UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCUS AURELIUS FARIAS

Operações Booleanas entre Objetos Delimitados por Surfels Usando Constrained BSP-trees

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. João Luiz Dihl Comba
Orientador

Prof. Dr. Luiz Velho
Co-orientador

Porto Alegre, março de 2006

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Farias, Marcus Aurelius

Operações Booleanas entre Objetos Delimitados por Surfels Usando Constrained BSP-trees / Marcus Aurelius Farias. – Porto Alegre: PPGC da UFRGS, 2006.

63 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2006. Orientador: João Luiz Dihl Comba; Co-orientador: Luiz Velho.

1. Surfels. 2. BSP-tree. 3. CSG. I. Comba, João Luiz Dihl. II. Velho, Luiz. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Pró-Reitor de Coordenação Acadêmica: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profª. Valquíria Linck Bassani
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

RESUMO 5

ABSTRACT 6

1 INTRODUÇÃO 7

2 CONCEITOS 10
2.1 Surfels 10
2.2 Operações booleanas de objetos descritos por surfels 13
2.3 Estruturas de dados hierárquicas 13
2.3.1 Quadtree e Octree 14
2.3.2 KD-Tree 15
2.3.3 BSP-Tree 16
2.4 Componentes principais 18
2.4.1 Matriz de covariância 18
2.4.2 Autovetores e autovalores 19
2.4.3 Componentes principais 20

3 CBSP-TREE 21
3.1 Motivação 21
3.2 Conceito de CBSP-Tree 21
3.3 Alternativas de criação de CBSP-trees 23

4 OPERAÇÕES BOOLEANAS 24

5 PARTICIONAMENTO 27
5.1 Escolha de particionadores 27
5.1.1 Escolha PCA 27
5.1.2 Escolha de candidatos 28
5.2 Critérios de parada 29
5.2.1 Critério de alinhamento de surfels 29
5.2.2 Critério da profundidade da árvore 29
5.3 Particionadores de *trim* 30
5.3.1 Particionadores de *trim* em 2D 30
5.3.2 Particionadores de *trim* em 3D 33
5.3.3 Particionamento especial das células-folhas 37
5.4 Discussão 38

6 CLASSIFICAÇÃO DE INTERIOR E EXTERIOR 40
6.1 Células vizinhas 40
6.2 Classificação 43
6.3 Discussão 45
7 RESULTADOS 2D 46
7.1 Particionamento 46
7.2 Operações booleanas 47
8 RESULTADOS 3D 52
8.1 Particionamento 52
8.2 Operações booleanas 56
9 CONCLUSÃO 59
9.1 Trabalhos futuros 59
REFERÊNCIAS 61

RESUMO

As áreas de visualização e modelagem baseados em pontos têm sido pesquisadas ativamente na computação gráfica. Pontos com atributos (por exemplo, normais) são geralmente chamados de *surfels* e existem vários algoritmos para a manipulação e visualização eficiente deles. Um ponto chave para a eficiência de muitos métodos é o uso de estruturas de particionamento do espaço. Geralmente octrees e KD-trees, por utilizarem cortes alinhados com os eixos são preferidas em vez das BSP-trees, mais genéricas. Neste trabalho, apresenta-se uma estrutura chamada *Constrained BSP-tree* (CBSP-tree), que pode ser vista como uma estrutura intermediárias entre KD-trees e BSP-trees. A CBSP-tree se caracteriza por permitir cortes arbitrários desde que seja satisfeito um critério de validade dos cortes. Esse critério pode ser redefinido de acordo com a aplicação. Isso permite uma aproximação melhor de regiões curvas. Apresentam-se algoritmos para construir CBSP-trees, valendo-se da flexibilidade que a estrutura oferece, e para realizar operações booleanas usando uma nova classificação de interior/exterior.

Palavras-chave: Surfels, BSP-tree, CSG.

Boolean Operations on Surfel-Bounded Objects using Constrained BSP-Trees

ABSTRACT

Point-based modeling and rendering is an active area of research in Computer Graphics. The concept of points with attributes (e.g. normals) is usually referred to as surfels, and many algorithms have been devised to their efficient manipulation and rendering. Key to the efficiency of many methods is the use of partitioning schemes, and usually axis-aligned structures such as octrees and KD-trees are preferred, instead of more general BSP-trees. In this work we introduce a data structure called Constrained BSP-tree (CBSP-tree) that can be seen as an intermediate structure between KD-trees and BSP-trees. The CBSP-tree is characterized by allowing arbitrary cuts as long as all cuts satisfy a certain criterion, which can be specified differently for every application. These features allow better approximation of curved regions. We discuss algorithms to build CBSP-trees using the flexibility that the structure offers, and present a modified algorithm for boolean operations that uses a new inside-outside object classification.

Keywords: Surfels, BSP-tree, CSG.

1 INTRODUÇÃO

A computação gráfica é utilizada em várias aplicações de natureza tanto científica como robótica, simulações e prototipagem; quanto de entretenimento, como filmes e jogos. A modelagem de objetos necessita de uma forma apropriada de representação, para poder armazenar, exibir e manipular no computador modelos dos objetos desejados. Uma forma amplamente conhecida e muito usada é a representação poligonal. Conectando vários polígonos, de modo a formar uma malha, é possível representar uma enorme variedade de formas e objetos, bastando diminuir o tamanho desses polígonos à medida que mais detalhamento é necessário, e aumentar nas partes mais simples do objeto. Essa representação atinge um bom equilíbrio entre poder descritivo e custo computacional. Os triângulos são o tipo de polígono mais usado para esse propósito, já que seus três vértices são garantidamente coplanares (o mesmo não pode ser dito de um quadrilátero: o quarto vértice pode não estar no mesmo plano descrito pelos três primeiros vértices). O uso de malhas de triângulos é tão importante que os processadores das placas gráficas atuais têm uma arquitetura especialmente projetada para o processamento eficiente de vários triângulos em paralelo, aplicando transformações geométricas, texturas e efeitos de iluminação durante o processo.

Por outro lado, existem elementos naturais em que a representação poligonal não é muito apropriada. Por exemplo, para representar nuvens, fogo, água, fumaça, poeira e até mesmo plantas seria necessário utilizar uma quantidade muito grande de pequenos triângulos, desperdiçando poder computacional em algo que poderia ser modelado mais facilmente por simples pontos, como é feito em sistemas de partículas [1]. A forma indefinida e desconexa desses elementos também contribui para a dificuldade em aplicar a representação poligonal. Para contrastar as características das abordagens por polígonos e por pontos, podemos usar o exemplo da água: enquanto uma malha de polígonos pode servir convenientemente para representar e fazer a animação de uma superfície d'água, é muito mais apropriado representar e fazer a animação das gotículas d'água usando simplesmente um conjunto de pontos.

Existem casos, porém, em que mesmo a representação de objetos sólidos pode se beneficiar com uma representação por pontos. Pode-se citar dois exemplos importantes, descritos a seguir.

O primeiro exemplo refere-se a objetos com detalhes finos, por exemplo grande quantidade de curvas na sua forma. Quanto mais detalhes se deseja num

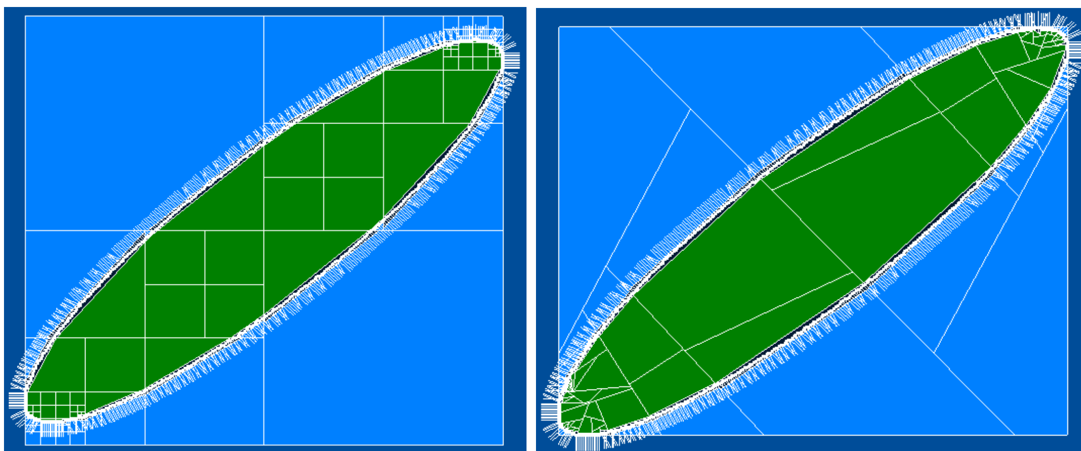
modelo, menores têm que ser os triângulos usados. É fácil perceber que, quando um modelo usa triângulos grandes, apenas seus três vértices são suficientes para descrever um grande pedaço de superfície (ainda que isso possa causar uma aparência não-natural em objetos curvilíneos). Já no caso de triângulos muito pequenos, o processo de transformação dos três vértices num conjunto de pontos que representa o triângulo (processo chamado de *rasterization* ou *rasterização*) se torna mais custoso, e às vezes até desnecessário, pois certos triângulos representam um pedaço de superfície tão pequeno que aparecem na tela do computador como apenas uns poucos pixels. Por outro lado, uma representação direta por pontos poderia exigir menos processamento na rasterização.

O segundo exemplo, mais importante, refere-se ao uso de *scanners 3D*. Esses aparelhos, com algoritmos adequados, são capazes de obter a forma de objetos tridimensionais do mundo real e levá-la ao computador. A forma natural de fazer isso é obter as coordenadas de vários pontos da superfície do objeto, junto com os vetores normais, gerando, portanto, uma nuvem de pontos. Esses pontos com normais, por representarem uma superfície, são chamados de *surfels*, isto é *surface elements*. Esses fatos impulsionaram muita pesquisa para ter-se um conjunto completo de técnicas para processamento de surfels, desde a aquisição de modelos até a visualização dos pontos, passando por técnicas de modelagem, como CSG.

Geometria sólida construtiva (CSG — *Constructive solid geometry*) é uma ferramenta útil de modelagem de objetos gráficos. Com a CSG, pode-se aplicar operações booleanas entre objetos primitivos para construir objetos com formas mais complexas. Também é possível usar essas operações para editar sólidos obtidos pelos *scanners 3D*. A habilidade de efetuar as operações de união, interseção e diferença entre modelos de surfels aumenta significativamente o poder dessa primitiva gráfica. Com esse objetivo, vários algoritmos já foram implementados. Um deles foi descrito em [2] onde se usa a estrutura de dados *octree* para executar as operações booleanas, servindo como auxiliar na classificação dos surfels que comporão o resultado da operação.

O objetivo deste trabalho é apresentar uma nova estrutura de dados, a CBSP-tree (Constrained BSP-tree) para as operações booleanas entre modelos descritos por surfels. Essa estrutura surgiu da necessidade de uma melhor aproximação de formas curvas do que as octrees e kd-trees permitem e ao mesmo tempo limitando a complexidade das células de uma BSP-tree. A figura 1.1 ilustra a diferença no particionamento que pode ser obtida, aproximando melhor a forma do objeto com uma CBSP-tree. A CBSP-tree será usada no algoritmo proposto por Adams e Dutré [2]. Ao contrário do que acontece com uma octree, na construção de estruturas de particionamento espacial como a CBSP-tree e a BSP-tree, existem muitas alternativas a considerar, e não se pode dizer que uma delas seja necessariamente melhor que as outras. Por isso, este trabalho apresentará algumas dessas alternativas de construção e sugerirá algumas outras para trabalhos futuros.

A construção de uma CBSP-tree depende de várias decisões a serem tomadas durante a implementação. Uma delas é a escolha da posição e da inclinação do particionador do espaço. Este trabalho descreverá várias estratégias de particionamento e alternativas serão discutidas visando alcançar uma divisão espacial eficiente.



Quadtree, CBSP-tree

Figura 1.1: Comparação das estruturas Quadtree e CBSP-tree, ilustrando a diferença no estilo de particionamento. A primeira repete sempre o mesmo tipo de corte, enquanto que a CBSP-tree procura seguir a forma do objeto.

O objetivo é ter células pequenas representando as bordas dos objetos, com pouco espaço vazio; enquanto células grandes representam o espaço vazio do interior e do exterior do objeto. Outro item discutido é o critério de parada da divisão celular recursiva. Vários testes podem ser aplicados a uma célula para descobrir se ela ainda precisa ser subdividida, como seu tamanho, quantidade de surfels e posicionamento deles. Neste trabalho, as diferentes estratégias de particionamento e critérios de parada foram implementadas independentemente, de modo que possam ser combinadas para descobrir uma variedade de particionamentos possíveis.

O presente trabalho está estruturado da seguinte forma: O capítulo 2 descreve os conceitos básicos necessários para o resto do texto. O capítulo 3 apresenta a idéia da CBSP-tree, com a motivação e descrição da estrutura. O capítulo 4 descreve brevemente a idéia de operações booleanas entre sólidos, mostra exemplos e dá uma visão geral do algoritmo. Os capítulos 5 e 6 discutem a implementação do trabalho, sendo que o capítulo 5 descreve as várias estratégias para a construção da estrutura de dados enquanto que o capítulo 6 detalha a implementação do algoritmo de classificação de interior e exterior. Por último, os capítulos 7 e 8 mostram os resultados obtidos e o capítulo 9 conclui o trabalho.

2 CONCEITOS

Este capítulo apresenta os conceitos básicos utilizados nesta dissertação. A seção 2.1 descreve a primitiva gráfica usada na modelagem de objetos. A seção 2.2 dá uma rápida visão geral sobre operações booleanas entre objetos de surfels. A seção 2.3 explica as várias estruturas de dados hierárquicas (também conhecidas como *estruturas de dados espaciais*) existentes e faz comparações entre elas. Por fim, os conceitos matemáticos sobre componentes principais são explicados na seção 2.4.

2.1 Surfels

Quando se diz, neste trabalho, que um modelo é representado por pontos, quer dizer que o objeto é descrito com amostras dos infinitos pontos que formam a sua superfície, e cada um desses pontos contém propriedades que aproximam localmente as características dessa superfície. Essa é a definição de *surfel*, dada no ano 2000 por Pfister et al. [3]. Mas já em 1985 [4], os pontos foram propostos como primitiva de representação para a qual todas as outras representações poderiam ser convertidas para a renderização. O nome *surfel* vem da expressão em inglês *surface element*, que quer dizer *elemento da superfície*, em analogia a outros termos já existentes, como *pixel* (*picture element*) e *voxel* (*volume element* ou *volume pixel*). As propriedades básicas de um surfel são sua posição e seu vetor normal. Propriedades adicionais podem ser: cor (que pode vir de uma textura), raio, e quaisquer outras. Apesar de chamarmos os surfels de *amostras de pontos*, é comum dar-lhes uma propriedade *raio*, para definir quanto de superfície eles representam. Assim, pode-se usar uma amostragem não-homogênea de surfels, já que cada surfel conterá o seu raio de alcance. Essa propriedade permite, por exemplo, representar melhor as bordas do modelo.

O uso de surfels como primitivas de renderização e de modelagem, como é de se esperar, possui tanto vantagens como desvantagens. Em geral, diz-se que os pontos complementam os triângulos, mas não os substituem. O desenho de cada surfel é simples e rápida de ser feita, mesmo em software apenas (sem *hardware* gráfico especializado). Os surfels permitem objetos com alto grau de detalhamento, e os algoritmos são dependentes da complexidade da saída; ou seja, quando é necessário exibir pouco detalhamento, o tempo de processamento é reduzido. Isso também quer dizer que há muitas possibilidades de escolher diferentes níveis de detalhamento e

equilibrar desempenho e qualidade de imagem. É também possível beneficiar-se de grande paralelização no algoritmo de renderização, aproveitando também a *localidade* da informação (já que cada surfel carrega consigo todos os seus atributos). Uma das desvantagens dos surfels fica por conta da ineficiência ao representar grandes superfícies planas, pois a coerência não é aproveitada. [5]

É interessante notar que não há informação de conectividade entre os surfels. Isso pode ser visto como um benefício na obtenção de modelos através de *scanners 3D* e em manipulações da topologia dos modelos [5]. Além disso, considerando que o objetivo final de produzir imagens computacionalmente é gerar uma matriz de pontos (pixels) que deve ser desenhada na tela do computador, a informação de conectividade não é estritamentente necessária. Naturalmente, os algoritmos usados neste processo são diferentes daqueles usados no desenho das tradicionais malhas de triângulos.

Quanto à obtenção dos surfels, existe mais de uma forma de gerar modelos. Uma delas é converter um modelo existente a partir de outro formato, que pode ser: superfícies implícitas, NURBS, malhas de triângulos ou outro. Outra maneira é adquirir o modelo a partir de um objeto do mundo real, usando algum tipo de *scanner 3D* [6, 7]. O *scanner* por *contato* toca a superfície do objeto para obter sua posição. Os métodos por transmissão usam ondas, como por exemplo, o *raio X*. Já os métodos por reflexão utilizam ondas como a da luz ou do som e, a partir da reflexão dessas ondas, são capazes de detectar os detalhes do objeto.

Esses processos de escaneamento, entretanto, não obtêm a cor do objeto e as propriedades de reflexão da luz em cada ponto. É possível obter-se uma aproximação da aquisição de cores do objeto tirando fotos dele. Depois disso, é necessário fazer um mapeaento das fotos 2D para o modelo 3D, utilizando os parâmetros de projeção da câmera.

Depois de obtidos os surfels, um importante passo a ser feito é a renderização, isto é, o desenho dos surfels para formar uma imagem no computador. Em 1998, inspirado pelos avanços em na área de *image based rendering*, o trabalho de Grossman and Dally [8, 9] desenvolveu um algoritmo que permite iluminação dinâmica e tem desempenho sensível à complexidade da saída. O trabalho de Pfister et al. [3] extendeu a idéia de Grossman e Dally com um controle de *nível de detalhe* (LOD) hierárquico, e também com uma técnica para eliminar surfels ocultos. Posteriormente o artigo [10], focando em filtragem de texturas com alta qualidade, melhorou a renderização dos surfels. O trabalho [11] descreveu um eficiente algoritmo de renderização valendo-se da estrutura hierárquica de representação dos surfels.

A figura 2.1 ilustra como um surfel pode ser representado por discos que se sobrepõem. O raio de cada surfel deve ser suficiente para que a superfície desenhada não contenha buracos e pode ser conseguido a partir da taxa de amostragem do *scanner*. Mesmo assim, o algoritmo de renderização precisa fazer interpolação para evitar buracos causados pela projeção perspectiva ou pela magnificação da imagem. As figuras 2.2 e 2.2 mostram algumas visualizações de modelos feitos com surfels.

