simulações é alto pois o tempo para executar uma operação booleana depende do número de *voxels* usados no volume. E o número de *voxels* de cada movimento da ferramenta depende do tamanho da ferramenta e da distância do movimento. Assim, como as simulações contêm diversos movimentos curtos e longos, o des-

Figura 6.1: Consumo de memória com o modelo *bear*



Fonte: Produção do próprio autor.

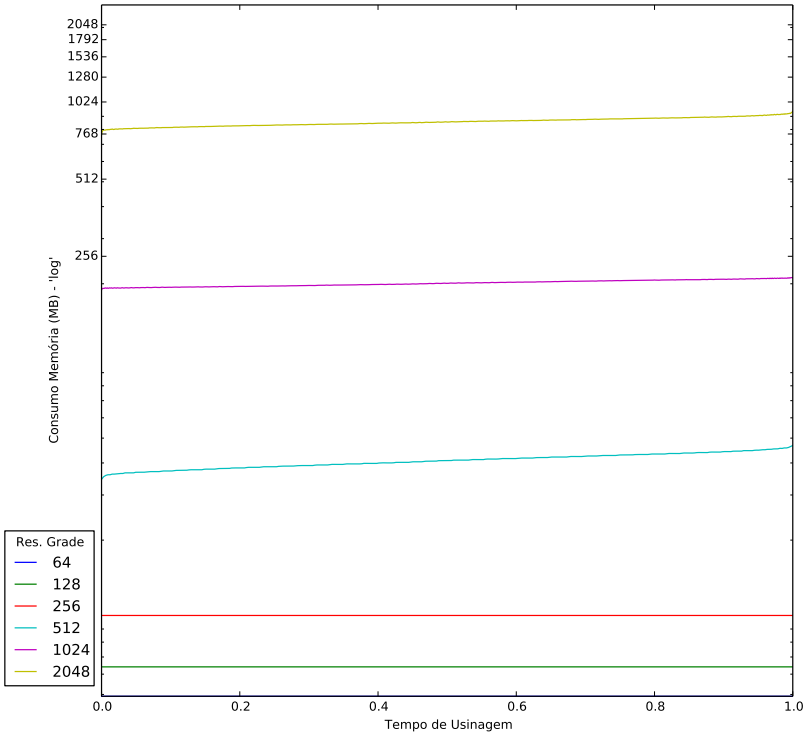
vio padrão acaba sendo grande.

6.1.4 Tempo total da simulação geométrica

O tempo total da simulação foi coletado após execução de 20 casos de testes para cada parâmetro. As Tabelas 6.5 e 6.6 apresentam os resultados coletados para as simulações *bear* e *cameo*, respectivamente.

Cada linha apresenta o tempo médio (em milissegundo), e seu desvio padrão (ω), para execução de uma simulação de usinagem completa para criação do volume de deslocamento. As colunas demarcam o

Figura 6.2: Consumo de memória com o modelo *cameo*



Fonte: Produção do próprio autor.

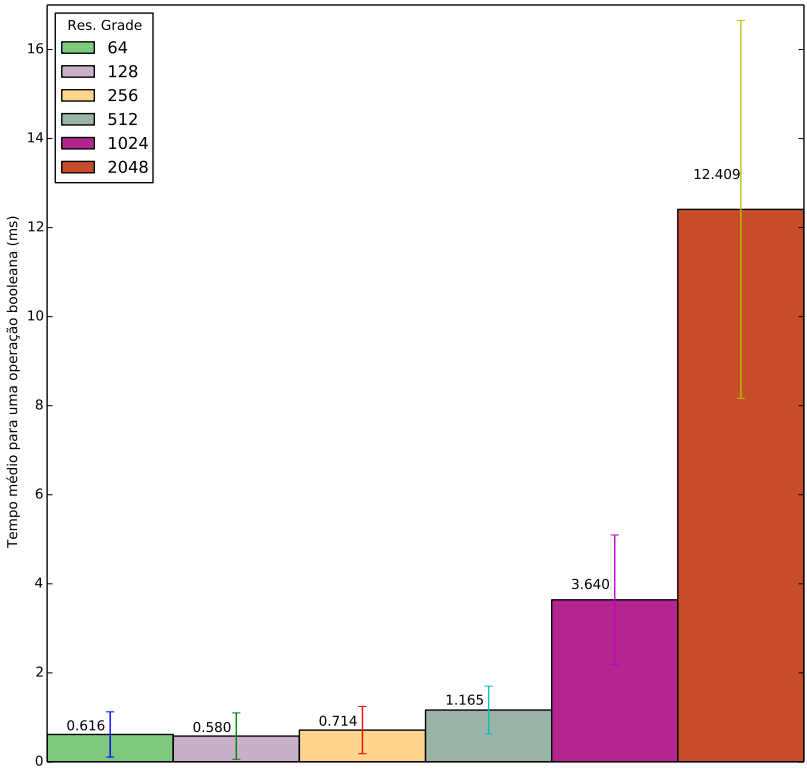
número de *threads* usadas na manipulação geométrica.

A simulação *bear* apresenta um tempo superior a *cameo*, que apresenta mais que o dobro de blocos de comandos. O principal motivo para essa diferença é o número de amostras usadas para criação dos volumes. A ferramenta na simulação *bear* apresenta um volume maior, por isso a simulação acaba demorando mais tempo.

As Figuras 6.5 e 6.6 mostram o *speed-up*[5] da manipulação geo-

[5] Métrica para desempenho relativo entre execução linear e paralela.

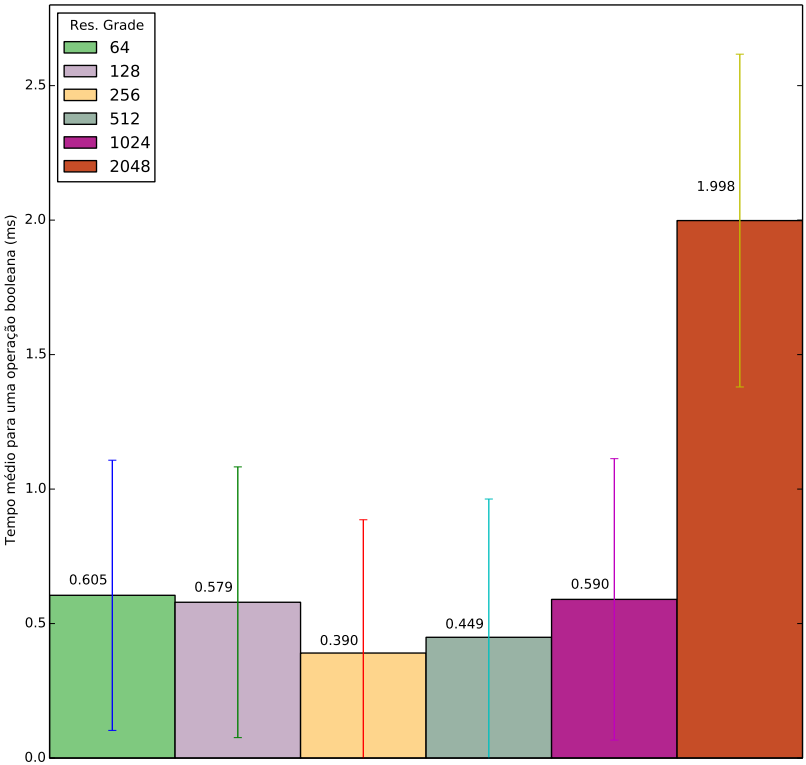
Figura 6.3: Tempo médio (ms) das operações booleanas com o modelo *bear*



