Análise de Desempenho de Algoritmos para Detecção de Colisão em Ambientes Gráficos Interativos

Rafael de Sousa Rocha[1], **Maria Andréia Formico Rodrigues**[1]

[1]Mestrado em Informática Aplicada (MIA) – Universidade de Fortaleza (UNIFOR)
60811-905 – Fortaleza – CE – Brasil

rafaelrocha@edu.unifor.br, mafr@unifor.br

Abstract. *Interactive graphical environments containing a great number of 3D objects need fast, accurate and scalable collision detection methods. This work presents a detailed performance analysis of broad phase collision detection algorithms: Brute Force, Grid, Octree, and Sweep & Prune. The broad phase algorithm with the best performance (Sweep & Prune) was integrated to a narrow phase algorithm based on Sphere-Trees, composing a hybrid collision detection algorithm. The results obtained show that this algorithm is fast, accurate and scalable, hence, it is recommended for interactive applications that contain a great number of colliding objects with complex geometry.*

Resumo. *Ambientes gráficos interativos, contendo um grande número de objetos 3D, necessitam de métodos para detecção de colisão rápidos, precisos e com alta escalabilidade. Este trabalho apresenta uma análise de desempenho detalhada de algoritmos para detecção de colisão broad phase: Força Bruta, Grid, Octree, e Sweep & Prune. O algoritmo de broad phase com melhor desempenho (Sweep & Prune) foi integrado a um algoritmo de narrow phase baseado em Sphere-Trees, compondo um algoritmo híbrido para detecção de colisão. Os resultados obtidos mostram que este algoritmo é rápido, preciso e escalável, portanto, sendo recomendado para aplicações interativas que contêm um grande número de objetos colidentes com geometria complexa.*

1. Introdução

Ambientes gráficos contendo um grande número de objetos 3D necessitam de métodos para detecção de colisão rápidos, precisos e com alta escalabilidade, de tal forma a garantir a interatividade do usuário com a aplicação. Entretanto, apesar dos recentes avanços das tecnologias de computação, a detecção de colisão é ainda um dos grande gargalos para o desenvolvimento de aplicações gráficas interativas. Estas possuem taxas interativas e constantes de quadros por segundo (q/s), restringindo o tempo disponível para a detecção [Bergen 2004]. Contudo, existem alternativas de otimização, tais como, o uso de coerência espacial e/ou temporal. A coerência espacial pode ser alcançada realizando-se a detecção de colisão em duas fases: na primeira (*broad phase*), os pares de objetos próximos são calculados; já na segunda (*narrow phase*), o teste de interseção entre esses pares é refinado. Outra abordagem para otimização da detecção de colisão é a substituição de objetos complexos (utilizados no processo de renderização do ambiente gráfico) por representações geométricas com menor nível de detalhamento. Existem basicamente dois métodos para detecção de colisão: contínuos e discretos [Redon 2004].

Algoritmos contínuos não são apropriados para ambientes interativos com um grande número de objetos, pois interpolar suas posições para efetuar o cálculo do instante das colisões é computacionalmente caro. Por outro lado, algoritmos discretos, amplamente utilizados em ambientes interativos, amostram as posições dos objetos em intervalos de tempo para determinar a ocorrência de colisões. Contudo, algoritmos discretos podem falhar para objetos em alta velocidade (neste caso, um objeto poderá eventualmente atravessar um outro, termo conhecido como *collision tunneling* [Ericson 2005]).

Este trabalho apresenta inicialmente uma análise detalhada do desempenho de algoritmos para detecção de colisão *broad phase* em ambientes virtuais interativos: Força Bruta [Rocha e Rodrigues 2005]; utilizando *Grid* [Rocha e Rodrigues 2006a]; utilizando *Octree* [Rocha e Rodrigues 2006b]; e *Sweep & Prune* [Cohen et al. 1995]. A partir dos resultados obtidos em nossos experimentos, o algoritmo com melhor desempenho de *broad phase* (*Sweep & Prune*) foi integrado a um algoritmo de *narrow phase* baseado em *Sphere-Trees* (hierarquia de esferas) [Hubbard 1996], compondo um algoritmo híbrido de detecção de colisão [Rocha et al. 2006]. O desempenho deste algoritmo híbrido foi então avaliado exaustivamente, utilizando duas abordagens para a geração de *Sphere-Trees* (construção através de *Octrees* e o algoritmo *Combined* [Bradshaw e O'Sullivan 2004]). Os resultados obtidos mostram que o algoritmo híbrido é rápido, preciso e escalável, portanto, sendo recomendado para aplicações gráficas interativas contendo um grande número de objetos de geometria complexa, potencialmente colidentes. Mais especificamente, obtemos taxas médias de aproximadamente 24 q/s e 16 q/s para cenários com 1000 objetos movendo-se em xyz, utilizando *Sphere-Trees* com 2 e 3 níveis de profundidade, respectivamente. Uma descrição completa e detalhada dos algoritmos implementados e dos experimentos conduzidos pode ser obtida em [Rocha 2006].