Além de métodos de aquisição e de renderização de amostras de pontos, é importante haver algoritmos para a edição dos dados. A edição pode ser feita tanto

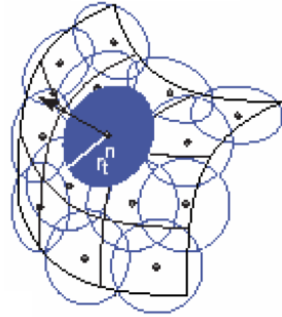
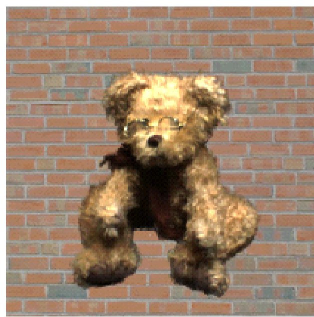


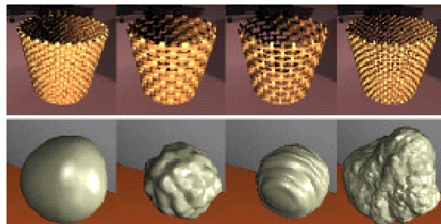
Figura 2.1: Surfels como discos tangentes, com as propriedades *centro*, *normal* e *raio*. [3]



[Matusik et al. 2002]

Figura 2.2: Exemplos de objetos de surfels. Estas figuras mostram como os surfels podem representar detalhes como o pêlo do ursinho e as asas do anjo, ambos obtidos por scanners. [12]

[Zwicker et al. 2001]



[Stamminger et al. 2001]

[Deussen et al. 2001]

Figura 2.3: Exemplos de objetos de surfels. Estas figuras mostram objetos sintéticos variados. [12]

para criar novos modelos ou adicionar detalhes quanto para corrigir imperfeições do processo de aquisição. O Pointshop3D é um programa de código aberto que permite editar interativamente superfícies descritas por surfels. A sua arquitetura permite adicionar novos *plug-ins* e novas ferramentas (*tools*), o que torna o programa bastante flexível e útil para a pesquisa de computação gráfica baseada em pontos.

2.2 Operações booleanas de objetos descritos por surfels

Além de criar modelos usando *scanners 3D* ou convertendo outras representações para a representação por pontos, existe a possibilidade de combinar objetos com operações booleanas. Essas operações são importantes, pois permitem criar objetos novos e mais complexos a partir da união, interseção e diferença de objetos já existentes. A figura 2.4 mostra vários exemplos de operações booleanas entre um modelo de uma cabeça e de uma espiral.

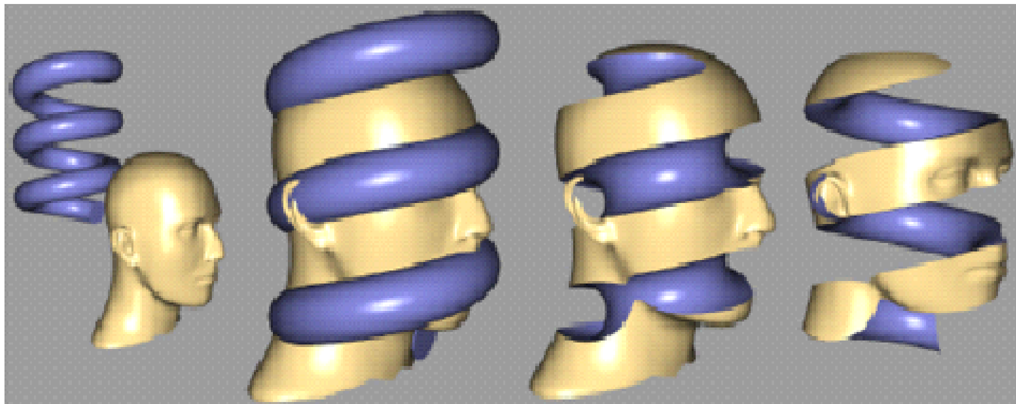


Figura 2.4: Exemplos de operações booleanas feitas entre uma cabeça e uma espiral. Da esquerda para a direita: objetos separados, união, diferença (cabeça-espiral) e interseção [2].

Algoritmos eficientes foram propostos para resolver as operações booleanas entre objetos delimitados por surfels [2, 13, 14]. Nesses algoritmos, é necessário testar freqüentemente que partes de um objeto estão dentro do outro. Esse problema exigiria muitos testes de todos os surfels de um objeto contra todos os surfels do outro. Por isso, para reduzir a complexidade dele, se usam estruturas de dados espaciais como octrees e KD-trees, que serão explicadas em mais detalhes na seção 2.3.

2.3 Estruturas de dados hierárquicas

Ao lidar com formas e gráficos, como conjuntos de polígonos, conjuntos de *surfels* ou conjuntos de *pixels* numa imagem, é freqüentemente desejável fazer uma subdivisão do espaço, de forma que dados próximos e semelhantes sejam agrupados. Essa estratégia pode também ser utilizada em algoritmos não relacionados com computação gráfica, como em pesquisa e ordenação de dados.

Para agrupar os dados, ou objetos, várias estruturas de dados hierárquicas foram criadas [15]. Elas funcionam dividindo o espaço do problema recursivamente, como exemplifica a seqüência de passos abaixo:

1. dividir o espaço de dados em subespaços menores, usando hiperplanos[1];
2. para cada um dos sub-espaços, testar se é necessário subdividi-lo;
3. em caso afirmativo, repetir o algoritmo nesse sub-espaço a partir do passo 1.

O diagrama da figura 2.5 mostra uma forma comum de representar este tipo de estrutura de dados. Um nome mais usual para a estrutura é *árvore*. O elemento que dá origem aos outros (o superior) é chamado de *raiz*, enquanto que os últimos (aqueles que não têm mais sub-itens) são chamdos de *folhas* da árvore.

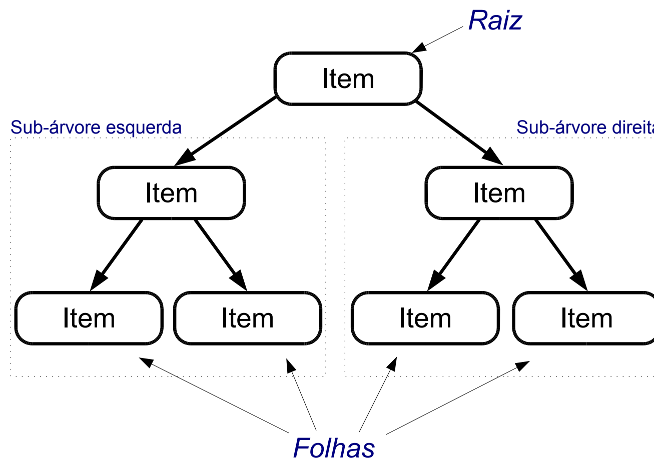


Figura 2.5: Exemplo de árvore

Cada aplicação pode usar suas próprias regras para escolher como particionar o espaço, quais condições devem ser satisfeitas para terminar as subdivisões e também quais informações guardar nas células. Árvores que fazem o particionamento sempre em duas partes são bastante comuns e são chamadas de *árvores binárias*. As árvores BSP-tree (*Binary Space Partitioning Tree*) e KD-tree, bastante utilizadas em computação gráfica, são exemplos de árvores binárias. Outros tipos de árvores usadas em computação gráfica são as quadtrees e octrees. Estas duas últimas levam em consideração a natureza 2D ou 3D (respectivamente) dos dados para dividi-los em quatro ou oito partes por vez. Todas as árvores supracitadas serão descritas com mais detalhes nas próximas seções.

2.3.1 Quadtree e Octree

A quadtree é uma estrutura usada em regiões bidimensionais. A cada nível da árvore, o espaço é dividido em quatro partes, usando sempre dois cortes alinhados

[1] Um hiperplano num espaço n-dimensional possui dimensão $n-1$, por exemplo: um hiperplano 2D é uma reta, um hiperplano 3D é um plano

com os eixos cartesianos. A figura 2.6 mostra vários níveis de uma quadtree, desde o espaço vazio (raiz da árvore) até o terceiro nível de profundidade, onde algumas células foram subdivididas e outras não.

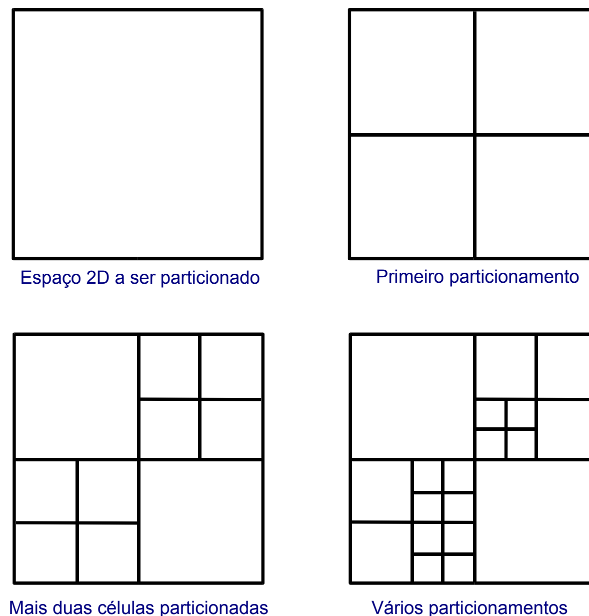


Figura 2.6: Exemplo de quadtree

As quadtrees são estruturas simples de se implementar e possuem várias aplicações. Os cortes, sendo sempre verticais e horizontais (normalmente centralizados na célula, mas não necessariamente), não introduzem muita instabilidade numérica, o que poderia ocorrer com cortes inclinados. A estrutura também é compacta, pois poucas informações são necessárias para cada particionamento. Entretanto, algumas aplicações podem necessitar de particionadores mais bem posicionados, para isso veremos outras estruturas possíveis no resto deste capítulo.

As aplicações das quadtrees incluem: representação de imagens, detecção de colisão em duas dimensões e eliminação de polígonos não-visíveis na exibição de terrenos.

A octree é a árvore correspondente à quadtree em 3 dimensões. A cada nível, o espaço é dividido em 8 partes, usando três planos, alinhados com os planos xy, xz e yz. A figura 2.7 mostra alguns exemplos de octrees. Assim como a quadtree, a octree também pode ser usada para eliminar partes não-visíveis de uma cena, por exemplo, se o volume de visualização não intersecta uma célula da octree, o conteúdo dela e de suas sub-árvores não precisa ser desenhado. Operações booleanas de objetos delimitados por surfels foram implementadas usando octrees em [2].

2.3.2 KD-Tree

A *KD-tree* foi criada como uma generalização de árvores binárias de busca para dimensões maiores [16]. Comparada com a quadtree, a KD-tree permite mais alternativas de posicionamento dos cortes, pois divide o espaço sempre em duas

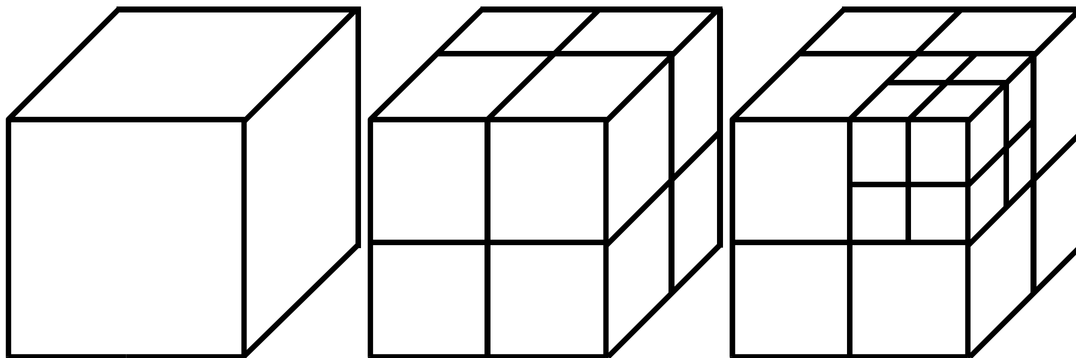


Figura 2.7: Exemplos de octrees

partes usando cortes livremente posicionados, mas alinhados com cada eixo alternadamente. A figura 2.8 mostra uma simples comparação entre os dois tipos de árvore.

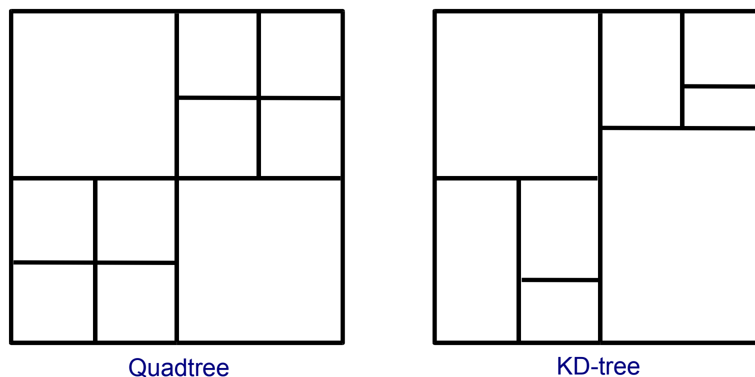


Figura 2.8: Comparação entre uma quadtree e uma KD-tree

Uma aplicação de KD-trees em duas dimensões é fazer busca de pontos que estejam localizados dentro de um dado retângulo no plano. A figura 2.9 mostra um exemplo de KD-tree 2D[2]. Dados quatro pontos de entrada, foi feito o primeiro corte paralelo ao eixo x, passando por um ponto da entrada. O segundo corte foi feito paralelo ao eixo y, passando por outro ponto dos dados de entrada. É interessante notar que tanto as folhas quanto os nós internos possuem um ponto dos dados de entrada. Outra alternativa seria guardar os pontos somente nas folhas, usando os outros nós da árvore apenas para a organização do espaço.

2.3.3 BSP-Tree

A *BSP-tree* (*Binary Space Partitioning Tree*) também é uma árvore binária de particionamento do espaço. A estrutura foi apresentada por Fuchs et al. [17] como uma solução ao problema de superfícies ocultas. Sua principal característica é ser

[2] Embora o K represente o número de dimensões da árvore, não é muito comum usar os termos 2D-tree e 3D-tree para se referir às KD-trees

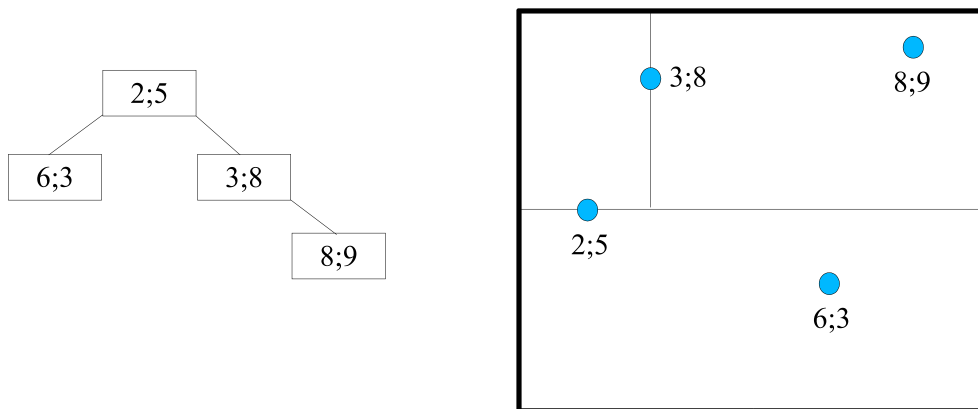


Figura 2.9: Exemplo de uma KD-tree em 2D.

a mais genérica das árvores de particionamento de espaço. Enquanto os cortes das KD-trees são sempre alinhados com os eixos, os cortes usados nas BSP-trees não têm restrições de direção. Por isso, pode-se dizer que as KD-trees são um caso específico das BSP-trees.

Cada particionador é um hiperplano qualquer, com um lado definido como *frente* e outro definido como *trás*. Na figura 2.10, mostra-se, ao mesmo tempo, um espaço 2D dividido em 6 regiões numeradas e a árvore correspondente a essa divisão. Os hiperplanos são designados por letras de A a E e possuem um vetor normal que indica qual é o lado *frente*.

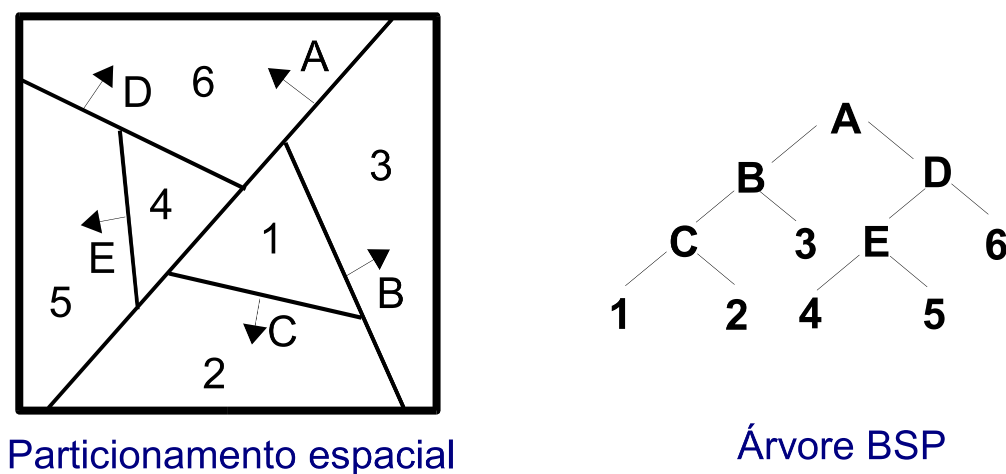


Figura 2.10: Exemplo de árvore BSP em 2D

As BSP-trees foram adaptadas para vários propósitos diferentes, entre eles: ordenação de objetos de uma cena independentemente da posição do observador [17], testes de visibilidade, detecção de colisão [18] e operações booleanas entre sólidos [19].

Usando uma BSP-tree, é possível desenhar os objetos de uma cena tanto de *frente-para-trás* quanto de *trás-para-frente* para qualquer posição do observador. Para isso, a cada nó da árvore, testa-se a posição do observador em relação ao hiperplano do nó, percorre-se recursivamente as sub-árvores na ordem desejada. Desenhando de *trás-para-frente*, se garante que o último polígono desenhado esteja mais na frente dos outros, sobrescrevendo os que estavam atrás. Já a ordem *frente-para-trás* combinada com um *depth-buffer* permite eliminar rapidamente os polígonos que estão ocultos por outros.

