Detecção de Colisão Broad Phase com RAY KD-Tree: Uma Abordagem Eficiente e Adaptativa

Ygor Rebouças Serpa[1], Maria Andréia Formico Rodrigues[2]

[1]Centro de Ciências Tecnológicas – UNIFOR
[2]Programa de Pós-Graduação em Informática Aplicada – UNIFOR
60811-905 – Fortaleza-CE – Brasil

{ygor.reboucas, andreia.formico}@gmail.com

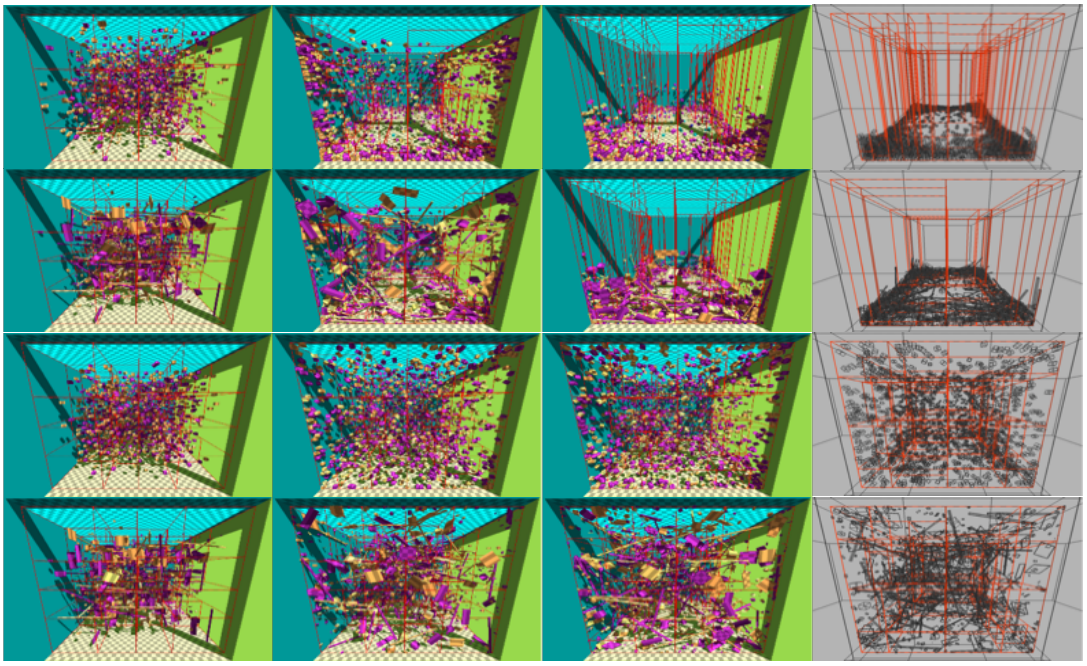


Figura 1: De cima para baixo, os cenários: *Queda Livre Uniforme*, *Queda Livre Variado*, *Partículas Uniforme* e *Partículas Variado*. Cada linha mostra 3 quadros-chaves das respectivas animações e 1 quadro em *wireframe com* a estrutura dos planos de corte. Animações disponíveis em https://www.youtube.com/watch?v=XGSxysp69gw

Abstract. *Several fast algorithms for Broad Phase collision detection have been proposed, but few of them have diverse applicability being, in most cases, limited to a small class of scenarios and/or of difficult configuration. In this work, we describe RAY KD-Tree, a new solution based on the use of k-dimensional trees for an efficient, generic, and easily configurable execution, which is able to adapt automatically to the test scenarios and to the dynamics of their objects.*

Resumo. *Vários algoritmos rápidos para a detecção de colisão Broad Phase têm sido propostos, mas poucos deles possuem aplicabilidade diversificada sendo, na maioria das vezes, limitados a uma pequena classe de cenários e/ou de difícil configuração. Neste trabalho, descrevemos RAY KD-Tree, uma nova solução baseada no uso de árvores k-dimensionais para uma execução eficiente, genérica e facilmente configurável, a qual é capaz de adaptar-se automaticamente aos cenários de teste e à dinâmica dos seus objetos.*

1. Introdução

Aplicações gráficas interativas cada vez mais realistas, as quais têm em seu cerne objetos 3D regidos por Dinâmica, vêm produzindo resultados de alto impacto em diversas áreas, entre as quais: Prototipagem Visual, Robótica, Jogos Digitais, Engenharia, Medicina, etc. Mais especificamente, o controle robusto de objetos por Dinâmica está diretamente relacionado a métodos otimizados para detecção de colisão [Bergen 2004]. Contudo, apesar dos avanços das tecnologias de computação, a detecção de colisão ainda é um dos grandes gargalos dos ambientes gráficos interativos.