2. Trabalhos Relacionados

Vários algoritmos eficientes de detecção de colisão para ambientes interativos têm sido propostos [Cohen et al. 1995, O'Sullivan e Dingliana 1999, Bergen 2004]. No entanto, muitos se limitam a uma classe específica de aplicações e nenhum soluciona o problema da detecção de colisão para todos os casos. Comumente, volumes envoltórios são utilizados para a rejeição rápida de pares de objetos distantes. Exemplos são caixas alinhadas aos eixos [Bergen 2004], caixas orientadas [Gottschalk et al. 1996], esferas [Hubbard 1996] e *k-DOPs* [Klosowski et al. 1998]. No entanto, para ambientes com um grande número de objetos colidentes, o teste entre volumes envoltórios pode ser um gargalo. É tarefa da *broad phase* diminuir o número de testes realizados. Uma forma de se implementar a *broad phase* é utilizar estruturas de partição espacial (por exemplo, *Grids* [Lawlor e Kalée 2002], *Octrees* [Bandi e Thalmann 1995] e *BSP-Trees* [Luque et al. 2005]), pois estas permitem o uso de coerência espacial entre os objetos. Já outros algoritmos de *broad phase* não utilizam estruturas de partição espacial (por exemplo, o *Sweep & Prune*, que pode utilizar a coerência temporal entre os quadros da animação [Cohen et al. 1995]).

Para a *narrow phase* da detecção de colisão, é possível usar a própria geometria do objeto (como no algoritmo GJK [Bergen 2004], por exemplo) ou uma representação simplificada desta. Hierarquias de volumes envoltórios (*AABB-Trees* [Bergen 2004], *OBB-Trees* [Gottschalk et al. 1996], *Sphere-Trees* e hierarquias de *k-DOPs*) são exemplos de estruturas que aproximam a geometria dos objetos. *Sphere-Trees* são estruturas particular-

mente interessantes, pois as esferas que as compõem podem ser utilizadas para aproximar o cálculo da detecção e resposta à colisão [O'Sullivan e Dingliana 1999]. Isso possibilita a implementação de um algoritmo interruptível [Hubbard 1996], bastante apropriado para aplicações interativas nas quais precisão e velocidade devem ser balanceadas. Utilizando esta técnica, é possível alocar um intervalo de tempo para o processo de detecção de colisão. Nos casos em que este não possa ser concluído no intervalo de tempo alocado, o algoritmo interrompe o processo e utiliza uma heurística para aproximar o resultado.

3. *Broad Phase*

O algoritmo de Força Bruta para a *broad phase* verifica a existência de interseção entre os volumes envoltórios de todos os objetos da cena. Portanto, para n objetos, são necessários $\binom{n}{2}$ testes de interseção, resultando em um algoritmo de complexidade $O(n^2)$.

3.1. *Grids* e *Octrees*

Uma forma simples de otimizar o algoritmo de Força Bruta é o uso de estruturas de partição espacial, as quais particionam o espaço considerado para a detecção de colisão em regiões. Apenas os objetos pertencentes a uma mesma região são testados por colisão. É possível particionar o espaço em um grande número de regiões, diminuindo consideravelmente o número de testes. Contudo, a sobrecarga de atualização das estruturas pode comprometer o desempenho da detecção de colisão.

As estruturas de partição espacial mais utilizadas são os *Grids* e as *Octrees*. *Grids* regulares particionam o espaço em *voxels* (células cúbicas) de mesmas dimensões, de acordo com uma granularidade pré-definida. Estas estruturas são estáticas e, geralmente, não se alteram durante a geração da animação (apenas as listas de objetos de cada região são atualizadas), sendo indicadas para cenários onde os objetos estejam uniformemente distribuídos. Já as *Octrees* são *Grids* hierárquicos de granularidade $2 \times 2 \times 2$ (8 octantes), as quais particionam recursivamente os *voxels* que interceptam, pelo menos, o volume envoltório de um objeto, até uma profundidade pré-definida. *Voxels* que não interceptam nenhum objeto são descartados da hierarquia. Uma vez construída a *Octree* para um determinado quadro da animação, não há garantia de que esta continuará válida para o próximo quadro, portanto, sendo necessário reconstruir a *Octree* a cada novo quadro.