Fonte: Produção do próprio autor.

métrica com relação ao número de *threads* usado. Quanto maior a resolução da grade, melhor é o aproveitamento dos recursos de paralelismo. Porém, um *speed-up* linear não foi alcançado. Isso porque há partes do sistema que apresentam processamento serial, entre elas as mais significativas são as operações booleanas e construção da BVH da cena. Além disso, operações de criação dos volumes não apenas calculam a distância de pontos com o volume de deslocamento, elas também precisam atravessar a árvore do modelo geométrico para acessar seus *voxels*. Esse tipo de acesso dificulta na ocupação da CPU, pois em situações

Figura 6.4: Tempo médio (ms) das operações booleanas com o modelo *cameo*



Fonte: Produção do próprio autor.

de *cache miss* e de escrita a memória RAM, a CPU pode ficar diversos ciclos sem utilização. Esses problemas são realçados na simulação *cameo*, onde o volume da ferramenta é menor, logo menos amostras são computadas assim tendo ainda mais regiões seriais.

Tabela 6.5: Tempo (ms) total da simulação geométrica do modelo *bear*

		# Threads				
Res.		1	2	4	8	12
64	Tempo	21.257	14.232	14.700	14.086	14.083
	ω	468	20	578	142	13
128	Tempo	26.598	16.794	15.534	15.277	15.322
	ω	321	13	74	68	80
256	Tempo	48.269	29.035	21.887	20.695	20.640
	ω	881	21	34	18	25
512	Tempo	150.145	89.826	62.828	52.260	50.675
	ω	53	37	33	65	165
1024	Tempo	664.244	382.731	256.404	208.690	200.550
	ω	590	521	449	323	233
2048	Tempo	3.096.055	1.746.505	1.123.902	909.186	874.438
	ω	1.460	370	1.495	2.151	1.800

Tabela 6.6: Tempo (ms) total da simulação geométrica do modelo *cameo*

		# Threads				
Res.		1	2	4	8	12
64	Tempo	49.638	37.528	37.690	37.867	37.888
	ω	1	30	128	281	323
128	Tempo	49.430	36.261	36.498	37.041	36.336
	ω	540	86	521	752	498
256	Tempo	38.885	25.372	24.277	24.369	23.934
	ω	106	35	11	188	172
512	Tempo	54.956	35.034	29.457	29.068	28.084
	ω	364	140	746	1.061	561
1024	Tempo	132.662	86.769	67.466	63.282	61.848
	ω	30	138	1.601	2.854	2.052
2048	Tempo	448.058	321.152	238.183	222.581	212.171
	ω	25	8.393	5.729	4.038	4.537

6.2 Análise do Módulo de Ray Casting

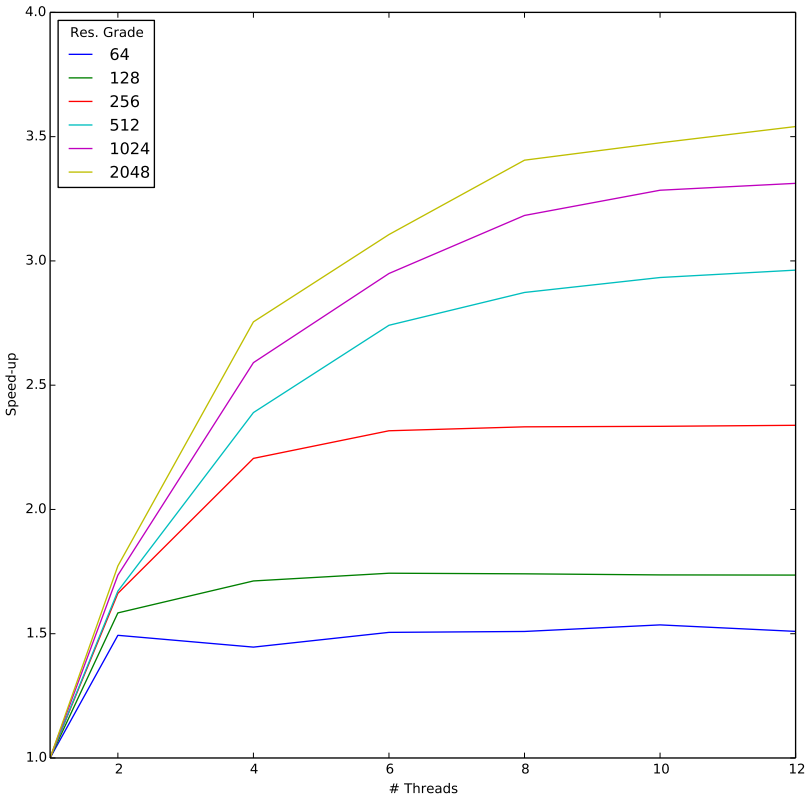
Esta seção apresenta os resultados relacionados ao módulo responsável pelo *ray casting*.

6.2.1 Construção da Cena

Como visto na Seção 5.3, o algoritmo de *ray casting* faz uso de uma BVH para aceleração da travessia dos raios. A construção dessa BVH é custosa, como pode ser visto nas Figuras 6.7 e 6.8 onde é mostrado o tempo médio para construção da BVH nas cenas da simulação *bear* e *cameo*, respectivamente.

Em ambas simulações, o tempo para construção da BVH não

Figura 6.5: *Speed-up* do tempo total da simulação do modelo *bear*



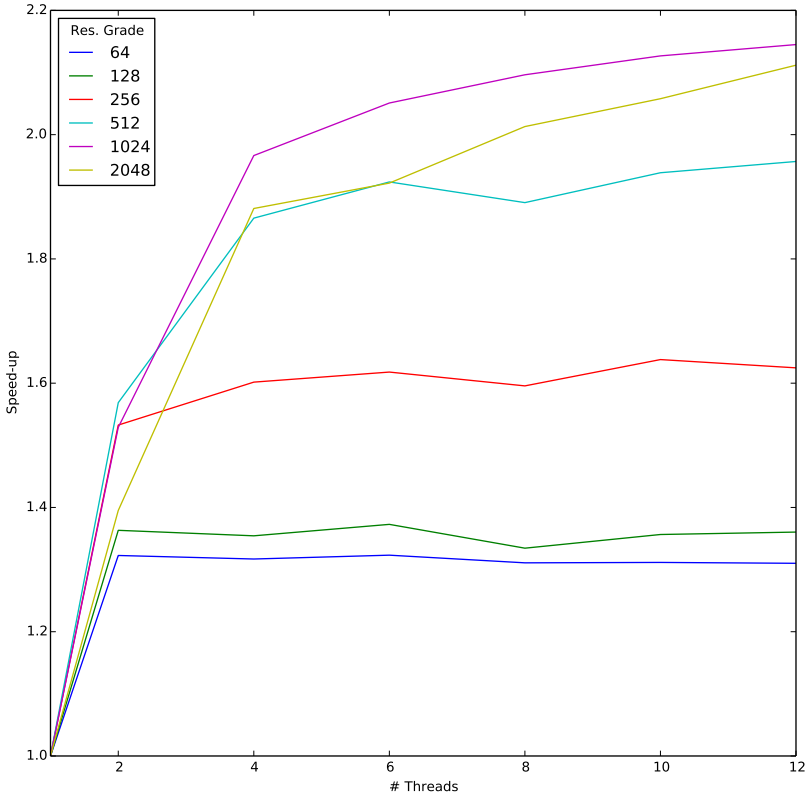
Fonte: Produção do próprio autor.

permite que essa operação seja feita a cada quadro. Entretanto, esse problema pode ser contornado fazendo a construção da árvore no plano de fundo, e ativando-a apenas quando ela estiver pronta.

