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

RODRIGO GHELLER LUQUE

# Árvores BSP Semi-Ajustáveis

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. Carla M.D.S. Freitas
Co-orientadora

Porto Alegre, julho de 2005.

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Luque, Rodrigo Gheller

Árvores BSP semi-ajustáveis / Rodrigo Gheller Luque – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

60 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2005. Orientador: João Luiz Dihl Comba; Co-orientadora: Carla M.D.S Freitas.

1. Detecção de colisão. 2. Árvore BSP 3. Estruturas semi-ajustáveis. I. Comba, João Luiz Dihl. II. Freitas, Carla M.D.S. III. Árvores BSP semi-ajustáveis

# AGRADECIMENTOS

Ao meu orientador João Comba, e co-orientadora Carla Freitas, por todo o suporte e estímulo recebido durante a execução deste trabalho.

A todo o pessoal do grupo de computação gráfica da UFRGS e demais amigos que estiveram em contato nesses dois anos.

Agradecimentos muito especiais à minha família: minha mãe Neila, meu pai Francisco, minha irmã Luciana e meu irmão Ricardo.

Por fim, mas não menos importante, meus agradecimentos a todos os contribuintes deste país, que acreditam e ajudam a manter em funcionamento as nossas instituições públicas de ensino.

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AABB | Caixa alinhada aos eixos de coordenadas |
| BSP | Árvore BSP |
| QT | Quadtree |
| LO | Loose Octree |
| SABSP | Árvore BSP Auto-Ajustável |
| SH | Hash Espacial |
| SW | Sweep-and-Prune |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

A etapa de *broad-phase* para a detecção de colisão em cenas compostas de *n* objetos que se movimentam é um problema desafiador, pois enumerar os pares de colisão revela uma complexidade quadrática. Estruturas de dados espaciais são desenvolvidas para acelerar o processo, mas muitas vezes a natureza estática dessas estruturas dificulta o manejo de cenas dinâmicas. Nesse trabalho, é proposta uma nova estrutura chamada de árvore BSP semi-ajustável para representar cenas compostas de milhares de objetos dinâmicos. Um algoritmo de agendamento avalia onde a árvore BSP torna-se desbalanceada, usa várias estratégias para alterar os planos de corte e atualizações preguiçosas para reduzir os custos de reconstrução. É mostrado que a árvore não precisa uma total reconstrução mesmo em cenas altamente dinâmicas, ajustando-se e mantendo propriedades desejáveis de balanceamento e profundidade.

**Palavras-Chave:** Detecção de Colisão, Árvore BSP, Estruturas Semi-Ajustáveis.

**Semi-Adjusting BSP tree**

# ABSTRACT

The broad-phase step of collision detection in scenes composed of $n$ moving objects is a challenging problem because enumerating collision pairs has an inherent $O(n^2)$ complexity. Spatial data structures are designed to accelerate this process, but often their static nature makes it difficult to handle dynamic scenes. In this work we propose a new structure called Semi-Adjusting BSP-tree for representing scenes composed of thousands of moving objects. A scheduling algorithm evaluates locations where the BSP-tree becomes unbalanced, uses several strategies to alter cutting planes, and defers updates based on their re-structuring cost. We show that the tree does not require a complete re-structuring even in highly dynamic scenes, but adjusts itself while maintaining desirable balancing and height properties.

**Keywords:** Collision Detection, BSP-tree, Semi-Adjusting Structures.

# 1 INTRODUÇÃO

## 1.1 Motivação

Tratar cenas dinâmicas de maneira eficiente requer a descoberta dos objetos visíveis considerando o ponto de vista de uma câmera para mostrar tudo o que está acontecendo. Entretanto, detectar a colisão entre os objetos que pertencem ao cenário também é uma tarefa que deve ser feita para simular corretamente o ambiente. A noção de cena dinâmica não se limita a uma câmera deslocando-se em um ambiente composto de objetos estáticos. Entende-se como dinâmico o cenário cujos objetos podem se deslocar livremente e interagir com os outros objetos do ambiente, o que leva à possibilidade de colisão entre eles.

A detecção de colisão entre *n* objetos dinâmicos pode ser resolvida por um algoritmo de complexidade $O(n^2)$, na medida em que se pode, ao movimentar-se um objeto, testar a ocorrência de colisão entre este e todos os demais presentes no cenário. A fim de reduzir a complexidade do problema, um algoritmo de dois passos foi apresentado por Hubbard (HUBBARD, 1995). O primeiro passo é conhecido como *broad phase* e serve para determinar todos os pares de objetos potencialmente em colisão, os quais devem ser testados no teste de colisão exato, que corresponde ao segundo passo, chamado de *narrow phase*. Uma vez que a *broad phase* precisa ser extremamente rápida, todos os testes são realizados usando volumes limitantes tais como as AABBs (*Axis-Aligned Bounding Boxes,* caixas alinhadas aos eixos), OBBs (*Oriented Bounding Boxes,* caixas alinhadas aos objetos) ou k-DOPs (*Discrete Oriented Polytopes*) (EBERLY, 2001).

Estruturas de dados espaciais (SAMET, 1990) são amplamente usadas para acelerar muitos dos algoritmos de *broad-phase.* Porém, poucas são adequadas para tratar cenas dinâmicas, e devem ser estendidas para tal. Uma solução de força bruta é reconstruir a estrutura a cada momento em que as posições dos objetos são atualizadas. Essa abordagem, além de extremamente ineficiente, não é aplicável para simulações em tempo real, devido ao custo computacional de reconstruir a estrutura. É possível tornar essas estruturas híbridas para árvores *BSP* (NAYLOR, 1992). Nesse caso, a estrutura de dados espacial seria construída baseada nos objetos estáticos existentes e os dinâmicos seriam posteriormente inseridos. Essa solução, apesar de adequada para muitos casos, não é sensível ao contexto, uma vez que a estrutura continuará a mesma, independente da quantidade de objetos dinâmicos que forem inseridos no ambiente.

Existem outros métodos mais elaborados de atualizar estruturas de dados espaciais (TORRES, 1990; CHRYSANTHOU, SLATER, 1992; SNYDER, LENGYEL, 1998). Nesses trabalhos, geralmente, são feitas atualizações locais que servem para evitar uma completa reconstrução da estrutura. Entretanto, essas atualizações locais tendem a degradar a estrutura ao longo do tempo, subsistindo a necessidade de reconstruí-la em algum momento no futuro.

O enfoque nessa dissertação é o desenvolvimento de uma estrutura de dados espacial, uma árvore *BSP* (*binary space partitioning tree),* utilizada na *broad-phase* de um processo de detecção de colisão, que se atualize dinamicamente e nunca precise ser reconstruída do zero devido a auto-manutenção constante. As estratégias de atualização foram elaboradas para que se mantenha a estrutura em boas condições e não haja degradação de performance ao longo da simulação. Essa estrutura não requer nenhum pré-processamento e é suficiente para suportar um cenário no qual todos os objetos são dinâmicos e não há nenhum conhecimento sobre a trajetória dos objetos. Assim, todo objeto é considerado dinâmico, mesmo que permaneça a maior parte do tempo em estado de repouso. Essa condição torna o trabalho suficientemente genérico, uma vez que tanto a detecção de colisão como a visibilidade não pode ser auxiliada por uma fase de pré-processamento.

A estrutura foi denominada **Árvore BSP Semi-Ajustável** devido ao artigo de TARJAN e SLEATOR (1985) onde se define árvore de procura semi-ajustável como aquela que reduz o grande montante de reestruturação requerido para um subconjunto de operações. As demais operações são distribuídas nos próximos passos de atualização para amortizar o custo de manutenção.

## 1.2 Objetivo e Contribuições

O objetivo a ser atingido como resultado desta dissertação é a enumeração de pares de colisão em cenas com grande número de objetos dinâmicos em tempo real. As simulações devem abranger situações variadas, com número variado de objetos, e o comportamento em termos de movimento deve ser fisicamente embasado.

Como objetivo específico estão a aceleração da *broad phase* do método de detecção de colisão através do uso de uma nova estrutura de dados espacial, a árvore BSP semi-ajustável.

As principais contribuições dessa dissertação podem ser especificadas pelos seguintes itens:

- Proposição da árvore BSP semi-ajustável e de um conjunto de operações utilizadas para realizar a manutenção da árvore: *split*, *shift split*, *merge*, *swap* e *balance*.
- Algoritmo de agendamento que realiza atualizações de baixo custo na árvore BSP à medida que os objetos se movimentam.
- Comparação com métodos existentes (*quadtree*, *loose octree*, *hash*, *sweep&prune*) em termos de performance, tempo de colisão, tempo de atualização, número de pares de colisão.

## 1.3 Organização do Texto

A dissertação é organizada da seguinte forma: o Capítulo 2 faz uma revisão de trabalhos relacionados ao tema de detecção de colisão, em especial aquele utilizados para comparação com a proposta aqui apresentada. O Capítulo 3 discute o algoritmo de *broad-phase*, destacando a construção da árvore BSP e estratégias de armazenamento dos objetos. O Capítulo 4 e 5 apresentam o algoritmo, descrevendo as operações na árvore e o algoritmo de agendamento que é responsável pela manutenção da árvore. O Capítulo 6 descreve os resultados obtidos e comparação com outros métodos através da apresentação de simulações. Finalmente, o Capítulo 7 apresenta as conclusões e extensões que podem ser aplicadas em trabalhos futuros.

# 2 DETECÇÃO DE COLISÃO

A detecção de colisão é um dos problemas mais estudados na computação gráfica. Diversos tipos de aplicativos, em particular na área de jogos ou simuladores de ambientes virtuais, dependem desses métodos para simularem as situações a que se propõem. Esses aplicativos precisam ser executados em tempo real, necessitando, portanto, algoritmos ou abordagens de baixo custo.

Existe uma extensa literatura propondo soluções (COHEN et al., 1995; HUDSON et al., 1997; GOVINDARAJU et al., 2003; TESCHNER et al., 2003; JAMES, PAI, 2004), além de várias investigações sobre o assunto (DAVE, EBERLE, 2004; HEIDELBERGER 2004; BERGEN 2004). Pacotes de software bem conhecidos para resolver esse problema incluem o SOLID (BERGEN, 2004), I-COLLIDE (COHEN, 1995), V-COLLIDE (HUDSON 1997), ODE (SMITH 2004).

Conforme já mencionado, a detecção de colisão em cenas dinâmicas é normalmente resolvida com um algoritmo de dois passos (HUBBARD, 1995). Na *broad phase*, são enumerados pares de objetos que podem estar colidindo, os chamados *pares de colisão*. Na *narrow phase*, é realizado um teste de colisão exato, envolvendo a geometria dos objetos dos pares enumerados na *broad phase*.

Há duas abordagens comuns entre os algoritmos de detecção de colisão que servem para acelerar os testes de colisão. A primeira delas é o uso de coerência temporal que explora a possibilidade de, entre diferentes *frames*, objetos estarem na mesma posição ou em posições próximas. A segunda seria usar volumes limitantes para acelerar o teste de colisão de objetos mais complexos (BERGEN, 2004). Hierarquias de volumes limitantes, tais como árvores de esferas (*sphere trees*) (HUBBARD, 1995), *oriented-bounding-box trees* (*OBB trees*) (GOTTSCHALK et al., 1996), *axis-aligned-bounding-box trees* (*AABB trees*) (BERGEN, 2004), da mesma forma que particionamento de espaço como *quadtrees*, *octrees*, *kd-trees* e *árvores-BSP* têm sido explorados para otimizar tanto a *broad* como a *narrow phase*.

Nas seções a seguir, exploramos diversas alternativas de detecção de colisão, em particular abordagens que aceleram a *broad phase*, em função do objetivo deste trabalho.

## 2.1 Abordagens da Broad Phase

A proposta da *broad phase* é determinar eficientemente o conjunto de pares de colisão a ser passado para a *narrow phase*. Além disso, existem diferentes métodos para evitar o custo quadrático da comparação dos pares de colisão que usam estruturas hierárquicas para minimizar o número de objetos a serem testados. Projetar os extremos dos volumes limitantes sobre os eixos de coordenadas é a base da técnica *sweep-and-prune* (COHEN, 1995). Outra abordagem é armazenar os volumes limitantes identificados em uma tabela espacial *hash* que representa uma repetição 3D do espaço em diferentes níveis de resolução (MIRTICH, 1997). Apenas os volumes dentro do mesmo espaço ou de espaço em níveis de resolução menor formam pares de colisão.

O uso de estruturas de dados hierárquicas para acelerar a *broad phase* requer a atualização destas estruturas quando objetos se movem. As questões sobre atualizações dinâmicas em estruturas hierárquicas surgiram na literatura em problemas relacionados a colisão, como por exemplo no cálculo de visibilidade. As soluções propostas, entretanto, podem ser aplicadas em ambos problemas, e por isso são discutidas aqui.

As soluções apresentadas usualmente seguem três abordagens: força-bruta, conservadora e auto-ajustável. Na abordagem força-bruta descrita por CHRYSANTHOU e SLATER (1992), objetos que se movimentam são removidos a cada passo e re-inseridos para manter a estrutura atualizada. Em um outro trabalho, TORRES (1990) usa cortes de planos auxiliares para minimizar o impacto na árvore *BSP* dos objetos que se movimentam.

A abordagem conservadora vai além da força bruta ao aplicar um esquema de re-balanceamento, mas eventualmente uma total re-construção é necessária quando a árvore se degrada após vários passos de atualização. Essa abordagem muitas vezes é custosa pelo fato de estar atualizando a árvore a todo o momento para tentar evitar o processo de degradação estrutural. SNYDER e LENGYEL (1998) usam uma *kd-tree* para indexar objetos. A inovação deste trabalho esta no fato que após a movimentação de objetos, os cortes na *kd-tree* são deslocados para permitir o re-balanceamento da árvore, evitando assim uma reconstrução total a cada movimentação. Note que apenas a posição dos cortes é alterada, e não a seqüência de eixos usados para fazer particionamento. Esta simplificação faz com que em algumas situações a estrutura degrade severamente e uma reconstrução total é necessária.

A abordagem mais elaborada utiliza estruturas auto-organizáveis (ou auto-ajustáveis) com a premissa que a estrutura deve adaptar-se a si mesma dependendo de como é usada. AR e CHAZELLE (2002) e AR e MONTAG (2002) adotaram auto-organização da árvore *BSP* para mostrar cenas estáticas complexas, onde a construção da árvore é incremental e leva a uma estrutura parcial que tem suficiente informação para responder perguntas de visibilidade e colisão. Ao fazer isso, a atualização da estrutura é postergada, uma vez que a configuração completa da árvore é adiada até quando realmente se precisa da informação.

Estruturas de dados cinéticas (*kinetic data structures)* foram discutidas para árvores *BSP* (AGARWAL et al., 1998; COMBA, 2000). A principal idéia é usar um modelo de predição para estimar quando a estrutura combinatorial da árvore *BSP* muda, seguido pelo passo de auto-ajuste que realiza atualizações locais na árvore *BSP*. O modelo de predição requer que os objetos tenham trajetórias conhecidas e a atualização pode ser mais cara, uma vez que esses dados podem fazer com que a árvore fique se modificando continuamente para atender às demandas. Esse trabalho é baseado no agendamento de

modificação estrutural (representado por um conjunto de operações) em intervalos regulares.

Assim, estas operações são distribuídas ao longo dos passos de simulação para não afetar a performance e manter a árvore o melhor possível. Dessa maneira, o método posterga também o conjunto mínimo de operações para re-configurar a árvore, estando de acordo com a característica semi-ajustável apontada por TARJAN e SLEATOR (1983; 1985; 1986).

Embora existam muitas estruturas para *broad phase* com diferentes abordagens, poucas estruturas são de fato utilizadas para processamento em tempo real. Esse fato ocorre porque muitas das estruturas propostas possuem implementação complicada ou não tem um ganho significativo. Nas próximas seções, as estruturas mais amplamente usadas são descritas em maior detalhe, para posteriormente serem comparadas com o método proposto nesta dissertação.

## 2.2 Quadtree

A *quadtree* (SMITH 2004) representa uma árvore de particionamento no espaço bidimensional que geralmente tem seu domínio alinhado aos eixos coordenados. Portanto, o nodo raiz da *quadtree*, representado pelo nodo vermelho na figura 2.1, corresponde a todo o espaço descrito pelas bordas vermelhas. Esse nodo é particionado em quatro nodos iguais, usando a mediana do espaço representado por aquele nodo. Assim, quatro nodos representados pelas bordas laranja. Esses nodos são novamente divididos surgindo 4 nodos definidos pelas bordas azuis. Considera-se aqui o uso de uma *quadtree* completa, ou seja, a árvore é completa (com todos os nodos filhos) até um determinado nível.

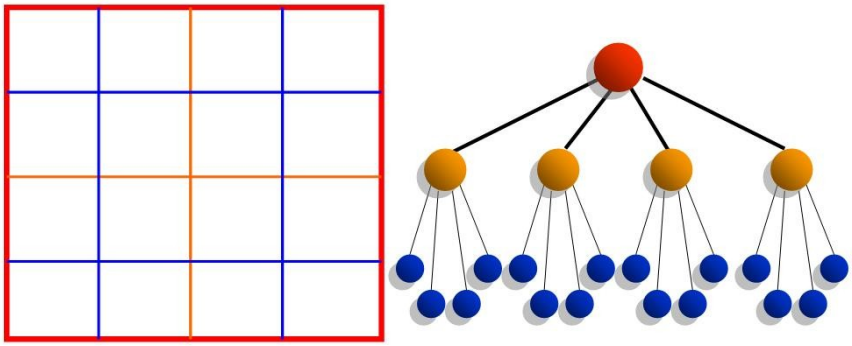


Figura 2.1 - *Quadtree* completa

Os volumes limitantes são inseridos na árvore dependendo do espaço que ocupam. Se o volume limitante interseccionar uma determinada borda de um nodo, como por exemplo, a borda laranja, esse volume limitante será inserido no nodo vermelho ao invés de em seus filhos, pois ele está em mais de um filho do nodo vermelho. Para cada volume limitante, tenta-se inseri-lo no nível mais profundo possível. Um problema comum dessa abordagem ocorre quando muitos volumes limitantes interseccionam bordas de nodos próximos à raiz. Isso faz com que os objetos se mantenham próximos a raiz em um grande subespaço ao invés de serem refinados para subespaços menores.

Para testar possíveis colisões, cada um dos volumes limitantes dentro dos nodos é testado contra os outros dentro do mesmo nodo. Portanto, dentro de um nodo, o teste de colisão tem ordem quadrática. O maior problema é que os volumes limitantes contidos em nodos próximos às folhas ou até nas folhas têm que testar colisão contra os volumes limitantes dos nodos ascendentes. É por isso que é um problema ter muitos volumes limitantes próximos ao nodo raiz, pois estes serão testados contra todos os filhos do nodo em que se encontram.