Apesar da otimização significativa obtida pelos algoritmos de *broad phase* que utilizam *Grids* e *Octrees* em relação ao algoritmo de Força Bruta, estes ainda possuem outras limitações, além das já mencionadas. Por exemplo, para diferentes cenários, é difícil estimar a melhor granularidade do *Grid* e a melhor profundidade da *Octree*. Outro problema é o aumento da memória necessária para a execução destes algoritmos, pois um objeto pode interceptar vários *voxels* simultaneamente, devendo ser referenciado várias vezes.

3.2. Sweep & Prune

O algoritmo *Sweep & Prune* busca diminuir o número de testes da *broad phase* através do cálculo eficiente de interseção entre caixas alinhadas aos eixos. Esse volume envoltório pode ser representado pelos três intervalos resultantes de sua projeção, nos três eixos coordenados x, y e z. Além disso, é possível mostrar que se os três intervalos resultantes das projeções possuem interseção, então, as duas caixas alinhadas aos eixos possuem

interseção [Cohen et al. 1995]. A idéia principal deste algoritmo é manter os intervalos de todas as caixas em três listas ordenadas (uma para cada eixo) e, a partir destas, derivar os pares colidentes. Uma vantagem desta abordagem é a possibilidade da utilização de coerência temporal, pois, geralmente, os objetos se movem a pequenas distâncias entre quadros da animação. Como as três listas do quadro corrente estão ordenadas, é provável que, no próximo quadro, a lista esteja próxima de estar ordenada. O algoritmo *insertion sort* de ordenação pode, então, ser utilizado para manter as listas ordenadas em tempo linear, fazendo com que o algoritmo tenha baixa sobrecarga de atualização, se comparado às abordagens apresentadas na Seção 3.1. O *Sweep & Prune* possui complexidade média de $O(n+m)$ [Bergen 2004], onde n é o número total de caixas e m é o número de caixas em estado de colisão.

A implementação do *Sweep & Prune* neste trabalho foi baseada no algoritmo apresentado em [Ericson 2005]. Uma limitação do *Sweep & Prune* ocorre para arranjos de objetos em fila, nos quais as caixas envoltórias possuem interseção em dois eixos coordenados. Nesta situação, o algoritmo deverá testar todas as caixas, portanto, sendo indicado para cenários onde não ocorra esse tipo de distribuição dos objetos.

4. *Narrow Phase*

Na *narrow phase* da detecção de colisão é realizado um teste mais refinado dos pares resultantes da *broad phase*. Em particular, utilizamos *Sphere-Trees* para a representação dos objetos.

4.1. *Sphere-Trees*

Sphere-Trees correspondem a hierarquias de esferas comumente utilizadas para aproximar a geometria dos objetos. Optamos por utilizar *Sphere-Trees* devido aos seguintes fatos: esferas são rotacionalmente invariantes, o que facilita a atualização das *Sphere-Trees*; o cálculo de interseção entre esferas é simples e de baixo custo computacional; existem vários algoritmos eficientes para geração automática de *Sphere-Trees*; e *Sphere-Trees* podem ser usadas na implementação de um algoritmo interruptível para tratamento de colisões [Bradshaw e O'Sullivan 2004].

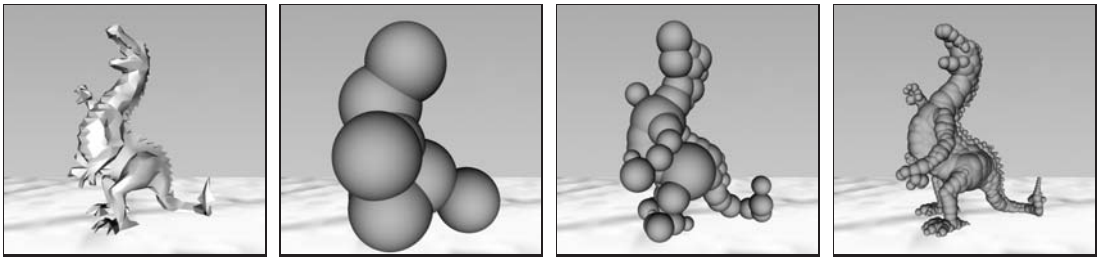
4.2. Construção de *Sphere-Trees*

Uma forma para construção de *Sphere-Trees* é via *Octrees*, que são construídas utilizando-se a caixa envoltória do objeto como raiz e circunscrevendo-se cada um de seus nós. As esferas resultantes deste processo formam a *Sphere-Tree* do objeto. Apesar deste método ser simples e rápido, este não gera *Sphere-Trees* precisas.