Desta forma, ambientes gráficos que não tratam a ocorrência de colisões entre objetos dinâmicos de forma robusta, não conseguem simular diferentes comportamentos de forma realista e convincente [Coutinho 2001]. Assim, vários tipos de otimização têm sido propostos usando-se a distância relativa entre os objetos (coerência espacial) ou a direção de seus deslocamentos (coerência temporal), de tal forma a filtrar um grande número de pares de objetos, por exemplo, os que estão distantes entre si ou deslocando-se em direções opostas. Uma outra forma de otimização consiste em englobar objetos 3D em envoltórios de geometria mais simples, tais como, caixas. O processo de colisão pode então ser subdividido em duas fases [Mirtich 1997]: (1) *Broad Phase*, cujo foco é encontrar os pares de envoltórios próximos que se interceptam; e (2) *Narrow Phase*, cujo objetivo é testar a geometria real desses pares de objetos, de forma mais refinada.

Fundamentais no desempenho geral da aplicação, algoritmos de colisão *Broad Phase* focam no uso de técnicas de busca e/ou estruturas de dados robustas para acelerar o descarte de pares de objetos não colidentes. Comumente, são apresentadas soluções baseadas em *grids* [Lo *et al*. 2013], *quadtrees*, *octrees* ou árvores de volumes envoltórios [Coumans 2008], bem como na ordenação espacial dos objetos [Liu *et al*. 2010, Zomorodian and Edelsbrunner 2000]. No entanto, um dos grandes problemas da área é a dificuldade em se aplicar um mesmo algoritmo, de forma otimizada, em cenários contendo objetos com características geométricas e de movimento diferentes. Por exemplo, métodos baseados em *grid*s sofrem grandes penalidades quando a distribuição dos objetos não é uniforme. Soluções baseadas em árvores, ora oferecem estruturas rígidas com ótimos particionamentos; ora flexíveis, porém, pouco expressivas, apresentando dificuldades no tratamento de objetos dinâmicos e baixo desempenho quando há muitos objetos estáticos. Métodos baseados em ordenação focam na escolha de algoritmos otimizados, bem como na escolha do melhor eixo de varredura. Por fim, diversos algoritmos têm aplicabilidade prática reduzida devido à necessidade de uma extensa configuração de diversos parâmetros iniciais, que muitas vezes são pouco compreensíveis e sensíveis às variações no cenário que possam ocorrer em tempo de execução como, por exemplo, o número máximo de interseções entre objetos pequenos pertencentes a uma dada célula de um *grid*.

Este trabalho estende consideravelmente a pesquisa realizada em [Rocha 2010]. Mais especificamente, apresentamos uma nova solução, a qual nomeamos *RAY KD-Tree,* para a detecção de colisão *Broad Phase,* baseada no uso de uma árvore *k*-dimensional, capaz de tratar diversos tipos de cenários 3D aplicando-se um processo dinâmico de adaptação do algoritmo à configuração espacial dos objetos. Em particular, a adaptação é regida apenas por um único parâmetro de configuração T, responsável por controlar a profundidade da estrutura, o qual garante simultaneamente uma execução eficiente, genérica e de fácil uso, capaz de tratar diversos tipos de cenários contendo

objetos com geometrias variadas e diferentes comportamentos de movimento. Além disso, 4 casos de teste foram modelados com o objetivo de avaliar o desempenho do *RAY KD-Tree* em relação a outros algoritmos especializados em detecção de colisão *Broad Phase,* disponíveis na biblioteca *Bullet* [Coumans 2015], bem como frente a um método genérico para a busca de interseções entre objetos de um conjunto, fornecido pela biblioteca *CGAL* [Zomorodian and Edelsbrunner 2000, Kettner *et al*. 2015].