É interessante notar que, embora cada estrutura de árvore descrita tenha aplicações diferentes que motivaram sua criação, há uma grande sobreposição de funcionalidade. A escolha da estrutura ideal precisa considerar fatores como: a necessidade de particionamentos com inclinação arbitrária, a complexidade das células geradas, a profundidade da árvore criada na aplicação desejada e a eficiência ao percorrer a estrutura.

2.4 Componentes principais

Esta seção descreve o conceito de *Componentes Principais* [20, 21], necessário para a escolha de particionadores neste trabalho. Primeiramente, a seção 2.4.1 descreve como calcular a matriz de covariância de um conjunto de dados. Depois, a seção 2.4.2 explica o conceito de autovetores e autovalores. Por último, a seção 2.4.3 mostra como calcular as componentes principais usando os conceitos descritos previamente.

2.4.1 Matriz de covariância

Em estatística, existem várias análises que podem ser feitas sobre um conjunto de dados, como a média aritmética, o desvio padrão e a variância. Os dois últimos medem o quanto os dados estão afastados em relação à média, sendo que a variância é igual ao quadrado do desvio padrão.

Todas essas medidas, porém, consideram separadamente cada dimensão dos dados. Por sua vez, a covariância sempre é medida entre duas dimensões (calcular a covariância entre uma dimensão e ela mesma resulta na variância). A fórmula da covariância é a seguinte:

$$cov(X,Y) = \frac{\sum_{i=1}^{n}[(X_i - \bar{X}) \cdot (Y_i - \bar{Y})]}{n}$$

Na fórmula acima, X e Y são listas de dados, onde X é a primeira e Y é a segunda dimensão. $\bar{X}$ e $\bar{Y}$ são as médias das listas e X_i e Y_i são os elementos das listas X e Y na i-ésima posição. A variável n representa o número de itens de dados obtidos. Quando os dados representam uma *amostra*, usa-se $n-1$ no denominador, e quando os dados representam o conjunto total da “população”, usa-se simplesmente n no denominador.

Como a covariância é medida entre duas dimensões, se os dados tiverem mais de duas dimensões, é necessário ter a covariância entre cada par de dimensões. A partir dessa idéia, surge a *matriz de covariância*. Se forem usadas três dimensões (x, y e z), a matriz de covariância terá este formato:

$$matrix_cov = \begin{pmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{pmatrix}$$

A diagonal principal da matriz contém as variâncias e a matriz é simétrica acima e abaixo da diagonal principal.

2.4.2 Autovetores e autovalores

Diz-se que um vetor v é autovetor de uma matriz quadrada M se $M \cdot v$ resulta num múltiplo de v, ou seja, em $\lambda \cdot v$. Nesse caso, λ é o autovalor de M associado ao autovetor v. Quando se fala em autovetores, subentende-se "autovetores de comprimento 1", já que a propriedade desejada é apenas a direção do vetor. O problema descrito neste trabalho requer autovetores de comprimento 1.

Uma propriedade dos autovetores é que eles são perpendiculares (ortogonais) entre si. Essa propriedade é importante porque torna possível expressar os dados em termos dos autovetores, em vez de em termos dos eixos x, y e z.

Para matrizes de dimensões 2×2 ou também 3×3, os autovalores podem ser calculados da seguinte forma:

$$\det(M - \lambda \cdot I) = 0$$

Por exemplo, no caso de uma matriz M 2×2:

$$\det \begin{pmatrix} m_{11} - \lambda & m_{12} \\ m_{21} & m_{22} - \lambda \end{pmatrix} = 0$$

Isso resulta numa equação de 2º. grau, cujas raízes podem ser calculadas e substituídas no sistema abaixo para encontrar os autovetores correspondentes:

$$\begin{pmatrix} m_{11} - \lambda & m_{12} \\ m_{21} & m_{22} - \lambda \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

No caso de dimensões maiores, ou para algoritmos genéricos para qualquer número de dimensões, o usual é aplicar um algoritmo numérico iterativo (isto é, repetitivo) como o existente em [22]. O último passo do algoritmo é ordenar os autovetores de acordo com seus autovalores.

2.4.3 Componentes principais

A análise de componentes principais (PCA — *Principal Components Analysis*) é uma maneira de identificar padrões em dados. É bastante útil quando os dados têm muitas dimensões, onde uma representação gráfica não é possível, mas também pode ser útil em dimensões menores, como mostra a figura 2.11. A componente principal é a dimensão que melhor representa a distribuição dos dados (linha vermelha na figura 2.11) e a componente secundária é perpendicular à componente principal (linha azul na figura 2.11).

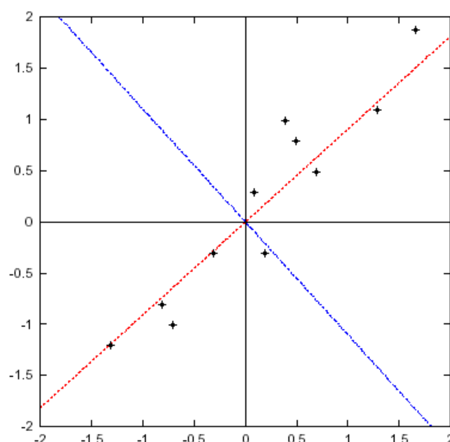


Figura 2.11: A linha vermelha pontilhada mostra a distribuição principal dos dados (autovetor de maior autovalor correspondente) e a linha azul mostra a componente secundária.

Os passos para calcular as componentes principais são:

1. obter os dados;
2. calcular a média de cada dimensão
3. subtrair a média de todos os itens em cada dimensão;
4. calcular a matriz de covariância;
5. calcular os autovetores e autovalores da matriz de covariância.

   O autovetor com o maior autovalor associado, corresponde à *componente principal* do conjunto de dados usado. Isso significa que essa é o relacionamento mais significativo entre as dimensões dos dados. A figura 2.11 ilustra esse ponto.

As componentes principais podem então ser usadas conforme desejado, seja apenas para visualização, seja para compressão ou até mesmo para reconhecimento de faces [23, 20]. Vários exemplos são descritos em [20]. As seções 5.1.1 e 5.3 valem-se da análise das componentes principais para escolher os particionadores da árvore e na seção 5.2.1, a PCA é utilizada como critério de parada.

3 CBSP-TREE

3.1 Motivação

Se as BSP-trees forem comparadas com as KD-trees, vê-se que a primeira nos dá total liberdade de escolha de particionadores, enquanto a segunda permite apenas particionadores alinhados com os eixos. A CBSP-tree (*Constrained BSP-tree*, ou BSP-trees restrita) pode ser entendida como uma estrutura intermediária entre as KD-trees e as BSP-trees, pois permite cortes inclinados mas mesmo assim é necessário definir quais cortes são considerados válidos.

A CBSP-tree foi criada a partir da necessidade de cortes inclinados mas sem gerar células complexas (com muitos lados). Por exemplo, na figura 3.1, as células de números *3* e *5* possuem 5 lados cada. Em árvores mais profundas, a complexidade de células pode aumentar cada vez mais. Entretanto, células com complexidade limitada podem ser usadas mais eficientemente em algoritmos geométricos como localização de pontos, procura de vizinhos mais próximos e detecção de interseções entre formas usando o teorema *separating axis* [24]. O particionamento de polígonos (em 2D) e poliedros (em 3D) que representam as células também pode ser executado de maneira mais simples e eficiente quando as células não aumentam em complexidade. Tanto em 2D quanto em 3D é possível ganhar desempenho usando estruturas de dados de tamanho fixo em vez de estruturas realocáveis ao armazenar o polígono ou poliedro que representa a célula.

A necessidade de manter as células simples é apenas uma das possibilidades da CBSP-tree, já que é possível aplicar amplamente a idéia de BSP-tree com restrições. Na próxima seção, o conceito de CBSP-tree será explicado e algumas alternativas serão apresentadas.

3.2 Conceito de CBSP-Tree

Define-se uma CBSP-tree como sendo uma BSP-tree com um predicado que define quais direções de particionamento são válidas e, por conseguinte, quais células são válidas. Por exemplo, é possível considerar uma KD-tree como uma CBSP-tree onde apenas cortes alinhados com os eixos são permitidos. Várias outras possibilidades também existem, como:

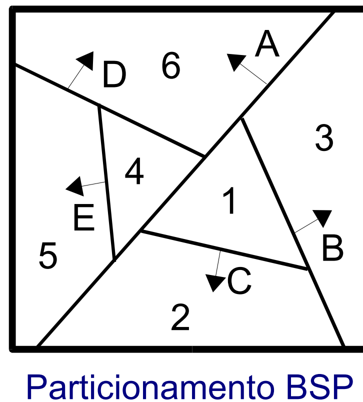


Figura 3.1: Exemplo de árvore BSP em 2D

- Limitar os ângulos permitidos de particionadores. Por exemplo, permitir apenas as inclinações de 0°, 45° e 90°.
- Procurar centralizar os cortes. Isto pode auxiliar a evitar células muito finas.
- Limitar o número máximo de lados das células. Simplifica o armazenamento da estrutura e também simplifica alguns algoritmos.

Neste trabalho, decidiu-se pela seguinte definição de célula válida: *Uma célula em R^n é válida se ela contém no máximo 2^n vértices e $2 \cdot n$ faces.* Isso quer dizer que, em duas dimensões, as células têm geralmente 4 vértices, mas podem ter apenas 3 vértices. Em três dimensões uma célula válida é aquela que contém no máximo 8 vértices e 6 faces (um hexaedro). Naturalmente, todas as células são convexas, como numa BSP-tree. Um corte é dito válido se ele particiona uma célula válida em duas células válidas também. A figura 3.2 mostra exemplos de células válidas e inválidas em 2D.

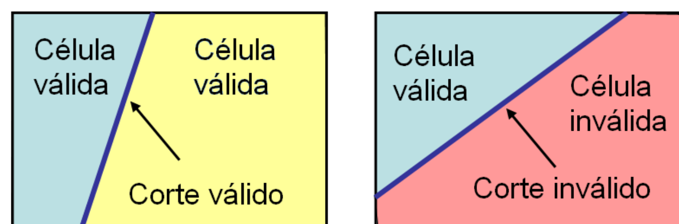


Figura 3.2: Células e cortes válidos e inválidos em 2D.

As aplicações que usam a CBSP-tree podem precisar de regras mais restritivas ou mais flexíveis que a definição dada acima. Um exemplo de regra mais restritiva é a relação de aspecto das células. Essa é uma importante característica a ser mantida para o bom funcionamento de certos algoritmos, como as operações booleanas, já que células longas e finas não funcionarão bem se a distância entre um surfel e outro for maior que a largura das células. Um exemplo de regra mais flexível usada neste trabalho se refere ao último nível da árvore. Às vezes, para se ter um particionamento ideal, pode ser necessário criar uma célula inválida apenas

nas células-folhas. Como as células-folhas não serão mais subdivididas, não haverá um aumento cumulativo de complexidade das células. Esta e outras alternativas para lidar com cortes inválidos, mas que às vezes são necessários, são discutidas na próxima seção.

3.3 Alternativas de criação de CBSP-trees

A maneira mais simples de construir uma CBSP-tree é usar apenas cortes que sigam alguma regra básica de validade de cortes. Essa idéia pode ser usado em problemas onde nenhum corte pré-definido é necessário, como na aproximação de funções implícitas ou representação de formas por imagens. O algoritmo de operações booleanas deste trabalho usa apenas cortes válidos nos primeiros níveis da árvore, já que, devido à grande variedade de cortes possíveis, o programa pode escolher dentre apenas os válidos e não considerar cortes inválidos. Se alguma métrica do algoritmo sugerir o uso de um corte inválido, ainda assim é possível escolher um corte semelhante, ou fazer os particionamentos numa ordem diferente sem grandes perdas na qualidade da estrutura.

Existem algumas aplicações de BSP-trees que exigem que os cortes sejam posicionados de uma certa forma pré-especificada. Um exemplo disso é a *auto-partition BSP-tree*, um tipo de BSP-tree que usa particionadores a partir de um conjunto de entrada de primitivas geométricas. Uma CBSP-tree pode ser usada para o mesmo propósito, porém é preciso escolher alguma forma de tratar os cortes inválidos que podem surgir. O tratamento dado aos cortes inválidos pode ser feito relaxando a restrição escolhida para a CBSP-tree em questão. Como dois exemplos possíveis, existem os cortes estruturais e existe o particionamento especial de células-folhas.

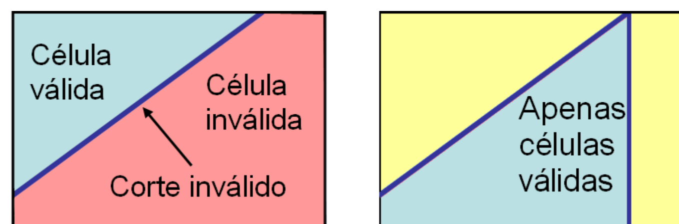


Figura 3.3: Exemplo de corte estrutural em 2D, corrigindo um corte inválido.

Cortes estruturais são cortes que, ao serem adicionados numa célula, transformam um corte inválido num válido. A figura 3.3 mostra um exemplo de corte estrutural corrigindo um corte inválido. Essa estratégia pode ser usada livremente em qualquer nível da árvore, mas nem sempre é extensível para três dimensões, como será visto na seção 5.3.2. Por outro lado, existe a alternativa de fazer um particionamento especial nas células-folhas. O particionamento especial permite usar o corte inválido diretamente, desde que *as células filhas não sejam subdivididas*. Isso garante que a complexidade das células não crescerá mais a cada nível da árvore.

4 OPERAÇÕES BOOLEANAS

Uma das maneiras mais intuitivas de modelar objetos sólidos é construí-los fazendo combinações de objetos mais simples até criar os modelos da complexidade desejada. Essa é a idéia da *Geometria Sólida Construtiva* (CSG — *Constructive solid geometry*) [25, 26]. Estritamente falando, CSG é uma forma de representação e uma metodologia de modelagem. Por isso, um objeto CSG é construído hierarquicamente, através de sucessivas operações booleanas entre objetos primitivos. O resultado é representado pela seqüência de operações usada para criá-lo. As operações possíveis são a *união*, a *interseção* e a *diferença*, conforme definidas na teoria matemática de conjuntos e também se usam operações de posicionamento e rotação das primitivas e dos objetos intermediários.

Entretanto, as operações booleanas não são úteis somente como uma forma de representação, como é feito na CSG. Freqüentemente se deseja usar operações booleanas em modelos representados de outras maneiras como a *B-rep* (*Boundary representation*) e *surfels*.

A figura 4.1 mostra as três operações booleanas executadas sobre dois círculos. Primeiramente, a união, depois a diferença e por último a interseção entre os círculos. A operação de diferença não é comutativa: a figura mostra apenas uma das duas possibilidades. A tabela 4.1 mostra como os resultados das operações booleanas são gerados, de acordo com a posição da superfície de um objeto relativa ao outro objeto. Disso, infere-se que a execução de uma operação booleana consiste em classificar pedaços de superfície e gerar um objeto com os pedaços relevantes.

Operação	**Superfície de A mantida**	**Superfície de B mantida**
$A \cup B$	Fora de B	Fora de A
$A \cap B$	Dentro de B	Dentro de A
$A - B$	Fora de B	Dentro de A (normais invertidas)
$B - A$	Dentro de B (normais invertidas)	Fora de A

Tabela 4.1: Partes das superfícies mantidas ao executar operações booleanas [2].

Vários algoritmos para operações booleanas são criados e aprimorados com freqüência. Um deles, descrito por [2], executa as operações com objetos modelados por surfels usando árvores para selecionar os surfels que farão parte do resultado.

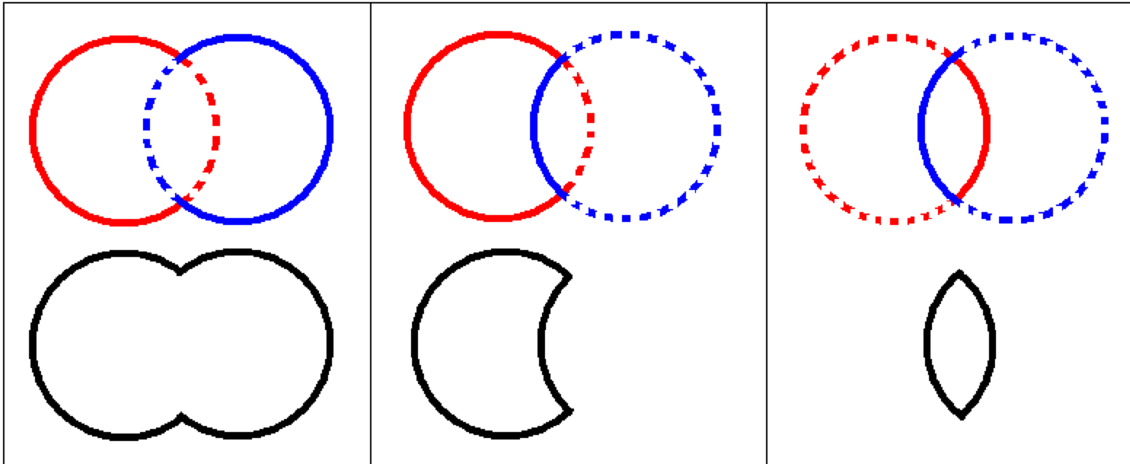


Figura 4.1: Exemplo demonstrando como as operações de união, diferença e interseção (respectivamente) funcionam.

O trabalho aqui descrito usa a mesma idéia, porém adiciona novas possibilidades de construção da árvore e compara os resultados. De um modo geral, o algoritmo pode ser resumido nos seguintes passos:

Construir as árvores dos dois objetos: O primeiro passo consiste construir uma árvore de modo a descrever a forma do objeto através dos particionamentos recursivos. O capítulo 5 explicará com mais detalhes esta parte do algoritmo.

Classificar as células-folhas sem surfels: As células-folhas com surfels representam a borda (ou fronteira) do objeto. Já as células-folhas sem surfels podem representar um pedaço do espaço de dentro ou de fora do objeto. É necessário então classificá-las como internas ou externas (figura 4.2). O capítulo 6 explicará esta parte do algoritmo em mais detalhes.

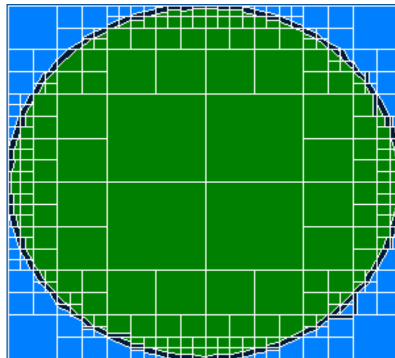


Figura 4.2: Exemplo de uma quadtree construída para um *dataset* em forma de círculo. As células verdes representam o interior do círculo e as azuis representam o exterior.