Hubbard desenvolveu uma abordagem mais sofisticada para a construção de *Sphere-Trees* [Hubbard 1996], baseada na aproximação do eixo medial do objeto (definido como sendo a união dos centros de todas as esferas máximas dentro da figura, resultando em uma estrutura que se assemelha a um esqueleto para o objeto). Para calcular uma aproximação do eixo medial, Hubbard utilizou diagramas de Voronoi. A partir desta estrutura, o autor mostra que é possível derivar um conjunto inicial de esferas e, destas, a própria *Sphere-Tree*. O nó raiz da *Sphere-Tree* pode ser calculado como sendo a esfera envoltória do objeto. Cada nó após a raiz é construído em um processo *bottom-up*, através da fusão recursiva de pares de esferas. A adição dos nós na *Sphere-Tree*, no entanto, é feita em um processo *top-down*.

Bradshaw e O'Sullivan propuseram outros métodos para a construção de *Sphere-Trees* baseados no algoritmo de Hubbard (algoritmos *Merge*, *Burst*, *Expand*) [Bradshaw e O'Sullivan 2004]. Estes algoritmos melhoram o algoritmo de Hubbard através da tentativa de redução do número de esferas na *Sphere-Tree*. Os autores também apresentaram outras formas para a construção de *Sphere-Trees* (algoritmos *Grid* e *Spawn*). Adicionalmente, também desenvolveram uma abordagem nomeada *Combined*, que pode utilizar vários algoritmos de redução de esferas em conjunto (geralmente os algoritmos *Merge* e *Expand*) para gerar *Sphere-Trees* bastante precisas. Neste trabalho, utilizamos o *Sphere-Tree Construction Toolkit* [Bradshaw 2003] para gerar as *Sphere-Trees* dos objetos e armazená-las em arquivos. A aplicação que desenvolvemos carrega as *Sphere-Trees* dos arquivos e as posiciona na superfície dos objetos.

A Figura 1 exibe os níveis 0, 1, 2 e 3 da *Sphere-Tree* do objeto "dragão", gerada pelo algoritmo *Combined* em (a), (b), (c) e (d), respectivamente. Observe que o nível 3 da *Sphere-Tree* em (d) possui um alto grau de precisão. Uma desvantagem do algoritmo *Combined* é que este pode demorar para calcular as *Sphere-Trees*, porém, este cálculo pode ser realizado em uma fase de pré-processamento.

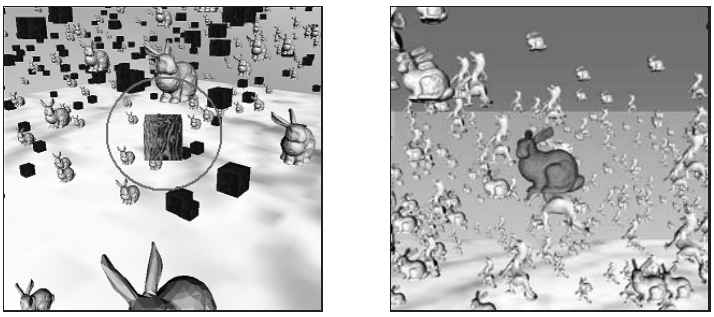


(a) "Dragão" (b) Nível 1 (c) Nível 2 (d) Nível 3

Figura 1. *Sphere-Tree* gerada com o algoritmo *Combined*.

5. Cenários da Aplicação

Os cenários que projetamos para realizar todos os experimentos foram implementados utilizando-se o pacote gráfico Java3D [Rowe 2001] e um *joystick* com *force feedback* para controlar o movimento dos objetos circunscritos na Figura 2 ("caixa" em (a) e "coelho" em (b)) e interagir mais realisticamente com a aplicação [Rocha e Rodrigues 2006b]. O ambiente gráfico é representado por uma sala, contendo 1000 objetos potencialmente colidentes. Implementamos um cenário 1 com objetos do tipo "caixa" e "coelho" (em (a) da Figura 2), e um cenário 2 com objetos do tipo "coelho" e "dragão" (em (b) da Figura 2). Os objetos movem-se no espaço xyz, possuem velocidade linear constante e relativamente pequena (evitando a ocorrência de *collision tunnelling*), e não rotacionam (evitando a necessidade de recalcular as caixas alinhadas aos eixos que envolvem os objetos). Apesar do cenário implementado possuir estas restrições, em aplicações reais, a velocidade dos objetos é geralmente controlada e relativamente baixa. Além disso, para cenários onde os objetos rotacionam, é possível utilizar caixas envoltórias amplas o suficiente de tal forma a garantir que o objeto, mesmo rotacionado, permaneça englobado pelo envoltório. Esta técnica foi utilizada com sucesso, por exemplo, em [Cohen et al. 1995].