Vale ressaltar que trabalhos prévios publicados pelos autores em co-autoria foram fundamentais para a evolução natural na área de pesquisa em questão e nas tomadas de decisão que levaram ao desenvolvimento da nova solução aqui apresentada. Mais especificamente, em [Serpa e Rodrigues 2013] conduzimos um estudo comparativo entre soluções de colisão em nível de CPU e GPU e, em [Serpa e Rodrigues 2014], estendemos a publicação anterior, enfatizando a aplicabilidade da solução na área de jogos digitais. Em [Oliveira *et al*. 2013], técnicas de detecção de colisão foram empregadas para simular acidentes de trânsito em forense. Em [Rodrigues *et al*. 2015], implementamos a detecção de colisão para fins não físicos, em um amplo cenário de um *serious game* voltado para o aprendizado de regras de trânsito e direção segura, detectando infrações e gerenciando todos os veículos não controlados pelo jogador. Finalmente, em [Rodrigues *et al*. 2014], focamos nossa contribuição na representação de colisões entre objetos rígidos 3D e malhas dinâmicas, garantindo a geração de taxas interativas de quadros por segundo. Detalhes dos resultados, na forma de animações podem ser visualizados em https://www.youtube.com/watch?v=XGSxysp69gw

2. Trabalhos Relacionados

Diversos trabalhos têm explorado o uso de diferentes tipos de estruturas de dados para a detecção de colisão *Broad Phase*: *grids* [Lo *et al*. 2013]; *BVHs* [Coumans 2008]; *BSPs* [Luque *et al*. 2005]; e *KD-Trees* e *Octrees* [Glass 2005]. No entanto, a maioria destes concentra-se na qualidade do particionamento espacial das estruturas e faz pouco uso da coerência temporal das animações, exibindo uma deficiência de soluções para cenários dinâmicos. Em [Glass 2005], são comparadas as estruturas *Octree*, *KD-Tree* e *BSP*, concluindo-se que estas duas últimas atingem os melhores desempenhos e que a *KD-Tree* realiza o menor número de testes de colisão. Em [Luque *et al*. 2005], é apresentada uma *BSP* capaz de se auto-ajustar, através da identificação de desbalanceamentos e do agendamento de operações de ajuste. Comparativamente, nossa abordagem usa *KD-Trees* devido à alta capacidade de particionamento, focando exaustivamente no quesito adaptabilidade. Diferentemente de trabalhos como [Luque *et al*. 2005], que agendam ajustes custosos na estrutura a fim de amortizar custos, optamos por uma abordagem gulosa, cujos ajustes são realizados assim que for caracterizado o desbalanceamento.

No contexto das bibliotecas mais atuais para detecção e simulação da dinâmica entre corpos colidentes, a *Bullet* [Coumans 2015] provê dois métodos de *Broad Phase*: *DBVT* (*Dynamic Bounding Volume Tree*) e *AxisSweep* [Coumans 2008]. O primeiro usa uma árvore de volumes envoltórios e possui duas variantes, uma para cenários com pouco dinamismo e outra para cenários mais dinâmicos. Neste trabalho, referenciaremos essas variantes como *DBVT F* (*forward*) e *DBVT D* (*deferred*), respectivamente. O segundo método, *AxisSweep,* é baseado no algoritmo *Sweep and Prune* [Cohen 1995], também aplicado em cenários cujos objetos são, na maioria, estáticos. Fora do escopo das aplicações voltadas exclusivamente para a Física, a biblioteca *CGAL* oferece uma

solução genérica para a busca de interseções entre pares de caixas em *k*-dimensões [Kettner *et al*. 2015], baseada em [Zomorodian and Edelsbrunner 2000].

3. RAY KD-Tree

Nossa solução divide-se em duas partes: (1) a estrutura para a organização espacial dos objetos, incluindo suas rotinas de atualização e adaptação; e (2) o algoritmo capaz de encontrar as interseções geométricas entre todos os objetos presentes na estrutura.

3.1. Estrutura de Dados

Organizamos os objetos da simulação com uma árvore *k*-dimensional estritamente binária, ou seja, cada nó tem obrigatoriamente zero ou dois filhos, cujos nós são compostos de uma lista de objetos e um plano de corte. A estrutura é persistente, ou seja, a cada quadro é ajustada ao invés de reconstruída. Além disso, apenas 2 restrições R_1 e R_2 são aplicadas à estrutura: (1) R_1: cada objeto deve residir no nó mais profundo capaz de contê-lo completamente; e (2) R_2: cada nó folha deve conter no máximo T objetos, onde T corresponde a um valor limiar.