6.2.2 Contagem de Raios por Segundo

Diversos fatores influenciam na contagem de raios que são processados por segundo, como: disponibilidade do processador, largura da

Figura 6.6: *Speed-up* do tempo total da simulação do modelo *cameo*



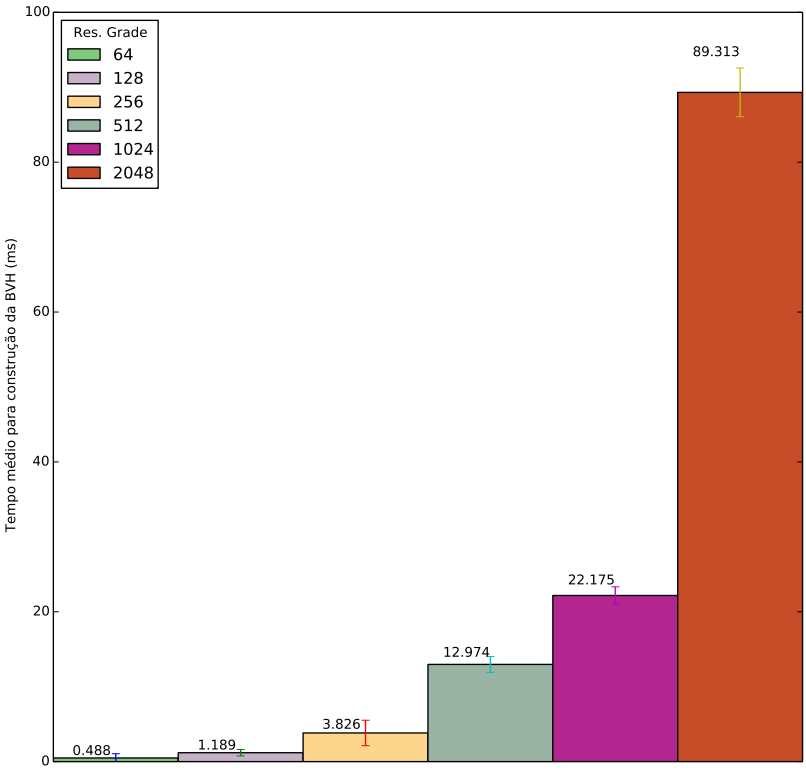
Fonte: Produção do próprio autor.

banda da memória sendo usada, ângulo de visão, número de *voxels* em frente a câmera, número de *voxels* percorridos até interseção, etc.

A metodologia adotada para obtenção do número de raios por segundo neste trabalho foi colocar uma câmera estática no topo da peça, como pode ser visto nas Figuras 6.11a e 6.11b para as simulações *bear* e *cameo*, respectivamente.

A Tabela 6.7 apresenta a média de raios por segundo que este

Figura 6.7: Tempo médio (ms) para construção da BVH com o modelo *bear*

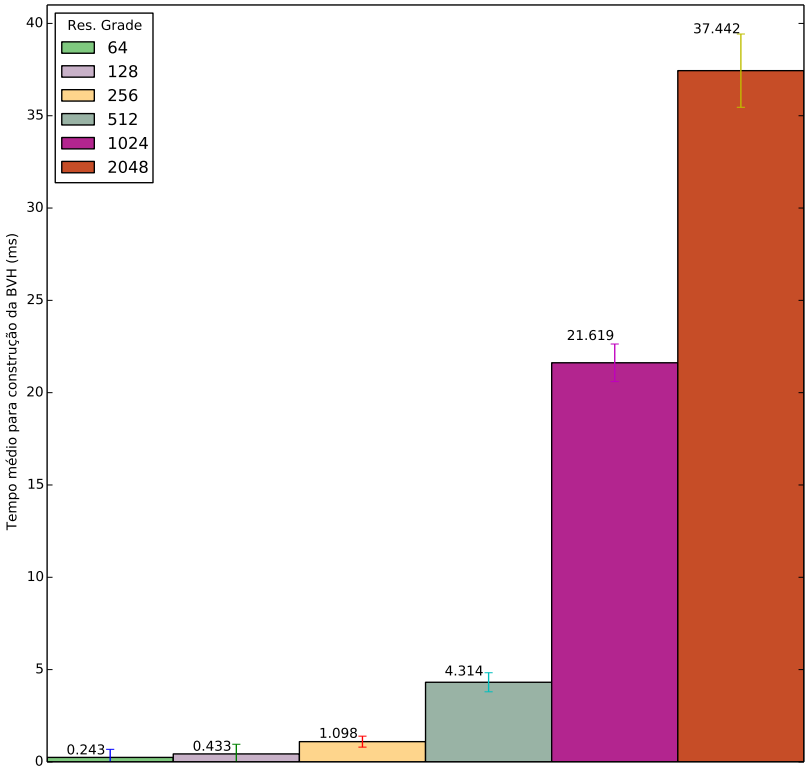


Fonte: Produção do próprio autor.

trabalho processa, seguido do seu desvio padrão. A quinta coluna, "*FPS (HD)*", mostra quantos quadros por segundo é feita a visualização para uma resolução de 1280×720 *pixels*.

Um dos objetivos deste trabalho é a simulação interativa, ou seja, permitir que o usuário possa acompanhar o processo de usinagem visualizando a peça de diversos ângulos. Cenas simples de usinagem alcançaram taxas de $\sim$16 FPS, o que permite uma navegação suave na cena. Para as cenas mais complexas este trabalho alcançou taxas de $\sim$4 FPS,

Figura 6.8: Tempo médio (ms) para construção da BVH com o modelo *cameo*



Fonte: Produção do próprio autor.

que, enquanto baixas e deixam a navegação na cena não suave, elas permitem que o usuário consiga, de forma interativa, posicionar a câmera no ângulo desejável.

6.3 Análise dos Gargalos

Uma análise geral do sistema foi feita, onde os diversos componentes foram avaliados para verificar as regiões críticas de desempenho.

Tabela 6.7: Raios por segundo

Simulação	Resolução	Raios/segundo	ω	FPS (HD)
Bear	64	8253,44	256,14	8,96
	128	15428,26	1941,55	16,74
	256	12639,08	328,41	13,71
	512	8025,60	304,44	8,70
	1024	4881,06	563,20	5,30
	2048	3115,88	337,75	3,38
Cameo	64	14509,29	184,05	15,74
	128	14100,48	174,38	15,29
	256	13926,40	217,22	15,11
	512	8685,38	143,49	9,42
	1024	6144,00	663,62	6,66
	2048	3801,60	158,99	4,12

A ferramenta *CPU Profiler* da gperftools[6] foi utilizada para verificar quais partes do sistema usam mais recursos computacionais. Essa ferramenta funciona interrompendo a execução do programa em torno de 100 vezes por segundo e coletando uma amostra contendo a pilha da rotina sendo executada no momento.