(a) Cenário 1 (b) Cenário 2

Figura 2. Cenários da aplicação.

6. Experimentos e Resultados

No cenário 1, avaliamos o desempenho dos algoritmos, sem levar em consideração o processo de renderização da animação, utilizando um computador o qual chamaremos de computador 1 (com uma placa gráfica *on-board* regular, processador Pentium HT 3,06Ghz e memória RAM de 768Mb, da qual a placa gráfica utiliza 64Mb). Já no cenário 2, levamos em consideração também o processo de renderização da animação, escolhendo para nossas novas análises o algoritmo que obteve o melhor desempenho nos experimentos anteriores, utilizando, para fins comparativos, além do computador 1, um outro, o qual chamaremos de computador 2 (com uma placa gráfica GeForce 7300 GT da NVIDIA com 256Mb de memória, processador Pentium D 3,00Ghz e memória RAM de 512Mb).

6.1. Cenário 1

Inicialmente, para o cenário 1, foram realizados vários experimentos utilizando somente os algoritmos de *broad phase*. A resposta à colisão foi implementada aplicando-se um algoritmo heurístico que utiliza os intervalos das caixas envoltórias dos objetos colidentes para determinar quais componentes da velocidade inverter.

Inicialmente, avaliamos o desempenho dos algoritmos de Força Bruta e dos que utilizam *Grids* e *Octrees*, para um número de objetos variando de 100 a 500 (em (a) da Figura 3). Nota-se que as abordagens que utilizam estruturas de partição espacial têm desempenhos notadamente superiores ao do algoritmo de Força Bruta. Além disso, devido ao fato do *Grid* ser uma estrutura estática, construída em uma fase de pré-processamento, a abordagem que a utiliza possui desempenho melhor do que a que utiliza *Octree*, já que essa última necessita que a *Octree* seja reconstruída a cada quadro da animação. Em particular, os algoritmos que utilizam *Grid* e *Octree* necessitam de parâmetros específicos: a granularidade do *Grid* e a profundidade da *Octree*, respectivamente. Em nossos experimentos, os melhores parâmetros encontrados e utilizados na comparação dos algoritmos de *broad phase* foram: uma granularidade de $10 \times 4 \times 10$ para o *Grid* e uma profundidade de 3 níveis para a *Octree*. Por sua vez, o algoritmo *Sweep & Prune* (em (b) da Figura 3) tem desempenho superior ao algoritmo que utiliza *Grid*, alcançando uma alta taxa de q/s, sem degradar drasticamente o seu desempenho à medida que o número de objetos aumenta. Para 1000 objetos, por exemplo, enquanto o algoritmo que utiliza *Grid* alcança uma taxa de 14 q/s, o algoritmo *Sweep & Prune* atinge 54 q/s. Isto demonstra que esta abordagem, além de rápida, também apresenta maior escalabilidade e é, portanto,

a mais apropriada para detectar colisões na *broad phase* para o ambiente interativo que implementamos.

Mais especificamente, para analisar as diferenças entre o algoritmo que utiliza *Grid* e o *Sweep & Prune*, foram comparados os tempos gastos na atualização das estruturas de dados utilizadas e nos testes de interseção entre os pares de caixas envoltórias, durante 500 quadros da animação (Figura 4). Em particular, observamos uma grande disparidade entre ambos algoritmos no tempo gasto para a realização dos testes de interseção. O algoritmo *Sweep & Prune* calcula os pares colidentes em muito menos tempo. O tempo necessário para a atualização das estruturas de dados é relativamente maior no algoritmo *Sweep & Prune* do que no que utiliza *Grid* (o primeiro consome aproximadamente 78% a 98% do tempo total da *broad phase*, enquanto que o segundo varia de 53% a 62%).