Os planos de corte dos nós, quando criados, são posicionados na média do eixo de maior variância dos centros dos objetos. A sucessão de planos, da raiz até um nó, delimita uma região, a qual chamamos de *AABB* (caixa envoltória alinhada aos eixos) do nó. Adicionalmente, definimos como população de um nó, a soma do número de seus objetos com a de seus filhos esquerdo e direito. Cada objeto é representado por uma *AABB*, capaz de englobá-lo integralmente. Para evitar varreduras desnecessárias, armazenamos na estrutura do nó a sua *AABB* e a sua população.

A cada quadro, são executadas 4 tarefas: (1) T_1: atualização dos objetos; (2) T_2: reposicionamento; (3) T_3: particionamento/poda da estrutura; e (4) T_4: otimização do particionamento. Basicamente, a T_1 consiste em atualizar o envoltório de cada objeto, ajustando-o à nova posição e orientação. Encarregada da restrição R_1, T_2 requer o uso de dois operadores: *Preparação* e *Otimização*, responsáveis, respectivamente, por garantir que o nó contenha o objeto e que esteja no menor nó que o contenha. Em T_3, devemos garantir a restrição R_2, com os operadores de *Particionamento* e *Poda*, para transformar, respectivamente, um nó folha em interno e um interno em folha. Já T_4 busca manipular os nós internos para melhor distribuir os objetos na estrutura. Para tal, aplicamos mais 3 operadores: *Avaliação*, *Translação* e *Troca*. *Avaliação* verifica se um nó deve (ou não) ser alterado; *Translação* altera o plano de corte do nó, melhorando-o; e *Troca* descarta o nó e sua sub-árvore menos populosa, mantendo apenas a sub-árvore restante.

Neste fluxo, primeiramente, atualizamos os objetos e preparamos os nós. O operador *Preparação* garante que cada objeto esteja em um nó que o contenha, ou seja, objetos que saíram da *AABB* de seus nós serão posicionados de forma a estarem em nós que os contenham. Aplicamos então o operador *Otimização* em nós internos, verificando, para cada objeto, se este pode ser transferido para o filho esquerdo ou direito (transferindo-o, se possível). Este processo moverá, nó à nó, os objetos para o menor nó que os contenha, atendendo à restrição R_1 e concluindo T_2. Após a otimização, verificamos se o nó deve ser podado. De forma antagônica, sempre verificamos cada nó folha para o particionamento. Ambas as verificações realizam a R_2 e T_3. Para completar T_4, nós internos não podados são avaliados para a execução do operador *Translação* ou *Troca*. A execução deste fluxo é capaz de forçar R_1 e R_2 e realizar T_1, T_2, T_3 *e* T_4,

concluindo a atualização e a adaptação da estrutura. Nas próximas subseções, devido às restrições de espaço, apresentaremos de forma sintetizada, cada um dos operadores.

3.1.1. Operador *Preparação*

Prepara um nó, removendo todos os objetos que não estão completamente dentro de sua *AABB*, transferindo-os para o seu nó pai. Aplicamos este operador realizando uma varredura em largura inversa (das folhas à raiz). Desta forma, antes de preparar um nó *N*, preparamos seus dois nós filhos. Portanto, objetos dos filhos de *N,* que também não estão contidos em *N,* serão enviados para o pai de *N*.

3.1.2. Operador *Otimização*

Busca otimizar o posicionamento dos objetos, movendo-os para nós com *AABBs* menores, porém, que ainda os contenham integralmente. Aplicado a um nó, varre seus objetos, verificando se este está completamente à esquerda ou à direita do plano de corte. Caso posicionado à esquerda, é transferido para o filho esquerdo; caso contrário, para o direito. Caso parcialmente à esquerda e à direita, é mantido no nó atual.

3.1.3. Operador *Particionamento*

Aplicado apenas aos nós folhas contendo um número de objetos superior a *T*, transformando-os em nós internos e criando, para cada um, dois novos nós (filhos) e um plano de corte que corta o espaço na média do eixo de maior variância dos centros dos objetos. Após isso, aplicamos o operador *Otimização* no nó, distribuindo seus objetos.