Classificar os surfels: Tendo a classificação das células, os surfels de um objeto podem ser classificados em relação ao outro objeto. Para isso, faz-se um caminhamento na árvore do outro objeto. Se o surfel se localiza numa célula-folha

do tipo *interior*, o surfel também é classificado como *dentro do outro objeto*. Se a célula for do tipo *exterior*, o surfel é, por conseguinte, classificado como *fora do outro objeto*. O último caso é quando um surfel se localiza numa célula de borda. Nesta situação, a solução é fazer uma pesquisa do surfel mais próximo que existe dentro dessa célula. Usando a normal e a posição desse surfel mais próximo, descreve-se um plano. Se o surfel que está sendo classificado estiver no lado da normal (sub-espaço positivo) do plano, ele está *fora do objeto*; se estiver no lado oposto (sub-espaço negativo), ele é classificado como *dentro do objeto* (figura 4.3).

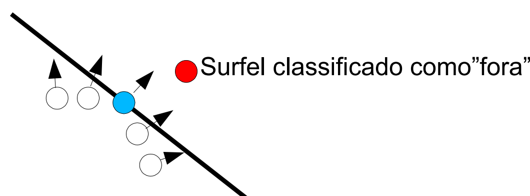


Figura 4.3: Classificação de um surfel posicionado numa célula de borda do outro objeto

A pesquisa de vizinho mais próximo pode ser feita, por exemplo, usando um pré-processamento como o do método TINN [27]; ou uma simples busca linear, caso a célula contenha poucos surfels. Pode-se acelerar essa classificação *surfel-por-surfel* se células inteiras (e suas filhas) forem classificadas de uma vez só. Isso é possível quando uma célula de um objeto intersecta apenas células do tipo *interior* ou *exterior* do outro objeto, mas nenhuma célula do tipo *borda*. E quando uma célula não intersecta nenhuma célula da outra árvore, também é possível fazer uma classificação rápida, afinal, todos os surfels estão fora do objeto.

Gerar o objeto resultante: Depois que essa classificação está terminada, é simples combinar os surfels usando operações booleanas. O resultado será a combinação de surfels de acordo com o indicado na tabela 4.1. Na operação de diferença, a superfície do segundo objeto que estiver dentro do primeiro precisará ser invertida para criar um modelo consistente. Para isso, basta inverter a orientação das normais dos surfels em questão, conforme também indicado na tabela.

O algoritmo da operação booleana pode ser implementado de maneira genérica e aproveitado tanto na versão 2D quanto na versão 3D. Para isso, basta que ambas estruturas de dados implementem a mesma interface para acessar suas operações de busca, recursão e iteração pelos surfels. Os tipos de dados (*surfel*, *ponto*, *árvore*, etc.) serão diferentes, mas se todos puderem ser acessados através dos mesmas funções, é possível implementar um algoritmo genérico. As linguagens de programação possuem ou deixam de possuir diferentes mecanismos para a programação genérica, às vezes com checagem de tipos durante a execução, outras vezes durante a compilação. O mecanismo usado para esse propósito neste trabalho foram os *templates* da linguagem C++.

5 PARTICIONAMENTO

Este capítulo discute as técnicas usadas para construir a CBSP-tree. Para construir uma BSP-tree ou uma CBSP-tree é necessário usar alguma estratégia para escolher quais particionadores serão usados. Isso não é necessário numa quadtree, já que o particionamento é sempre igual, bastando testar o critério de parada. Várias estratégias para cada parte do algoritmo foram implementadas, e o usuário é livre para escolher qualquer combinação de estratégias e ver o particionamento criado.

A seção 5.1 descreve as estratégias que podem ser usadas para escolher os particionadores iniciais. A seção 5.2 descreve alternativas de *critérios de parada* para o algoritmo decidir se o particionamento da árvore é suficiente. A cada nível de recursão, antes de escolher um particionador, o critério de parada deve ser testado. Se o critério for satisfeito, a subdivisão pára e o algoritmo passa ao estágio de *trim*, descrito na seção 5.3. Este último estágio consiste em particionadores posicionados de forma mais planejada, visando obter os particionadores mais precisos de acordo com o posicionamento dos surfels.

5.1 Escolha de particionadores

A construção da CBSP-tree para representar os surfels é controlada por uma operação de particionamento. A flexibilidade de escolha de particionadores é explorada por várias estratégias implementadas usando o padrão de projeto *Strategy* [28]. Isso permite alternar entre as estratégias durante a execução do programa para analisar os resultados. Duas estratégias serão apresentadas nas seções 5.1.1 e 5.1.2. A implementação usou ambas alternadamente, pois elas são complementares. A facilidade de adicionar novos algoritmos de particionamento também permitiu a inclusão de, por exemplo, estratégias que simulam outros tipos de árvores, como *quadtrees* e *octrees*, foram usadas para a comparação com as CBSP-trees.

5.1.1 Escolha PCA

O primeiro modo de escolha de particionadores usa a Análise de Componentes Principais (PCA) [21]. Esta é a estratégia chamada de *escolha PCA*. A *escolha PCA* consiste em calcular os auto-vetores que representam as direções principal e secundária das posições dos surfels. Os auto-valores correspondentes representam a

significância de cada direção. A figura 5.1 mostra as componentes principal e secundária para uma seqüência de surfels.

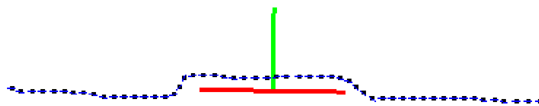


Figura 5.1: Componentes principais da posição dos surfels. Vermelho: componente principal; verde: componente secundária.

Após calcular as componentes principais, os resultados obtidos podem ser usados de várias formas na construção da árvore. Na fase inicial do particionamento, é bastante útil usar a direção secundária e o centro do grupo de surfels. Dessa forma, cria-se um particionador perpendicular ao grupo de surfels, dividindo-o convenientemente ao meio. É interessante notar que calcular a média dos vetores normais dos surfels geralmente vai gerar um vetor que pode ser usado no particionamento da mesma forma que a componente secundária, já que os vetores normais são, por definição, perpendiculares à superfície. Entretanto, a análise PCA também calcula uma informação muito importante: o alinhamento dos surfels, através dos auto-valores retornados. O algoritmo da seção 5.2.1 usará os auto-valores calculados aqui como critério de parada. Já os cortes alinhados com a componente principal são usados próximo ao fim do particionamento, conforme descrito na seção 5.3.

5.1.2 Escolha de candidatos

Outra estratégia é escolher um particionador a partir de uma lista de candidatos. Esta é a estratégia chamada de *escolha de candidatos*. Existem algumas propriedades desejáveis de um corte que são difíceis de atingir diretamente, por exemplo, obter o corte que maximize o espaço vazio em um dos seus lados. Num caso como esses, a melhor alternativa é experimentar alguns particionadores em várias posições e/ou inclinações e escolher aquele que atinge melhor a propriedade desejada. Não é necessário testar muitos particionadores: a maioria dos testes foi feita com 8 candidatos, para não deixar o algoritmo muito lento. A figura 5.2 mostra dois conjuntos de candidatos que podem ser usados. O primeiro testa várias inclinações (embora todos os candidatos passem pelo centro) e o segundo testa apenas cortes horizontais e verticais (deixando a árvore com as características de uma KD-tree). Alguns candidatos mostrados na figura podem ter inclinações extremas, possivelmente indesejáveis em alguns casos. Entretanto, a própria natureza da escolha de candidatos permite evitar esse tipo de corte: basta limitar os candidatos a apenas posições mais centrais da célula. Esta estratégia é também fácil de implementar em três dimensões, enquanto que um corte inválido sugerido pela PCA pode ser difícil de corrigir mantendo suas propriedades.

É interessante escolher o candidato que maximiza o espaço vazio num dos seus semi-espaços porque a *escolha PCA* reparte os surfels de uma célula pela direção perpendicular ao surfels. Assim, alternando a *escolha PCA* e a *escolha de candidatos*, geramos particionadores que são aproximadamente ortogonais ao particionador anterior e, pouco a pouco, separam o espaço vazio do espaço com surfels.

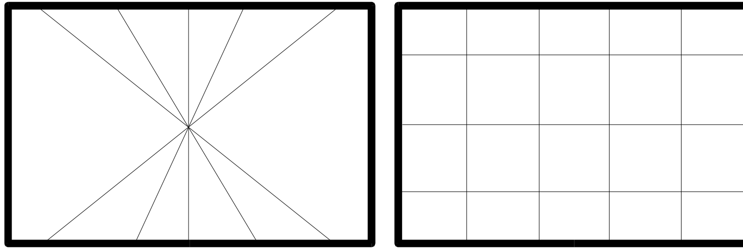


Figura 5.2: Exemplos de cortes candidatos que podem ser usados

Caso nenhum dos candidatos satisfaça o critério definido para a escolha de candidatos (por exemplo, todos os cortes deixam surfels nos dois sub-espaços), pode-se usar então uma estratégia mais simples, como fazer uma divisão perpendicular à maior dimensão da célula (para ajudar a manter a relação de aspecto das células: evitando células finas) ou então repetir a *escolha PCA*.

5.2 Critérios de parada

O particionamento pára quando certas condições são atingidas. Duas estratégias foram testadas. A primeira usa o alinhamento dos surfels, enquanto a segunda pára quando um certo nível de profundidade é atingido. Em ambos os casos, as dimensões da célula podem ser testadas primeiro, para evitar gastar tempo processando células muito pequenas, e também para evitar erros numéricos. Como é de se esperar, o número de surfels numa célula também é usado como critério de parada. Nos testes, o critério de parada foi escolhido de modo que o particionamento seja detalhado mas ao mesmo tempo sem muitos cortes desnecessários.

5.2.1 Critério de alinhamento de surfels

Nesta estratégia, primeiro testa-se o número de surfels. Se a célula tem menos que lim surfels, o particionamento acaba. O parâmetro lim pode ser configurado tão baixo quanto 1 ou 2 (já que dois surfels vão estar sempre alinhados) ou pode-se escolher um número mais alto quando tal grau de precisão se torna desnecessáriio.

Se a célula tem mais que lim surfels, testa-se o alinhamento dos surfels usando PCA. Usando uma fórmula para comparar a proporção entre os valores do primeiro e do segundo auto-valores, descobre-se o quão alinhados os surfels estão. A fórmula usada pode ser esta: $sec_eigval/(princ_eigval + sec_eigval)$. Se o resultado for menor ou igual a um dado valor, os surfels são considerados como alinhados e o particionamento pára.

5.2.2 Critério da profundidade da árvore

Uma alternativa mais simples de critério de parada é baseada na profundidade da árvore. Esta é a estratégia usada em [2]. Ao usar esta estratégia, o particio-

namento é feito enquanto a profundidade da árvore é menor que *level* (por exemplo, $level = 4$). Quando a profundidade da árvore é maior que *level*, testa-se o número de surfels na célula. Isso significa que o particionamento termina quando a árvore atinge um nível mínimo de profundidade e a célula tem menos que um dado número de surfels.

5.3 Particionadores de *trim*

Células que contêm surfels podem necessitar de particionamentos extras após o critério de parada ser satisfeito. Uma célula-folha geralmente contém um pequeno pedaço da superfície do objeto, e os surfels estão próximos do alinhamento. Por isso, é possível adicionar cortes extras para delimitar melhor a borda do objeto. Isso pode ser feito usando dois cortes paralelos de modo que todos os surfels fiquem entre esses cortes. Essa operação é chamada de *trim* e os cortes são chamados de *particionadores de trim*. Entretanto, alguns particionadores de *trim* podem violar a restrição da CBSP-tree, gerando células inválidas. Existem duas abordagens que podem ser tomadas. A primeira é adicionar cortes extras (estruturais) que tornem válido o particionamento desejado. A segunda é simplesmente permitir qualquer tipo de particionamento nas folhas (somente nelas). Como as folhas não serão mais subdivididas, a complexidade das células ainda se mantém controlada. Nas seções seguintes serão descritas as estratégias usadas em 2D (seção 5.3.1) e em 3D (seção 5.3.2). A seção 5.3.3 descreve como permitir qualquer tipo de particionamento. Em duas dimensões, é relativamente simples adicionar cortes estruturais, por isso os vários casos possíveis serão analisados. Em três dimensões, apenas alguns casos de cortes estruturais são suficientemente simples. Nos outros casos, usam-se os cortes diretamente, conforme descrito em 5.3.3.

5.3.1 Particionadores de *trim* em 2D

Os particionadores de *trim* são usados diretamente se os dois cortes paralelos são válidos. Caso contrário, de modo a manter a complexidade limitada das células e todos os surfels numa pequena área, adicionam-se alguns cortes a mais para tornar válidos os particionadores de *trim*. A figura 5.3 ilustra as diferentes situações que podem ocorrer. As próximas seções descrevem como lidar com os cortes inválidos.

5.3.1.1 Dois cortes inválidos, mesmos lados

A figura 5.3.b mostra dois particionadores inválidos (note que eles cortam dois lados adjacentes da célula). Para corrigi-los, é necessário primeiro adicionar o corte A (figura 5.4) de modo que o corte original passe a intersectar lados opostos da nova célula. Pode-se fazer o extremo superior de A coincidente com o extremo superior de B, gerando uma célula com três lados. Apesar disso, preferiu-se manter todas as células com o mesmo número de lados por questão de simplicidade. A mesma idéia então é aplicada ao outro corte inválido, criando os cortes C e D.

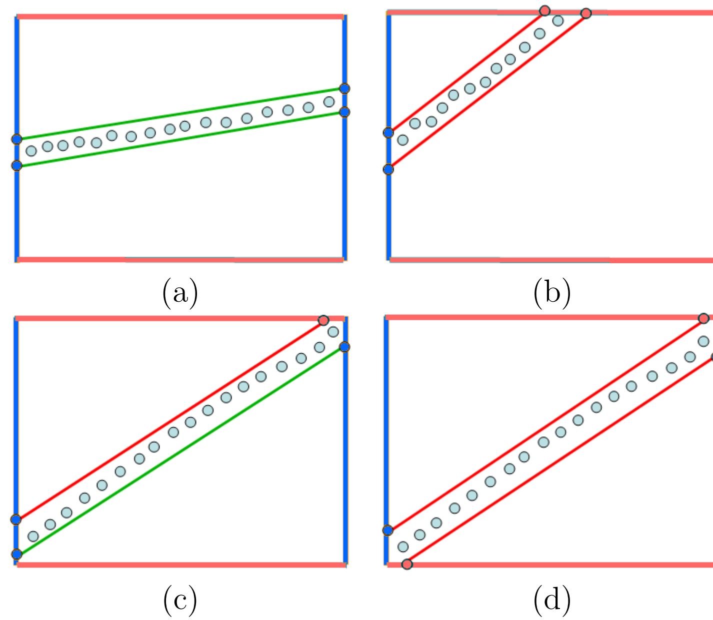


Figura 5.3: Casos de *Trim*. (a) 2 cortes válidos; (b) 2 inválidos (mesmos lados); (c) 1 inválido e 1 válido; e (d) 2 inválidos (lados diferentes)

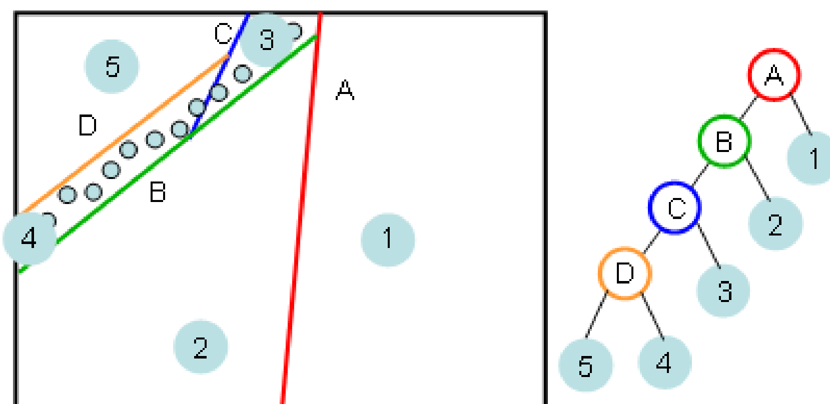


Figura 5.4: *Trim*: corrigindo dois particionamentos inválidos, quando ambos intersectam os mesmos lados da célula (cortes A e C adicionados).

5.3.1.2 *Um corte inválido*

A figura 5.3.c mostra um caso onde um particionador é inválido. Como no caso anterior, adiciona-se um corte extra (corte B) que intersecta o corte inválido antes que ele atinja a borda da célula. B poderia ser qualquer corte que torne o particionamento válido (como um corte vertical). O exemplo mostrado usa o ponto médio do corte válido como um dos extremos de B, porque é mais simples e sempre gera um corte válido.

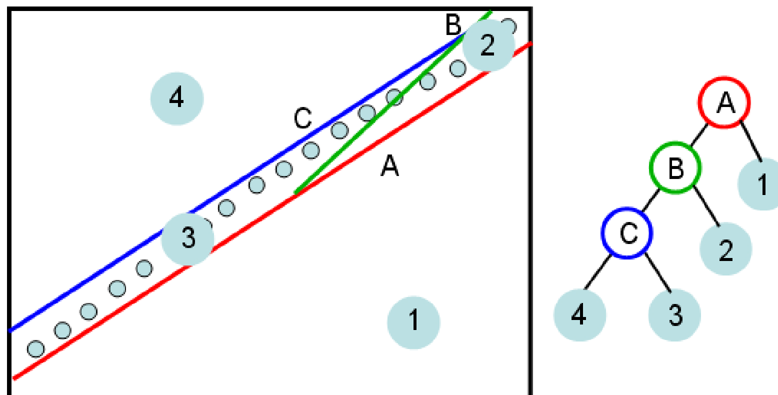


Figura 5.5: *Trim*: corrigindo o caso onde um corte é inválido e o outro é válido. O corte B torna válido o corte C.

5.3.1.3 *Dois cortes inválidos, lados diferentes*

Na figura 5.3.d, há dois particionadores inválidos, mas cada um intersecta um par de lados diferentes da célula, por isso não se pode aplicar o caso A aqui. Em vez disso, a solução consiste em dividir a célula em duas pela metade (preferivelmente perpendicular à dimensão mais longa, de modo a evitar células muito finas) e então o caso B é aplicado nos dois lados (figura 5.6). Embora todos os exemplos usem células retangulares para ilustração, os algoritmos funcionam para qualquer célula de quatro lados.