## 2.3 Loose Octree

Para evitar o problema de ter muitos volumes limitantes próximos ao nodo raiz, pode-se adotar o método *loose octree* (THATCHER, 2000). Como é de conhecimento geral, uma *octree* é uma estrutura similar a *quadtree*, mas definida no espaço tridimensional.

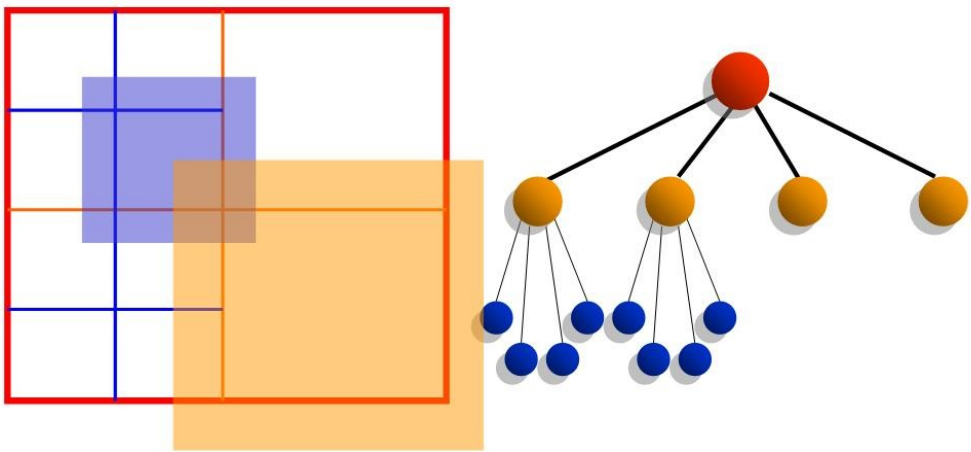


Figura 2.2: *Loose Quadtree*

Na *loose octree,* diferentemente do método adotado com a *quadtree*, não será usada uma estrutura completa por causa dos requerimentos exponenciais de memória. Nesse caso, cada vez que o nodo precisa ser dividido, são criados novos nodos. O mesmo se aplica para nodos vazios, pois estes serão destruídos para aliviar o consumo de memória. Essa gerência de nodos é uma diferença vital em relação à *quadtree* completa, uma vez que há um *overhead* para criar e destruir nodos. Para resolver o problema de

volumes limitantes ficarem interseccionando bordas de nodos próximos a raiz, a *loose octree* expande o volume dos nodos (de onde advém o nome "*loose" octree)*, fazendo com que estes intersecionem seus volumes entre si, como mostrado na figura 2.2. Nessa figura, está sendo considerada uma *quadtree* para facilitar a visualização da estrutura.

A *loose octree* tem um tratamento diferenciado para a determinação da colisão. Ao invés de testar apenas colisão com os nodos ascendentes, é necessário verificar a colisão com os demais nodos irmãos e seus filhos. Tal fato ocorre pela intersecção entre os volumes representados pelos nodos. Assim, o número de testes de colisão entre objetos até pode ser reduzido, mas há a necessidade de testar quais nodos estão se intersecionando. A enumeração dos pares de colisão é um dos pontos fracos desse método, uma vez que o percurso pela árvore para gerar os pares é mais complicado.

## 2.4 Hash espacial

O método utiliza uma função *hash* para mapear volumes do espaço para células de uma lista de tamanho qualquer (SMITH 2004). Quanto maior o tamanho da lista, menor a possibilidade de dois volumes limitantes em posições distintas serem mapeados para a mesma célula da lista. Entretanto, um volume limitante pode ter tamanho maior que o volume representado por uma célula da lista. Por isso, é necessário utilizar mais de uma lista para representar volumes distintos (figura 2.3). O número de listas depende do menor e maior volume limitante que será usado. Uma célula de uma lista do nível n é sempre representada por duas células da lista do nível n + 1.

Como um volume limitante deve ser mapeado necessariamente para apenas uma célula de uma lista, deve-se encontrar a célula cujo volume é o mais próximo do objeto.

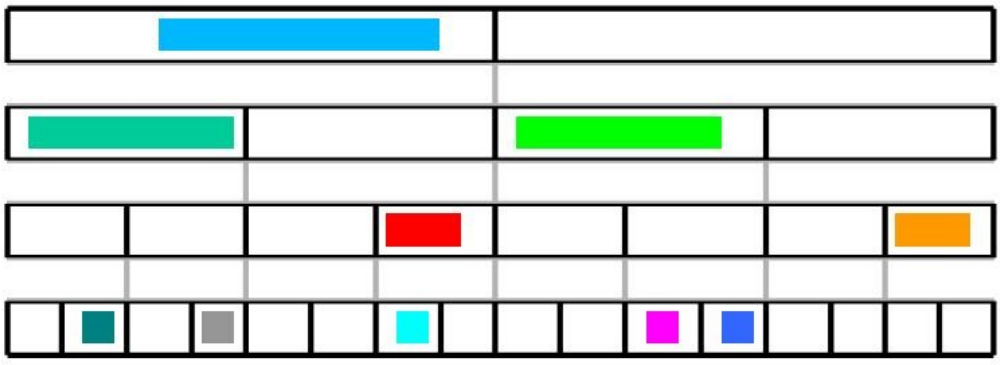


Figura 2.3: *Hash* espacial

A geração dos pares de colisão é relativamente simples. Dentro de cada célula das listas deve-se testar todos contra todos, logo a complexidade é quadrática. Deve-se fazer testes entre as células de diferentes listas, pois o volume representado por uma célula da

lista no primeiro nível do *hash*, como vemos na figura 2.3, contém duas células do nível dois, e quatro do nível três. Como elas representam volumes do espaço que se interseccionam, os volumes limitantes indexados por estas células devem ser testados entre si.

## 2.5 Sweep-and-Prune

O último método a ser descrito é bastante conhecido: o método *sweep-and-prune* (COHEN et al., 1995). O método *sweep-and-prune* baseia-se em operadores de projeção, que é uma abordagem comum para calcular intersecção entre objetos. Os volumes limitantes são projetados sobre os eixos X,Y,Z para enumerar os pares de colisão.

Para cada eixo, o conjunto de volumes limitantes é varrido, é armazenado o mínimo e o máximo de cada volume limitante e a geração dos pares ocorre quando há intersecção de intervalos. A geração dos pares para cada eixo pode ocorrer em tempo linear e a eliminação dos pares ocorre se um dado par não é gerado por pelo menos um eixo. Essa fase de *prunning* ocorre normalmente utilizando uma matriz NxN, sendo N o número máximo de objetos, para ocorrer em tempo constante. A figura 2.4 mostra o método operando nos eixos X, Y.

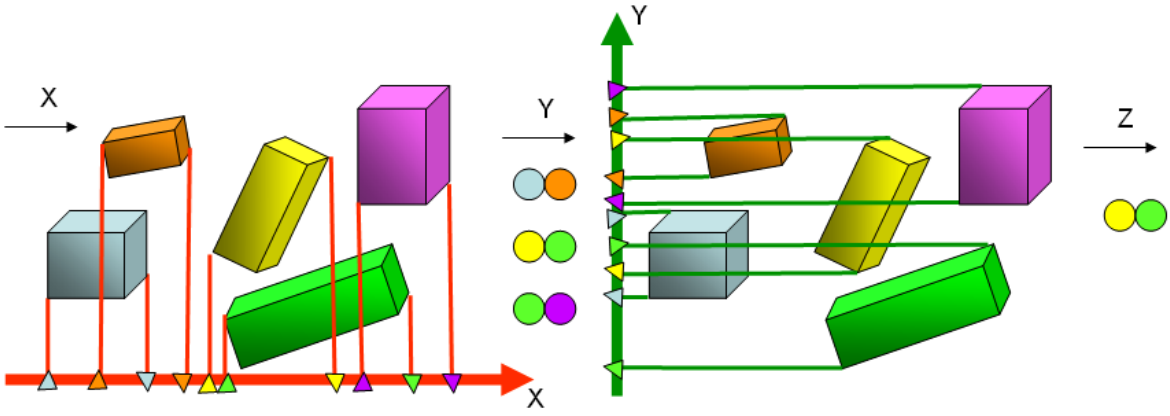



Figura 2.4: Projeção dos objetos nos eixos e remoção dos pares.

# 3 CONSTRUÇÃO DE ÁRVORE BSP PARA DETECÇÃO DE COLISÃO

Inúmeras estratégias devem ser avaliadas para desenvolver uma *broad-phase* eficiente e foram revistas no capítulo 2. Dentre as árvores existentes, a árvore *bsp* foi escolhida por ser flexível, uma vez que o corte do espaço pode ser feito em qualquer direção. Tal propriedade a torna mais flexível à quebra espacial para se adaptar a diferentes distribuições de volumes limitantes pelo espaço. As demais árvores apresentam restrições como os cortes serem paralelos aos eixos de coordenadas. Ainda há a necessidade de avaliar outros aspectos dessa estrutura.

O custo de manutenção da árvore, quando os volumes limitantes se movimentam, deve ser baixo e, ao mesmo tempo, o número de pares de colisão deve ser reduzido ao máximo. Por conseguinte, é necessário avaliar como as volumes limitantes devem ser armazenados na árvore, e como podem auxiliar na construção e avaliação da qualidade da árvore.

Diferentemente das estratégias adotadas em outros trabalhos relacionados, a estrutura proposta combina o uso de hierarquia com *sweep* para minimizar ao máximo o número de pares de colisão. Tal fato evita a necessidade de uma hierarquia grande, minimizando custos para sua manutenção. As seções seguintes descrevem princípios básicos para a construção da árvore BSP aqui proposta enquanto o próximo capítulo descreve as operações que a tornam semi-ajustável.

## 3.1 Projeção em eixo arbitrário

Projeção em um eixo de coordenadas reduz o número de pares de colisão se esse eixo representa a normal de um plano separador (GOTTSCHALK et al., 1996). Dependendo da configuração dos volumes limitantes, a projeção em um eixo arbitrário pode retornar resultados similares ou melhores que utilizar o eixo de coordenadas, além de reduzir o custo linear por apenas necessitar uma passada ao invés de três e remover a fase de *prunning* em comparação com o método *sweep&prune*. A figura 3.1 mostra vários volumes limitantes sendo projetados em um eixo.

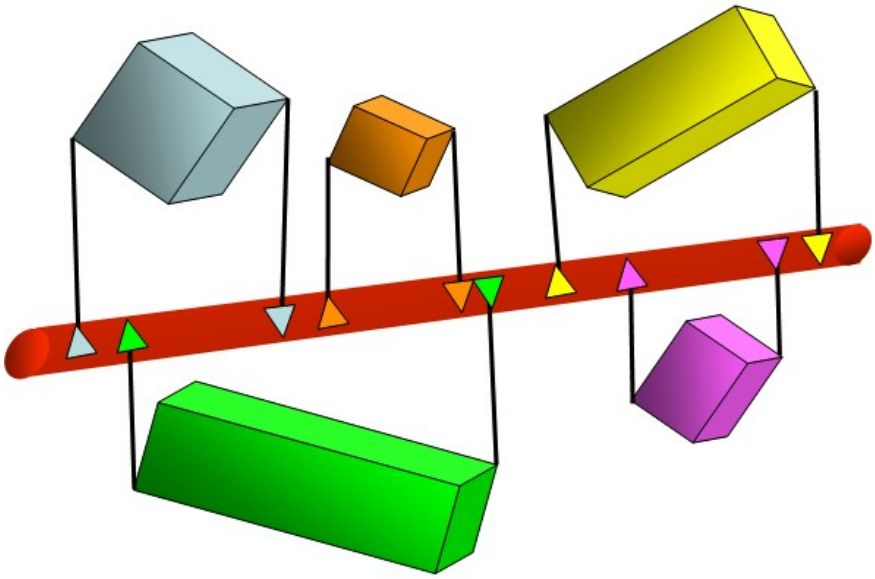


Figura 3.1: Projeção dos volumes limitantes em um eixo arbitrário

Cada um desses volumes limitantes possui um intervalo de projeção no eixo arbitrário. Nota-se que a intersecção de intervalos de projeção representa uma possível colisão entre os objetos. Assim, facilmente podem ser extraídos os pares de colisão.

## 3.2 Projeção dos volumes limitantes (AABBs)

Projetar um objeto qualquer pode ser relativamente complexo dependendo do número de vértices que este objeto contém. Podem-se utilizar estruturas hierárquicas para facilitar a projeção e calcular mais facilmente os vértices da extremidade da projeção. Entretanto, muitas vezes o custo benefício de usar o objeto original não compensa. Assim, estruturas mais simples, como caixas, esferas ou até cilindros são utilizadas ao invés disso. Essas estruturas encapsulam o objeto original (logo podem dar origem a falsos positivos em testes) e são denominados como volumes limitantes (envelopes de colisão). Hierarquias ou a composição dessas estruturas podem ser usadas para minimizar a presença de falsos positivos. A projeção de um volume limitante requer que sejam computados os vértices da extremidade com respeito ao eixo de projeção arbitrário fornecendo um intervalo com dois valores escalares que indicam o mínimo e o máximo na equação paramétrica do eixo.

A projeção de uma AABB nos eixos coordenados é simples, uma vez que a AABB contém o mínimo e máximo para cada eixo. Projetar os extremos da AABB em um eixo qualquer requer o uso de produto escalar com o uso da normal do plano separador. A normal é usada para determinar quais valores de mínimo e máximo da AABB devem ser usados no produto escalar a fim de projetar as extremidades. A Figura 3.2 mostra o procedimento que deve ser feito.

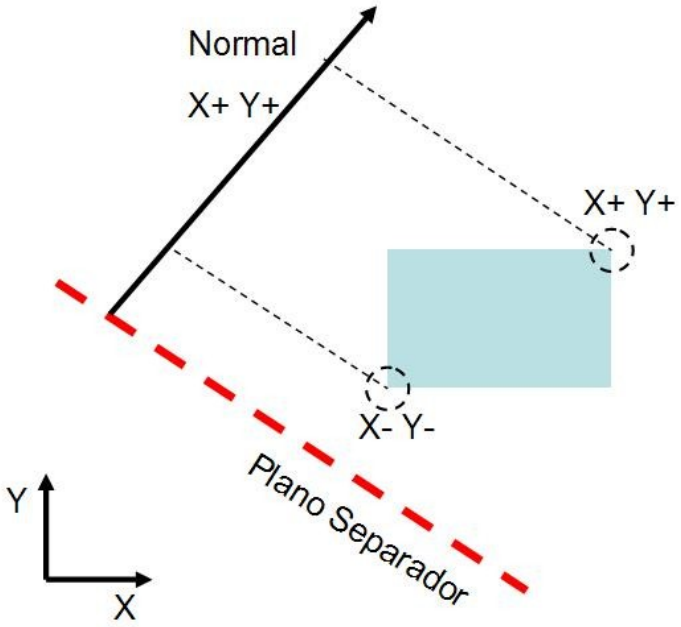



Figura 3.2: Determinação do vértice relevante da AABB

No exemplo da Figura 3.2, um dos vértices da AABB a ser considerado é aquele que formando o vetor entre um vértice da AABB e o centro da AABB possui o sentido mais próximo da normal. Como a normal possui coeficientes positivos para X e Y, o vértice com deslocamento positivo em X,Y em relação ao centro da AABB será considerado. O mesmo se aplica ao sentido inverso da normal para descobrir o outro vértice significativo.

## 3.3 Determinação do intervalo de corte

O plano separador para duas AABBs pode ser deduzido com facilidade se seus intervalos não se interseccionam. Entretanto, para mais AABBs, a determinação do plano separador não é tão óbvia, uma vez que deve ser escolhido o plano separador que maximize o número de pares de objetos separados. Esse problema se torna bem evidente quando ocorrem muitas intersecções entre intervalos próximos a intervalos que dividem em partes similares o número de volumes limitantes. Na Figura 3.3, podemos facilmente separar o espaço no intervalo dado entre o máximo da AABB verde e o mínimo da AABB amarela. Entretanto, é facilmente visível também que o ponto escolhido não é a mediana da lista ordenada, mas um membro mais próximo ao final da lista. Caso separássemos na mediana da lista ordenada, separaríamos entre o máximo da AABB laranja e o máximo da AABB verde. Apesar de particionar a lista em partes iguais, o intervalo pertencente a AABB verde estaria em ambas as listas, o que significa que a AABB verde seria duplicada.

Apesar de intuitivamente se pensar que a mediana da lista ordenada pode ser um bom particionador, isso não é necessariamente verdade como pode ser observado na Figura 3.3. A mediana pode ser um bom ponto de partida para a procura de intervalos

de corte do subespaço. Além disso, existem outras propriedades que devem ser respeitadas e que serão discutidas mais adiante.

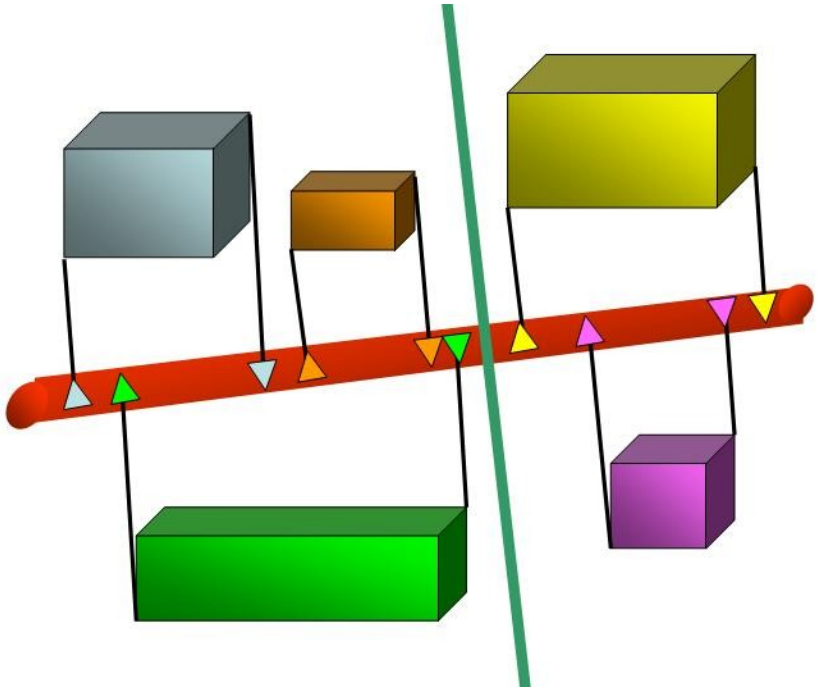


Figura 3.3: Corte do espaço

## 3.4 Estratégia de armazenamento

O particionamento de espaço é usado quando o número de AABBs fica significativamente grande. A árvore BSP é uma estrutura que permite gerar planos de corte em direções arbitrárias. Particionadores são armazenados nos nodos da árvore binária e as AABBs são armazenadas nas folhas ou nos nodos intermediários. O armazenamento das AABBs nas folhas ou nodos normalmente é feito através de listas. O armazenamento nos nodos não é normalmente recomendado para AABBs que se movimentam livremente pelos seguintes motivos:

- Criação e destruição de listas em nodos por AABBs que estão migrando entre células.
- Maior dificuldade para gerar os pares de colisão, uma vez que se devem considerar todos os nodos da árvore.
- Aumento no número de listas a serem atualizadas.