3.1.4. Operador *Poda*

Aplicado apenas aos nós internos, verificando se o nó possui uma população inferior a *T*. Em caso afirmativo, será transformado em nó folha, removendo ambas suas sub-árvores, transferindo seus objetos para o nó. Embora este operador não seja fundamental para a execução, contribuí para diminuir a altura da árvore, removendo ramos extensos e pouco povoados, tornando a execução e o uso de memória mais eficientes.

3.1.5. Operador *Avaliação*

Avalia o particionamento de um nó, tendo papel crucial na capacidade de adaptação da estrutura de forma eficaz, através de duas heurísticas que definimos: *Balanceamento* e *Custo*. Se o valor do *Balanceamento* for maior que o *Custo*, o nó é avaliado positivamente, não sendo necessário ajustá-lo; caso contrário, será alterado. Considerando *M* o número total de objetos da sub-árvore, *N* o número de objetos do nó, *P* a população da sub-árvore mais populosa e *p* a da menos populosa, definimos: *Balanceamento* = *p*/*P*. Calculamos então o *Custo* a partir de uma estimativa do número de testes de colisão feitos para a sub-árvore, onde: $Estimativa = \binom{N}{2} + \binom{P}{2} + \binom{p}{2} + N(P+p)$.

Considerando que, no pior caso (*pc*), todos os objetos estarão em uma das sub-árvores e, no melhor caso (*mc*), haverá exatamente metade dos objetos em cada sub-árvore, então: $pc = \binom{M}{2}$ e $mc = 2\binom{M/2}{2}$. Portanto, $Custo = (Estimativa - pc) / (mc - pc)$.

3.1.6. Operador *Translação*

Trata-se da primeira escolha para a otimização do particionamento de um nó, visando alterá-lo de forma rápida e potencialmente suficiente. Primeiramente, transfere todos os objetos das sub-árvores esquerda e direita para o nó, recalculando a média dos objetos

no eixo do plano de corte e atualizando o plano com o novo valor, transladando-o. O operador *Otimização* é então acionado para redistribuir os objetos. Conforme a execução do fluxo continuar, os objetos serão recolocados em suas posições originais.

3.1.7. Operador *Troca*

Em contraste com o operador anterior, *Troca* realiza uma alteração mais drástica na estrutura, removendo o nó e sua sub-árvore menos populosa, mantendo apenas a mais populosa em seu lugar. Todos os objetos dos nós removidos são transferidos para a sub-árvore restante.

3.2. Detecção de Colisão

Nesta etapa, o objetivo é usar a estrutura que desenvolvemos para encontrar todos os pares de objetos colidentes. A partir da árvore atualizada e dos objetos posicionados, para cada nó, há duas verificações: (1) V_1: teste de todos os objetos do nó entre si; e (2) V_2: teste de todos os objetos do nó contra os objetos que interceptam a *AABB* do nó. Enquanto V_1 é trivial, V_2 requer uma forma eficiente de encontrar todos os objetos que interceptam a *AABB* do nó. Nomeamos este grupo de objetos de *Lista de Objetos de Fronteira (LOF)*. Sabemos que todos os objetos da *LOF* de um nó estão nos nós que ligam o nó ao nó raiz, pois se um objeto não está completamente em um nó, estará em um de seus ancestrais. Adicionalmente, objetos pertencentes a nós internos interceptam o plano de corte de seus nós, caso contrário, o operador *Otimização* o transferiria para um de seus nós filhos. Além disso, sabemos que a região de um nó é a parte esquerda *E* ou direita *D* da região de seu nó pai. Portanto, dado um objeto x, unindo estas três informações construímos as *LOFs* dos filhos *E* e *D* de um nó n, desta forma:

$LOF(E) = \text{objetos}(n) \cup \{x \mid x \in LOF(n)$ e está à esquerda do plano de corte de $n\}$
$LOF(D) = \text{objetos}(n) \cup \{x \mid x \in LOF(n)$ e está à direita do plano de corte de $n\}$

Finalmente, encontramos todos os pares colidentes de objetos, percorrendo a estrutura proposta. Começamos pelo nó raiz, o qual não possui *LOF* (pois não possui ancestral), realizando V_1 e calculando a *LOF* de seus dois filhos. Em seguida, para cada nó, realizamos V_1 e V_2 e, caso o nó avaliado seja interno, calculamos as *LOFs* de seus dois filhos. A busca em profundidade é preferível nesta execução, pois reduz a quantidade de *LOFs* em memória.