Os resultados desta sessão foram baseados na análise de aproximadamente 100 mil amostras coletadas durante a simulação do modelo *bear* com resolução 2048^3.

Os resultados mostram que $59,8\%$ do tempo são gastos na criação dos volumes de deslocamento. Dentro desses $59,8\%$, aproximadamente 94% do tempo são gastos calculando a função de distância para o volume de deslocamento, deixando apenas 6% para a gerência da estrutura de dados. Isso significa que o cálculo da função de distância é o principal fator para o desempenho deste trabalho.

As operações booleanas tiveram um custo de apenas $1,4\%$ do tempo total da simulação. Interessante notar que essas operações são feitas de forma sequencial. Como elas não contribuem de forma signifi-

[6] gperftools. Disponível em: <http://code.google.com/p/gperftools>. Acesso em: 24/11/14.

cante para o tempo total da aplicação, mesmo se elas fossem feita de forma paralela, o *speedup* seria insignificante.

Na parte de *ray casting*, $17,4\%$ do tempo total (note que esse tempo inclui todas as partes do sistema, incluindo manipulações geométricas) é usado na requisição dos *voxels* da estrutura de dados. O teste de interseção requer uma célula contendo oito *voxels*. Assim que a célula é buscada na estrutura de dados, o teste de interseção de Marmitt, como visto na Seção 5.4, contribui em apenas $1,2\%$ no tempo total da aplicação.

A construção da BVH ocupou apenas $2,1\%$ do tempo total da simulação. Enquanto a construção dos raios, incluindo o tempo para atualização da câmera e da criação dos pacotes de raios, usou apenas $1,6\%$ do tempo total.

6.4 Comparação com OpenSCAM

Esta seção apresenta uma comparação entre o método desenvolvido neste trabalho com a OpenSCAM. A OpenSCAM foi escolhida para comparação pois é uma solução *opensource*, assim permitindo análise de sua implementação e algoritmos usados, além do fácil acesso para realização de testes. Em seu núcleo, a OpenSCAM faz uso da técnica de MC para geração do modelo geométrico da peça final. Por causa disso, a OpenSCAM oferece meios para comparar o método desenvolvido neste trabalho com a técnica de MC.

As mesmas simulações efetuadas anteriormente com os modelos *bear* e *cameo* foram efetuadas com a ferramenta OpenSCAM e comparada com este trabalho. Ou seja, os mesmos arquivos NC contendo o código-G, e as mesmas configurações de ferramentas e ambiente de trabalho foram usadas.

Foram executados diversos testes para coleta do tempo total da simulação. Esses tempos foram coletados manipulando um cronometro de forma manual. Por isso, testes com altas tolerâncias foram ignorados

por causa da impressão do método. Entretanto, em tolerâncias pequenas, onde o tempo de execução pode ser medido em segundos, essa impressão tem menos importância. A Tabela 6.8 mostra o tempo total da simulação usando as duas simulações descritas anteriormente com diferentes níveis de tolerância. Todos os tempos são mostrados em segundos e a tolerância em milímetro.

Tabela 6.8: Tempo em segundos da simulação usando OpenSCAM

Simulação	Tol.(mm)	Tempo	ω
Bear	0.078	560	8,18
	0.039	4227,5	34,65
Cameo	0.203	42,33	2,52
	0.102	258,67	7,03

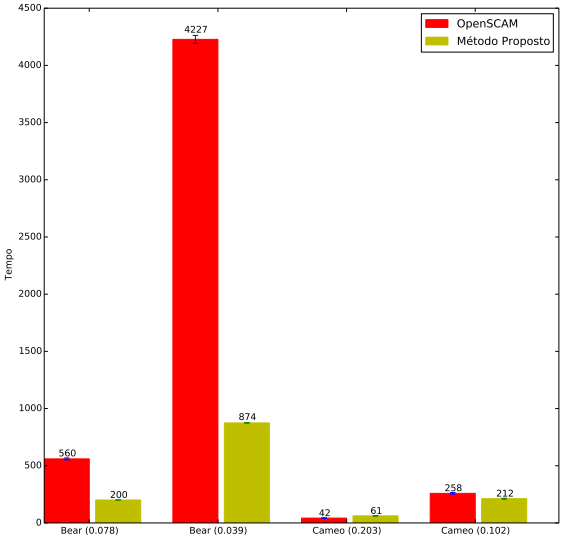
A Figura 6.9 apresenta uma comparação do tempo médio das simulações usando a OpenSCAM e a solução implementada neste trabalho. Os barras verticais indicam o tempo em segundos. A distribuição horizontal separa os testes indicados pelo nome da simulação e a tolerância adotada em parênteses.

A simulação *bear* é a que apresenta o maior contraste entre as duas tecnologias. Quanto menor a tolerância, maior a diferença entre o tempo total de simulação indicando que este trabalho tem uma melhor escalabilidade. Com uma tolerância de $0,039mm$ nesta simulação, este trabalho executa em média $4,8$ vezes mais rápido que a OpenSCAM.

Para simulações onde a peça é significativamente menor, como no caso da simulação *cameo*, não há uma diferença significativa no tempo da usinagem. Com uma tolerância de $0,203mm$, a OpenSCAM é aproximadamente 20 segundos mais rápida, porém, com uma tolerância de $0,102mm$, este trabalho ficou 46 segundos mais eficiente, em média.

Enquanto as simulações com baixa tolerância foram computadas e concluídas usando a OpenSCAM, nenhum modelo foi mostrado na

Figura 6.9: Comparação do tempo total em segundos da simulação com a OpenSCAM



Fonte: Produção do próprio autor.

tela[7]. Na simulação *bear*, apenas simulações com tolerância de $0,078mm$ ou maior foram mostradas na tela. Para a simulação *cameo*, a tolerância mínima foi de $0,203mm$. Já o método proposto não teve nenhum problema para visualizar as simulações com uma pequena tolerância. Para as simulações onde a OpenSCAM conseguiu mostrar na tela a peça final, a taxa de quadros por segundo foi alta o suficiente para não ser perceptível nenhuma lentidão no sistema.

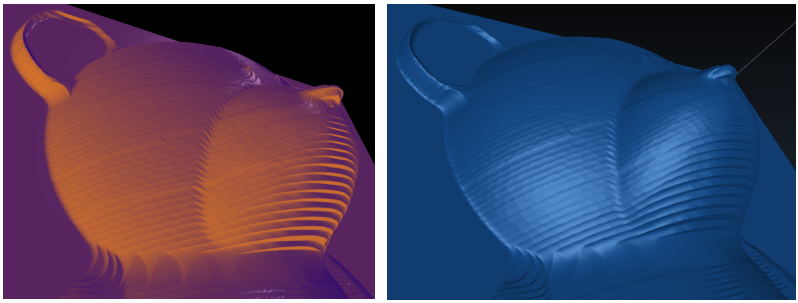
A Figura 6.10 compara os resultados visuais obtidos. OpenSCAM usa modelos geométricos representados por polígonos e um algoritmo de sombreamento simples. Para as imagens com o mesmo nível de tolerância, ambas as técnicas apresentam resultados visuais similares. Entretanto, o sombreamento de Gooch, como visto na Seção 5.5, consegue

[7] A mensagem *"radeon: Failed to allocate a buffer"* foi enviada para o terminal informando o erro.