Entretanto, existem vantagens em utilizar os nodos intermediários para o armazenamento de AABBs que devem ser levadas em consideração. Podemos destacar as principais vantagens como:

- Evitar redundância de AABBs pelo seu armazenamento em nodos intermediários quando esse intersecciona um particionador.

- Facilitar a avaliação de um corte e sua aceitação, uma vez que o balanceamento seria o único fator a ser considerado.
- Ser mais flexível para atingir metas de partição de espaço.

Apesar das vantagens, uma grande desvantagem é ter que consultar nodos intermediários na hora de gerar os pares de colisão. Tal fato torna a árvore custosa e também esconde valores de redundância verdadeiros que estão implícitos nos nodos intermediários. É claro que se torna mais fácil na hora de particionar o espaço, pois apenas são levadas em consideração as AABBs que estão apenas no subespaço a ser particionado. Assim, pode ser criada uma árvore que atinge os valores de granularidade requeridos, mas que provavelmente estará além do esperado em termos de redundância, gerando, por conseguinte, muitos pares de colisão. Os pares de colisão precisam ser gerados também de uma maneira eficiente, e o fato de ter que visitar não somente as folhas para isso, mas os nodos, também, torna a geração dos pares mais custosa. Portanto, foi decidido armazenar AABBs na folhas para simplificar a enumeração de pares de colisão (Figura 3.4).

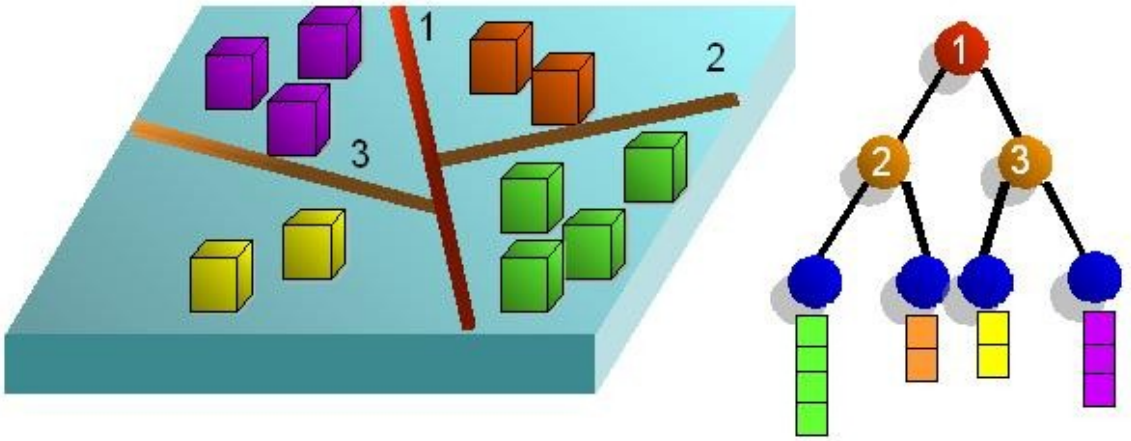



Figura 3.4: Particionamento de espaço

## 3.5 Escolha da direção de corte

Escolher bons particionadores é essencial para performance nessa abordagem. A análise de componentes principais (PCA) pode ser usada, mas requer uma solução cara de um sistema linear que tem uma matriz de covariância com a dimensão proporcional ao número de AABBs. Esse método pode ser usado para a fase de pré-processamento ou com um pequeno número de AABBs. Para cenas com AABBs que se movimentam ou se deformam constantemente, a necessidade de computar planos separadores rapidamente leva a uma aproximação ou uma solução heurística.

Foi utilizada uma aproximação que enumera os candidatos a serem particionadores e escolhe um deles de acordo com um determinado critério que deve ser maximizado. As

direções dos particionadores são limitadas a um conjunto pré-definido que se assemelha ao hemisfério de possíveis direções seguindo a aproximação de repulsão de ponto (HASAN et al., 2001). A Figura 3.5 mostra o conjunto de cortes possíveis.

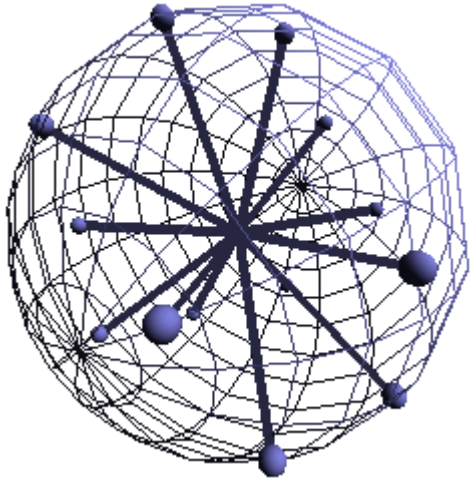


Figura 3.5: Direções de corte

O conjunto de cortes candidatos contém ao todo 13 direções. Essas direções são criadas a partir da combinação dos sentidos nos eixos ordenados x,y,z. Posteriormente, a aproximação de repulsão de ponto processa as direções de corte, criando um novo conjunto final de direções de corte. Entretanto, processar as direções de corte pelo método de HASAN et al. (2001) é opcional.

Treze direções de corte é um valor adequado, uma vez que não possui um imenso conjunto de cortes e ao mesmo tempo não restringe demasiadamente a árvore. O uso de um conjunto maior que este com um grande número de direções pode ocasionar perda de performance pelo fato de requerer a avaliação de um conjunto muito grande de direções ao particionar o espaço. Além disso, o ganho para escolher um particionador ótimo é pequeno, uma vez que no próximo passo esse particionador pode deixar de ser ótimo e ter que ser substituído por outro. Por outro lado, usar um conjunto reduzido como o da *kd-tree* dificulta o particionamento. A falta de direções para particionamento força a escolha de particionadores não ótimos, degradando a qualidade dos cortes e aumentando a profundidade da árvore para obter células abaixo de um determinado limiar de população.

Uma abordagem que poderia ter sido usada é escolher particionadores perpendiculares ao deslocamento das AABBs. Esses particionadores não teriam que ser reavaliados ou deslocados freqüentemente se a movimentação das AABBs fosse linear. Entretanto, essa abordagem teria que considerar particionadores com direções arbitrárias ao invés de um conjunto. Além disso, funcionaria se as AABBs se deslocassem uniformemente em uma determinada direção. Normalmente, não é possível considerar tal fato, pois as AABBs podem estar se deslocando em direções arbitrárias, dificultando o uso de uma análise de baixo custo.

## 3.6 Propriedades do particionador

A fim de escolher o particionador, alguns critérios devem ser considerados para avaliar a qualidade do particionamento. Esses critérios podem ser extraídos das listas ordenadas para a determinação do intervalo ótimo de corte. Considerando um particionador *p*, temos que:

**população**(*p*): número de AABBs testados contra *p*.

**balanceamento**(*p*): diferença entre AABBs num subespaço *p* positivo e um subespaço *p'* negativo.

**redundância**(*p*): número de AABBs contidos em ambos subespaços positivo e negativo de *p*.

A Figura 3.6 mostra um exemplo prático dos critérios acima descritos.

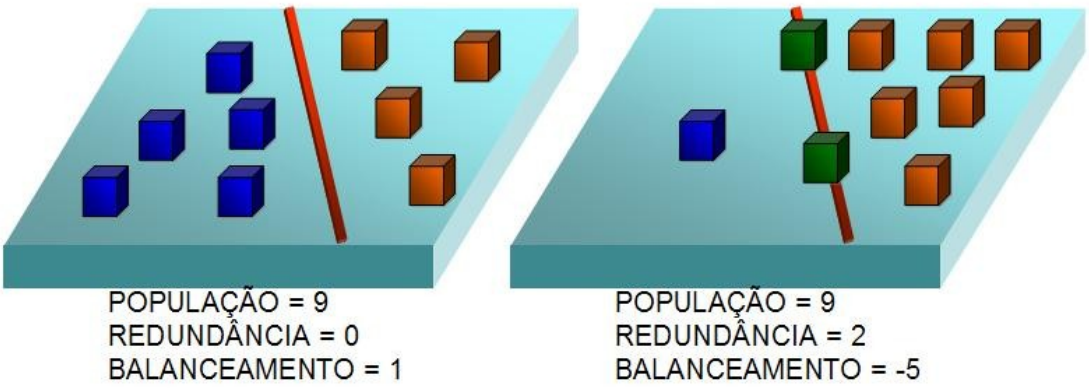



Figura 3.6: Demonstração de população, redundância e balanceamento.

O balanceamento representa como as AABBs são distribuídas nas sub-árvores de um nodo. Se o particionador gera um balanceamento inadequado, neste caso, uma sub-árvore com uma grande população e a outra com uma pequena população, a árvore tenderá a ficar desbalanceada devido a grande diferença de profundidade entre as folhas. Tal situação é indesejável por gerar uma árvore ineficiente.

Por outro lado, há redundância quando o plano particionador intercepta AABBs, levando ao armazenamento dessas AABBs em ambas as sub-árvores. Por conseguinte, há um aumento no custo de atualização da estrutura, pois a mesma AABB será atualizada mais de uma vez e irá pertencer a mais de uma lista ordenada. A ausência de redundância é um fator determinante para a eficiência da estrutura, uma vez que aumenta o custo de atualização da árvore e ao mesmo tempo pode afetar o número de pares de colisão gerados, podendo gerar pares redundantes.

Após a escolha da direção do corte de particionamento, as AABBs são projetadas no eixo definido pela normal e armazenadas usando uma lista ordenada. Essa lista é usada para selecionar um intervalo apropriado e, entre todos os candidatos, é selecionado o intervalo de referência que maximize o balanceamento enquanto minimiza a redundância (Figura 3.7).

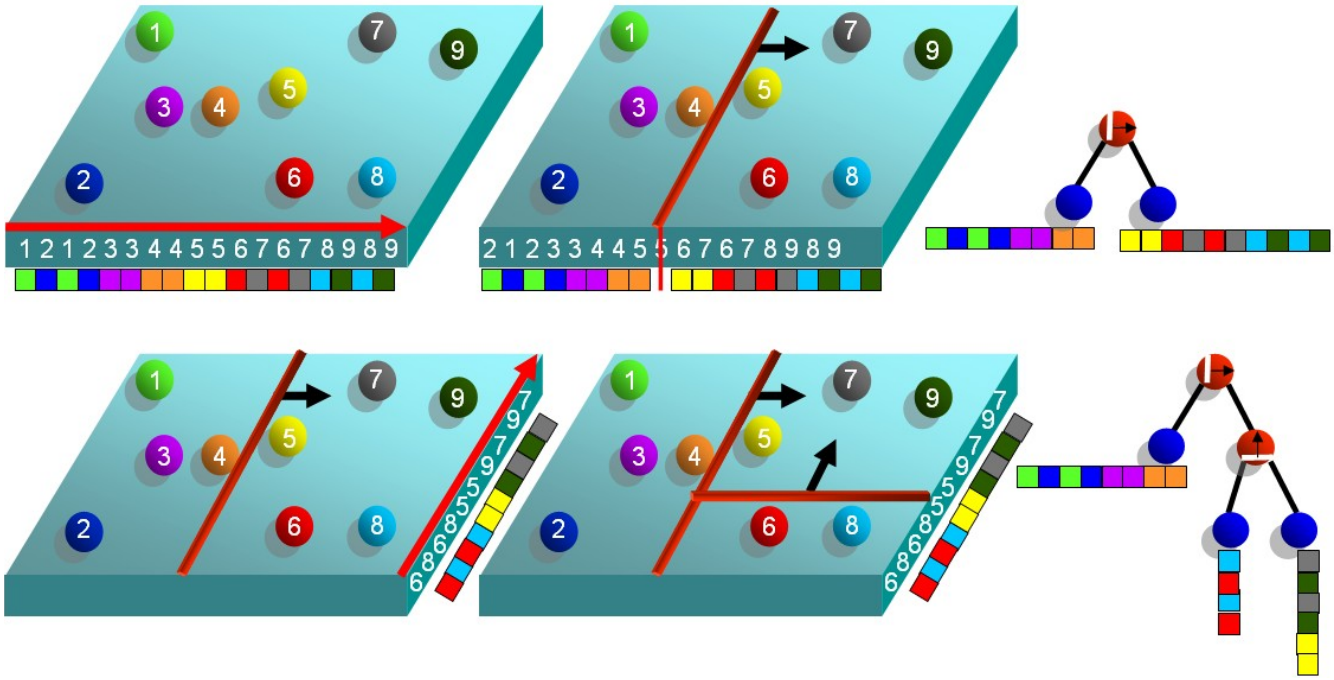


Figura 3.7: Projeção das AABBs em uma normal fornece uma lista ordenada que é usada como ponto de referência para o particionamento.

A lista ordenada precisa ser percorrida para se determinar um ponto de corte ideal. Ao percorrer a lista, se armazena a qualidade do último intervalo de corte e se compara com o novo intervalo de corte para ver se este é melhor ou não. O número de intervalos abertos representa o valor de redundância que tal intervalo de corte tem. Assim, um bom ponto de corte é aquele que não possui nenhum intervalo aberto no momento da análise. Além disso, o valor de balanceamento é facilmente dedutível, uma vez que se sabe o número de AABBs que já foram analisadas (número de intervalos fechados) e o número de AABBs restantes. Com isso, ao se percorrer a lista, é fácil encontrar o ponto de corte ideal para aquela lista ordenada.

Entretanto, nem toda a lista precisa ser avaliada, uma vez que é intuitivo pensar que intervalos próximos ao início e fim da lista contém valores de balanceamento que não satisfazem o critério mínimo. Portanto, existem otimizações que podem ser feitas para não visitar partes não-úteis da lista. A determinação do número de intervalos abertos força a análise da lista desde o seu início, mas a análise pode parar no momento que se aproximar do final da lista. Tal fato ocorre pois quanto mais próximo do final, pior o valor de balanceamento. Caso esse valor saia fora do critério mínimo de escolha do corte, é garantido que o final da lista apenas conterá valores piores ou iguais. Portanto, torna-se evidente que não é necessário visitar todas as AABBs da lista intervalar ordenada.

# 4 OPERAÇÕES SEMI-AJUSTÁVEIS SOBRE ÁRVORES BSP.

Modificações na árvore BSP causada pelas movimentações das AABBs tem de ser feitas para preservar certas propriedades de pesquisa. Além disso, essas modificações só podem ser realizadas dentro de um certo tempo disponível, e, neste caso, as atualizações devem ocorrer no menor tempo possível. Tais modificações não podem levar a uma degradação da estrutura que atinja um estado irrecuperável. Atualizações ótimas são caras de computar e muitas vezes desnecessárias. Neste capítulo, será mostrada uma lista das operações estruturais e no capítulo 5 será discutido como essas operações são agendadas para realizarem adequadamente as atualizações.

## 4.1 Operação *Split*

A operação *split* divide uma folha com uma população maior que um determinado limiar (Figura 4.1). A folha é dividida com um novo particionador, tornando-se um nodo e duas folhas são geradas. A Figura 4.1 ilustra uma operação de divisão considerando um limiar de 5 AABBs.

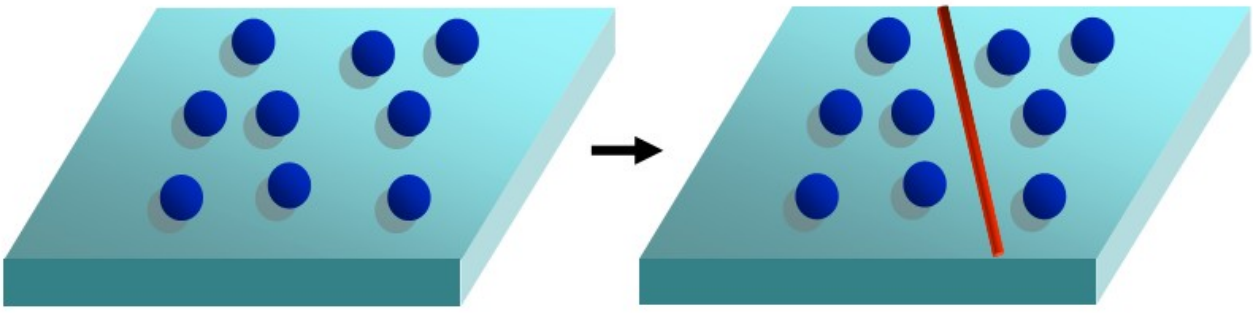


Figura 4.1: Operação de *split*

Escolher um plano particionador segue a abordagem discutida no capítulo 3. Ordenar as projeções das AABBs contra as normais candidatas é o fator mais custoso para essa operação e para reduzir o custo dessa operação é considerado apenas um subconjunto desses objetos.

Assim, para avaliar a melhor direção de corte, listas ordenadas temporárias são criadas a partir de uma amostragem da população e, posteriormente, avaliadas. A direção que possui o corte que maximize o critério discutido no capítulo 3 é escolhida. É importante lembrar que a operação de divisão não necessariamente tem sucesso. Podem-se ter casos que dentre todos particionadores, nenhum deles atinja o critério mínimo desejado.

## 4.2 Operação *Shift-Split*

A operação *Shift-Split* representa uma variação muito mais eficiente para dividir uma folha. Esta operação usa a informação armazenada na folha da árvore binária de particionamento para eliminar o custo de ordenamento de uma operação *split*. A Figura 4.2 ilustra a operação.

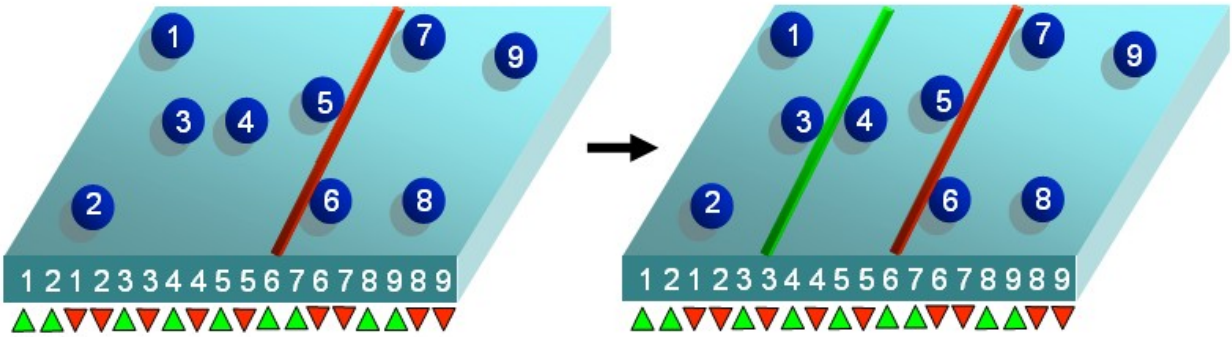



Figura 4.2: Operação *shift-split*.