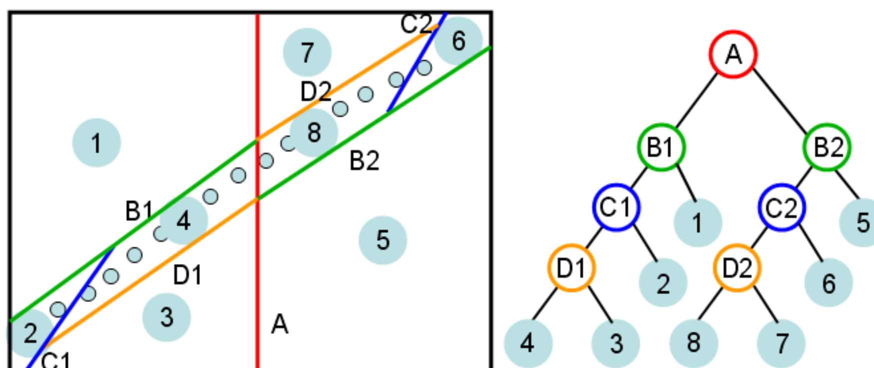


Figura 5.6: *Trim*: corrigindo dois cortes inválidos, quando cada um intersecta lados diferentes da célula.

5.3.2 Particionadores de *trim* em 3D

Assim como na versão 2D, em 3D também é necessário particionar as células-folhas de uma maneira que deixe os surfels no menor espaço possível. O desejável é que cada célula seja preenchida com surfels que representem um pedaço de superfície. A maneira mais simples de particionar as células-folhas, naturalmente, é construir dois planos paralelos em volta dos surfels, segundo seu posicionamento e/ou seus vetores normais. Deseja-se manter as células com as propriedades de terem 8 vértices e 6 faces. Os cortes *estritamente válidos* sempre mantêm essas propriedades das células. As células filhas podem ser facilmente criadas, já que não são criadas nem destruídas nem faces nem vértices, apenas a disposição dos vértices muda. Cortes válidos, mas não estritamente válidos, são os que intersectarem vértices da célula, alterando um pouco a topologia dela. Exemplos deste segundo tipo de corte podem ser vistos nas figuras 5.9 e 5.11.

Conforme será explicado a seguir, existem casos em que o uso de apenas cortes válidos se torna desnecessariamente complexo. Para esses casos, usa-se uma estratégia alternativa de trim que é explicada na seção 5.3.3.

Apenas algumas das estratégias de *trim* feitas em 2D (usando cortes adicionais) podem ser extendidas dirtamente para 3D. Entretanto, em três dimensões, a quantidade de casos a ser tratada é muito maior. É importante perceber que em 2 dimensões, apenas um tipo de corte é inválido: aquele que intersecta lados adjacentes da célula (figura 3.2). Os casos adicionais que tratados na seção 5.3.1 se devem apenas ao fato de que são usados *dois* cortes paralelos, fazendo-se necessário lidar com as combinações de particionamentos válidos e inválidos. Já em 3 dimensões, podemos identificar os seguintes casos, caso 4-4-simples, caso 2-6, caso 7-1, caso 3-5 e caso 4-4-complexo.

Caso 4-4 simples: A figura 5.7 mostra o tipo de particionamento mais simples possível em 3 dimensões, onde o particionador deixa 4 vértices da célula de cada lado. Se os dois planos paralelos forem desse caso, ambos podem ser usados diretamente.

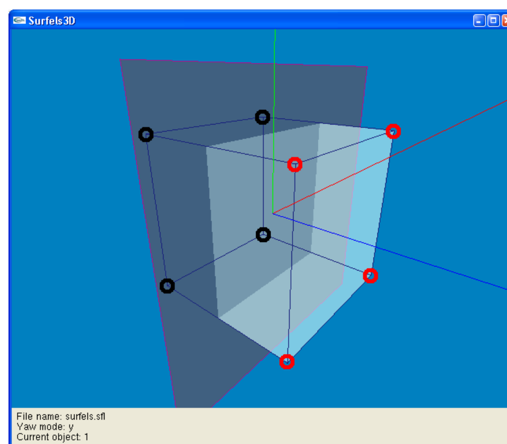


Figura 5.7: Caso 4-4-simples de *trim* em 3D.

Caso 2-6: A figura 5.8 mostra um caso em que o plano particionador deixa dois vértices da célula de um lado e os outros 6 vértices do outro. Este caso pode ser

tratado em 3D de forma semelhante ao que é feito em 2D (ver figura 5.9). Porém, tratar dois cortes paralelos deste tipo pode tornar-se desnecessariamente complexo. Por isso, apenas um corte é tratado, o que cria a maior célula com espaço vazio, adicionando-se cortes extras para torná-lo válido.

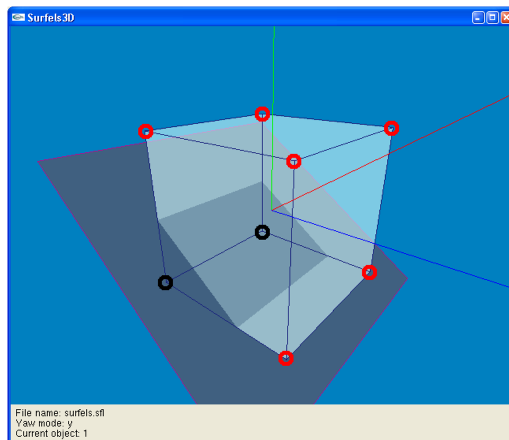


Figura 5.8: Caso de *trim* em 3D chamado de *2-6* porque deixa 2 vértices de um lado do plano e 6 do outro.

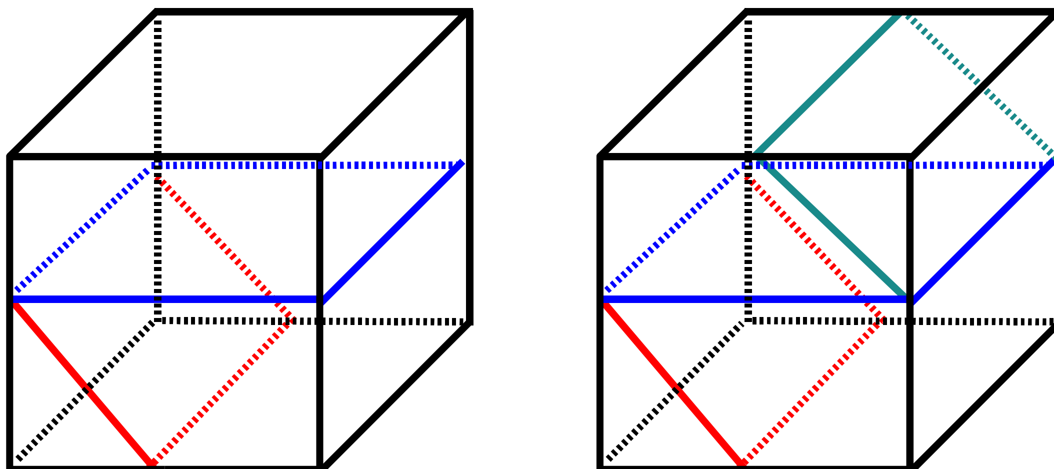


Figura 5.9: Soluções possíveis para o caso 2-6. A primeira é usada se os surfels estão num canto e a segunda solução é usada quando os surfels estão no meio (deixando, portanto, os planos paralelos em cantos opostos da célula).

Caso 7-1: O caso da figura 5.10 ocorre quando o plano deixa um vértice da célula de um lado e os outros 7 vértices do outro. Ele pode ser transformado no caso 2-6 e então o particionamento mostrado pode ser usado diretamente. Um exemplo de solução é mostrado na figura 5.11. A célula com surfels gerada terá a forma de um tetraedro.

Caso 3-5: Por sua vez, o caso mostrado na figura 5.12, que deixa 3 vértices de um lado e 5 do outro, é mais difícil de ser tratado: seriam necessários muitos cortes adicionais e maior probabilidade de erros numéricos apareceriam (principalmente se

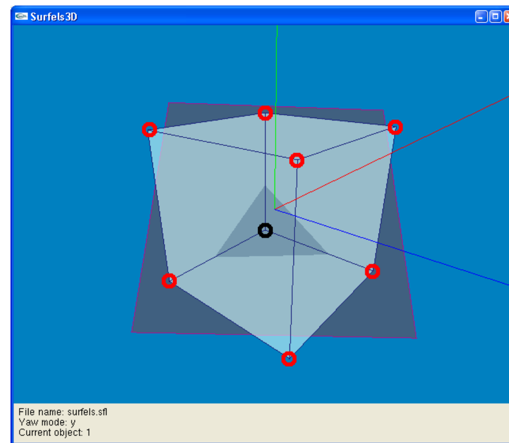


Figura 5.10: Caso de *trim* em 3D que deixa apenas 1 vértice de um lado do plano.

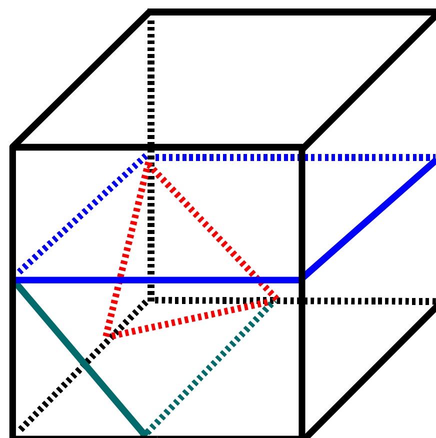


Figura 5.11: Exemplo de como solucionar um caso 7-1.

for necessário tratar ao mesmo tempo células muito grandes e muito pequenas). O recomendável é usar outra estratégia, como descrito em 5.3.3.

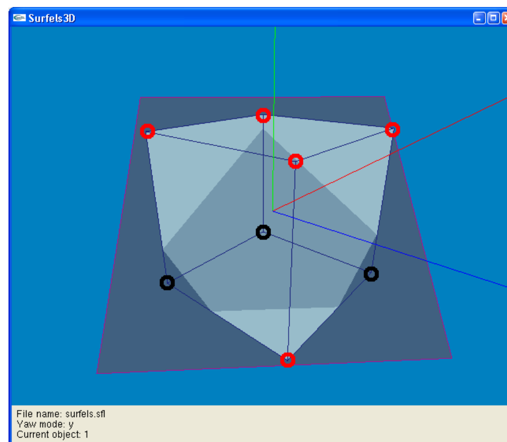


Figura 5.12: Caso de *trim* em 3D em que particionadores adicionais não são uma boa solução.

Caso 4-4 complexo: O *caso 4-4 complexo*, ilustrado na figura 5.13, é difícil de ser tratado, assim como o *caso 3-5*. Adicionalmente, este caso deixa quatro vértices da célula original de cada lado do plano particionador, podendo ser confundido com o *caso 4-4 simples*, da figura 5.7.

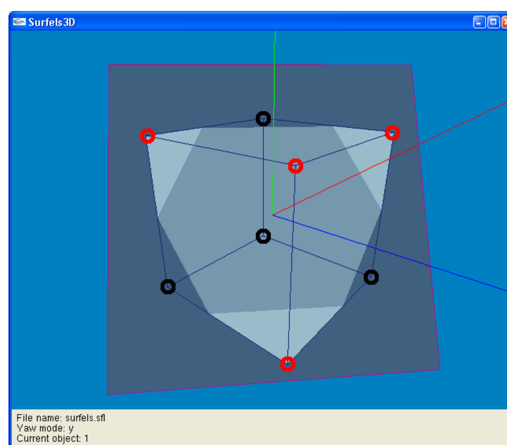


Figura 5.13: Caso de *trim* em 3D que, apesar de deixar 4 vértices de cada lado do plano, é difícil de ser tratado com particionadores adicionais.

Como pôde ser visto, os casos lembram as diferentes configurações do algoritmo *Marching Cubes* [29]. Porém nestes casos sempre existe apenas um plano, pois se uma célula contiver mais de um pedaço de plano, o ideal é particioná-la novamente — a *análise de componentes principais* ou um *agrupamento de surfels com normais semelhantes* poderiam ser usados para detectar isso. Facilmente percebe-se que, às vezes, fazer tantos cortes extras deixa de ser uma vantagem, devido à complexidade do algoritmo para tratar cada caso possível. Por isso, quando as estratégias descritas acima não são suficientes, se usa também a técnica descrita na próxima seção.

5.3.3 Particionamento especial das células-folhas

A segunda possibilidade de fazer o *trim* consiste em adicionar atributos especiais nas células-folhas que as particionem com os dois planos. Existem várias razões para preferir esta técnica:

- caso a distribuição de surfels na célula seja difícil de ser tratada de outra maneira (caso 3-5 e caso 4-4 complexo, descritos na seção 5.3.2);
- caso o uso de cortes inclinados nos primeiros níveis da árvore gere células com formatos onde não é possível aplicar as técnicas da seção 5.3.2;
- caso se deseje utilizar um tratamento genérico para todas as células folhas.

Esses planos paralelos, quando armazenados como atributos das células, são computacionalmente mais leves que um particionamento verdadeiro, pois não alocam memória para as três células-filhas[1] nem necessitam que o poliedro que representa cada célula seja particionado; basta armazenar a equação dos dois planos paralelos nas células-folhas. Com um simples teste contra a equação dos planos, desde que eles sempre estejam consistentemente orientados, é possível descobrir a localização de um ponto: se ele está entre os planos (borda), dentro do objeto ou fora dele. Além disso, qualquer particionamento é válido, já que essas informações são considaradas apenas como *propriedades da célula-folha*. Mesmo assim, esta técnica não deixa de limitar as células mais complexas, pois essas novas células não serão mais subdivididas e existirão apenas no último nível. Esta abordagem é equivalente à usada em [2].

A figura 5.14 mostra um exemplo de uma célula subdividida desta maneira. A parte do meio possui os surfels enquanto que as partes superior e inferior estão vazias. Os dois planos particionadores (ambos do tipo 2-6) estão marcados com linhas vermelhas. A célula continua sendo uma só, mas possui dentro de si, três regiões diferentes.

As desvantagens desta estratégia são pequenas, mas existem. Primeiramente, a visualização das células com estas propriedades exige código especial, já que os poliedros não estão verdadeiramente particionados. Esse código pode ser mais complexo do que o usado no particionamento normal, porque cortes com inclinações arbitrárias podem mudar radicalmente a topologia da célula. Outra dificuldade ocorre na implementação do algoritmo de classificação de interior/exterior (que será explicado no capítulo 6): uma célula que usa o particionamento especial de células-folhas não pode ser simplesmente classificada como interior, exterior ou borda, porque ela contém essas três partes ao mesmo tempo. Quando uma vizinha dessa célula for ser classificada, será necessário considerar a posição da vizinha relativa aos dois planos particionadores. No caso de uma célula normal, bastaria copiar sua classificação. E, finalmente, o algoritmo de caminhamento na árvore também deve ser alterado para que, ao chegar a uma célula-folha, o algoritmo considere os dois planos internos à célula como particionamentos verdadeiros.

[1] ou mais de três células filhas, considerando os cortes estruturais

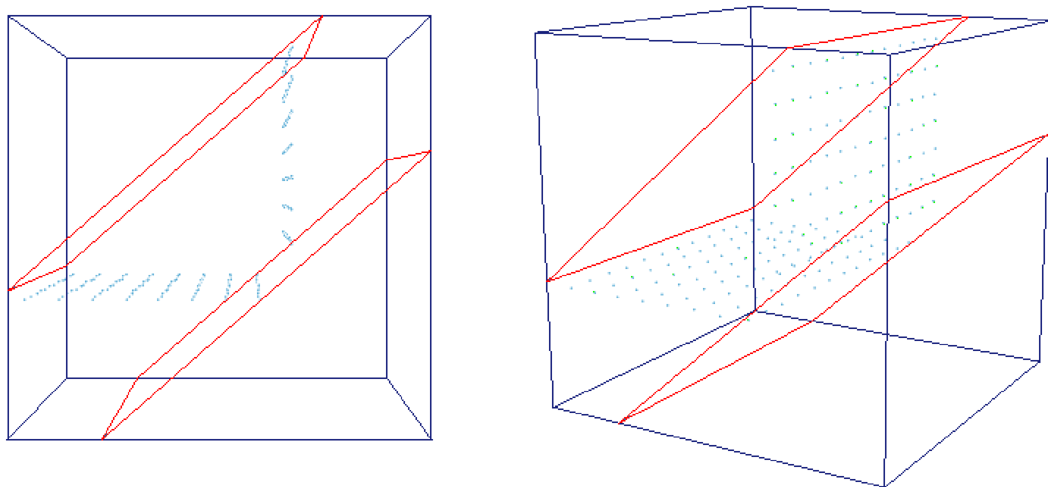


Figura 5.14: Exemplo do particionamento especial de células-folhas. Uma única célula possui três regiões, delimitadas por dois planos paralelos.

5.4 Discussão

Neste capítulo, foram apresentadas várias estratégias para a construção de uma CBSP-tree para operações booleanas. As estratégias foram separadas em três categorias: escolha de particionadores, critérios de parada e particionadores de *trim*, na ordem em que são executados.

Combinando as diferentes estratégias e ajustando seus parâmetros, é possível experimentar vários particionamentos para um mesmo objeto. Entretanto, é difícil escolher um conjunto de algoritmos que sejam adequados para qualquer objeto que venha a ser usado nas operações booleanas. Alguns dos fatores que podem exigir ajustes no particionamento são as proporções do objeto, quantidade de detalhes finos e densidade dos surfels.

O *trim* é o algoritmo mais complexo deste capítulo. São vários casos que precisam ser tratados com cuidado para o correto posicionamento dos cortes. Além disso, as diferenças entre as versões 2D e 3D do *trim* são significativas, o que significa que a implementação de uma não facilita muito a implementação da outra. Algumas circunstâncias podem fazer com que o *trim* não encontre um bom particionamento. O exemplo mais comum é quando a célula contém mais de um pedaço de superfície, ou contém um pedaço de superfície e alguns surfels de outra superfície do objeto. Para evitar isso, o critério de parada é responsável por verificar as propriedades da célula, seja usando as componentes principais, seja verificando as normais dos surfels ou alguma outra estratégia. Entretanto, também existe a alternativa de permitir quaisquer tipos de cortes no *trim* (seção 5.3.3). Cada implementação pode escolher a melhor forma de balancear a complexidade: seja com células simples e cortes posicionados de forma mais complexa, seja com cortes simples gerando células mais complexas.

A abordagem extensível da implementação (pelo uso do *strategy pattern*

[28]) permite explorar diferentes particionamentos e critérios de parada, bastando criar novas classes que implementem novas estratégias. Por exemplo, é possível imaginar um critério de parada ou uma estratégia de particionamento que agrupe os surfels de acordo com a semelhança de suas normais. Isso permitiria gerar células com pedaços de superfície mais homogêneos, facilitando, portanto, o trabalho do *trim*. Outras possibilidades serão discutidas na seção *trabalhos futuros* (seção 9.1).