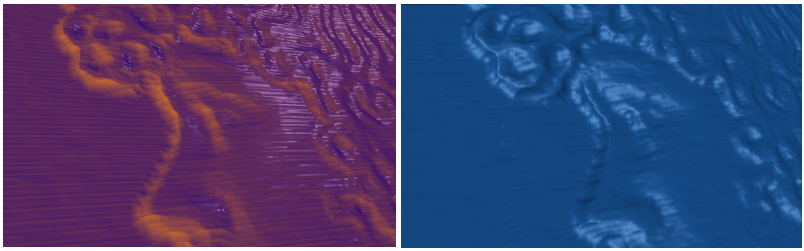
realçar melhor as deformidades do modelo, facilitando a visualização das marcas deixadas pela ferramenta.

Figura 6.10: Comparação dos resultados visuais entre o método proposto (MP) e a OpenSCAM

(a) MP - Bear(0,078mm) (b) OpenSCAM - Bear(0,078mm)



(c) MP - Cameo(0,203mm) (d) OpenSCAM - Cameo(0,203mm)



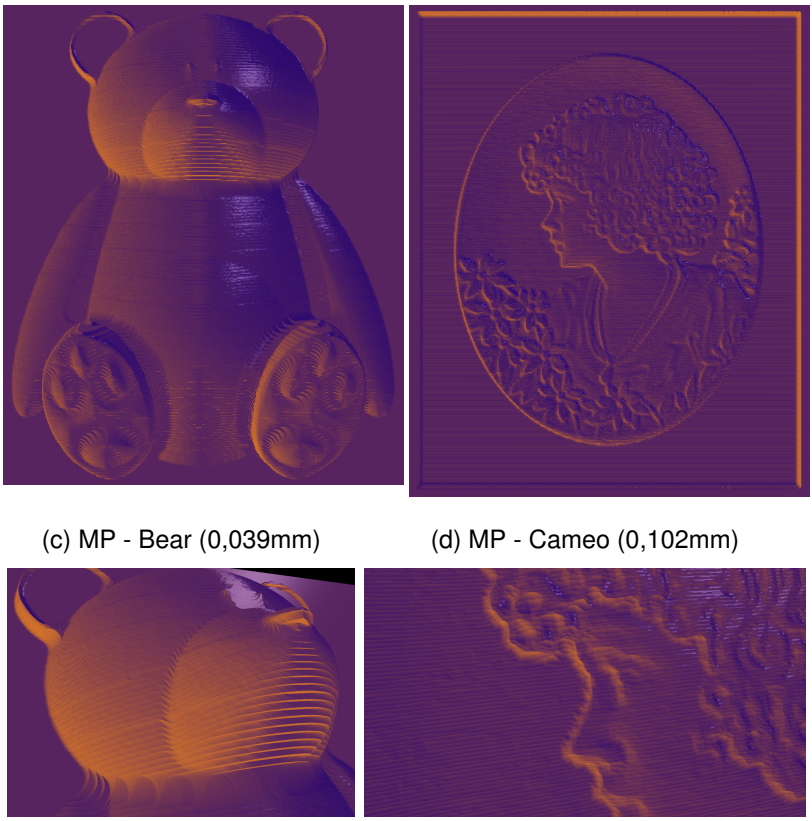
Fonte: Produção do próprio autor.

A Figura 6.11 apresenta os resultados visuais usando a metade da tolerância das imagens anteriores. Apenas o método proposto foi capaz de mostrar na tela o resultado final.

A OpenSCAM oferece tempo de processamento inferior ao do método proposto e uma qualidade de imagem similar para modelos com alta tolerância. Além disso, a OpenSCAM oferece um ambiente gráfico e taxas altas de atualização dos quadros, permitindo assim uma melhor

Figura 6.11: Resultados da simulação com baixa tolerância usando o método proposto

(a) MP - Bear (0,039mm) (b) MP - Cameo (0,102mm)



(c) MP - Bear (0,039mm) (d) MP - Cameo (0,102mm)

Fonte: Produção do próprio autor.

experiência para o usuário. Entretanto, modelos com baixa tolerância demoraram mais para serem processados na OpenSCAM. Além disso, a OpenSCAM não conseguiu mostrar o resultado final para o usuário, indicando falta de memória gráfica para visualização do modelo geométrico gerado. Já o método proposto conseguiu mostrar o mesmo modelo para o usuário, embora com taxas baixas de atualização dos quadros ($\sim$4 FPS).

7 CONCLUSÕES

Este trabalho teve como objetivo geral o desenvolvimento de um simulador de usinagem de alta precisão e interativo. Isso implica no uso de uma estrutura de dados que permite operações booleanas entre a peça e a ferramenta de forma eficiente, e que ocupe uma quantidade prática de memória. Além disso, a solução precisa oferecer visualização em tempo real da peça usinada.

As principais tecnologias na área de simulação de usinagem foram estudadas e comparadas. Entre elas, a representação usando campos de distância (DF) se mostrou a mais adequada para atender as necessidades do objetivo geral. A técnica de campos de distância torna operações booleanas triviais e eficientes. Além disso, DF são fáceis de serem manipuladas e armazenadas, pois podem ser interpretadas como *voxels*. Este trabalho fez uso de estruturas de dados *narrow-band level sets* para armazenar os *voxels* que representam a superfície da peça. Uma implementação eficiente, a OpenVDB, foi usada pois ela oferece acesso constante aos *voxels* da superfície além de usar uma estrutura hierárquica para agrupar regiões homogêneas, assim economizando espaço em memória.

Para a visualização interativa da peça foram estudadas as duas principais técnicas para renderização de cenários 3D: a rasterização e *ray tracing*. A principal desvantagem da rasterização é que ela exige uma transformação da representação usando campos de distância em uma malha de triângulos antes da visualização, o que dificulta manter uma taxa de quadros aceitável.

Por causa disso a técnica de *ray casting* foi escolhida para visualização interativa pois ela consegue renderizar superfícies representadas por DF de forma direta. Este trabalho estudou diversas técnicas de *ray tracing*, onde cada uma delas apresenta pontos positivos e negativos. Técni-

cas como *path tracing* e *metropolis light transport*, por exemplo, tendem a focar na geração de imagens de alta qualidade gráfica, principalmente em cenas foto-realísticas. Entretanto, o custo computacional dessas técnicas é elevado quando comparado com a técnica de *ray casting* ou WRT de Whitted (1980). A técnica de *ray casting* gera imagens não realísticas, porém é simples de implementar e eficiente. *Path tracing* pode necessitar de mais de 16 amostras por pixel para visualizar uma cena sem muito ruído, já a *ray casting* usa apenas uma amostra por pixel. Como o objetivo deste trabalho é a representação e visualização de pequenos detalhes em modelos geométricos, a técnica de *ray casting* é a mais adequada. Com a *ray casting* foi possível alcançar taxas de quadros por segundo aceitáveis (entre $\sim$3 e $\sim$16 quadros por segundo) para uma simulação de usinagem interativa.

Testes foram realizados para testar o tempo da usinagem e consumo de memória do processo de simulação de usinagem. Comparando com a OpenSCAM o método proposto apresenta desempenho similar para modelos com pouca resolução, porém é significantemente mais rápido para modelos mais complexos.

O consumo de memória do método proposto se mostrou estável durante todas as simulações, e previsível pois é proporcional ao número de *voxels* da peça inicial. Em simulações de alta resolução (2048^3 *voxels*), o consumo de memória foi de aproximadamente 2GB.