Para cada folha da árvore, há uma lista ordenada com as AABBs utilizando a normal do nodo pai. Essa lista pode ser usada para encontrar outro particionador com a normal paralela a normal do pai. Se o novo particionador satisfaz um determinado critério estabelecido, este é usado ao invés de utilizar a operação *split*. Tal operação pode aumentar o tamanho da árvore, uma vez que um particionador perpendicular ao pai tende a ser supostamente mais eficiente que um paralelo. Entretanto, nos testes, tal operação mostrou aumentar mais a performance no geral do que reduzir a eficiência geral da árvore. O aumento de profundidade não foi significativo, uma vez que se o particionador for ruim, este não atingirá o critério mínimo e será descartado.

## 4.3 Operação Merge

Essa operação agrega um nodo interno com outro nodo que tenha uma população menor que um determinado limiar. É responsável por remover particionadores em regiões que se tornaram praticamente vazias devido à migração de AABBs (Figura 4.3).

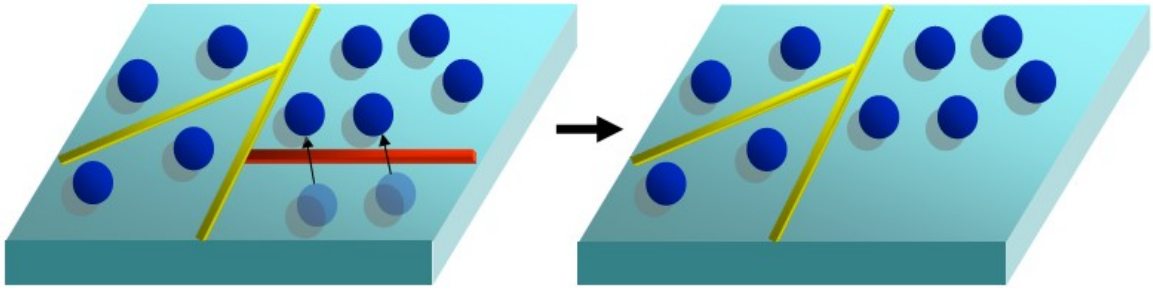


Figura 4.3: Operação *merge*

Todas as listas ordenadas de AABBs são armazenadas nas folhas da sub-árvore do nodo que sofreu a operação de junção, tornando-se apenas uma única lista ordenada, e portanto, uma nova folha, deixando de ser um nodo. As sub-árvores esquerda e direita do nodo são combinadas utilizando uma abordagem *bottom-up* e designadas para o nodo folha que substitui o nodo original (Figura 4.4). Cada lista precisa ser reordenada contra a nova direção apenas uma vez, se for necessário, pois algumas listas podem ter a mesma direção de ordenamento. Uma vez que apenas nodos com baixa população sofrem a operação *merge*, sub-árvores que precisam sofrer essa operação são usualmente pequenas.

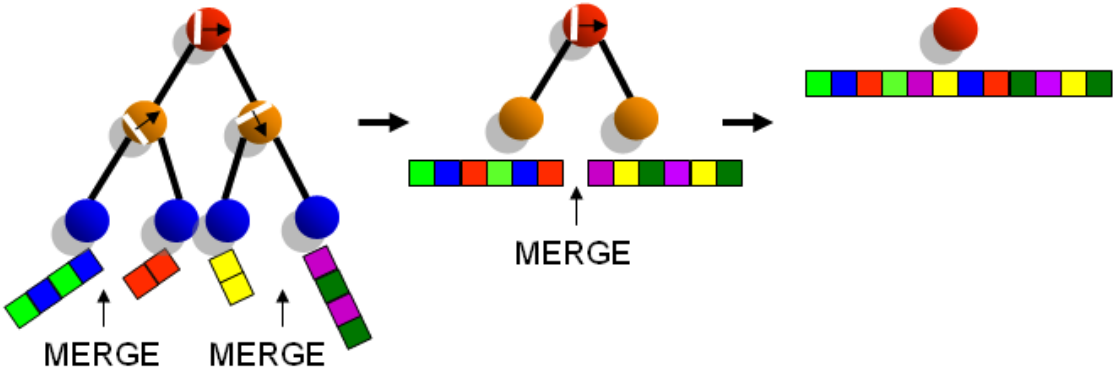



Figura 4.4: Listas ordenadas são processadas através de uma abordagem *bottom-up*.

## 4.4 Operação *Balance*

O operador *balance* foi desenvolvido para reparar nodos que ficam desbalanceados através de modificações mínimas na árvore. É útil quando um nodo tem uma sub-árvore com alta população e o outro está quase vazio. Ao invés de aplicar a operação *merge* no nodo para apagar as sub-árvores, é feita uma tentativa de restabelecer o balanceamento através de um deslocamento do plano particionador. Como não é necessário armazenar a distribuição das AABBs entre suas sub-árvores, é usado o procedimento descrito na operação *merge* para recuperar a informação. Se o re-posicionamento do particionador satisfaz os critérios estipulados para redundância e balanceamento, a operação de balanceamento é aplicada para esse nodo com sucesso.

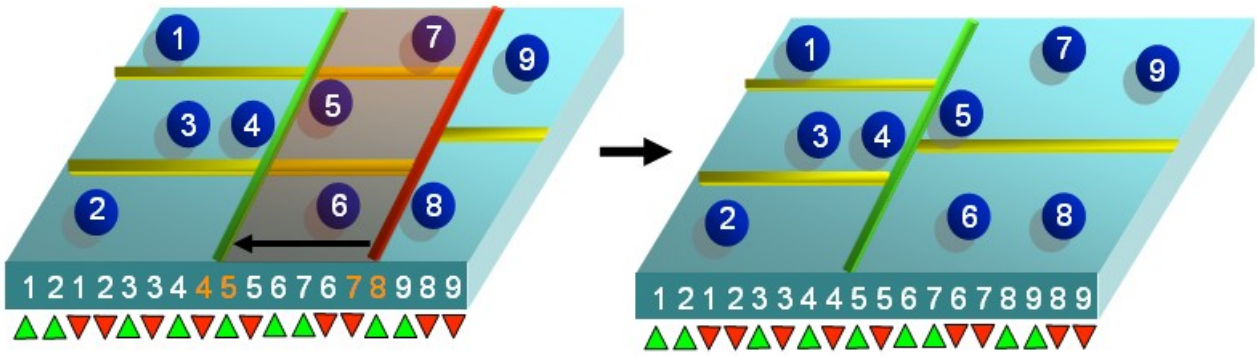



Figura 4.5: Região intervalar (vermelho) entre as posições anterior e atual do particionador. Listas ordenadas são recriadas e usadas para reposicionar o particionador.

Modificar o particionador pode requerer que algumas AABBs migrem entre sub-árvores. Na Figura 4.5, por exemplo, apenas AABBs que pertençam à região intervalar precisam ser consideradas. É criada uma lista ordenada temporária para determinar a nova posição do particionador.

## 4.5 Operação *Swap*

Existem situações em que não é possível balancear um nodo. Tal fato ocorre quando muitas AABBs estão parcialmente alinhadas com o plano separador. Assim, na projeção dos intervalos, existem muitos intervalos que ficam se sobrepondo. Quebrar nessa zona intervalar causaria um excesso de redundância, o que não é vantagem. Portanto, o particionador usado torna-se ineficiente, tendo que ser eliminado.

Para essas situações em que a operação *balance* falha ao re-estabelecer o balanceamento, existe outra operação que pode ser usada ao invés de usar a operação *merge*. A Figura 4.6 ilustra essa operação.

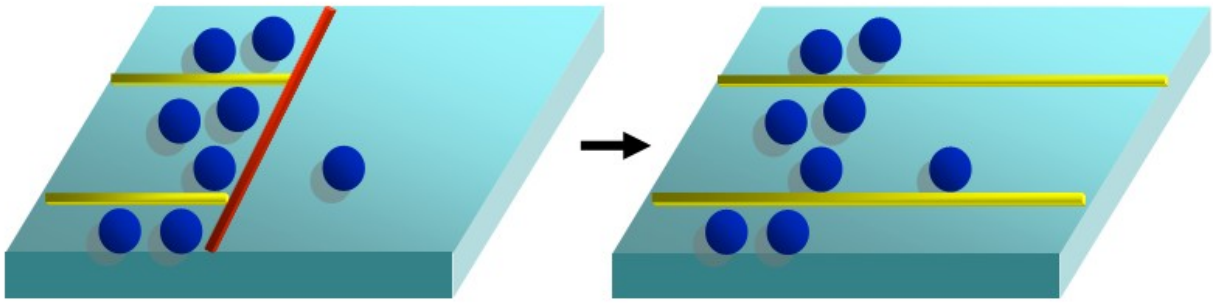


Figura 4.6: O particionador em vermelho é recolocado pela sub-árvore de maior população. Objetos que apenas existem na sub-árvore de menor população precisam migrar para a sub-árvore de maior população.

A operação *swap* apaga o nodo desbalanceado ou com redundância em excesso, e o substitui pela raiz da sub-árvore de maior população. Essa operação é fundamental para a manutenção de uma boa árvore, pois elimina a degradação que poderia ser causada por um particionador ineficiente. Se essa operação não existisse, a árvore poderia aumentar de profundidade a medida que a operação de balanceamento não conseguisse consertar particionadores intermediários, levando à criação de novos particionadores para solucionar o problema e manter sob controle a população nas folhas. Assim, a profundidade da árvore tenderia a crescer, levando a árvore a uma constante degradação, que tornaria necessária uma reconstrução.

A única mudança estrutural requerida para essa operação é a inserção das AABBs unicamente contidas na sub-árvore de menor população. Entretanto, isso é rápido para ambos os casos, uma vez que no caso de desbalanceamento há poucas AABBs e caso haja redundância alta existem muitas AABBs que são compartilhadas.

## 4.6 Efeitos colaterais das operações

Operações que afetam os nodos, como as operações *balance* e *swap*, podem modificar o formato das folhas. Essas alterações levam as AABBs a migrarem de uma folha a outra. Supondo que em um determinado nodo foi constatada a necessidade de aplicar uma operação para melhorar a qualidade do particionador, após o nodo ser afetado por essa operação de *balance* ou *swap*, as operações previamente detectadas em ambas sub-árvores podem se tornar inválidas.

A Figura 4.7 demonstra claramente que a necessidade da operação *split* desaparece após a aplicação da operação *swap*.

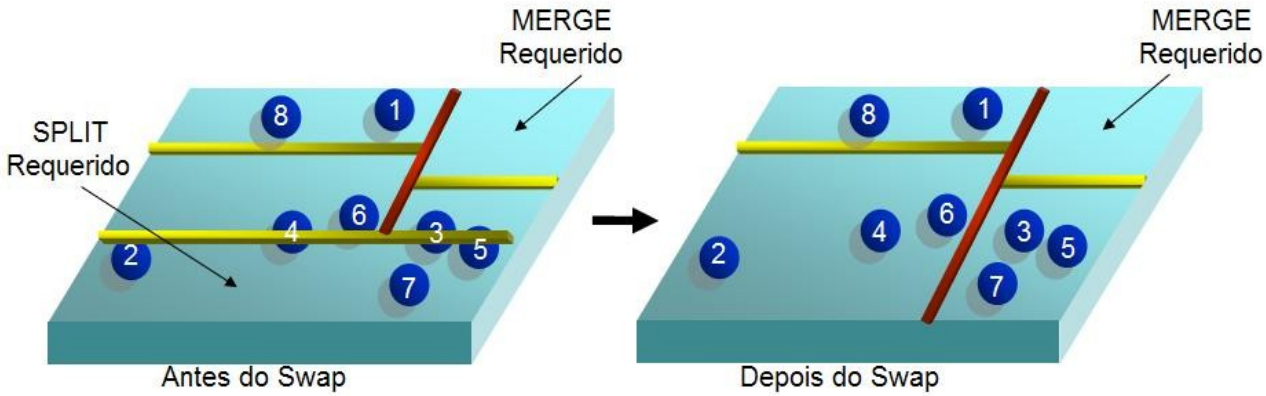



Figura 4.7: Efeitos colaterais das operações

É importante reparar que tal fato influência a estratégia de aplicação das operações na árvore. Como o efeito colateral é propagado de um determinado nodo para ambas as sub-árvores que pertencem a ele, não é plausível utilizar uma estratégia *bottom-up* para a detecção e aplicação de operações. Como pode ser visto na Figura 4.7, se tivéssemos aplicado a operação *split* antes da operação de *swap*, provavelmente teríamos perdido tempo, pois, após a operação de *swap*, foi desfeita a necessidade do *split*. Uma estratégia *bottom-up* levaria a aplicação de operações que poderiam não ser necessárias

caso os nodos com menos profundidade fossem reparados. Além disso, aumentaria a profundidade da árvore, pois a avaliação dos nodos próximos a raiz demoraria. Se o particionamento próximo à raiz é ineficiente, necessariamente os níveis mais profundos devem ajustar a árvore.

# 5 ALGORITMO DA ÁRVORE BSP SEMI-AJUSTÁVEL

A atualização estrutural da árvore BSP pode ser feita através de duas maneiras distintas. A primeira maneira, conhecida como método conservador, é fazer todas as operações assim que elas são detectadas. A cada momento que as AABBs são atualizadas na árvore, é feita uma procura por degradações estruturais e as devidas operações são aplicadas para consertá-la. Essa abordagem minimiza os pares de colisão e torna a árvore o mais próxima da ótima, segundo certos critérios de qualidade. Por outro lado, o desgaste em termos de performance é grande, além de acarretar os efeitos colaterais das operações como discutido no capítulo anterior.

Portanto, opta-se por uma segunda estratégia. A atualização estrutural da árvore BSP não é realizada na mesma proporção que as atualizações das AABBs. Mas, deixar de realizar manutenção estrutural pode levar a um aumento no número de pares de colisão e uma estrutura ineficiente em termos de quebra espacial. Assim, é definida uma atualização em intervalos regulares para alcançar um compromisso entre atualização, tamanho da árvore e pares de colisão gerados.

Neste capítulo são apresentadas estratégias de atualização. Primeiramente, é descrito como são feitas as atualizações das posições das AABBs na árvore. Após, é descrito como a árvore se atualiza para refletir a nova configuração das AABBs e como esta atualização é feita para ter o custo minimizado.

## 5.1 Atualização do posicionamento das AABBs

Diferentemente de outros métodos descritos na literatura, a árvore binária de particionamento não usa uma abordagem baseada em inserção e remoção. Essa abordagem além de ineficiente tende a degradar severamente a performance quando a profundidade da árvore é grande. Além disso, não basta determinar em que folha a AABB deve pertencer, mas atualizar também as listas ordenadas armazenadas nas folhas. Portanto, existem dois passos fundamentais na atualização da árvore: atualização sobre a hierarquia e atualização das listas ordenadas.

### 5.1.1 Atualização sobre a hierarquia

As posições das AABBs são atualizadas para cada etapa da simulação na árvore binária de particionamento por uma abordagem *top-down*. A atualização segue duas

etapas importantes. A primeiro etapa é a atualização das AABBs que migram entre folhas, causando remoção e inserção nas listas ordenadas. Todos as AABBs são verificadas na árvore para ver se continuam na mesma folha ou migraram entre folhas. Por conseguinte, para uma dada AABB, o caminho dela usando a posição nova e velha são percorridos e as diferenças nestes caminhos refletem a migração entre folhas. Operações de inserção e remoção devem ser feitas nas listas ordenadas para refletir essa migração entre folhas.

A Figura 5.1 mostra o processo de atualização para uma AABB verde que esta migrando entre folhas.

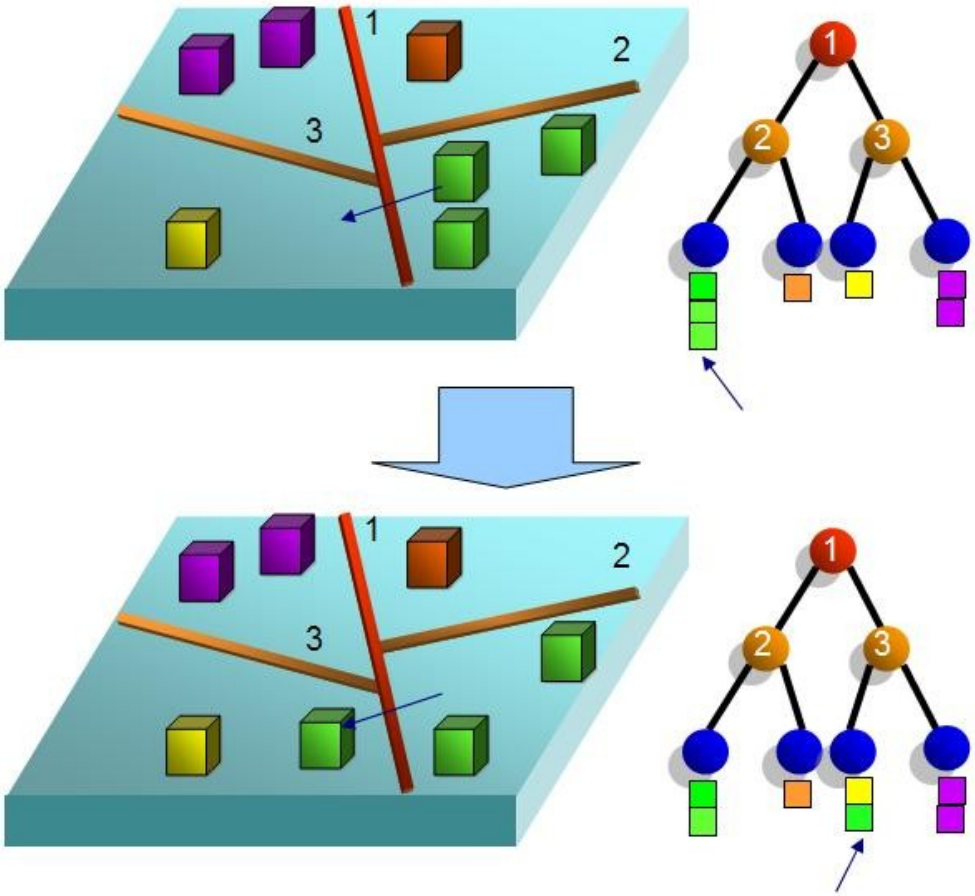



Figura 5.1: Atualização devido a migração de AABBs

### 5.1.2 Atualização das listas ordenadas

No segundo passo, as listas ordenadas de AABBs armazenadas nos nodos folhas são atualizadas para cada movimentação das AABBs com o novo valor de projeção, seguido pelo passo de ordenamento que apenas deve requerer algumas trocas de posição ou provavelmente nenhuma. Usualmente, tal ordenamento é feito em tempo linear uma vez que o número de troca de posição na lista ordenada tende a um valor baixo. As listas ordenadas são pequenas, se compararmos com métodos tipo s*weep-and-prune,* uma vez que apenas as AABBs dentro de uma folha são ordenadas.