Também utilizamos o algoritmo *Sweep & Prune* em conjunto com o algoritmo baseado em *Sphere-Trees*. A resposta foi implementada utilizando-se as esferas colidentes das *Sphere-Trees* para calcular quais componentes da velocidade inverter. A Figura 5 mostra o desempenho deste algoritmo híbrido utilizando *Sphere-Trees* construídas com *Octrees* de 4 e 5 níveis de profundidade, bem como construídas com o algoritmo *Combined* com 2 e 3 níveis de profundidade (considerando-se o nível 0 como o primeiro). O desempenho do algoritmo de *narrow phase* é diretamente afetado pela profundidade das *Sphere-Trees*. Portanto, o algoritmo *Combined* é mais apropriado para os cenários implementados, pois gera *Sphere-Trees* mais precisas e com menor profundidade.

6.2. Cenário 2

Para o cenário 2, coletamos então os percentuais de processamento da *broad phase* e da *narrow phase* do algoritmo híbrido (Figura 6). Constatamos que, à medida que o número de objetos aumenta, os percentuais da *broad phase* diminuem e os da *narrow phase* aumentam (cenários com poucos objetos possuem menos colisões e, portanto, a única sobrecarga do algoritmo híbrido será a manutenção das listas ordenadas do algoritmo *Sweep & Prune* para a *broad phase*). Por outro lado, para cenários com muitos objetos, o número de colisões é maior e, portanto, o algoritmo baseado em *Sphere-trees* para a *narrow phase* será invocado diversas vezes. Por exemplo, quando o ambiente possui 1000 objetos colidentes, o algoritmo híbrido utilizando *Sphere-trees* com 2 níveis de profundidade consome aproximadamente 70% do tempo de processamento da detecção na *broad phase* e 30% na *narrow phase* e, utilizando *Sphere-trees* com 3 níveis de profundidade, consome aproximadamente 40% do tempo de processamento da detecção na *broad phase* e 60% na *narrow phase*.

A Figura 7 mostra em (a) a taxa de q/s, considerando-se apenas o tempo gasto na detecção de colisão para o cenário 2. Note que o desempenho dos computadores sem e com placa gráfica aceleradora (computadores 1 e 2, respectivamente) são comparáveis, pois o processo de renderização não influencia as curvas de desempenho exibidas. Para 1000 objetos colidentes, utilizando-se *Sphere-trees* com 2 níveis de profundidade, obtemos taxas médias de 39 q/s e 38 q/s para os computadores 1 e 2, respectivamente. Já para *Sphere-trees* com 3 níveis de profundidade, obtemos taxas médias de 21 q/s em ambos computadores.

Já em (b) da Figura 7 são mostradas as taxas de q/s alcançadas no cenário 2, incluindo o processo de renderização. Obtemos um melhor desempenho utilizando *Sphere-*

trees com 3 níveis de profundidade no computador 2, do que utilizando *Sphere-trees* com 2 níveis de profundidade no computador 1. Especificamente, para 1000 objetos colidentes, alcançamos uma taxa média de 24 q/s e 16 q/s no computador 2 para *Sphere-trees* com 2 e 3 níveis de profundidade, respectivamente, enquanto 12 q/s e 10 q/s para os mesmos níveis de profundidade no computador 1.

7. Conclusões

Neste trabalho, implementamos e avaliamos o desempenho de quatro algoritmos para a *broad phase* (Força Bruta, utilizando *Grid*, utilizando *Octree*, e *Sweep & Prune*). O algoritmo *Sweep & Prune* obteve o melhor desempenho e, entre os algoritmos testados, possui a maior escalabilidade para os cenários da aplicação implementada. Já para a *narrow phase*, utilizamos um algoritmo baseado em *Sphere-Trees*. Foram comparadas duas abordagens de construção de *Sphere-Trees* (utilizando *Octree* e o algoritmo *Combined*). Mostramos que o algoritmo *Combined* gera *Sphere-Trees* mais precisas e, portanto, é mais apropriado para representar a detecção de colisão *narrow phase* dos objetos modelados.