4. Casos de Teste

Em busca de uma maior representatividade de casos de teste possíveis, desenvolvemos dois conjuntos de objetos do tipo corpo rígido (um com objetos *Uniformes* e outro com objetos *Variados*) e duas dinâmicas de movimentação (uma por *Queda Livre* e outra por *Partículas*), que quando combinados, totalizam 4 cenários diferentes de teste (Figura 1).

Mais especificamente, nas linhas 1 e 3 da Figura 1 (objetos *Uniformes*), todos os objetos são cubos idênticos. Já nas linhas 2 e 4 da Figura 1 há objetos de formatos e tamanhos variados, tais como, barras, placas e cubos (objetos *Variados*). Na dinâmica por *Queda Livre*, representada nas linhas 1 e 2 da Figura 1, não há interferência na dinâmica dos objetos, de tal forma que os objetos vão se depositando aos poucos na base do cubo que engloba o cenário. Na dinâmica por *Partículas*, mostrada nas linhas 3 e 4 da Figura 1, atribuímos uma velocidade aleatória a uma pequena parcela dos objetos a cada quadro da animação, adicionando energia cinética à simulação, de forma que

estes se desloquem similarmente ao movimento de partículas de um gás ideal, chocando-se freqüentemente uns com os outros. Estes 2 cenários foram implementados na *Bullet* a fim de gerar animações fisicamente plausíveis. Com elas, executamos todos os algoritmos por diversos quadros ao longo do tempo, avaliando a desenvoltura de cada método, sob o ponto de vista das diversas dinâmicas apresentadas pelos objetos.

Cada um dos cenários foi inicializado com n objetos dispostos em posições aleatórias, dentro de um cubo de dimensões $100x100x100$ (metros), com velocidade v também aleatória e de magnitude entre [0,100] m/s, utilizando -9,81 m/s² como valor para a aceleração da gravidade. Executamos cada simulação durante 2.000 quadros da animação, utilizando um *timestep* fixo de 1/120 segundos e a mesma semente para o gerador de números aleatórios, de tal forma a garantir que a execução de cada simulação seja idêntica. Durante as simulações, foram cronometrados os tempos de processamento dos algoritmos e a altura média dos nós folha do nosso algoritmo *RAY KD-Tree*.

A fim de avaliar *RAY KD-Tree* frente aos métodos disponíveis em bibliotecas amplamente usadas no meio acadêmico e na indústria de jogos e cinematográfica, selecionamos os dois métodos mencionados na Seção 2 (*DBVT F* e *DBVT D)*, da *Bullet* [Lo *et al*. 2013, Liu *et al*. 2010] e o algoritmo presente na *CGAL* [Zomorodian and Edelsbrunner 2000]. Este último, reconhecido por outros autores [Lo *et al*. 2013] como muito eficiente e de uso prático, está disponível no módulo *Intersecting Sequences of dD Iso-oriented Boxes* [Kettner *et al*. 2015] e será referenciado neste trabalho, por questões de simplicidade, pelo nome da biblioteca a qual pertence, ou seja, *CGAL*. A escolha desses métodos é justificada pelo fato de que compõem uma métrica de referência bastante conhecida, fazendo parte de bibliotecas de uso bastante consolidado.

Por fim, todo o desenvolvimento deste trabalho foi realizado em C++ e compilado com o *MVC++ 12.0*, em modo *release*. Os testes foram executados na plataforma *Windows 8.1 Update 1 (x64)* em um computador com as seguintes especificações: processador *Intel Core i7 860 2.8GHz*, *8GB* de memória principal e placa gráfica *NVidia GT560*. Cada teste foi realizado 5 vezes e os resultados mostrados nos gráficos foram consolidados tomando-se como referência a média das 5 execuções.