A árvore construída com os algoritmos apresentados neste capítulo receberá atributos (em suas folhas) que permitirão a classificação de interior/exterior dos surfels de um objeto em relação ao outro. O capítulo 6 detalhará o algoritmo usado para fazer essa classificação das células.

6 CLASSIFICAÇÃO DE INTERIOR E EXTERIOR

Depois de construir a árvore, conforme descrito no capítulo 5, é necessário classificar cada célula-folha de acordo com o espaço do objeto que ela representa. Essa classificação pode ser:

- *interior*: a célula está vazia e está localizada na parte de dentro do objeto;
- *exterior*: a célula está vazia e está localizada na parte de fora do objeto;
- *borda*: a célula contém surfels, portanto representa uma parte da borda do objeto.

Este capítulo descreve como essa classificação pode ser feita a partir dos surfels que cada célula contém. A seção 6.1 explica como criar uma lista das células vizinhas para cada célula da árvore. A seção 6.2 explica como a informação de vizinhança é usada junto com os surfels para fazer a classificação das células.

6.1 Células vizinhas

Uma CBSP-tree não armazena explicitamente as células vizinhas de uma dada célula. Por isso, é necessário fazer um pré-processamento para que cada célula contenha uma lista de suas vizinhas. Armazenando também o polígono ou poliedro que representa cada célula, pode-se usar o algoritmo 1 para encontrar as vizinhas de todas as células-folhas.

O algoritmo 1 consiste, primeiramente, em passar por todos os particionadores de maneira recursiva. Para cada particionador, é necessário descobrir quais células são incidentes a ele no sub-espaço positivo e quais são incidentes no sub-espaço negativo. O algoritmo 2 é responsável por essa tarefa, devendo ser chamado para cada um dos sub-espaços.

O algoritmo 2 recebe uma célula e um particionador como parâmetros. Se a célula for folha, não é necessário fazer nenhum teste, pois ela será incidente ao particionador. Se a célula não for folha, pode ser necessário fazer recursão ou para ambos os lados ou apenas para um, dependendo se as duas células-filhas ou

Algorithm 1 AcharVizinhos

```
1: for all particionadores p do
2:    C_e = AcharCélulasIncidentesAoParticionador(célula-filha-esquerda, p);
3:    C_d = AcharCélulasIncidentesAoParticionador(célula-filha-direita, p);
4:    for all célula c_e ∈ C_e e célula c_d ∈ C_d do
5:       if c_e é vizinha de c_d then
6:          Adicionar c_e à lista de vizinhos de c_d, e vice-versa
7:       end if
8:    end for
9: end for
```

Algorithm 2 AcharCélulasIncidentesAoParticionador(célula, particionador)

```
1: if célula é folha then
2:    Adicionar esta célula à lista de células incidentes.
3: else
4:    if Célula filha esquerda encosta no particionador then
5:       Fazer recursão(célula-filha-esquerda, particionador).
6:    end if
7:    if Célula filha direita encosta no particionador then
8:       Fazer recursão(célula-filha-direita, particionador).
9:    end if
10: end if
```

apenas uma encosta no particionador (figura 6.1). O teste usado para isso pode ser implementado de várias formas. Em 2D, uma estratégia que obteve bons resultados em termos de robustez consiste em calcular a interseção entre os particionadores (aquele que é parâmetro da função com aquele que particiona o parâmetro *célula*). Se essa interseção for entre os limites da aresta que coincide com o particionador (ver figura 6.1.a), significa que se deve fazer recursão para as duas células-filhas; já se os particionadores forem paralelos ou a interseção ocorrer fora dos limites da aresta, só se faz a recursão para um dos lados (ver figura 6.1.b). Em 3D, a estratégia usada foi testar se o poliedro da célula tem alguma face com três vértices coplanares ao particionador. Mantendo as células com tamanhos e proporções que não excedam a precisão numérica utilizada, essa estratégia funciona bem e pode também ser usada em 2D.

Tendo as listas de células incidentes ao particionador, o último passo do algoritmo 1 é decidir quais são vizinhas entre si. Em 2D, isso pode ser feito usando a equação paramétrica sobre o particionador, desde que se saiba todos os pontos em que as células incidentes começam e terminam (ver marcações a, b, c, d e e na figura 6.2). A figura 6.2 mostra um exemplo de como a busca por viznhas é feita. O particionador destacado nessa figura incide nas seguintes células à esquerda: C_e=$\{2, 6, 7\}$ e nas seguintes células à direita: C_d=$\{3, 8\}$. As células restantes (*1*, *5*, *4*, *9*), e quaisquer células filhas que elas venham a ter, não são consideradas para os testes subseqüentes. Testando cada célula do lado esquerdo contra as do lado direito permite concluir que as células *3* e *6* são vizinhas conectadas pelo *segmento de vizinhas c-d*. Por outro lado, as células *7* e *3* não são vizinhas porque elas não compartilham um segmento de linha. Vizinhas conectadas por um *segmento*

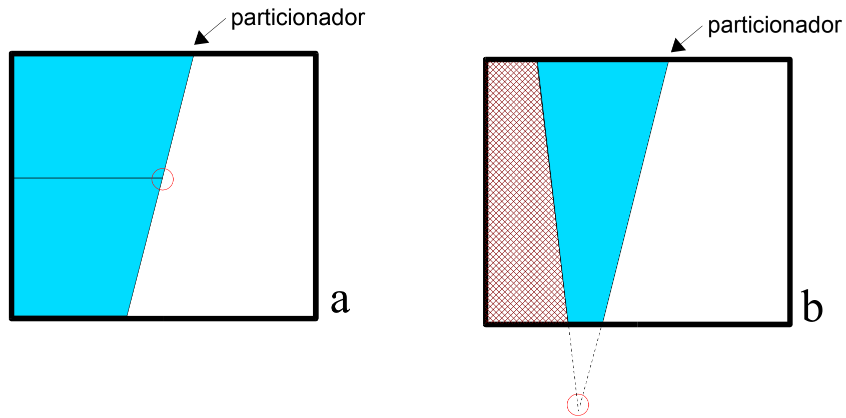


Figura 6.1: Descobrindo as células que enconstam no *particionador* pelo lado esquerdo. Na figura *a* as duas células (azuis) encostam no particionador. Na figura *b*, apenas uma célula encosta; enquanto que a outra, hachurada em vermelho, não encosta e não precisa mais ser testada.

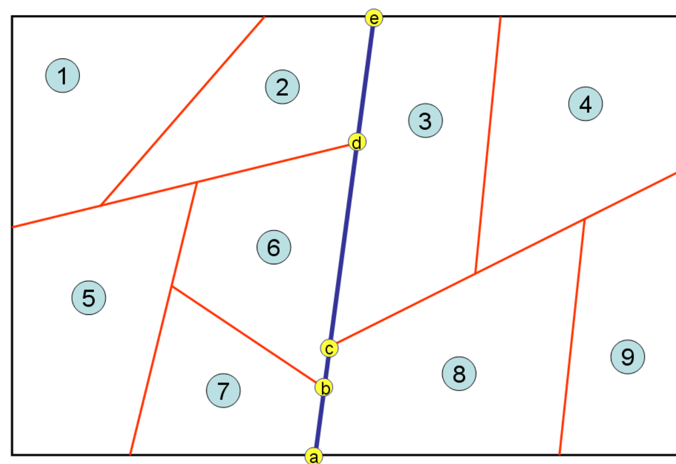


Figura 6.2: Encontrando células vizinhas.

de vizinhas muito pequeno (quando comparado com os lados das células) podem ser consideradas não-vizinhas, porque podem causar classificações incorretas de células no próximo passo. O *segmento de vizinhas* é guardado junto com o ponteiro para cada vizinha, pois será necessário na classificação de interior e exterior (seção 6.2) em duas dimensões. Já em três dimensões, foi usada outra abordagem, em que essa informação não é usada.

A verificação de quais células são vizinhas dentre todas as células incidentes a um particionador pode ser feita de maneira semelhante em 3D também. Como a equação paramétrica pode ser entendida como um modo de ver de maneira unidimensional um particionador que está num espaço bidimensional, tendo um particionador num espaço tridimensional, pode-se usar uma transformação de coordenadas para 2D (no plano do particionador) para analisá-lo de maneira bidimensional. Então, a interseção entre as faces (polígonos) pode ser testada usando o *Separating Axis Theorem* [24]. Entretanto, por ser um algoritmo bastante genérico, o *Separating Axis Theorem* também poderia ser usado sem essa transformação para 2D.

6.2 Classificação

Quando todas as células possuem ponteiros para as suas vizinhas, calcula-se a média dos vetores normais dos surfels de cada célula de *borda*. Como as normais não são usadas individualmente, refere-se a este vetor simplesmente como *vetor normal*. Usando o *vetor normal*, que aponta para fora, uma célula de borda pode identificar quando uma vizinha representa o interior ou o exterior do objeto. Esta parte do algoritmo foi implementada de forma diferente em 2D e em 3D. As duas estratégias são explicadas nos parágrafos seguintes.

Em duas dimensões, optou-se por implementar usando o *vetor normal* e os *segmentos de vizinhas*, conforme o exemplo a seguir. Supondo que a célula *6* seja uma célula de borda e a célula *3* não contenha surfels (figura 6.3), a célula *3* será classificada como *exterior*. Isso acontece porque o vetor normal (passando pelo centro dos surfels) intersecta o segmento das vizinhas e o sentido do vetor aponta para a célula *3*. Se o vetor apontasse no sentido oposto, a classificação seria *interior*. Essa distinção pode ser obtida facilmente vendo o sinal do produto escalar entre o *vetor normal* e um vetor *do centro dos surfels até a interseção*.

Todas as células que não foram classificadas pelo procedimento acima são classificadas com um simples algoritmo de *flood* ("inundação"). A figura 6.4 mostra um exemplo do resultado final obtido pelo algoritmo de classificação de interior e exterior.

Durante a implementação da versão em três dimensões, percebeu-se que é possível saber quais células criadas farão parte do interior e do exterior ao executar o procedimento de *trim*. Isso se deve ao fato de o *trim* ser o último passo do particionamento. Portanto, basta usar os vetores normais para classificar as células geradas pelo *trim* para obter imediatamente uma boa classificação inicial de células. A partir daí, é possível espalhar a classificação das células com o *flood*, da mesma maneira que é feito em duas dimensões. Naturalmente, as duas técnicas podem ser

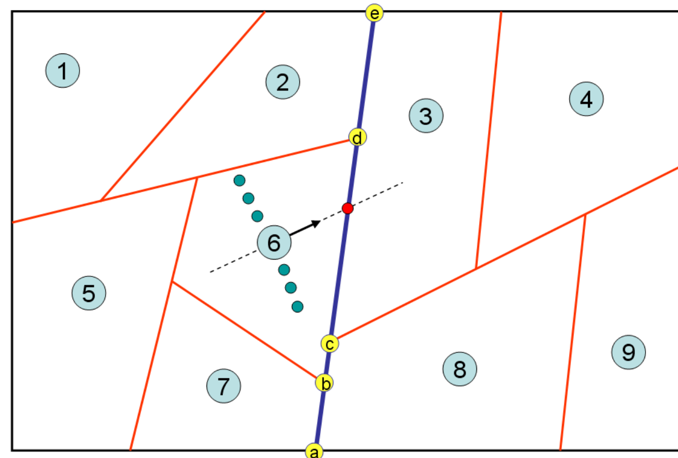


Figura 6.3: Classificando a célula *3* como *exterior*

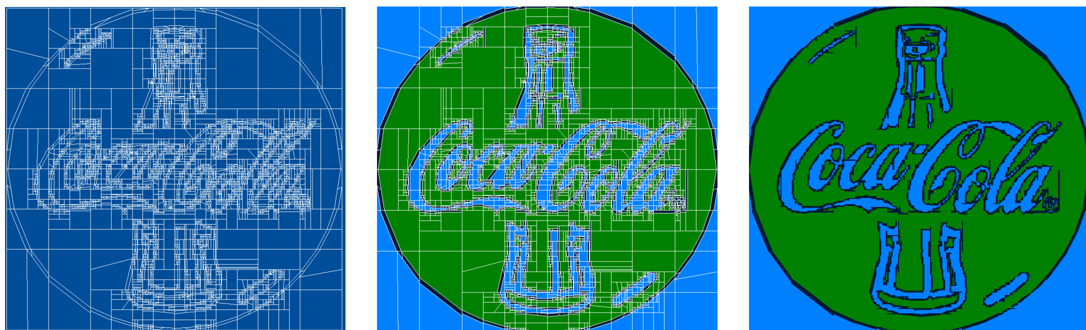


Figura 6.4: Exemplo do algoritmo de classificação de interior e exterior usando o dataset *coke*. Esquerda: subdivisão. Centro: classificação das células. Direita: classificação das células, com as linhas dos polígonos ocultas.

usadas nos dois casos. Entretanto, o conceito de *segmento de vizinhas* ficaria mais complexo na versão 3D, já que as células tocariam uma a outra por polígonos, e não por segmentos de retas.

Classificações incorretas podem acontecer caso haja células vizinhas que sejam uma do interior e uma do exterior do objeto. Sem uma célula de borda entre elas, a classificação por *flood* poderia repassar informações erradas. O algoritmo de particionamento deve, portanto, evitar células com dimensões menores que a densidade dos surfels.

6.3 Discussão

Este capítulo descreveu como classificar as células-folhas da árvore usando as normais dos surfels. Além disso, apresentou-se um algoritmo de busca de células vizinhas, já que é necessário que cada célula informe às suas vizinhas a classificação que lhe foi dada.

Após a classificação feita neste capítulo, a estrutura de dados poderá ser usada para efetuar as operações booleanas entre dois objetos. A classificação de um surfel de um objeto A em relação a um objeto B pode ser feita eficientemente, bastando fazer uma busca na árvore B. A classificação da célula em que o surfel de A estiver será dada a ele.

7 RESULTADOS 2D

Este capítulo mostra os resultados obtidos com a implementação 2D do programa. Vários objetos 2D foram usados nos testes, cada objeto com diferentes complexidades. A tabela 7.1 mostra os nomes dos conjuntos de dados e o número de surfels em cada um.

Dataset	**Número de surfels**
circle	472
elipse	560
elipse2	642
ufrgs	2.162
logo	2.278
koch	2.906
coca-cola	6.017

Tabela 7.1: Lista de *datasests* usados nos testes.

7.1 Particionamento

A criação das árvores dos objetos foi feita usando diferentes estratégias de particionamento e critérios de parada. O número de células-folhas nas CBSP-trees foi comparado com o número de células-folhas nas quadtrees e um sumário dos resultados está na tabela 7.2. Para cada arquivo de entrada, é necessário usar o mesmo critério de parada para a CBSP-tree e a quadtree, de modo a ter resultados comparáveis.

O particionamento da CBSP-tree foi feito de várias formas. Formas mais regulares, ou seja, mais "previsíveis" (por exemplo, o círculo e as elipses), costumam gerar bons particionamentos usando cortes com qualquer inclinação, já que a análise de componentes principais retorna resultados mais precisos. Por "bom particionamento", entende-se aquele que tem menos células e descreve bem o contorno do objeto. Já em figuras mais complexas, freqüentemente os particionamentos que geraram o menor número de células foram aqueles que usavam apenas alguns cortes inclinados ou que particionavam como se fosse uma KD-tree. Na figura 7.1 é possível ver três alternativas de particionamento, sendo uma a quadtree e as outras duas são CBSP-trees com diferentes limitações na inclinação dos cortes.

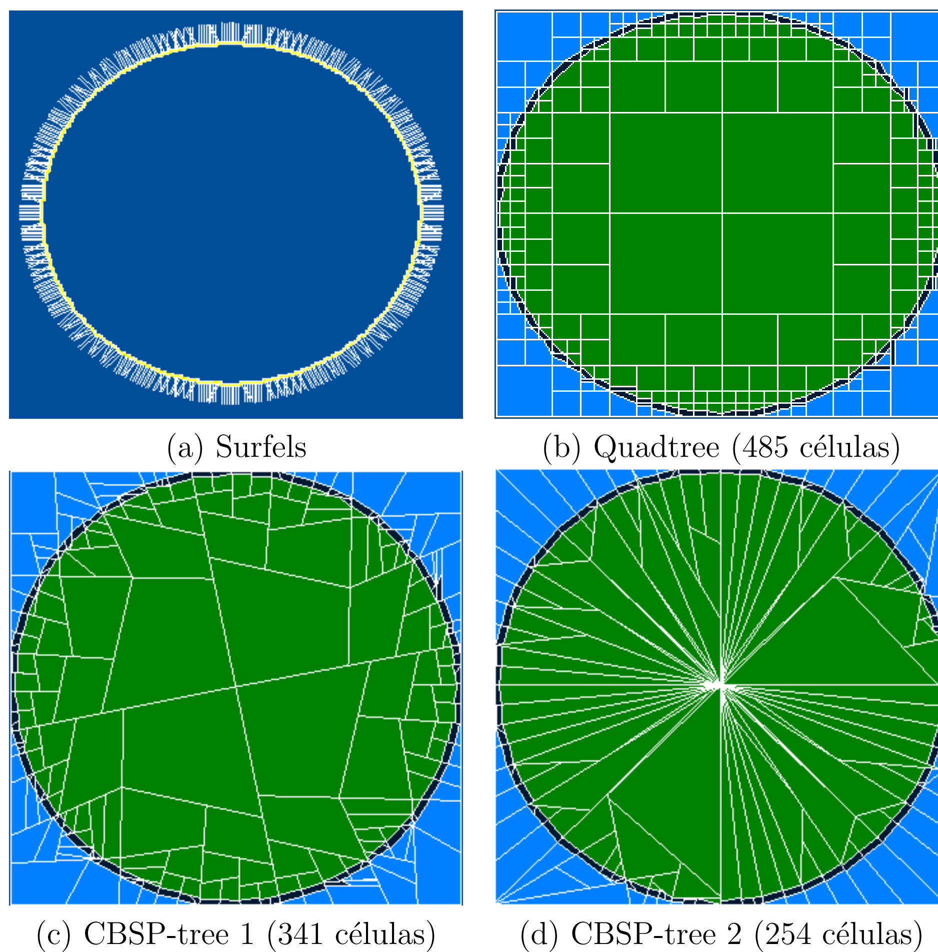
Dataset	Folhas CBSP	Folhas Quadtree	Critério de parada
circle	254	485	Profundidade máx.: 4 e máx. 8 surfels
elipse	209	287	Alinhamento dos surfels ou máx. 2 surfels
elipse2	151	207	Alinhamento dos surfels ou máx. 2 surfels
ufrgs	1.315	1.589	Alinhamento dos surfels ou máx. 8 surfels
logo	1.527	2.333	Alinhamento dos surfels ou máx. 1 surfel
koch	2.220	2.487	Profundidade máx.: 4 e máx. 8 surfels
coca-cola	5.294	7.688	Alinhamento dos surfels ou máx. 2 surfels

Tabela 7.2: Comparação entre CBSP-tree e quadtree.