Os testes mostraram que foi possível alcançar visualização interativa com *ray casting*. Cenas com uma peça com resolução 128^3 foi renderizada com uma média de 16 quadros por segundo, e uma cena de resolução 2048^3 com aproximadamente 3 quadros por segundo. Em ambos os casos foi possível navegar na cena e visualizar diferentes partes da peça usinada de forma interativa. A OpenSCAM usa o método tradicional de visualização interativa, a rasterização acelerada em hardware. Durante todos os testes a OpenSCAM apresentou taxas de quadros por segundo altas o suficientes para não ser perceptível a mudança de quadros. Entretanto, a OpenSCAM não foi capaz de mostrar modelos mais

complexos que foram exibidos pelo sistema que utiliza o método proposto.

Todos os testes foram realizados em um computador pessoal de mesa (Intel i7-4770k), fazendo com que o método proposto seja de fácil acesso. O método desenvolvido neste trabalho conseguiu alcançar um *speed-up* máximo de pouco mais de duas vezes da versão sequencial em um processador com quatro núcleos. Isso indica que processadores com mais núcleos podem melhorar os tempos obtidos.

7.1 Trabalhos Futuros

Este trabalho não implementa uma solução pronta para ser usada em produção. O foco principal ficou no núcleo do sistema responsável pela manipulação geométrica e visualização da peça final. Assim, a sua usabilidade não é fácil e nem prática quando comparada com outros software como a OpenSCAM. Trabalhos futuros podem melhorar a usabilidade do sistema implementando uma interface gráfica para melhor controle da simulação onde o usuário pode avançar e retroceder blocos de comandos, visualizar o caminho da ferramenta, mudar a resolução e tolerância do modelo, obter diversas estatísticas sobre a usinagem, alteração do código-G em tempo de execução da simulação e etc.

Este trabalho implementou e desenvolveu apenas dois tipos de ferramentas: fresas de topo plano e de topo esférico. Uma solução mais completa pode estender este trabalho incluindo funções de distância para outros tipos de fresas. Além do tipo das fresas, apenas a movimentação linear da ferramenta foi desenvolvida. Assim, uma possível extensão para o método seria no desenvolvimento de movimentação em arcos.

Em termos de desempenho, o método pode ser melhorado usando outras técnicas como GPGPU para aceleração dos cálculos de distância dos volumes de deslocamento. Como visto na Seção 6.3, quase 60% do tempo de processamento é gasto fazendo esses cálculos. Neste trabalho o paralelismo foi alcançado criando diferentes volumes em diferentes *threads*, onde cada amostra é computada de forma isolada e em

paralelo. Essa característica faz com que essa parte do algoritmo seja adequada para uso de GPGPU. Volumes grandes que ocupam mais espaço em memória do que disponível em GPUs ainda podem ser acelerados dessa forma usando *data streaming*.

O algoritmo de *ray casting* se mostrou suficiente para visualização da peça a uma taxa de atualização dos quadros aceitável, porém ainda baixa. Trabalhos futuros podem testar outras técnicas de visualização da cena. A Seção 6.3 mostrou que, durante a travessia do raio na cena, a parte mais custosa é a de aquisição dos *voxels* na estrutura de dados. Uma possível otimização seria a transformação dos *voxels* em uma malha poligonal e usar LOD (*level of details*). Entretanto, há o potencial para o aumento da complexidade do algoritmo caso esse caminho seja adotado.

A técnica de visualização implementada teve o foco principal na representação e percepção correta da forma geométrica da peça sendo usinada. Porém, outras aplicações podem exigir gráficos que simulam a realidade. Para isso, *ray tracing* é uma técnica adequada, podendo ser expandida implementado outras características como iluminação global, refração, sombras, métodos de renderização baseados na física, renderização volumétrica, *subsurface scattering* e outras.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDEL-MALEK, K. et al. Swept volumes: Fundation, perspectives, and applications. **International Journal of Shape Modeling**, v. 12, n. 01, p. 87–127, 2006.

ACCARY, A. B.; GALIN, E. **Fast distance computation between a point and cylinders, cones, line swept spheres and cone-spheres**. [S.l.], 2004. Validé par : comite Journal of Graphics Tools.

AILA, T.; LAINE, S. Understanding the efficiency of ray traversal on gpus. In: **Proc. High-Performance Graphics 2009**. [S.l.: s.n.], 2009.

ALIAGA, D. et al. **A Framework for the Real-Time Walkthrough of Massive Models**. Chapel Hill, NC, USA, 1998.

AMANATIDES, J.; WOO, A. A fast voxel traversal algorithm for ray tracing. In: **In Eurographics '87**. [S.l.: s.n.], 1987. p. 3–10.

ANDERSON, R. Detecting and eliminating collisions in {NC} machining. **Computer-Aided Design**, v. 10, n. 4, p. 231 – 237, 1978. ISSN 0010-4485.

BÆRENTZEN, J.; CHRISTENSEN, N. **Manipulation of volumetric solids with applications to sculpting**. Tese (Doutorado), 2002.

BAXTER III, W. V. et al. Gigawalk: Interactive walkthrough of complex environments. In: **Proceedings of the 13th Eurographics Workshop on Rendering**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002. (EGRW '02), p. 203–214. ISBN 1-58113-534-3.

BENTHIN, C. **Realtime Ray Tracing on current CPU Architectures**. Tese (Doutorado) — Saarland University, 2006.

BENTHIN, C. et al. Combining Single and Packet Ray Tracing for Arbitrary Ray Distributions on the Intel(R) MIC Architecture. **IEEE Transactions on Visualization and Computer Graphics**, 2011. (accepted for publication).

BIKKER, J. **RAY TRACING IN REAL-TIME GAMES**. Tese (Doutorado) — Technische Universiteit Delft, 2012.

BLOOMENTHAL, J. Implicit surfaces. **Encyclopedia of Computer Science and Technology**, Marcel Decker, 2001.

BOULOS, S. et al. Packet-based whitted and distribution ray tracing. In: **Proceedings of Graphics Interface 2007**. New York, NY, USA: ACM, 2007. (GI '07), p. 177–184. ISBN 978-1-56881-337-0.

BRENT, R. **Algorithms for Minimization Without Derivatives**. [S.l.]: Dover Publications, 1973. (Dover Books on Mathematics). ISBN 9780486419985.

BRUNET, P.; NAVAZO, I. Solid representation and operation using extended octrees. **ACM Transactions on Graphics**, ACM, v. 9, n. 2, 1990.

CHAPPEL, I. T. The use of vectors to simulate material removed by numerically controlled milling. **Computer-Aided Design**, v. 15, n. 3, p. 156 – 158, 1983. ISSN 0010-4485.

CIGNONI, P. et al. Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In: **ACM SIGGRAPH 2004 Papers**. New York, NY, USA: ACM, 2004. (SIGGRAPH '04), p. 796–803. Disponível em: <http://doi.acm.org/10.1145/1186562.1015802>.

COOK, R. L.; PORTER, T.; CARPENTER, L. Distributed ray tracing. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 18, n. 3, p. 137–145, jan. 1984. ISSN 0097-8930.

CRASSIN, C. et al. Interactive indirect illumination using voxel cone tracing. **Computer Graphics Forum (Proceedings of Pacific Graphics 2011)**, v. 30, n. 7, sep 2011.