5. Análise Comparativa de Desempenho e Resultados

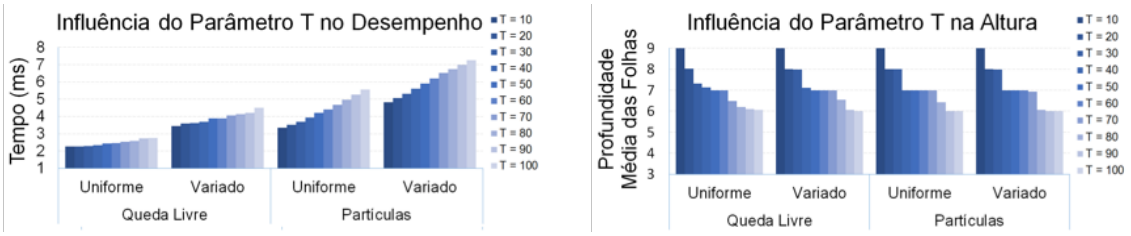
Primeiramente, analisaremos a influência da variação do *threshold T* no desempenho do *RAY KD-Tree* e na altura da estrutura implementada. Em seguida, apresentaremos resultados comparativos entre os algoritmos testados.

5.1. Variação do *Threshold T*

No *RAY KD-Tree*, o *threshold T* controla o particionamento dos nós folhas da estrutura e a poda dos nós internos. Um nó folha será particionado caso possua mais de T objetos e um nó interno; e será podado, caso duas sub-árvores vazias existirem ou se a sua população for inferior a T. Desta forma, valores pequenos para este limiar resultam em árvores mais profundas, enquanto valores altos produzem árvores mais rasas.

Para os quatro cenários da Figura 1, contendo 4.000 objetos, em (a) da Figura 2, é mostrado o gráfico da influência de T (variando de 10 a 100) no desempenho do *RAY KD-Tree*; e, em (b) da Figura 2, o gráfico da profundidade média das folhas da estrutura em função de T. Nestes dois gráficos, é possível observar que o desempenho do *RAY KD-Tree* é diretamente relacionado à altura da árvore: quanto maior a árvore, melhor o

desempenho. Desta forma, concluímos que o parâmetro *T* age como um controlador do *trade-off* entre desempenho e uso de memória, mostrando que valores pequenos resultam em um melhor desempenho, porém, em um maior consumo (árvore maior). Desta forma, a solução não só é fácil de ser configurada, como seu parâmetro é bastante compreensível. Para os testes a seguir, utilizaremos o valor 10 para *T* em todas as execuções do algoritmo *RAY KD-Tree*.



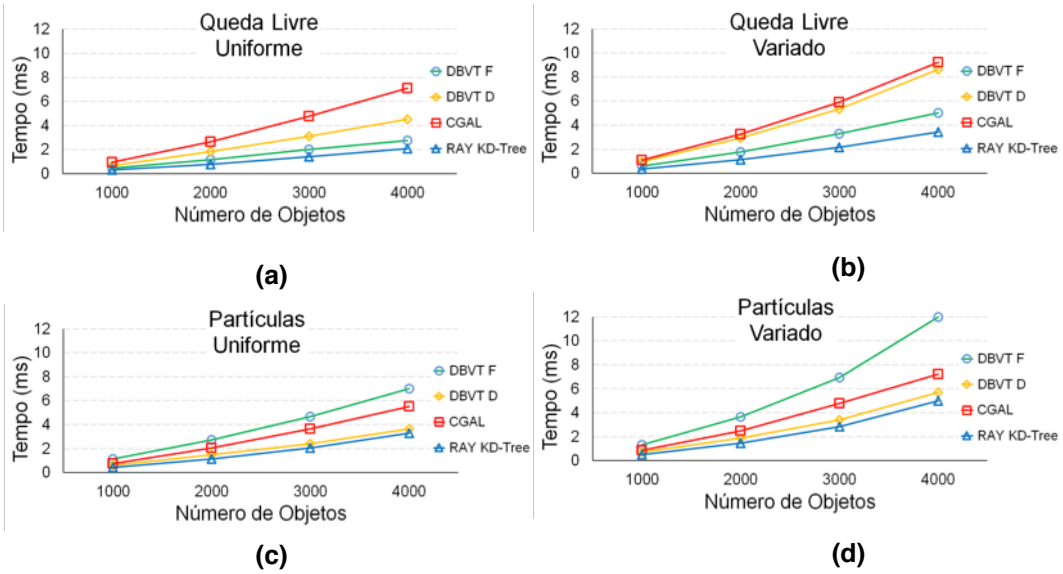
(a) **(b)**

Figura 2. Influência do parâmetro *T* no desempenho e na altura da árvore, sendo todos os testes realizados com 4.000 objetos.