Portanto, mesmo que haja deslocamento de intervalos, o número de operações de troca de posicionamento tende a ser muito menor que nas listas ordenadas dos eixos coordenados, uma vez que essas listas são grandes e não necessariamente refletem a distribuição das AABBs (muitos intervalos podem estar se interseccionando) e o número de trocas pode ser grande. Assim, pode-se afirmar que manter ordenadas pequenas listas contendo poucas AABBs é menos dispendioso de que uma lista única com todas as AABBs, uma vez que a distância média entre a posição velha na lista em relação a posição nova é minimizada.

A Figura 5.2 mostra o procedimento de atualização na folha de uma árvore. Após a movimentação da AABB amarela, algumas trocas nas listas ordenadas devem ser feitas para refletir a sua nova posição.

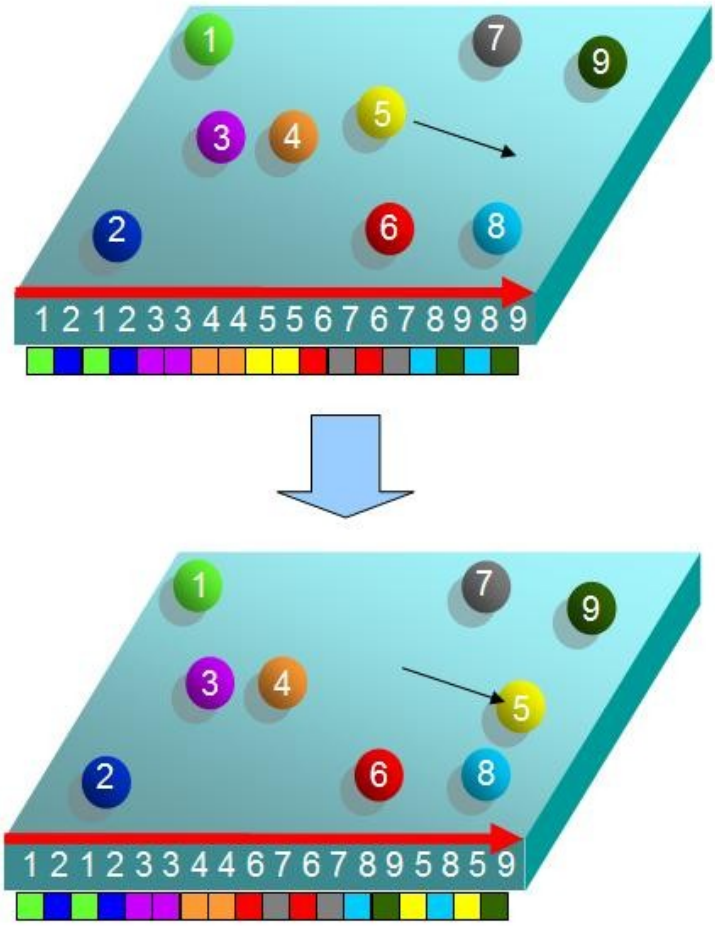



Figura 5.2: Atualização dentro das folhas

Nos testes feitos, a AABB forneceu a melhor performance. Entre as alternativas de usar listas encadeadas, tabelas *hash* ou arranjo, a estrutura de arranjo foi a que forneceu melhores resultados e, portanto, foi usada para armazenar as listas ordenadas nas folhas. Listas encadeadas, apesar de mais fáceis de aplicar em operações de inserção e remoção, são mais custosas para percorrer e atualizar.

## 5.2 Atualização estrutural da árvore BSP

A atualização do posicionamento das AABBs causa mudança na distribuição destas sobre as folhas da árvore. Algumas folhas podem ficar concentradas, enquanto outras podem se tornar vazias. Particionadores criados podem ser ineficientes para tratar a nova situação. Esta seção descreve como as operações sobre a árvore devem ser aplicadas a fim de evitar uma degradação estrutural.

### 5.2.1 Prioridade das operações

Ordenar a aplicação de operadores é muito importante, uma vez que existe diferença de prioridade entre as operações. O operador *balance* ajusta os particionadores para melhorar o balanceamento, mas tem efeitos colaterais na distribuição das AABBs na árvore, como foi discutido no item 4.6. Tal fato pode afetar outras operações, em alguns casos tornando-as desnecessárias (Figura 5.3). Como a operação *balance* tem a função de manter os particionadores com a melhor qualidade, é interessante executá-la mais freqüentemente que as outras operações para ter uma árvore adequada. Tal fato garante que os particionadores na árvore BSP fiquem em boas condições (balanceados), não necessitando criar novos particionadores para remediar a ineficiência de particionadores já existentes.

Além disso, essa operação é responsável por detectar quando operações *swap* devem ser feitas para eliminar particionadores inadequados. Para uma boa manutenção da árvore, a operação de *balance* é feita antes de qualquer outra operação.

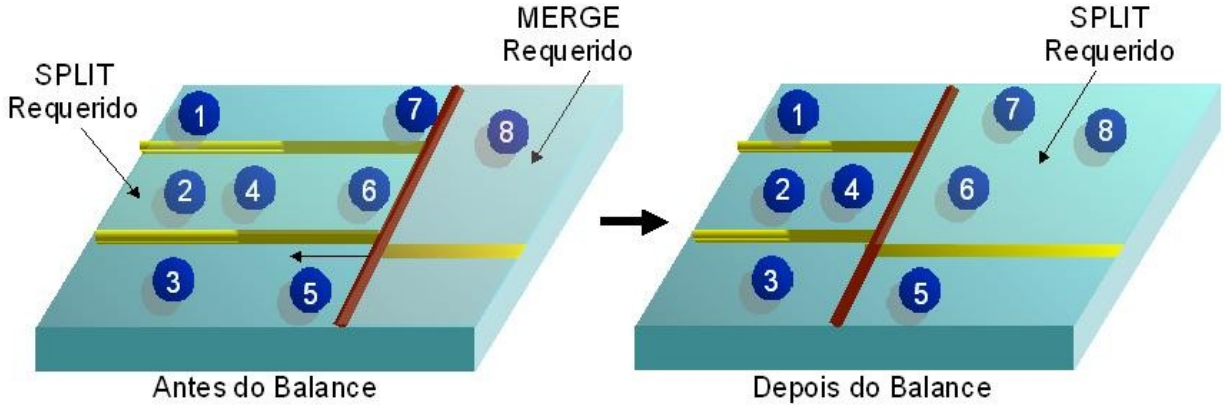



Figura 5.3: O impacto da operação *balance* (particionador vermelho) sobre as operações *split* e *merge*.

Em adição, a operação *swap* pode causar efeitos colaterais em suas respectivas sub-árvores. Portanto, os efeitos colaterais das operações levam a uma abordagem *top-down* na hora de verificar a árvore, como foi discutido no item 4.6. Quando é detectada uma operação para ser feita em um determinado nodo, suas respectivas sub-árvores são desconsideradas e se continua a procura por operações nas demais partes da árvore. Isso evita que operações inúteis sejam agendadas.

A operação *split* é crítica, uma vez que divide uma grande lista ordenada de AABBs em listas menores, reduzindo o custo de atualização da lista ordenada. Portanto, a primeira prioridade é manter a concentração de AABBs nas folhas abaixo de um

determinado limiar que é dinamicamente ajustável de acordo com o número de AABBs que existem na cena. A operação *shift-split* é sempre testada antes de uma operação de *split* e apenas se a operação *shift-split* falha, a operação *split* é realizada.

A operação *swap* é agendada quando o balanceamento falhou em ajustar um particionador ruim. Essa operação pode gerar efeitos colaterais como a operação de *balance* e, normalmente, um número pequeno de AABBs sofre migração. Essa operação lida com migração de AABBs entre sub-árvores, sendo menos custosa que a operação de *split*. Entretanto, esta operação não é considerada tão crítica como a operação *split* pelo fato de ser mais crítico reduzir o tamanho das listas ordenadas para um valor adequado do que remover uma sub-árvore inútil. Essa operação é realizada periodicamente para remover particionadores ruins ao longo da árvore.

A operação *merge* elimina as folhas com um número abaixo de um determinado limiar. A operação não é crítica porque listas pequenas são simples de manter e o custo adicional para manter nodos e folhas quase vazios é mínimo. A operação é efetuada apenas se a simulação dispõe de tempo livre em seus ciclos de atualização.

Para resumir, as operações são ordenadas da seguinte forma: *balance*, *shift-split, split*, *swap*, *merge*.

### 5.2.2 Sistema de agendamento das operações

As operações precisam ser aplicadas na árvore para evitar degradação estrutural. Como já foi mencionado no início do capítulo, a atualização conservadora, a qual aplica todas as operações necessárias, é custosa demais. É preciso aplicar as operações que sejam críticas, aquelas que afetam uma maior população, e, ao mesmo tempo, conservam a qualidade da árvore para minimizar o número de pares de colisão gerados e o tempo de atualização das listas ordenadas.

O método semi-ajustável distribui essas atualizações à medida que as operações são detectadas. Tal método alivia o custo de atualização estrutural, uma vez que a cada ciclo é aplicado um conjunto de operações pendentes. Como conseqüência, a estrutura tenderá a se reparar periodicamente para manter a qualidade da árvore. O número de operações por ciclo determina a qualidade da árvore que será mantida em função do dinamismo do ambiente. Neste caso, se o ambiente for caótico, a árvore manterá boa qualidade se aplicar um grande número de operações por ciclo. Caso contrário, ficará atrasando demais as operações e estará muito defasada em relação à configuração das AABBs.

Além da etapa de aplicação das operações, é necessária uma etapa que verifique as operações que devem ser aplicadas na árvore. O método *top-down* usado para percorrer a árvore à procura de operações evita efeitos colaterais de um operador em outro, caso ignore as sub-árvores de um nodo no qual foi constatada a necessidade de aplicar uma operação. Essa etapa é apenas aplicada quando todas as operações exceto *merge* forem aplicadas. Alem disso, há um intervalo de passos mínimo entre verificações da árvore que evita que a árvore seja verificada a toda hora, caso haja um cenário estável. Esse cenário estável seria caracterizado pela movimentação praticamente nula das AABBs, não requerendo operações para reparar a árvore.

Para resumir, a etapa de verificação da árvore é realizada segundo o diagrama mostrado na Figura 5.4. Note que a operação *balance* é a única operação que é aplicada

na fase de verificação da árvore. As demais operações são aplicadas na etapa de aplicação das operações pendentes como mostrado na Figura 5.5.

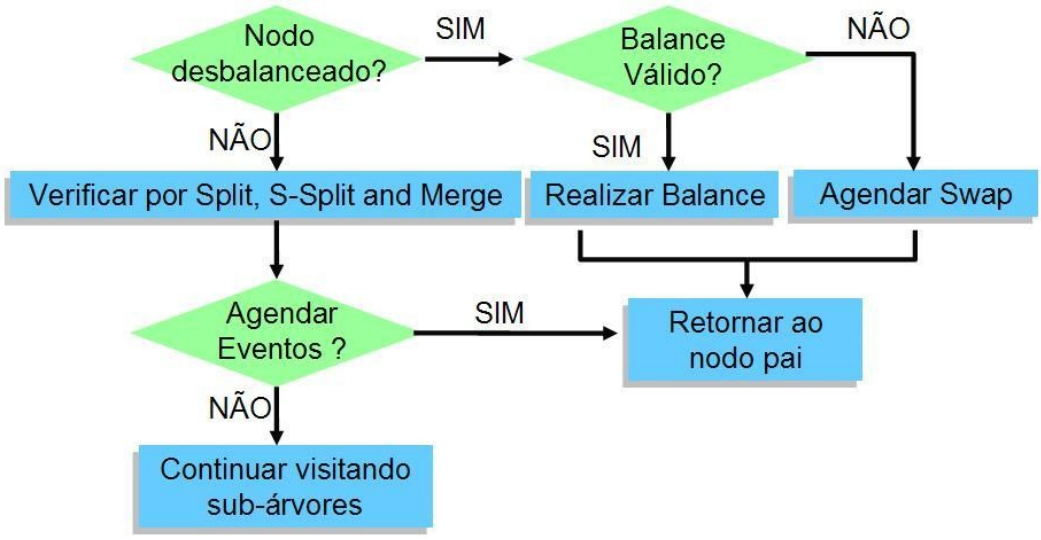



Figura 5.4: Sistema de verificação da árvore

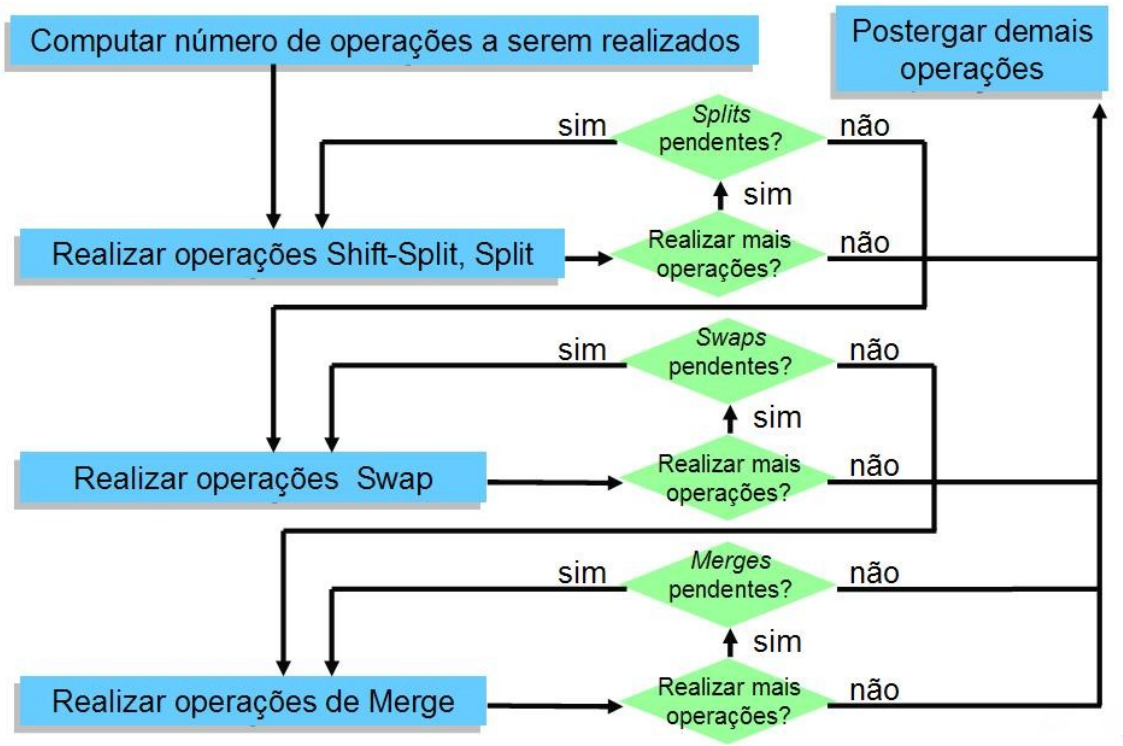



Figura 5.5: Aplicação de operações por ciclo

# 6 RESULTADOS E ANÁLISE

## 6.1 Escolhas de implementação

Foram feitas quatro simulações diferentes contendo até 4250 AABBs se movimentando para testar o algoritmo da árvore BSP semi-ajustável (Figuras 6.1, 6.3, 6.5, 6.7 respectivamente). O sistema de agendamento descrito no capítulo 5 e foi parametrizado da seguinte maneira:

- Tamanho do subconjunto de AABBs testadas para escolher um novo particionador na operação *split* foi de 2 * $\sqrt{n}$, onde $n$ é o número de AABBs no nodo folha.
- O nodo $p$ é considerado desbalanceado se *balanceamento(p)* < 0.5 (uma sub-árvore com mais que o dobro de AABBs que a outra)
- Máxima *redundância* permitida foi de 2.5% no aumento de número de AABBs.
- Limiar de *split* e *merge (t)*: os melhores resultados obtidos foram com $t$ de 16 a 128. Um limiar variável foi usado para todos os nodos dependendo do número de AABBs a serem particionados.
- Número de operadores processados para cada atualização da árvore *(s)*: maior tamanho de $s$ da lista de eventos foi de 64, e os melhores resultados obtidos foram ao processar $s$ / 16, levando a processar de 1 a 4 eventos por passo.
- Freqüência do agendamento das atualizações: o algoritmo de agendamento é configurado para fazer agendamento no mínimo a cada 10 passos de atualização, mas pode ser postergado se as filas de eventos *split* e *swap* não estiverem vazias.
- Freqüência para aplicar uma das operações: a cada passo se as listas de eventos não estiverem vazias.

Para avaliação do método proposto, foram utilizadas quatro outras estratégias: dois métodos implementados dentro do pacote de física Open Dynamics Engine

(ODE)(SMITH, 2004), **quadtree (QT)** e **spatial hash (SH)**, e dois algoritmos padrões, **loose octrees (LO)** e **sweep-and-prune(SP)**. A detecção de colisão exata e seu tratamento foram implementados utilizando o ODE. Uma vez que os algoritmos BSP, LO e SP são implementados fora do ODE, estes tem um *overhead* adicional para extrair as informações do ODE (tal como AABBs das malhas). Todos os testes foram realizados num Athlon XP 1900 PC com uma GeForce FX 5600 128Mb e 512Mb de memória DDR.

O algoritmo QT cria uma árvore *quadtree* estática no plano XZ. Cada quadrante representa um prisma ao longo da direção do eixo Y. Essa abordagem é extremamente custosa em termos de memória (uma *quadtree* de nível 11 foi o suficiente para indexar todos as AABBs sem redundância). O algoritmo SH usa uma tabela *hash* multi-nível que é reconstruída a cada passo da simulação. SP foi implementado usando uma lista ordenada para cada eixo de coordenadas com atualizações incrementais através de operações de troca. Uma matriz bidimensional foi utilizada para a fase de *prunning*. LO é baseado na descrição de THATCHER (2000), adaptado para utilizar AABBs ao invés de envelopes esféricos de colisão e parametrizado para reduzir o fator *looseness*.