DIETRICH, A.; STEPHENS, A.; WALD, I. Exploring a boeing 777: Ray tracing large-scale cad data. **Computer Graphics and Applications, IEEE**, v. 27, n. 6, p. 36–46, 2007. ISSN 0272-1716.

DU, S. et al. Formulating swept profiles for five-axis tool motions. **International Journal of Machine Tools and Manufacture**, v. 45, n. 7–8, p. 849 – 861, 2005. ISSN 0890-6955.

ERDIM, H.; ILIEŞ, H. T. Classifying points for sweeping solids. **Computer-Aided Design**, v. 40, n. 9, p. 987 – 998, 2008. ISSN 0010-4485.

FRISKEN, S. et al. Adaptively sampled distance fields: A general representation of shape for computer graphics. In: **ACM SIGGRAPH**. [S.l.: s.n.], 2000. p. 249–254. ISBN 1-58113-208-5.

FRISKEN, S. F.; PERRY, R. N. Designing with distance fields. In: **ACM SIGGRAPH 2006 Courses**. New York, NY, USA: ACM, 2006. (SIGGRAPH '06), p. 60–66. ISBN 1-59593-364-6.

GASTER, B. et al. **Heterogeneous computing with OpenCL, 2nd Edition**. [S.l.]: Morgan Kaufmann, 2011.

GEIMER, M.; MÜLLER, S. A cross-platform framework for interactive ray tracing. In: **Tagungsband Graphiktag der Gesellschaft für Informatik**. Frankfurt/Main, Germany: [s.n.], 2003. p. 25–34.

GOBBETTI, E.; KASIK, D.; YOON, S.-e. Technical strategies for massive model visualization. In: **Proceedings of the 2008 ACM symposium on Solid and physical modeling**. New York, NY, USA: ACM, 2008. (SPM '08), p. 405–415. ISBN 978-1-60558-106-4.

GOOCH, A. et al. A non-photorealistic lighting model for automatic technical illustration. In: **Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1998. (SIGGRAPH '98), p. 447–452. ISBN 0-89791-999-8.

HAGAN, R.; BRALEY, C. Numerical methods for isosurface volume rendering. **Virginia Tech**, 2009.

HANNIEL, I.; HALLER, K. Direct rendering of solid cad models on the gpu. In: **Computer-Aided Design and Computer Graphics (CAD/Graphics), 2011 12th International Conference on**. [S.l.: s.n.], 2011. p. 25–32.

HEUSINGER, S. et al. Integrating the cax process chain for step-compliant nc manufacturing of asymmetric parts. **International Journal of Computer Integrated Manufacturing**, v. 19, n. 6, p. 533–545, 2006. Disponível em: <http://dx.doi.org/10.1080/09511920600622098>.

HUGHES, J. F. et al. **Computer graphics: principles and practice (3rd ed.)**. Boston, MA, USA: Addison-Wesley Professional, 2013. 1264 p. ISBN 0321399528.

HUNTER, G. M.; STEIGLITZ, K. Operations on images using quad trees. **IEEE Trans. Pattern Anal. Mach. Intell.**, IEEE Computer Society, Washington, DC, USA, v. 1, n. 2, p. 145–153, fev. 1979. ISSN 0162-8828.

International Organization for Standardization (Ed.). **ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF**. 1996.

JAMRIŠKA, O. Interactive ray tracing of distance fields. In: CITESEER. **14th Central European Seminar on Computer Graphics**. [S.l.], 2010. p. 91.

JENSEN, H. W. Global illumination using photon maps. In: **Proceedings of the Eurographics Workshop on Rendering Techniques '96**. London, UK, UK: Springer-Verlag, 1996. p. 21–30. ISBN 3-211-82883-4.

JERARD, R. et al. Methods for detecting errors in numerically controlled machining of sculptured surfaces. **Computer Graphics and Applications, IEEE**, v. 9, n. 1, p. 26–39, Jan 1989. ISSN 0272-1716.

JONES, M.; BAERENTZEN, J.; SRAMEK, M. 3d distance fields: a survey of techniques and applications. **Visualization and Computer Graphics, IEEE Transactions on**, v. 12, n. 4, p. 581–599, July 2006. ISSN 1077-2626.

JU, T. et al. Dual contouring of hermite data. In: **Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 2002. (SIGGRAPH '02), p. 339–346. ISBN 1-58113-521-1.

KAJIYA, J. T. The rendering equation. In: **Proceedings of the 13th annual conference on Computer graphics and interactive techniques**. New York, NY, USA: ACM, 1986. (SIGGRAPH '86), p. 143–150. ISBN 0-89791-196-2.

KARUNAKARAN, K. et al. Octree-based nc simulation system for optimization of feed rate in milling using instantaneous force model. **The International Journal of Advanced Manufacturing Technology**, Springer-Verlag, v. 46, n. 5-8, p. 465–490, 2010. ISSN 0268-3768.

KARUNAKARAN, K. P.; SHRINGI, R. Octree-to-brep conversion for volumetric nc simulation. **The International Journal of Advanced Manufacturing Technology**, Springer, v. 32, n. 1, p. 116(16), 2007.

KAWASHIMA, Y. et al. A flexible quantitative method for nc machining verification using a space-division based solid model. **The Visual Computer**, Springer-Verlag, v. 7, n. 2-3, p. 149–157, 1991. ISSN 0178-2789.

KAY, T. L.; KAJIYA, J. T. Ray tracing complex scenes. In: **Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1986. (SIGGRAPH '86), p. 269–278. ISBN 0-89791-196-2.

KIRK, D.; HWU, W. **Programming Massively Parallel Processors: A Hands-on Approach**. [S.l.]: Morgan Kaufmann Publishers, 2010. (Applications of GPU Computing Series). ISBN 9780123814722.

KNOLL, A. et al. Interactive isosurface ray tracing of large octree volumes. In: **Interactive Ray Tracing 2006, IEEE Symposium on**. [S.l.: s.n.], 2006. p. 115–124.

KRAMER, T. R.; PROCTOR, F. M.; MESSINA, E. **The NIST RS274/NGC Interpreter- Version 3**. 2010.

LAFORTUNE, E. **Mathematical Models and Monte Carlo algorithms for physically based rendering**. Tese (Doutorado) — Katholieke Universiteit Leuven, 1996.

LAINE, S.; KARRAS, T. **Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation**. [S.l.], 2010.

LAUTERBACH, C. et al. Reducem: Interactive and memory efficient ray tracing of large models. In: **Proceedings of the Nineteenth Eurographics Conference on Rendering**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. (EGSR '08), p. 1313–1321. Disponível em: <http://dx.doi.org/10.1111/j.1467-8659.2008.01270.x>.

LORENSEN, W. E.; CLINE, H. E. Marching cubes: A high resolution 3d surface construction algorithm. In: **Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1987. (SIGGRAPH '87), p. 163–169. ISBN 0-89791-227-6.

MACDONALD, D. J.; BOOTH, K. S. Heuristics for ray tracing using space subdivision. **Vis. Comput.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 6, n. 3, p. 153–166, maio 1990. ISSN 0178-2789.

MARMITT, G. et al. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In: **in Proceedings of Vision, Modeling, and Visualization (VMV**. [S.l.: s.n.], 2004. p. 429–435.

MEAGHER, D. Geometric modeling using octree encoding. **Computer Graphics and Image Processing**, v. 19, n. 2, p. 129 – 147, 1982. ISSN 0146-664X.