A estratégia de *trim* usada em todos estes exemplos foi de usar cortes estruturais (seção 5.3.1) para as CBSP-trees e permitir qualquer tipo de corte nas folhas (seção 5.3.3) das quadtrees.

7.2 Operações booleanas

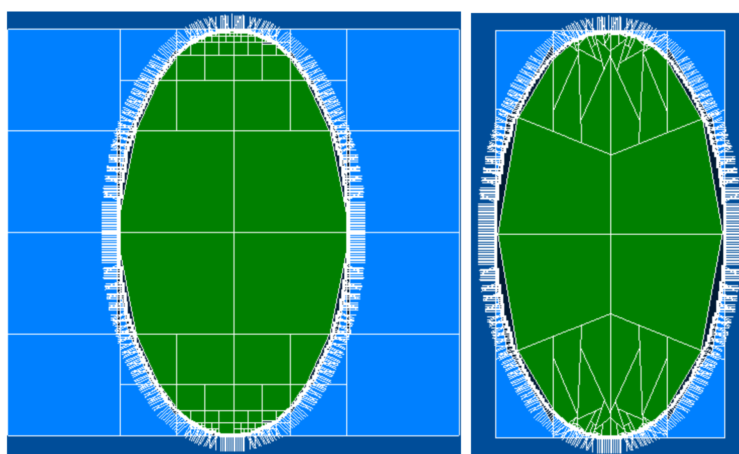
Esta seção contém exemplos dos resultados obtidos com as operações booleanas usando CBSP-trees. A figura 7.8 mostra as 4 operações implementadas, união, interseção e as duas operações de diferença (já que esta última não é comutativa). Os *datasets* usados nesses exemplos são o *circle* e o *logo*.



(a) Surfels (b) Quadtree (485 células)

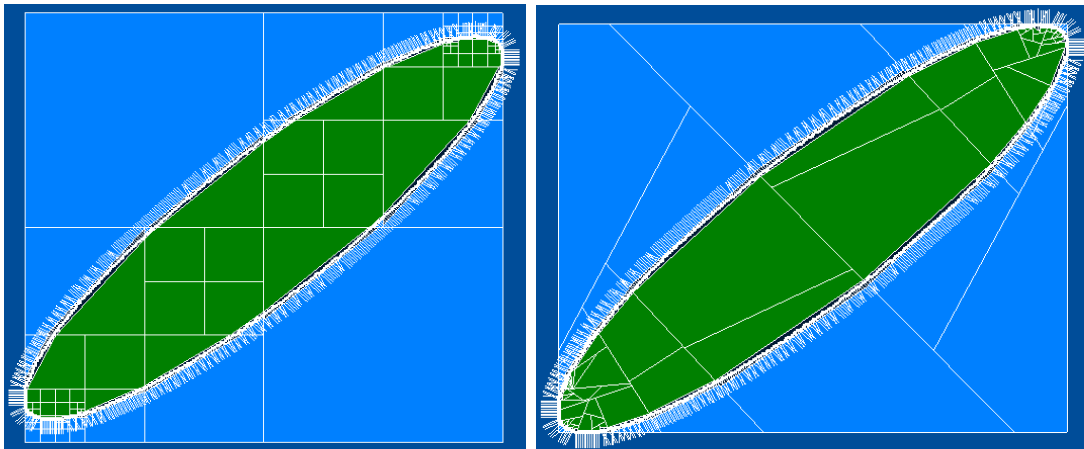
(c) CBSP-tree 1 (341 células) (d) CBSP-tree 2 (254 células)

Figura 7.1: Exemplo *circle*. (a) Surfels e (b) quadtree correspondente. (c) e (d) mostram duas possíveis CBSP-trees.



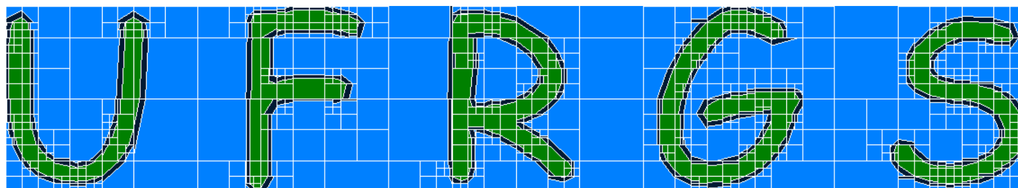
Quadtree (287 células), CBSP-tree (209 células)

Figura 7.2: Resultados do *dataset* Elipse

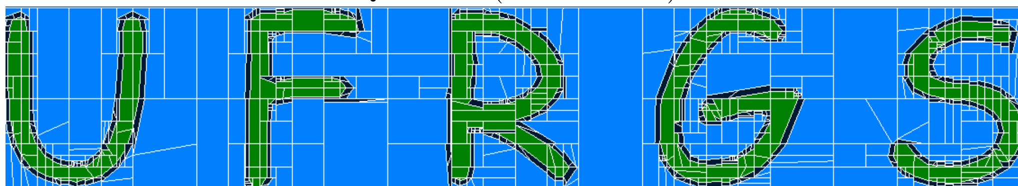


Quadtree (207 células), CBSP-tree (151 células)

Figura 7.3: Resultados do *dataset* Elipse2

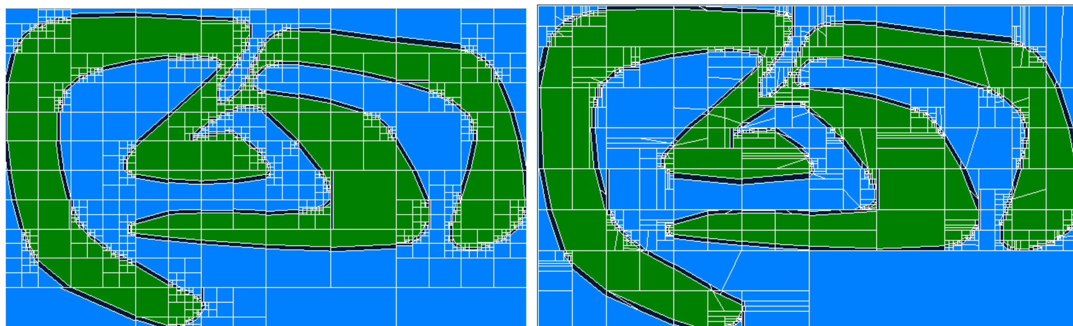


Quadtree (1589 células)



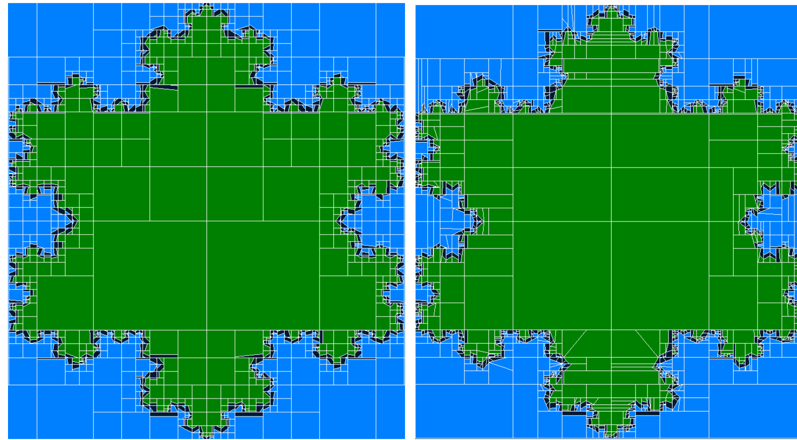
CBSP-tree (1315 células)

Figura 7.4: Resultados do *dataset* UFRGS



Quadtree (2333 células), CBSP-tree (1527 células)

Figura 7.5: Resultados do *dataset* Logo

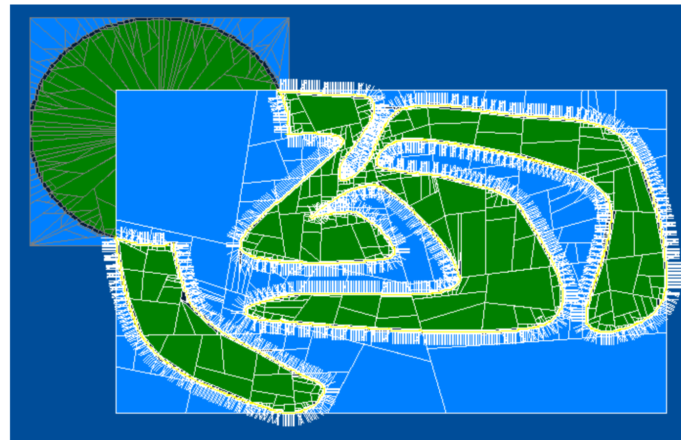


Quadtree (2487 células), CBSP-tree (2220 células)

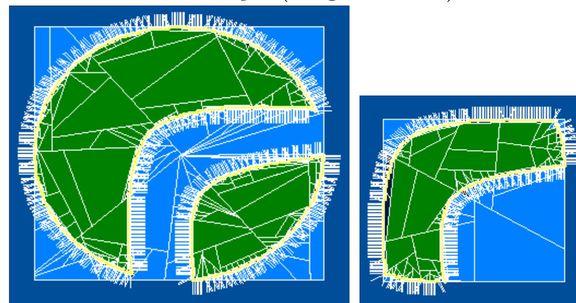
Figura 7.6: Resultados do *dataset* Koch

Quadtree (7688 células), CBSP-tree (5294 células)

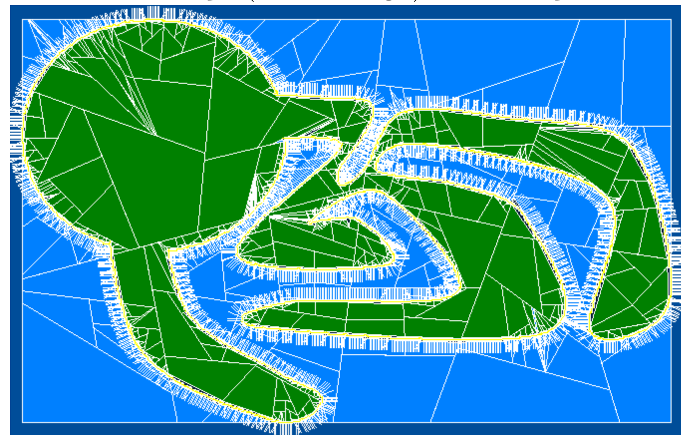
Figura 7.7: Resultados do *dataset* Coca-cola



Diferença (Logo-Circle)



Diferença (Circle-Logo) e Interseção



União

Figura 7.8: Exemplos de operações booleanas

8 RESULTADOS 3D

Este capítulo mostra os resultados obtidos com a implementação 3D do programa. A tabela 8.1 mostra os conjuntos de dados usados (*datasets*) e o número de surfels que cada um contém.

Dataset	**Número de surfels**
sphere	15.931
cylinder	60.000
papai noel	75.781
cube	99.846

Tabela 8.1: Lista de *datasests* usados nos testes.

8.1 Particionamento

O particionamento da esfera (*dataset sphere*) está ilustrado na figura 8.1. É possível ver também a classificação de interior e exterior (figura 8.1.b), onde as células internas estão marcadas em verde. A construção usando cortes inclinados é ilustrada na figura 8.1.c. É natural que neste último tipo de árvore, algumas células de borda fiquem com um formato maior ou mais irregular, mas o algoritmo de operações booleanas pode lidar com elas.

A figura 8.2 mostra o particionamento do *dataset cylinder* visto de dois ângulos diferentes. Na figura 8.2.a, apenas as células de borda estão visíveis. Já a figura 8.2.b mostra todas as células, permitindo ver também a classificação de interior/exterior. O *dataset papai noel* pode ser visto na figura 8.3. A figura contém tanto a renderização pelo Pointshop3D, quanto a árvore construída pelo programa implementado neste trabalho.

A tabela 8.2 mostra comparações do número de células-folhas com três tipos de árvores: CBSP-tree, KD-tree e octree (isto é, CBSP-tree com particionamentos no estilo octree).

Para visualizar a qualidade das bordas que pode ser obtida, dois *datasets* foram usados: *sphere* e *igea* (134.345 surfels). A figura 8.4 mostra como o plano de corte foi posicionado na *igea* e a figura 8.5 mostra *dataset* seccionado pelo plano em duas posições diferentes (mas com a mesma orientação). A figura 8.6 ilustra o

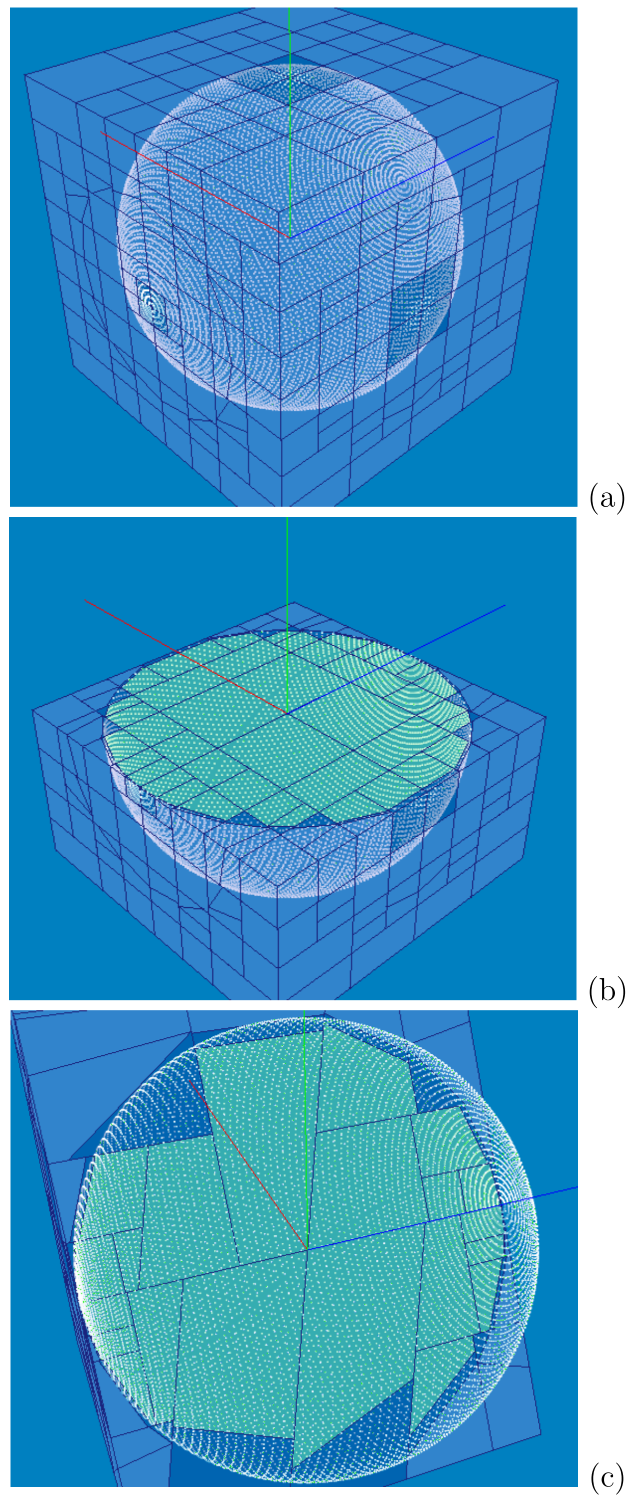


Figura 8.1: *Dataset sphere*. Mostrando a esfera inteira (a) e apenas a metade inferior (b), para visualizar a classificação das células. O particionamento com cortes inclinados é mostrado na terceira figura (c).

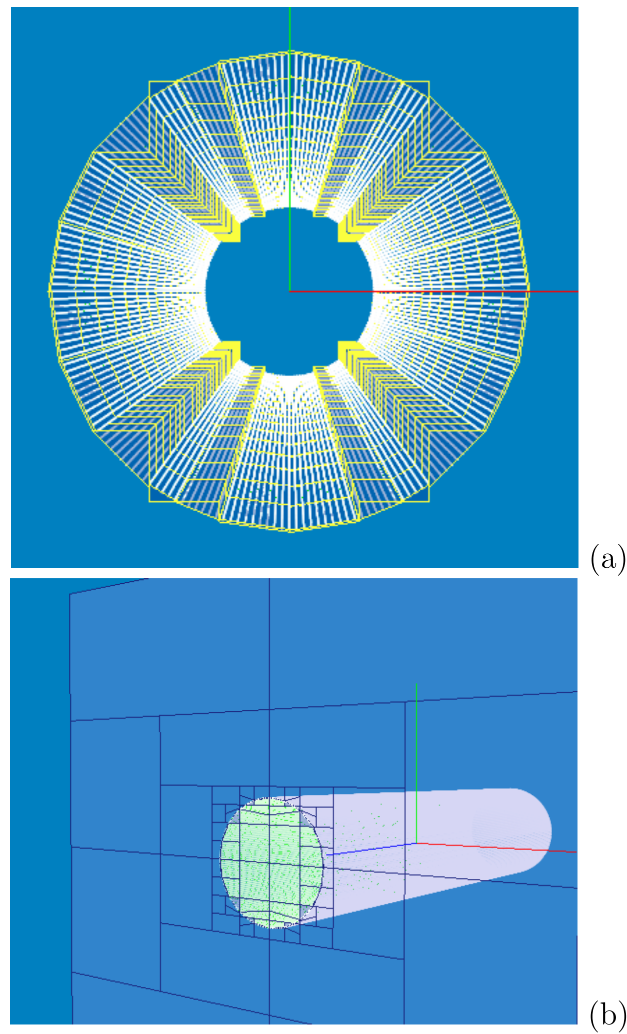


Figura 8.2: *Dataset cylinder*. Mostrando apenas células-folhas classificadas como *borda* (a) e mostrando todas as células (b).

Dataset	**Folhas CBSP-tree**	**Folhas KD-tree**	**Folhas Octree**
sphere	333	975	1.087
cylinder	1.394	1.472	3.392
papai noel	1.506	1.541	4.087
cube	2.005	2.391	5.252

Tabela 8.2: Comparação entre octrees, KD-trees e CBSP-trees em 3D.