Simulações foram desenvolvidas para avaliar o algoritmo em cenas com diferentes aspectos tal como tipo de AABBs, densidade, forma, velocidade e aceleração, sempre preservando a coerência espacial. Cada simulação foi executada por 4 minutos ao longo de 6000 passos (25 passos por segundo). Foram coletados os seguintes dados: número de frames por segundo (FPS), número de pares de colisão, tempo de colisão da *broad-phase* e tempo de *overhead* estrutural (atualização de AABBs e estrutura de dados) (Figuras 12-13). A Simulação 1 e 4 usaram apenas os eixos X e Z para SP, porque AABBs são alinhados com o eixo Y em ambas simulações.

## 6.2 Resultados da simulação

### 6.2.1 Simulação 1

A Simulação 1 (Figura 6.1) consiste em esferas e blocos caindo sobre um terreno. Tal simulação permite avaliar cenas com AABBs estáticas e dinâmicas onde a movimentação delas é dada pelas colisões que estas fazem no terreno. Assim, um grande número de pares de colisão são gerados entre as AABBs e o terreno, e isto predomina por quase toda a simulação. Também é analisado como cada algoritmo é afetado pelo aumento de número de AABBs e o posterior agrupamento dessas AABBs em vales do terreno. Assim, em termos de pares gerados, esse cenário extenua os métodos, uma vez que não é fácil reduzir o número de pares de colisão pela concentração de AABBs que ocorre.

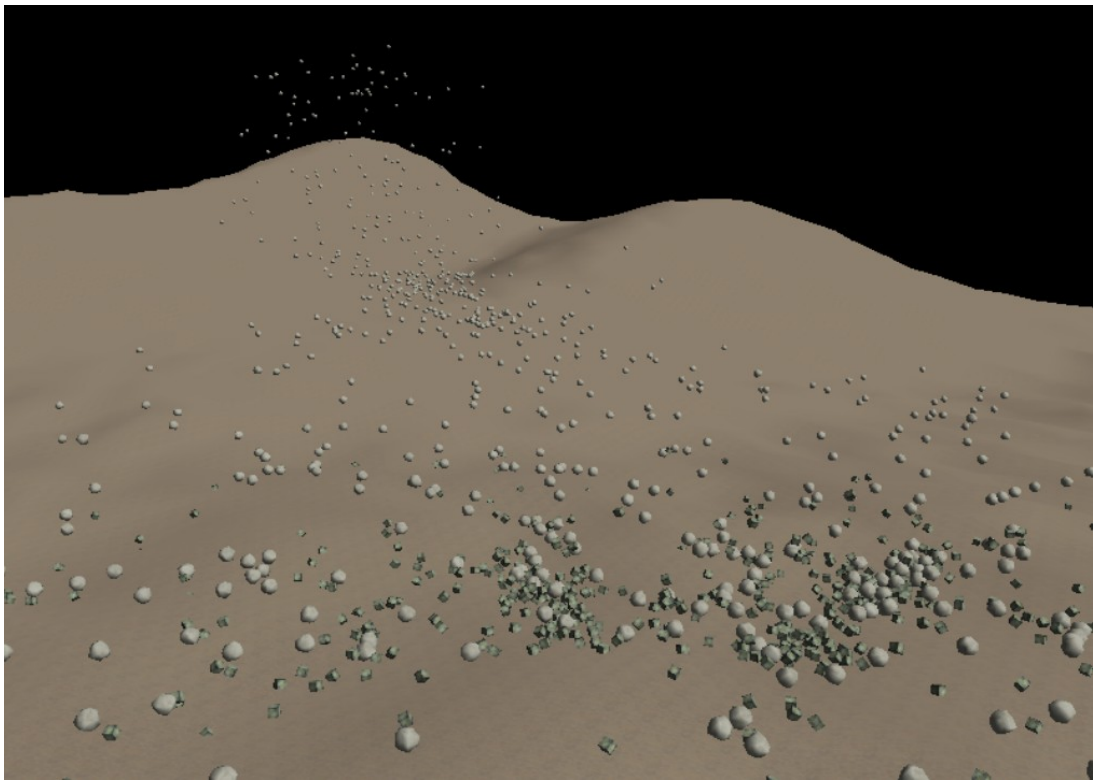
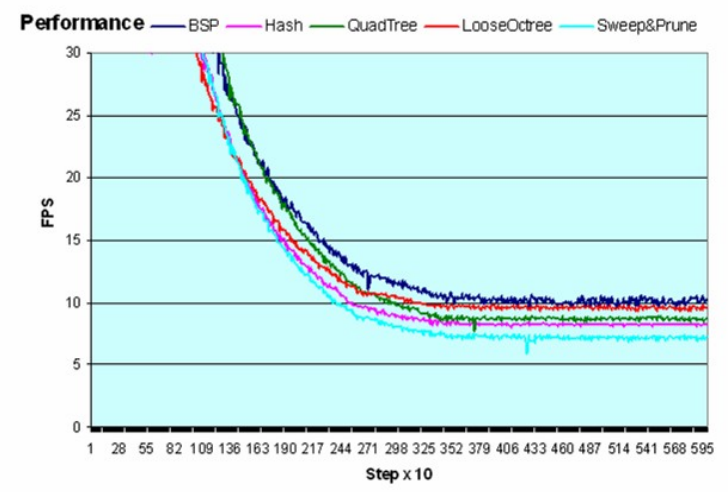
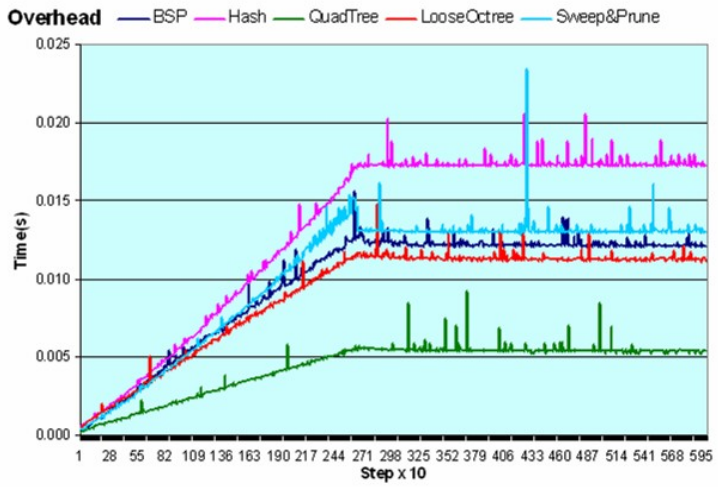


Figura 6.1: *Screenshot* da Simulação 1

A Figura 6.2 expressa o comportamento dos métodos. BSP e LO lideram em termos de performance, mas LO gera mais pares de colisão pelo fato da restrição estrutural imposta pela *octree*. Além disso, BSP possui uma fase de *sweep* que é feita nas folhas, algo que não ocorre no método LO. SP e SH geram menos pares de colisão entre todos, entretanto possuem um custo mais alto para gerá-los. Em SH, este custo é causado pelas colisões na tabela *hash* que fazem com o que o método degrade significativamente e que pode ser percebido quando AABBs tendem a se aglomerar. No caso da SP, o custo é grande devido ao tamanho das listas ordenadas e a posterior verificação na matriz de colisão. BSP tem a melhor performance entre todos, pois reduz os pares de colisão mais eficientemente. O *overhead* estrutural da BSP é pior que o do QT pelo fato de ter que atualizar as listas ordenadas, mas é similar ao SP e LO e melhor que o SH.


Performance
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
FPS
30
25
20
15
10
5
0
1 28 55 82 109 136 163 190 217 244 271 298 325 352 379 406 433 460 487 514 541 568 595
Step x 10


Overhead
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
Time(s)
0.025
0.020
0.015
0.010
0.005
0.000
1 28 55 82 109 136 163 190 217 244 271 298 325 352 379 406 433 460 487 514 541 568 595
Step x 10

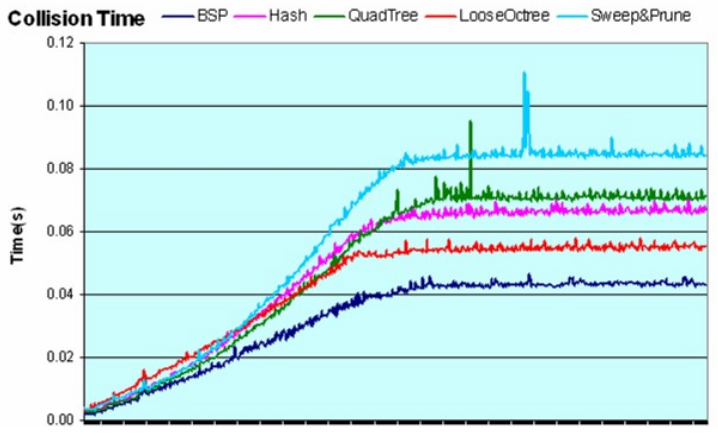

Collision Time
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
Time(s)
0.12
0.10
0.08
0.06
0.04
0.02
0.00


Figura 6.2: Resultados da Simulação 1

### 6.2.2 Simulação 2

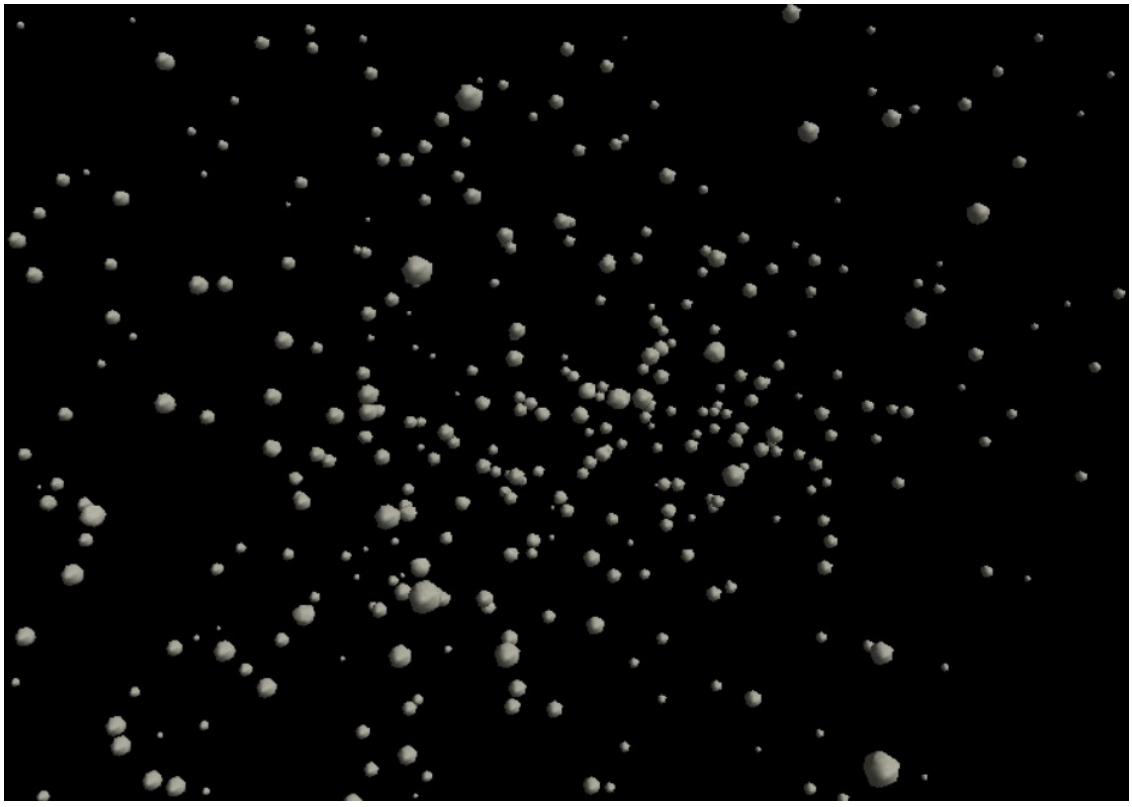
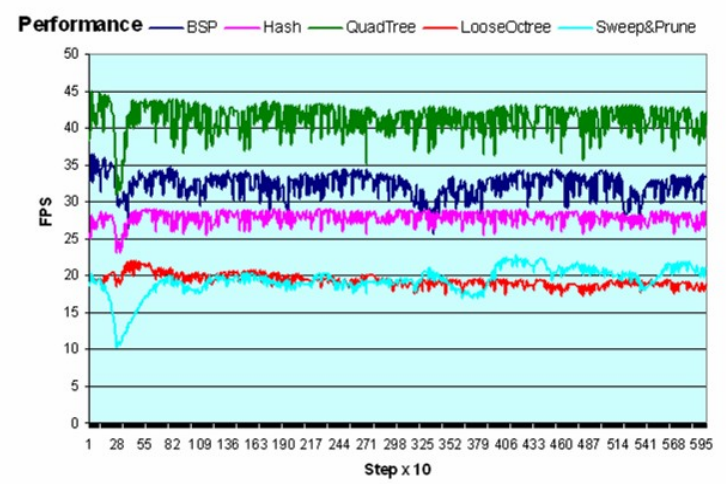


Figura 6.3: *Screenshot* da Simulação 2

A Simulação 2 (Figura 6.3) mostra um cena caótica com vários atratores de AABBs que estão se movimentando. Essa cena serve para avaliar a colisão de nuvens de AABBs com diferentes condutas e como as estruturas se comportam para se adaptar a novas situações. Esse método extenua muito a parte de atualização dos objetos,

enquanto não gera muitos pares de colisão pelo fato do ambiente ser a maior parte do tempo esparso.

A Figura 6.4 apresenta o desempenho dos métodos. A performance da BSP é razoável mesmo havendo AABBs que se movimentam rapidamente. Atualização postergada ajuda a reduzir custos, e força a altura da BSP até um determinado limiar para reduzir o número de AABBs que migram entre folhas. QT tem melhor performance uma vez que como é uma árvore completa, é ótimo para ambientes esparsos. QT e BSP mostram resultados similares para o número de pares de colisão gerados e o tempo gasto para gerar esses pares. Ambos os métodos não reduzem os pares de colisão tanto quanto os outros métodos o fazem, mas conseguem gerá-los muito mais rápido que os demais métodos. QT ainda tem o melhor tempo de *overhead* estrutural por ser uma estrutura estática e é seguido pela BSP. Ambos SH e LO mostraram similar tempo de *overhead* estrutural e SP gerou menos pares de colisão do que qualquer outro, mas com um custo de performance maior.