Adicionalmente, propomos e implementamos um algoritmo híbrido para a detecção de colisão que utiliza *Sweep & Prune* para a *broad phase* e o algoritmo baseado em *Sphere-Trees* para a *narrow phase*. Os resultados obtidos na análise de desempenho do algoritmo híbrido comprovam que este pode ser aplicado em ambientes interativos contendo um número considerável de objetos. Por exemplo, para cenários com até 1000 objetos movendo-se em xyz, alcançamos taxas superiores a 20 q/s, utilizando *Sphere-trees* com 2 níveis de profundidade, geradas pelo algoritmo *Combined*. Neste caso, mostramos que o algoritmo híbrido consome aproximadamente 70% do tempo de processamento da detecção na *broad phase* e 30% na *narrow phase*. Também demonstramos como o algoritmo híbrido pode ser utilizado para balancear velocidade, precisão e escalabilidade na detecção de colisão. Por exemplo, *Sphere-trees* com 2 níveis de profundidade podem ser utilizadas em aplicações que requerem maior velocidade e escalabilidade (por exemplo, na área de Jogos por computador, mono ou multi-usuário). Já *Sphere-trees* com 3 níveis de profundidade podem ser utilizadas em aplicações que requerem maior precisão na detecção de colisão (por exemplo, em Aplicações Gráficas em Medicina, Robótica, Engenharia, etc).

Agradecimentos

Rafael de Sousa Rocha gostaria de agradecer o apoio financeiro concedido para o desenvolvimento deste trabalho, enquanto bolsista de Iniciação Científica PIBIC/CNPq.

Referências

Bandi, S. e Thalmann, D. (1995). An Adaptive Spatial Subdivision of the Object Space for Fast Collision Detection of Animated Rigid Bodies. *CGF*, 14(3):259–270.

Bergen, G. V. D. (2004). *Collision Detection in Interactive 3D Environments*. Morgan and Kaufmann Publishers.

Bradshaw, G. (2003). Sphere-Tree Construction Toolkit. http://isg.cs.tcd.ie/spheretree/.

Bradshaw, G. e O'Sullivan, C. (2004). Adaptive Medial-Axis Approximation for Sphere-Tree Construction. *ACM Transactions on Graphics*, 23(1):1–26.

Cohen, J. D., Lin, M. C., Manocha, D., e Ponamgi, M. K. (1995). I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In *Proc. of the Symposium on Interactive 3D Graphics*, pages 189–196, USA. ACM Press.

Ericson, C. (2005). *Real-Time Collision Detection*. Morgan and Kaufmann Publishers.

Gottschalk, S., Lin, M. C., e Manocha, D. (1996). OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics*, 30:171–180.

Hubbard, P. M. (1996). Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Transactions on Graphics*, 15(3):179–210.

Klosowski, J. T., Held, M., Mitchell, J. S., Sowizral, H., e Zikan, K. (1998). Efficient Collision Detection using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36.

Lawlor, O. S. e Kalée, L. V. (2002). A Voxel-Based Parallel Collision Detection Algorithm. In *Proc. of the* 16^{th} *International Conference on Supercomputing*, pages 285–293, New York, USA. ACM Press.

Luque, R. G., Comba, J. L. D., e Freitas, C. M. D. S. (2005). Broad-Phase Collision Detection using Semi-Adjusting BSP-Trees. In *Proc. of the Symposium on Interactive 3D Graphics and Games*, pages 179–186. SIGGRAPH, ACM Press.

O'Sullivan, C. e Dingliana, J. (1999). Real-Time Collision Detection and Response Using Sphere-Trees. In *Proc. of the* 15^{th} *Spring Conference on Computer Graphics*, pages 83–92, Budmerice, Slovakia.

Redon, S. (2004). Continuous Collision Detection for Rigid and Articulated Bodies. Tutorial. ACM SIGGRAPH Course Notes.

Rocha, R. S. (2006). Avaliação Experimental de Métodos para Detecção de Colisão em Ambientes Gráficos Interativos. Monografia de Final de Curso, Universidade de Fortaleza, UNIFOR.

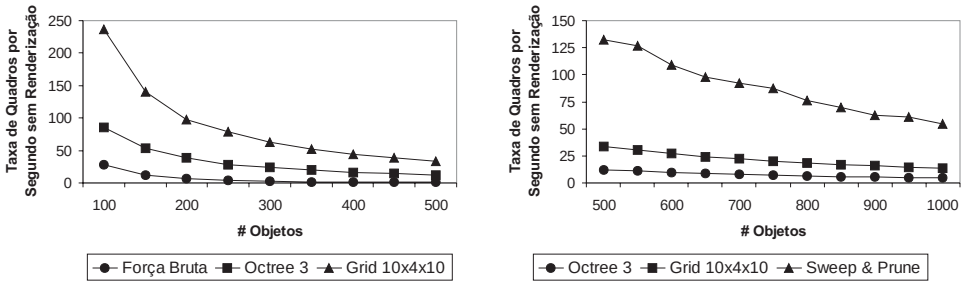
Rocha, R. S. e Rodrigues, M. A. F. (2005). Ambientes Interativos com Detecção de Colisão *Broad Phase* Utilizando *Grids*. In *Anais do III Workshop de Iniciação Científica do XVIII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, Natal, RN. Meio digital.