MORTENSON, M. **Geometric Modeling, 3rd Edition**. [S.l.]: Industrial Press, 2006.

MUSETH, K. Vdb: High-resolution sparse volumes with dynamic topology. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 32, n. 3, p. 27:1–27:22, jul. 2013. ISSN 0730-0301.

NEUBAUER, A. et al. Cell-based first-hit ray casting. In: **Proceedings of the Symposium on Data Visualisation 2002**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002. (VISSYM '02), p. 77–ff. ISBN 1-58113-536-X.

NEWMAN, T. S.; YI, H. A survey of the marching cubes algorithm. **Computers and Graphics**, v. 30, n. 5, p. 854 – 879, 2006. ISSN 0097-8493.

OVERBECK, R.; RAMAMOORTHI, R.; MARK, W. Large ray packets for real-time whitted ray tracing. In: **Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on**. [S.l.: s.n.], 2008. p. 41–48.

PARK, J. W.; SHIN, Y. H.; CHUNG, Y. C. Hybrid cutting simulation via discrete vector model. **Computer-Aided Design**, v. 37, n. 4, p. 419 – 430, 2005. ISSN 0010-4485.

PENG, C.; CAO, Y. A gpu-based approach for massive model rendering with frame-to-frame coherence. **Comp. Graph. Forum**, John Wiley & Sons, Inc., New York, NY, USA, v. 31, n. 2pt2, p. 393–402, maio 2012. ISSN 0167-7055.

PHARR, M.; HUMPHREYS, G. **Physically Based Rendering: From Theory to Implementation**. [S.l.]: Morgan Kaufmann, 2010. (Morgan Kaufmann series in interactive 3D technology). ISBN 9780123750792.

PORTO, A. J. V. et al. Manufatura Virtual: conceituação e desafios. **Gestão & Produção**, scielo, v. 9, p. 297 – 312, 12 2002. ISSN 0104-530X.

REINDERS, J. **Intel Threading Building Blocks**. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN 9780596514808.

REQUICHA, A. A. G.; ROSSIGNAC, J. Solid modeling and beyond. **Computer Graphics and Applications, IEEE**, v. 12, n. 5, p. 31–44, Sept 1992. ISSN 0272-1716.

ROMISCH, K. **Sparse Voxel Octree Ray Tracing on the GPU**. Dissertação (Mestrado) — Aarhus University, Dinamarca, 2009.

ROSSO JR, R. S. U. **STEP Compliant CAD/CAPP/CAM System For Rotational Asymmetric Parts**. Tese (Doutorado) — Loughborough University, 2005.

RUBIN, S. M.; WHITTED, T. A 3-dimensional representation for fast rendering of complex scenes. In: **Computer Graphics**. [S.l.: s.n.], 1980. p. 110–116.

SAMET, H. The quadtree and related hierarchical data structures. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 16, n. 2, p. 187–260, jun. 1984. ISSN 0360-0300.

SCHWARZE, J. Cubic and quartic roots. In: GLASSNER, A. S. (Ed.). **Graphics Gems**. [S.l.]: Academic Press, 1990. p. 404–407.

SHIRLEY, P.; MARSCHNER, S. **Fundamentals of Computer Graphics**. 3rd. ed. Natick, MA, USA: A. K. Peters, Ltd., 2009. ISBN 1568814690, 9781568814698.

SMITH, A. R. **A Pixel Is Not A Little Square, A Pixel Is Not A Little Square, A Pixel Is Not A Little Square! (And a Voxel is Not a Little Cube**. [S.l.], 1995.

SUFFERN, K. **Ray tracing from the ground up**. [S.l.]: A K Peters, 2007. (Ak Peters Series). ISBN 9781568812724.

SULLIVAN, A. et al. High accuracy {NC} milling simulation using composite adaptively sampled distance fields. **Computer-Aided Design**, v. 44, n. 6, p. 522 – 536, 2012. ISSN 0010-4485.

THEISEL, H. On properties of contours of trilinear scalar fields. In: ____. **Curve and Surface Design: Saint Malo 1999**. [S.l.]: Vanderbilt University Press, 2000.

TIAN, F. et al. Adaptive voxels: interactive rendering of massive 3d models. **The Visual Computer**, Springer-Verlag, v. 26, n. 6-8, p. 409–419, 2010. ISSN 0178-2789.

VEACH, E. **Robust Monte Carlo Methods for Light Transport Simulation**. Tese (Doutorado) — Stanford University, 1997.

VEACH, E.; GUIBAS, L. J. Metropolis light transport. In: **Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997. (SIGGRAPH '97), p. 65–76. ISBN 0-89791-896-7.

WÄCHTER, C.; KELLER, A. Instant ray tracing: The bounding interval hierarchy. In: **Proceedings of the 17th Eurographics Conference on Rendering Techniques**. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006. (EGSR'06), p. 139–149. ISBN 3-905673-35-5.

WALD, I. **Realtime Ray Tracing and Interactive Global Illumination**. Tese (Doutorado) — Computer Graphics Group, Saarland University, 2004.

WALD, I. On fast construction of sah-based bounding volume hierarchies. In: **Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on**. [S.l.: s.n.], 2007. p. 33–40.

WALD, I.; BOULOS, S.; SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 26, n. 1, jan. 2007. ISSN 0730-0301.

WALD, I.; DIETRICH, A.; SLUSALLEK, P. An interactive out-of-core rendering framework for visualizing massively complex models. In: **ACM SIGGRAPH 2005 Courses**. New York, NY, USA: ACM, 2005. (SIGGRAPH '05). Disponível em: <http://doi.acm.org/10.1145/1198555.1198756>.

WALD, I.; HAVRAN, V. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In: **Interactive Ray Tracing 2006, IEEE Symposium on**. [S.l.: s.n.], 2006. p. 61–69.

WALD, I. et al. Ray tracing animated scenes using coherent grid traversal. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 25, n. 3, p. 485–493, jul. 2006. ISSN 0730-0301.

WALD, I. et al. Interactive rendering with coherent ray tracing. **Computer Graphics Forum**, Blackwell Publishers Ltd, v. 20, n. 3, p. 153–165, 2001. ISSN 1467-8659.

WALD, I. et al. Embree: A kernel framework for efficient cpu ray tracing. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 33, n. 4, p. 143:1–143:8, jul. 2014. ISSN 0730-0301.

WANG, W.; WANG, K. Geometric modeling for swept volume of moving solids. **Computer Graphics and Applications, IEEE**, v. 6, n. 12, p. 8–17, Dec 1986. ISSN 0272-1716.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, ACM, New York, NY, USA, v. 23, n. 6, p. 343–349, jun. 1980. ISSN 0001-0782.

YAU, H. T.; TSOU, L. S.; TONG, Y. C. Adaptive nc simulation for multi-axis solid machining. **Computer-Aided Design and Applications**, v. 2, n. 1-4, p. 95–104, 2005.

ZHANG, Y.; XU, X.; LIU, Y. Numerical control machining simulation: a comprehensive survey. **International Journal of Computer Integrated Manufacturing**, v. 24, n. 7, p. 593–609, 2011.

ZHU, M.; HOUSE, D.; CARLSON, M. Ray casting sparse level sets. In: **Proceedings of the Digital Production Symposium**. New York, NY, USA: ACM, 2012. (DigiPro '12), p. 67–72. ISBN 978-1-4503-1649-1.