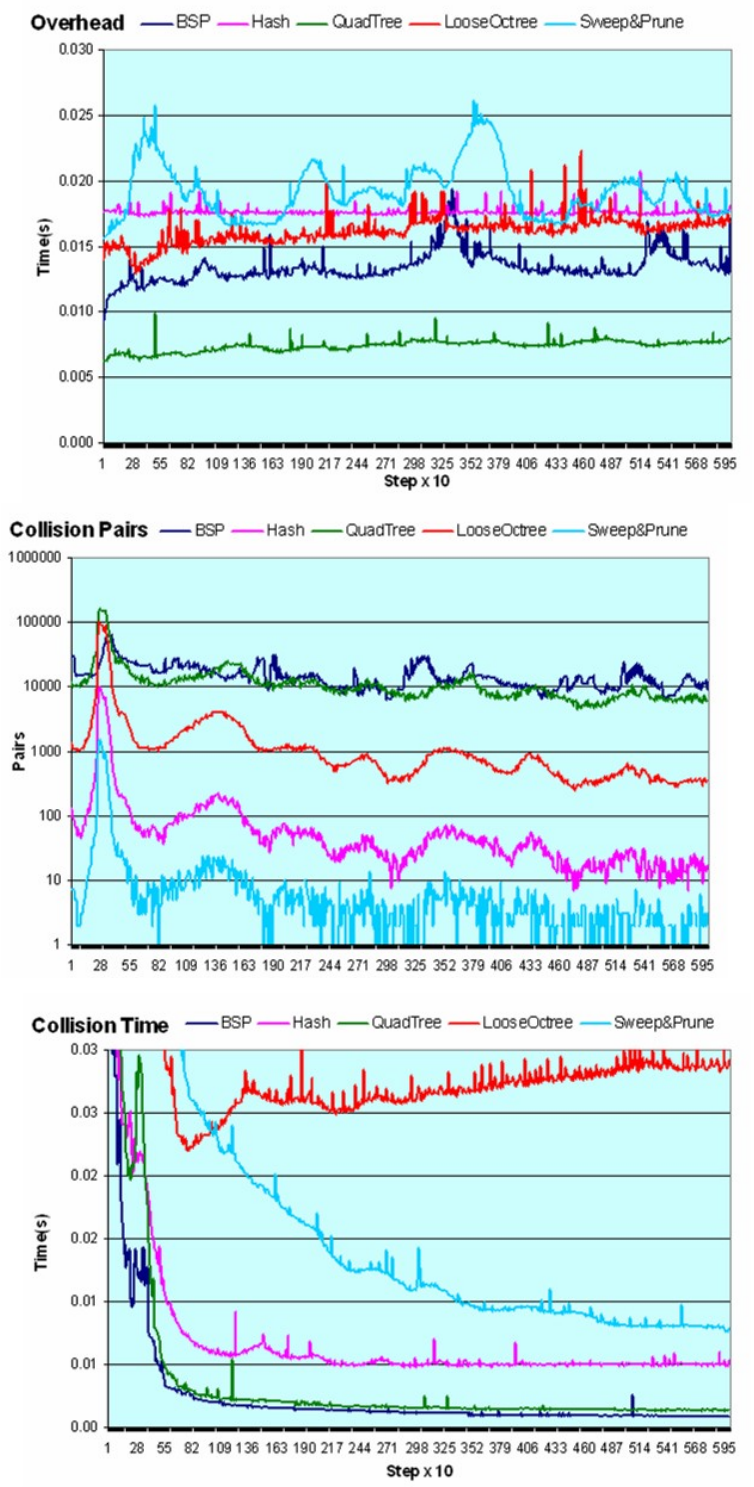



Figura 6.4: Resultados da Simulação 2

### 6.2.3 Simulação 3

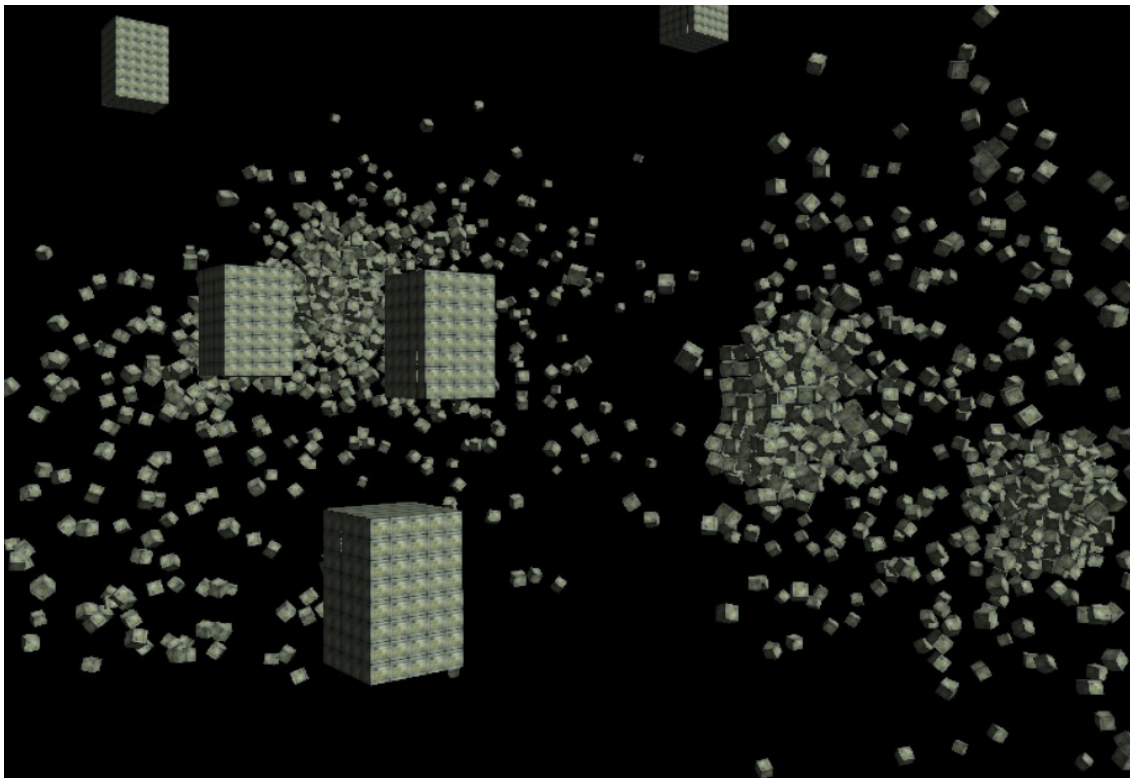
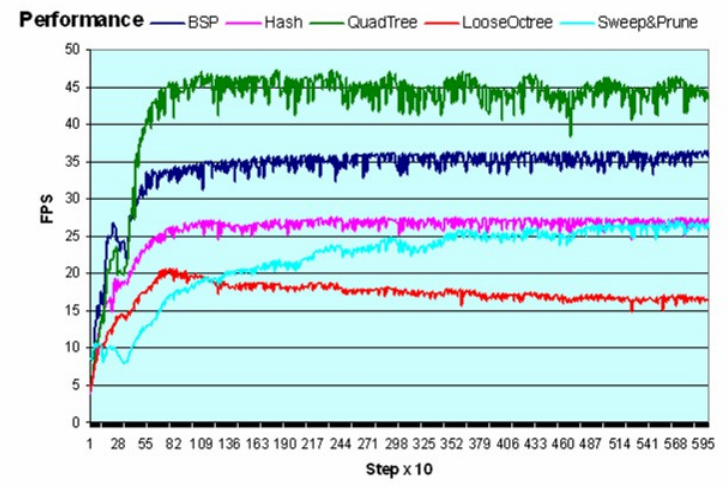
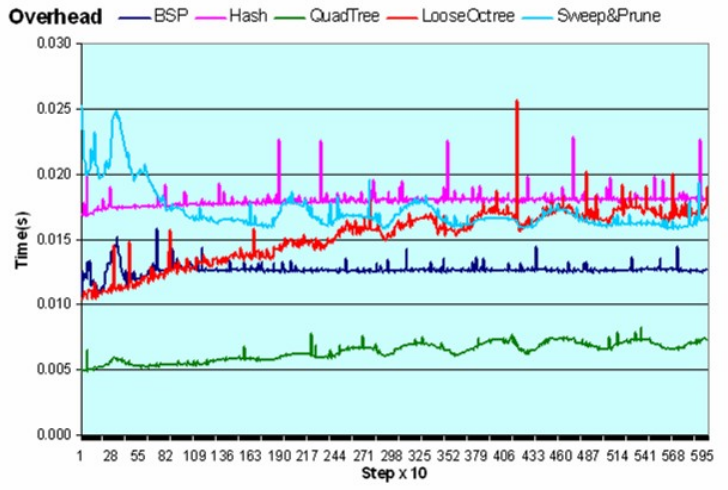


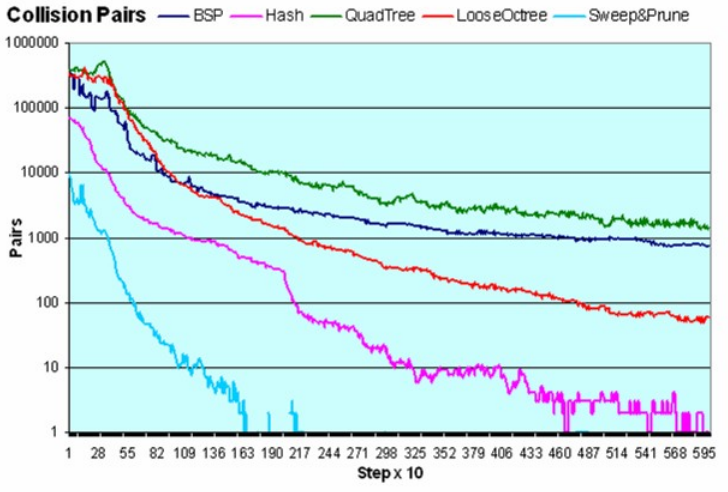
Figura 6.5: *Screenshot* da Simulação 3

A Simulação 3 (Figura 6.5) possui alguns blocos de concreto sendo atraídos por um atrator único. Após a primeira colisão, os blocos se quebram em vários bloquinhos. O objetivo dessa simulação é inicialmente exaurir os métodos com blocos difíceis de serem particionados e com um ambiente que tende a gerar muitos pares de colisão devido a proximidade das AABBs. À medida que os blocos se colidem e se quebram, as AABBs tendem a se espalhar pelo universo e o número de pares de colisão sofre uma queda drástica. O comportamento dos métodos é avaliado para ver como estes lidam com o ambiente concentrado mudando para um ambiente esparso.

A Figuar 6.6 mostra os gráficos de desempenho dos métodos nesta simulação. QT gera mais pares de colisão que qualquer outro método, especialmente antes das AABBs se quebrarem. Entretanto, melhora substancialmente após o primeiro impacto. Em todas as abordagens, o número de pares de colisão decresce após o primeiro impacto e a performance melhora substancialmente para a BSP, SH e SP, mas em maior escala para o QT. Uma vez que a BSP é a única que faz atualizações postergadas, ela precisa mais tempo para ajustar a estrutura quando as AABBs se separam. SH e SP mostraram o menor número de pares de colisão, mas não foram tão rápidos como o QT e BSP, pois tem um alto custo para gerá-los, além do custo de atualização estrutural. BSP tem uma pequena vantagem no número de pares de colisão gerados e no tempo de colisão que o QT, mas possui um maior *overhead* estrutural. LO tem o pior tempo de colisão, pois requer muitos testes entre AABBs e folhas da *octree,* devido a *looseness* da estrutura.


Performance
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
FPS
0
5
10
15
20
25
30
35
40
45
50
1 28 55 82 109 136 163 190 217 244 271 298 325 352 379 406 433 460 487 514 541 568 595
Step x 10


Overhead
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
Time(s)
0.000
0.005
0.010
0.015
0.020
0.025
0.030
1 28 55 82 109 136 163 190 217 244 271 298 325 352 379 406 433 460 487 514 541 568 595
Step x 10


Collision Pairs
BSP
Hash
QuadTree
LooseOctree
Sweep&Prune
Pairs
1
10
100
1000
10000
100000
1000000
1 28 55 82 109 136 163 190 217 244 271 298 325 352 379 406 433 460 487 514 541 568 595
Step x 10

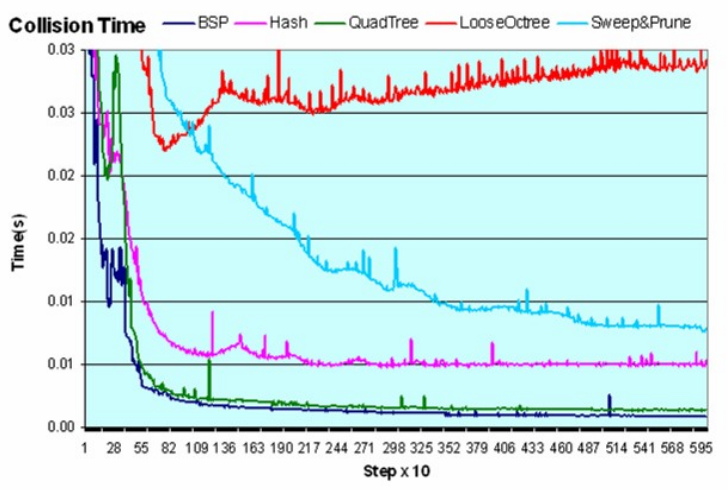



Figura 6.6: Resultados da Simulação 3

### 6.2.4 Simulação 4

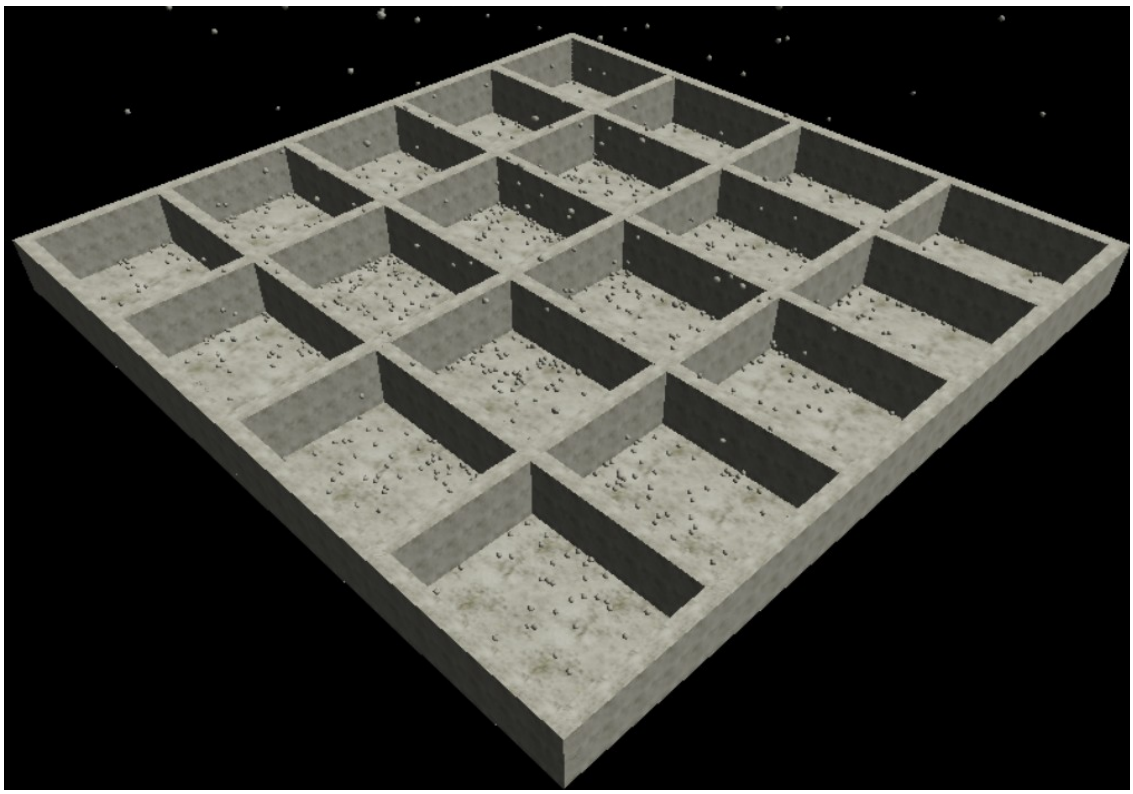
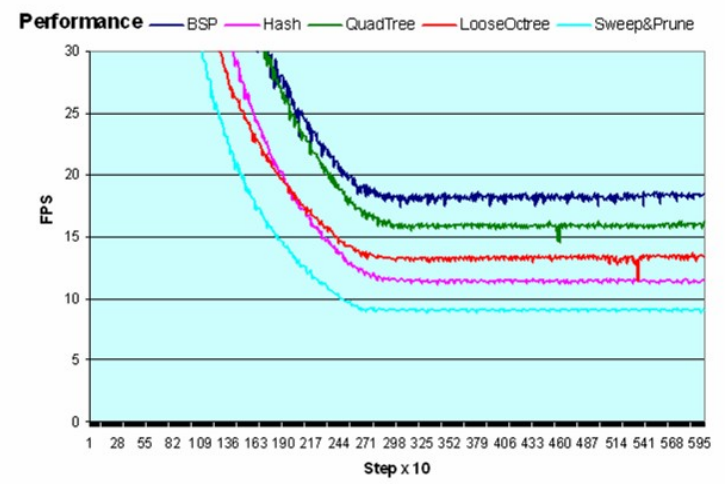


Figura 6.7: *Screenshot* da Simulação 4

Na simulação 4 (Figura 6.7), AABBs caem dentro de vários compartimentos e colidem contra outras AABBs que estão dentro deles. As estruturas estáticas impõem uma separação espacial entre AABBs dinâmicas e AABBs estáticas. As AABBs tendem a se aglomerar dentro desse compartimento causando a geração de muitos pares de colisão. Além disso, a situação não se estabiliza logo, uma vez que AABBs

constantemente ficam caindo nos compartimentos, causando movimentação e colisão das AABBs que lá estão.

A Figura 6.8 mostra a performance dos métodos. A separação espacial pelas AABBs estáticas auxilia na melhora de performance e esse é certamente o melhor cenário para a BSP. Como na simulação 1, o número de objetos dinâmicos é aumentado a cada passo para avaliar como cada um dos algoritmos escala. Os compartimentos estão alinhados com o plano XZ no qual é a melhor situação para o QT. BSP tem o melhor tempo de colisão seguido pelo QT. No início da simulação, a BSP adapta-se o mais rápido possível para reduzir o número de pares de colisão e estabilizar. SP comporta-se pior do que qualquer outro método em termos de tempo de colisão, mas tem um *overhead* similar ao da BSP. LO tem o melhor tempo *overhead* estrutural, uma vez que nodos são criados apenas em uma pequena região em que ocorre a simulação. SH tem o pior tempo de *ovehead* estrutural, pois existe mais de um objeto por célula, ocasionando colisões na tabela *hash*.



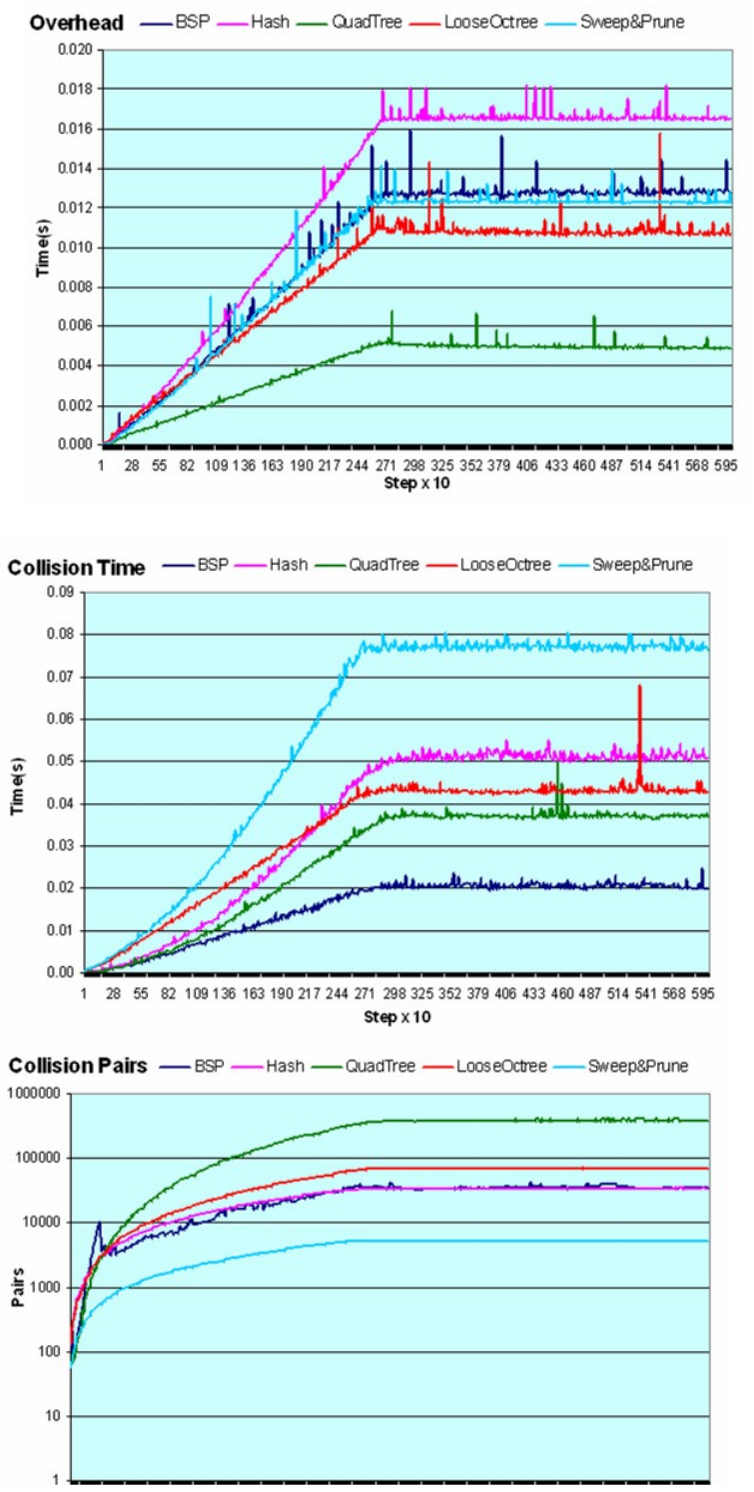
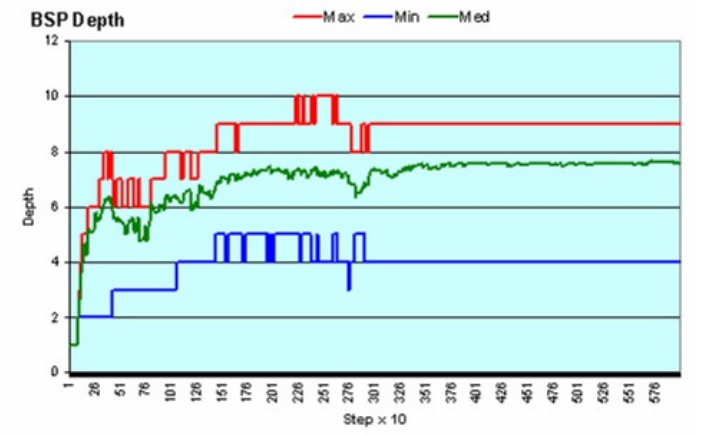



Figura 6.8: Resultados da Simulação 4

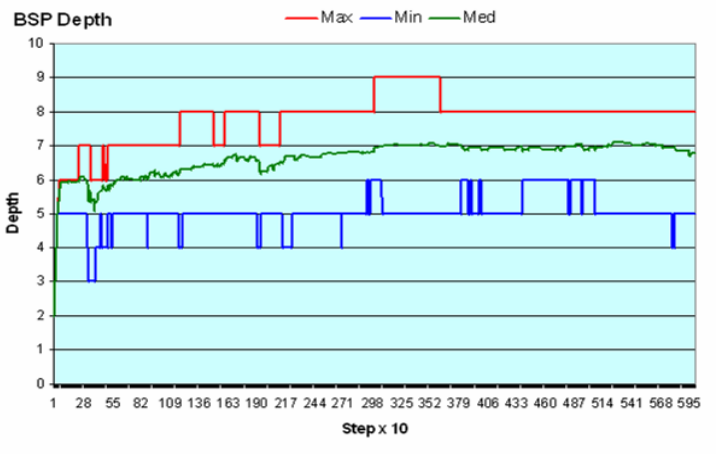
## 6.3 Profundidade da Árvore

Os dados de profundidade da BSP (Figura 6.9) mostram os efeitos das operações. A profundidade média da árvore não sofre dramáticas oscilações levando a um razoável número de folhas, e conseqüentemente, um número estável de pares de colisão e performance. A variação da profundidade no início da simulação é devido à construção da árvore. Podemos reparar que em todas as simulações, a árvore se estabilizou da metade para o final da simulação, não sofrendo grande oscilações na profundidade média. As simulações 1(a),2(b) e 4(d) apresentaram maior variação da profundidade média da árvore no início da simulação. Na simulação 3(c), a profundidade da BSP mudou rapidamente com a quebra dos blocos, mas se estabilizou logo após.