Rocha, R. S. e Rodrigues, M. A. F. (2006a). Detecção de Colisão *Broad Phase* utilizando *Grids* para Ambientes Interativos. *Revista Eletrônica de Iniciação Científica (REIC)*, 6(2):1–17.

Rocha, R. S. e Rodrigues, M. A. F. (2006b). Investigating Broad Phase Collision Detection Methods for 3D Scenarios Using Force Feedback Devices. In *Proc. of the XXXII Latin-American Conference on Informatics (CLEI)*, Santiago do Chile, Chile. Meio digital.

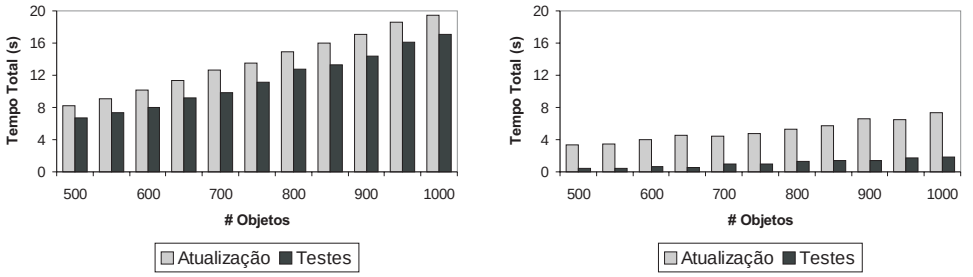
Rocha, R. S., Rodrigues, M. A. F., e Taddeo, L. S. (2006). Performance Evaluation of a Hybrid Algorithm for Collision Detection in Crowded Interactive Environments. In *Proc. of the XIX Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pages 86–93, Manaus, AM, Brazil. IEEE CS Press.

Rowe, G. W. (2001). *Computer Graphics with Java*. Grassroots Series. Palgrave.



(a) (b)

Figura 3. Comparação entre algoritmos de *broad phase*.



(a) Algoritmo que utiliza *Grid* (b) Algoritmo *Sweep & Prune*

Figura 4. Comparação entre o algoritmo que utiliza *Grid* e o *Sweep & Prune*.

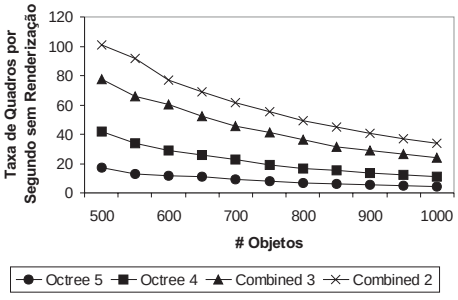


Figura 5. Desempenho do algoritmo híbrido (cenário 1).

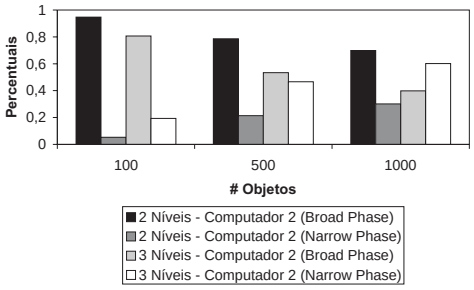
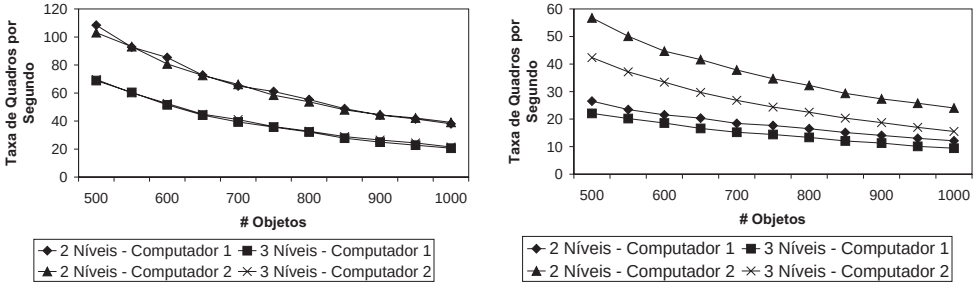


Figura 6. Percentuais do algoritmo híbrido.



(a) Sem renderização (b) Com renderização

Figura 7. Desempenho do algoritmo híbrido (cenário 2).