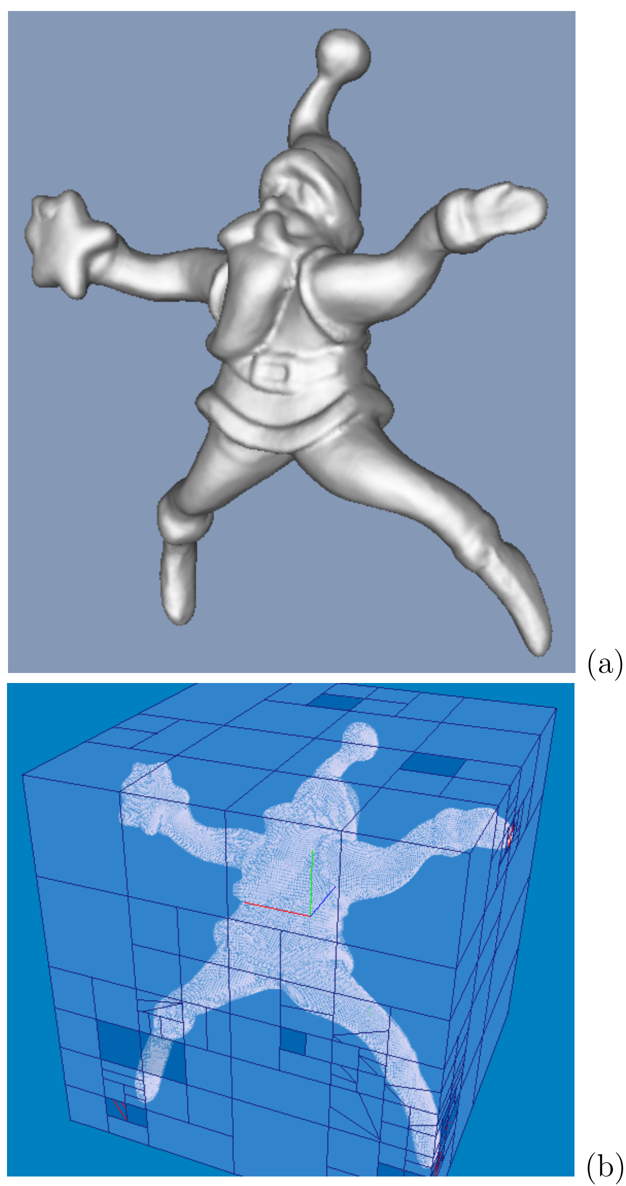


Figura 8.3: *Dataset papai noel.* Visualização pelo Pointshop3D (a) e visualização da árvore, usando a implementação deste trabalho (b).

mesmo procedimento para a esfera, com o plano em três posições. O *trim* usados nessas figuras, por ser o que apresenta os melhores resultados, é o *particionamento especial de células-folhas* (seção 5.3.3).

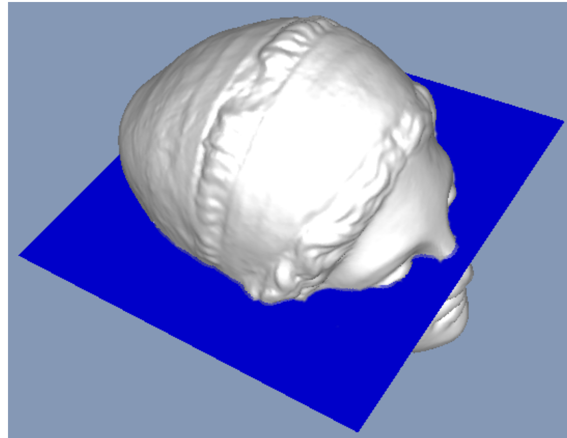


Figura 8.4: Visualizando o posicionamento da seção feita no *dataset igea* (pelo Pointshop3D).

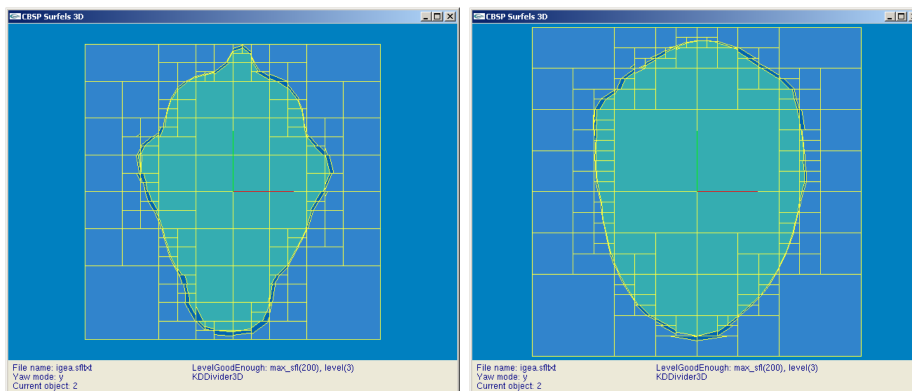


Figura 8.5: Visualizando as seções feitas no *dataset igea*.

8.2 Operações booleanas

A figura 8.7 mostra um clássico exemplo de operação booleana. Inicia-se fazendo a união entre um cubo e um cilindro (figura 8.7.a). Em seguida, outro cilindro, perpendicular ao primeiro, é subtraído do objeto (figura 8.7.b). Por último, um terceiro cilindro, perpendicular aos dois primeiros também é subtraído (figura 8.7.c). As operações foram feitas usando o programa implementado para este trabalho e a visualização foi feita usando o programa *Pointshop3D* [30], com a ajuda de um conversor de formato de arquivo. É possível notar que as bordas geradas na operação booleana possuem algumas imperfeições. Esse é um problema conhecido, e várias formas de melhorar a qualidade das bordas já foram propostas, como partir surfels em surfels de raio menor [2] ou aplicar um operador de suavização [31]. Esses problemas não foram tratados neste trabalho e são discutidos como trabalhos futuros.

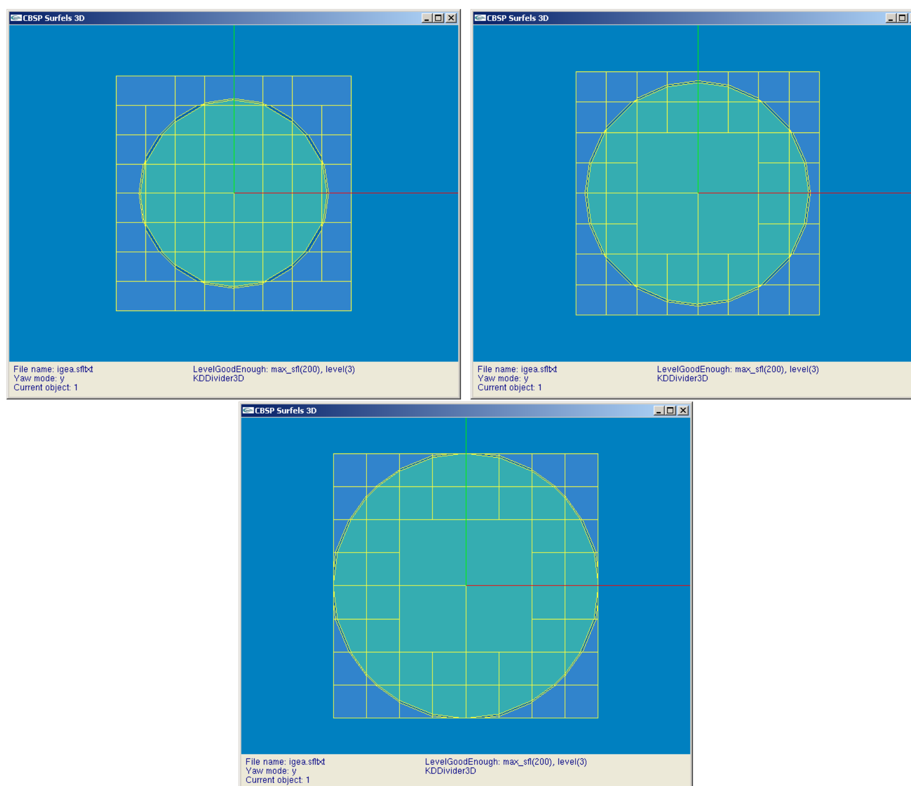


Figura 8.6: Visualizando as seções feitas no *dataset sphere*.

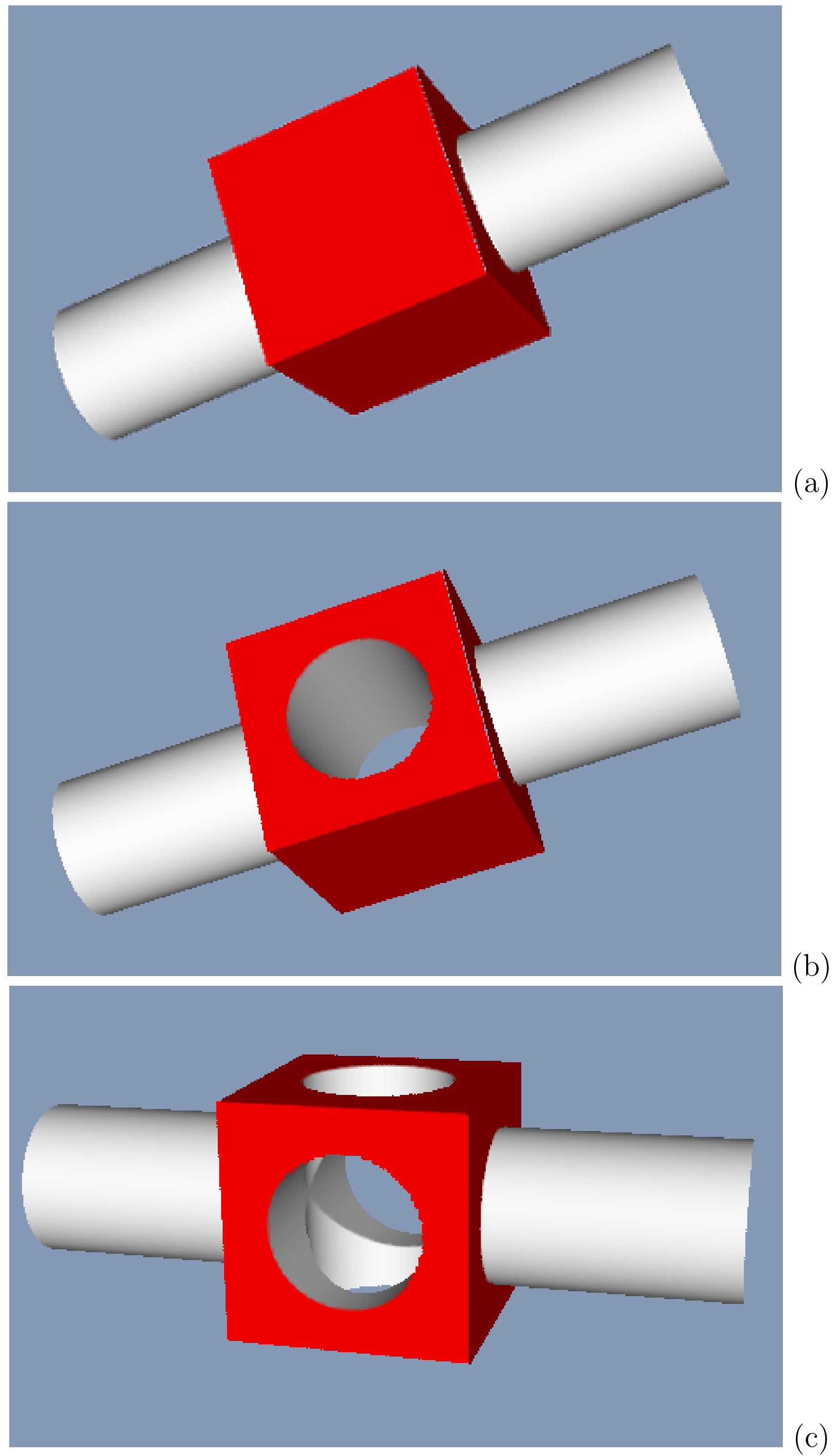


Figura 8.7: Exemplo de operação booleana, feita em três etapas, com um cubo e três cilindros. Visualização feita usando o *Pointshop3D*

9 CONCLUSÃO

A representação de objetos por surfels é bastante útil na computação gráfica, principalmente em conjunto com scanners 3D. Por isso, é necessário também que haja algoritmos para a edição de objetos representados por conjuntos de surfels. Uma das técnicas importantes para a edição de modelos são as operações booleanas, como implementadas em [2], por exemplo.

As estruturas de dados hierárquicas são amplamente utilizadas em computação gráfica, inclusive nas operações booleanas. Vendo a necessidade de combinar cortes inclinados e cortes alinhados na estrutura usada em [2], este trabalho propôs o conceito de BSP-tree (árvore de particionamento binário do espaço) com restrições de cortes (chamada de CBSP-tree).

A aplicação de CBSP-trees para operações booleanas exige que sejam resolvidos dois problemas principais: a escolha de quais cortes usar e a classificação de células. No capítulo 5 foram descritas e comparadas várias estratégias possíveis para a escolha dos particionadores. O algoritmo de classificação extendido para CBSP-trees foi explicado no capítulo 6.

Os capítulos 7 e 8 mostram os resultados obtidos, comparando o número de células necessárias para particionar vários objetos. Nos testes, o número de células foi menor nas CBSP-trees do que nos outros tipos de árvores. Entretanto, em alguns casos a margem de ganho foi muito menor do que o ganho obtido pela substituição de octrees por KD-trees, como pode ser visto mais claramente no capítulo 8. Isso indica que não é trivial escolher um conjunto de estratégias de particionamento que use o potencial de cortes inclinados e que, ao mesmo tempo, não gere células com formatos difíceis de tratar (como células muito finas).

9.1 Trabalhos futuros

Aplicação genérica de CBSP-trees. O conceito da CBSP-tree poderá ser aplicado em várias outras situações. Para isso, é necessário escolher que regras serão adotadas como restrição de cortes e quais benefícios se deseja extrair dos cortes inclinados. Algumas áreas de pesquisa que estão sob consideração são: representação de *distance fields* (ADFs) [32], marching cubes [29] e representação de imagens.

Qualidade das bordas. Quando dois planos não coincidentes se intersectam, gera-se o que é conhecido por "aresta viva". Esse é um problema difícil da área de operações booleanas com surfels, e pode ser interessante pesquisar novas soluções. Dentre as soluções conhecidas atualmente, pode-se citar a apresentada em [2], que usa surfels de raio menor para manter a qualidade das bordas geradas. Por outro lado, o trabalho de Pauly [33] resolve o problema marcando os surfels da aresta e alterando o processo de renderização para exibir a aresta com maior qualidade. É possível também suavizar as arestas criadas na operação booleana, conforme descrito em [31]. Nenhuma dessas alternativas foi implementada neste trabalho.

Melhorias no particionamento. A quantidade de critérios (e combinações deles) que podem ser consideradas durante o particionamento é praticamente infinita. O presente trabalho descreveu várias estratégias úteis, mas algumas não puderam ser suficientemente testadas. As seguintes melhorias poderiam ser exploradas em trabalhos futuros:

- agrupar conjuntos de surfels de acordo com a semelhança entre suas normais;
- usar a informação de densidade dos surfels para detectar quais surfels pertencem à mesma superfície;
- melhorar o *trim* para não só usar cortes paralelos, mas também detectar quando um conjunto de surfels alinhados não ocupa toda a célula (figura 9.1).

Figura 9.1: Se o trim fizer também cortes perpendiculares, será possível obter um particionamento ainda mais preciso.

REFERÊNCIAS

[1] William T. Reeves. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[2] Bart Adams and Philip Dutré. Interactive boolean operations on surfel-bounded solids. In *ACM Transactions on Graphics*, pages 651–656, 2003.

[3] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 335–342, 2000.

[4] M. Levoy and T. Whitted. The use of points as display primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, January 1985.

[5] Jaroslav Křivánek. Representing and rendering surfaces with points. Technical Report DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague, 2003.

[6] Brian Lee Curless. New methods for surface reconstruction from range images. Technical Report CSL-TR-97-733, Stanford University, 1997.

[7] Szymon Rusinkiewicz. *Real-time Acquisition and Rendering of Large 3D Models*. PhD thesis, Department of Electrical Engineering, Stanford University, 2001.

[8] J. P. Grossman and William J. Dally. Point sample rendering. In *9th Eurographics Workshop on Rendering*, pages 181–192, 1998.

[9] J. P. Grossman. Point sample rendering. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998.

[10] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH 2001, Computer Graphics Proceedings*, 2001.

[11] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering*, 2002.

[12] Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. Point-based computer graphics siggraph 2004 course notes. In *SIGGRAPH 2004*, 2004.

[13] Bart Adams and Philip Dutré. Boolean operations on surfel-bounded solids using programmable graphics hardware. In *Eurographics Symposium on Point-Based Graphics 2004*, June 2004.

[14] Mark Pauly. Point primitives for interactive modeling and processing of 3d geometry. Master's thesis, Federal Institute of Technology (ETH), 2003.

[15] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley, 1990.

[16] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[17] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press.

[18] Rodrigo G. Luque, João L. D. Comba, and Carla M. D. S. Freitas. Broad-phase collision detection using semi-adjusting bsp-trees. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186, New York, NY, USA, 2005. ACM Press.

[19] Bruce F. Naylor, John Amanatides, and William C. Thibault. Merging BSP trees yields polyhedral set operations. *Computer Graphics*, 24(4):115–124, August 1990.

[20] Lindsay I. Smith. A tutorial on principal components analysis, 2002.

[21] I. T. Jolliffe. *Principal Component Analysis.* Springer, 2nd edition, 2002.

[22] Jeffrey D. Taft and Rob Sheridan. The better C eigenvector source code page. http://www.nauticom.net/www/jdtaft/CEigenBetter.htm.

[23] Vincent Colin de Verdiere and James L. Crowley. Visual recognition using local appearance. In *ECCV (1)*, pages 640–654, 1998.

[24] S. Gottschalk. Separating axis theorem. Technical Report TR 96-024, The University of North Carolina at Chapel Hill, Department of Computer Science, 1996.

[25] Christoph M. Hoffmann. *Geometric & Solid Modeling.* Morgan Kaufmann Publishers, 1989.

[26] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics (2nd ed. in C): Principles and Practice.* Addison-Wesley Longman Publishing Co., 1996.

[27] Michael A. Greenspan, Guy Godin, and Jimmy Talbot. Acceleration of binning nearest neighbor methods. In *Vision Interface 2000*, 2000.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[29] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of SIGGRAPH 1987*, pages 163–170, 1987.

[30] Pointshop3D. http://www.pointshop3d.com.

[31] Bart Adams and Philip Dutré. A smoothing operator for boolean operations on surfel-bounded solids. Technical Report CW359, Katholieke Universiteit Leuven, Department of Computer Science, 2003.

[32] Sarah F. Frisken, Ronald N. Perry, Alyn P. Rockwood, and Thomas R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, pages 249–254. ACM Press, 2000.

[33] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. In *Proceedings of ACM SIGGRAPH 2003*, 2003.