A variação de profundidade média nas simulações reflete a aplicação das operações na árvore para alcançar um valor adequado de população de AABBs nas folhas. Assim, o aumento de profundidade reflete operações de *split* sendo aplicadas. Quando há um decréscimo na profundidade média, como pode ser visto claramente na simulação 1(a), isto é reflexo de aplicação de operações *merge* e *swap*. Essas operações causam um decréscimo na profundidade média através da eliminação de particionadores. A operação de *balance* não afeta a profundidade das folhas, uma vez que não insere ou remove nodos.

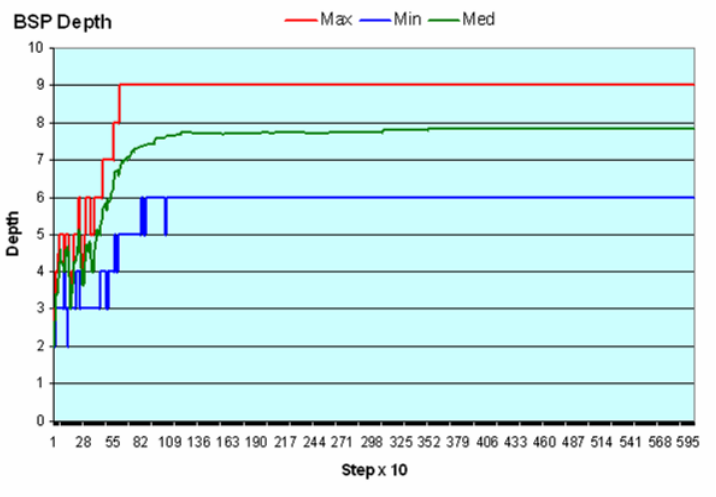



(a)




(b)


(c)




(d)

Figura 6.9 (a)-(d): Comportamento da profundidade da árvore gerada nas diferentes simulações 1 a 4, respectivamente

## 6.4 Análise

Para facilitar a análise dos resultados, a tabela 6.1 faz um sumário da classificação dos métodos aqui apresentados de acordo com os critérios analisados. Os métodos testados estão listados na tabela, em cada célula, de acordo com o seu desempenho na simulação frente ao critério indicado em cada coluna.

Tabela 6.1: Comparação dos resultados

| | ***Performance*** | ***Número de pares de colisão*** | ***Tempo de colisão*** | ***Tempo de atualização estrutural*** |
|---|---|---|---|---|
| Simulação 1 | BSP(1)<br>Loose Octree(2)<br>Quadtree(3)<br>Hash(4)<br>Sweep&Prune(5) | Sweep&Prune(1)<br>Hash(2)<br>BSP(3)<br>Loose Octree(4)<br>Quadtree(5) | BSP(1)<br>Loose Octree(2)<br>Hash(3)<br>Quadtree(4)<br>Sweep&Prune(5) | Quadtree(1)<br>Loose Octree(2)<br>BSP(3)<br>Sweep&Prune(4)<br>Hash(5) |
| Simulação 2 | Quadtree(1)<br>BSP(2)<br>Hash(3)<br>Loose Octree (4)<br>Sweep&Prune (4) | Sweep&Prune(1)<br>Hash(2)<br>Loose Octree(3)<br>BSP (4)<br>Quadtree (4) | Quadtree(1)<br>BSP(2)<br>Hash(3)<br>Sweep&Prune(4)<br>Loose Octree(5) | Quadtree(1)<br>BSP(2)<br>Loose Octree(3)<br>Hash(4)<br>Sweep&Prune(5) |
| Simulação 3 | Quadtree(1)<br>BSP(2)<br>Hash(3)<br>Sweep&Prune(4)<br>Loose Octree(5) | Sweep&Prune(1)<br>Hash(2)<br>Loose Octree(3)<br>BSP(4)<br>Quadtree(5) | BSP(1)<br>Quadtree(2)<br>Hash(3)<br>Sweep&Prune(4)<br>Loose Octree(5) | Quadtree(1)<br>BSP(2)<br>Loose Octree(3)<br>Sweep&Prune(4)<br>Hash(5) |
| Simulação 4 | BSP(1)<br>Quadtree(2)<br>Loose Octree(3)<br>Hash(4)<br>Sweep&Prune(5) | Sweep&Prune(1)<br>BSP(2)<br>Hash(3)<br>Loose Octree(4)<br>Quadtree(5) | BSP(1)<br>Quadtree(2)<br>Loose Octree(3)<br>Hash(4)<br>Sweep&Prune(5) | Quadtree(1)<br>Loose Octree(2)<br>Sweep&Prune(3)<br>BSP(4)<br>Hash(5) |

As Simulações 1 e 4 mostram alguns problemas com as outros métodos como pode ser visto na coluna de performance e tempo de geração dos pares de colisão. Após os compartimentos começarem a ser preenchidos na simulação 4, QT mostra um imenso aumento no número de pares de colisão. Por outro lado, SH tem um aumento no número de colisões na tabela *hash*, ocasionando um aumento no tempo para gerar os pares de colisão. O número de colisões na tabela também causa um *overhead* extra para gerar tais tabelas, degradando significativamente a performance da estrutura. Um problema similar ocorre com SP uma vez que cada eixo gera muitos pares de colisão num ambiente concentrado. Quando o número de intersecções intervalares é grande, o número de trocas de pares de colisão aumenta, assim como as consultas na matriz, custando mais tempo para gerar os pares. A BSP tem maior flexibilidade para se ajustar ao contexto, levando a um número de pares de colisão comparável ao gerado pelo SH, mas de uma maneira muito mais rápida. Tal fato ocorre pois, uma vez que ambos os cenários não são exatamente caóticos, eles tendem a se estabilizar. Assim, mesmo usando postergação, a estrutura tende a convergir para uma situação adequada, tanto em particionamento como em tamanho de listas e eficiência dos *sweeps*. Interessante reparar que LO teve um desempenho razoável em ambas as simulações. Para ambientes concentrados, esse método tem um bom comportamento pelo fato de não ter o mesmo

problema que a *Octree* normal. Assim, mesmo tendo que testar contra mais folhas as AABBs, houve uma significativa redução nos pares de colisão e uma performance adequada.

A simulação 2 mostra o quanto uma árvore estática como a *Quadtree* se beneficia pelo fato de só ter que atualizar a indexação das AABBs dentro dela. O tempo de atualização estrutural é o que mais afetou os demais métodos, fazendo-os ficarem significativamente atrás da *Quadtree*. Entretanto, a BSP mostrou ter dentre todos os métodos dinâmicos, o menor *overhead* estrutural, boa parte devido à postergação das operações. É claro que houve um custo no número de pares de colisão gerados, mas isso não afetou significativamente a performance. Caso a *narrow phase* fosse mais custosa, teria que se utilizar uma aproximação mais agressiva para a BSP, aumentando o número de operações por ciclo. Tal fato tenderia a manter a população das folhas sobre controle, reduzindo o número de pares de colisão. Entretanto, teria como efeito colateral o aumento do *overhead* estrutural.

A simulação 3 apresenta resultados semelhantes a simulação 2. Logo após a ruptura, as AABBs começam a se espalhar pelo universo a uma velocidade significativa. Entretanto, no momento da ruptura que ocorre logo no início da simulação, BSP lidera na performance, apesar de não apresentar um número de pares de colisão tão adequado quanto os demais métodos. A *Quadtree* é o método que apresenta maior ganho de performance após as AABBs se dividirem no espaço, seguido da BSP.

# 7 CONCLUSÃO E TRABALHOS FUTUROS

Nessa dissertação foi proposta uma estrutura semi-ajustável que mostrou resultados significativos para ambientes densos e esparsos com um *overhead* estrutural aceitável e com uma performance razoável.

O principal objetivo dessa abordagem foi estabelecer o compromisso entre o custo de enumerar os pares de colisão e a realização de atualizações estruturais na estrutura de dados espacial sem desconsiderar o número de pares de colisão gerados ao final. Resultados mostraram que a *broad phase* precisa ser extremamente rápida, mesmo que isso leve a um maior número de pares de colisão gerados. Trabalhos anteriormente publicados não priorizaram o fator performance e mostraram resultados relevantes a pares de colisão gerados ou algum valor numérico de comparação que não seja associado ao tempo de processamento. Esses valores não refletem a real capacidade do algoritmo, podendo esconder ou mascarar problemas inerentes a solução proposta. Portanto, nesse trabalho também se considerou a performance final do algoritmo. Além disso, comparou-se com outras abordagens existentes, sendo algumas delas já implementadas no ODE, como é o caso da *Quadtree* e *Spatial Hash.* Foram implementados baseado em bibliografia os métodos *Sweep&Prune* e *Loose Octree.*

A abordagem defendida mostrou melhor performance que o *Spatial Hash*, *Sweep&Prune* e *Loose Octree*, mesmo que em alguns casos gerando um número de pares de colisão próximo a *Quadtree*. Mesmo em relação ao *Sweep&Prune* em duas dimensões, o custo foi similar demonstrando que a manutenção de grandes listas ordenadas de uma certa forma quebra o custo linear das atualizações das listas ordenadas, porque muitas operações de troca podem ocorrer. No caso da *Loose Octree*, a solução apresentada para o problema de haver AABBs em níveis altos da árvore pelo fato de estarem migrando entre células levou a um desgaste pelo número de testes entre AABBs e folhas. Como as folhas se interseccionam, o número de testes de colisão de uma dada AABB não se limita ao seu nodo e sua sub-árvore, podendo necessitar ser realizado com um nodo irmão ou até um com outro mais distante. Isso causa um excessivo número de testes que degrada a performance. Comparado a *Quadtree*, a qual tem requerimento exponencial de memória, os resultados foram melhores quando AABBs estáticas induziam a um claro particionamento ou em ambientes extremamente concentrados que geram muitas colisões. A *Quadtree* é uma árvore estática que pode ser usada para casos 2.5D. Para casos 3D, *Quadtree* está em desvantagem a não ser que as AABBs estejam esparsas pelo universo como mostrado na simulação 2.

Uma questão interessante que deve ser respondida é o quanto e com qual freqüência o espaço deve ser particionado. Para algumas situações, aumentar o particionamento reduz o número de pares candidatos, mas há um maior custo para manter a estrutura devido a um aumento de profundidade. Por outro lado, reduzir o particionamento pode aumentar enormemente o número de pares de colisão, aproximando-se do método que apenas faz sweep em um eixo. A abordagem apresentada permite uma imensa flexibilidade para balancear esses dois argumentos conflitantes. Nesse trabalho apenas foram exploradas algumas possibilidades, tal como fixar a altura da árvore através de um determinado limiar, ou limitar o número de AABBs permitidos dentro de uma folha (o qual é determinado pelas operações de *split* e *merge*). Uma análise da dinamicidade do espaço pode levar a argumentos adaptativos, tornando o método mais agressivo (aumento de número de operações) quando há dinamismo inerente a simulação. Em situações mais estáveis, a árvore pode relaxar mais e reduzir o custo operacional para maximizar a performance.

O *design* das operações *shift split*, *swap* e *balance* permitiram uma melhor exploração da coerência temporal ao reusar informação armazenada na árvore para re-posicionar o particionador, remover particionadores inúteis ou adicionar novos particionadores. O agendamento de tais operações, usando uma abordagem que considera o tempo como um fator crítico, manteve a performance estável. Entretanto, há um vasto campo de soluções ainda em aberto para desenvolver outros algoritmos de agendamento. A *broad-phase* para detecção de colisão é apenas uma aplicação proposta por esse trabalho. Estendê-lo para acelerar outros problemas geométricos representa um outro caminho que pode ser trilhado como trabalho futuro. Aplicações que tratem AABBs deformáveis são também consideradas uma vez que AABBs deformáveis podem ser projetadas da mesma forma que AABBs rígidas.

# REFERÊNCIAS

AGARWAL, P. K.; ERICKSON, J.; GUIBAS, L. J. Kinetic BSPs for intersecting segments and disjoint triangles. In: SYMPOSIUM ON DISCRETE ALGORITHMS, 1998. **Proceedings…** [S.l.:s.n], 1998. p.107–116.

AR, S.; CHAZELLE, B.; TAL, A. Self-customized bsp trees for collision detection. **Computational Geometry: Theory and Applications**, [S.l.:s.n.], v.15, n.1-3, p. 91–102, 2000.

AR, S.; MONTAG, G.; TAL, A. 2002. Deferred, self-organizing bsp trees. **Computer Graphics Forum**, Amsterdam, v.21, n.3, 2002.

BERGEN, G. V. D. **Collision Detection In Interactive 3D Environments**. San Francisco: Morgan Kaufmann, 2004.

CHRYSANTHOU, Y.; SLATER, M. Computing dynamic changes to BSP trees. **Computer Graphics Forum**, Amsterdam, v.11, n.3, p.C321–C332, Sept. 1992.

COHEN, J. D.; LIN, M. C.; MANOCHA, D.; PONAMGI, M. K. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 1995. **Proceedings...** [S.l.:s.n.], 1995. p.189–196, 218.

COMBA, J. **Kinetic Vertical Decomposition Trees**. 2000. Ph.D. Thesis, Stanford University.

DAVE EBERLE, S. H. **Collision detection for deformable objects**. Curso apresentado no SIGGRAPH 2004.

EBERLY, D.H. **3D Game Engine Design**. San Francisco: Morgan Kauffman, 2001.

GOTTSCHALK, S.; LIN, M. C.; MANOCHA, D. OBBTree: A hierarchical structure for rapid interference detection. **Computer Graphics**, New Orleans, v.30, p.171–180, 1996.

GOVINDARAJU, N. K. et al. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In: ACM SIGGRAPH/EUROGRAPHICS CONFERENCE ON GRAPHICS HARDWARE, 2003. **Proceedings…** [S.l.:s.n.], 2003. p.25–32.

HASAN, K.M.; PARKER, D. L.; ALEXANDER, A. L. Comparison of gradient encoding schemes for diffusion tensor imaging. **Journal of Magnetic Ressonance and Imaging**, Salt Lake City, v.13, n.5, p.769–780, 2001.

HEIDELBERGER, T. K. Star – state of the art report collision detection for deformable objects. In: EUROGRAPHICS, 2004. **Proceedings…** [S.l.:s.n.], 2004.

HUBBARD, P. M. Collision detection for interactive graphics applications. **IEEE Transactions on Visualization and Computer Graphics**, [S.l.], v.1, n.3, p.218–230, 1995.

HUDSON, T. C. et al. V-COLLIDE: Accelerated collision detection for VRML. In**:** SYMPOSIUM ON THE VIRTUAL REALITY MODELING LANGUAGE, VRML 1997. **Proceedings…** New York: ACM Press, 1997.

JAMES, D. L.; PAI, D. K. BD-Tree: Output-sensitive collision detection for reduced deformable models. **ACM Transactions on Graphics**, New York, v.23, n.3, p.393-398, 2004.

MIRTICH, B. **Efficient algorithms for two–phase collision detection.** [S.l.]: Mitsubishi Electric Research Laboratory, 1997. p.203-223 (Technical Report TR-97-23).

NAYLOR, B. F. Interactive solid geometry via partitioning trees. In**:** GRAPHICS INTERFACE, 1992. **Proceedings…** San Francisco: Morgan Kaufmann, 1992. p.11–18.

SAMET, H. **The Design and Analysis of Spatial Data Structures**. Reading, MA: Addison-Wesley, 1990.

SLEATOR, D. D.; TARJAN, R. E. Self-adjusting binary search trees. **Journal of the ACM**, New York, v.32, n.3, p.652–686, 1985.

SLEATOR, D. D.; TARJAN, R. E. Self-adjusting heaps. **SIAM Journal on Computing**, Philadelphia, v.15, n.1, p.52–69, 1986.

SMITH, R. Open dynamics engine v 0.5 user guide. 2004. Disponível em: <http://ode.org/ode-latest-userguide.pdf>. Acesso em: 10 jan. 2005.

SNYDER, J.; LENGYEL, J. Visibility sorting and compositing without splitting for image layer decompositions. In: INTERNATIONAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1998. **Proceedings…** New York, 1998, p.219–230.

TARJAN, R. E. **Data Structures and Network Algorithms.** Philadelphia: Society for Industrial and Applied Mathematics, 1983.

TESCHNER,M. et al. Optimized spatial hashing for collision detection of deformable objects. In: VISION, MODELING, VISUALIZATION, 2003. **Proceedings…** Munich: [s.n.], 2003. p.47-54.

THATCHER, U. Loose octrees. In: **Game Programming Gems**. Rockland: Charles River Media, 2000. p.444–453.

TORRES, E. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In**:** EUROGRAPHICS, 1990. **Proceedings…** [S.l.]: North-Holland, 1990. p.507–518.