5.2 Análise de Desempenho

Como mostra a Figura 3, *RAY KD-Tree* apresenta, consistentemente, um melhor desempenho em relação aos outros métodos. Embora a diferença entre o algoritmo proposto e o segundo melhor algoritmo não seja tão expressiva na maioria dos casos, o ganho real em usar este método está na capacidade de oferecer o melhor desempenho, em todos os cenários explorados.

Analisando o desempenho dos algoritmos de referência nos cenários *Queda Livre*, como todos os objetos estão em repouso no final da simulação, observamos *DBVT F* à frente dos demais algoritmos. Em contrapartida, nos cenários *Partículas*, devido à realimentação do sistema com novas velocidades, *DBVT D* mostra-se melhor para tratar cenários com níveis de dinamismo. *CGAL*, por sua vez, demonstra uma maior habilidade em lidar com objetos de tamanhos variados, pois as curvas geradas para os cenários contendo objetos *Uniformes* são quase idênticas às curvas com objetos *Variados*, diferentemente dos algoritmos da *Bullet*, em especial, o *DBVT F*.



(a) **(b)**

(c) **(d)**

Figura 3. Tempo de execução médio por quadro de simulação para os 4 cenários.

De forma geral, *RAY KD-Tree* apresenta uma independência de cenário e de tipos de objetos, não demonstrando pior ou melhor caso para o algoritmo. O quarto cenário de teste é o mais próximo de exibir alguma tendência. Neste cenário, podemos notar que todos os algoritmos apresentam uma piora igual ou maior na escalabilidade, visto que é o mais complexo dos quatro cenários testados.

Esta independência de cenário, aliada à facilidade de configuração do algoritmo, constitui as principais contribuições do *RAY KD-Tree*, uma solução genérica e prática, com alta aplicabilidade e, simultaneamente, competitiva frente a outras soluções conhecidas.

6. Conclusões e Trabalhos Futuros

Neste trabalho, apresentamos *RAY KD-Tree*, uma nova solução para a detecção de colisão *Broad Phase* com árvores *k*-dimensionais. Dentre os cenários e os algoritmos selecionados como referência nos testes, *RAY KD-Tree* obteve o melhor desempenho. Não apenas eficiente, nossa solução se mostrou genérica o suficiente para lidar com diversos tipos de cenários, além de ser, na prática, bastante simples de ser configurada. Adicionalmente, *RAY KD-Tree* é extensível para *n* dimensões e aplicável a outras áreas de pesquisa, aumentando a sua relevância. Por exemplo, *RAY KD-Tree* também pode ser recomendada para aplicações não vinculadas a simulações físicas, como detecção de proximidade entre objetos, organização de dados *k*-dimensionais não pontuais e dinâmicos, balanceamento de cargas, gerenciamento de dados em simulações federadas (*HLA*), entre outras. Além disso, outras melhorias são possíveis, tanto em detecção de colisão *Broad Phase* em geral, quanto no *RAY KD-Tree*, justificando a continuidade na busca por soluções ainda mais robustas e otimizadas para o problema da detecção.

Embora bastante eficiente para a detecção de colisão *Broad Phase*, *RAY KD-Tree* não foi concebido, por exemplo, para testar todos os objetos contra uma *AABB* avulsa ou para realização de *raycasts*. Nossa estrutura otimiza estas operações, mas ainda não é tão especializada nestas tarefas quanto as *BVHs* (*Bounding Volume Hierarchies*) e *KD-Trees* com *SAH* (*Surface Area Heuristic*). Como trabalhos futuros, também planejamos estender *RAY KD-Tree*, propondo uma solução híbrida que combine a solução corrente e o algoritmo *Sweep and Prune* [Liu *et al*. 2010].

Agradecimentos

Ygor R. Serpa gostaria de agradecer à FUNCAP-CE (processo No. 0074-00006.01.40/13), pelo apoio financeiro recebido.

Referências

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

Cohen, J.D. e Lin, M.C. (1995) “I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments”, In Proc. of I3D’95, pages 189-ff, New York.

Coumans, E. (2008). “btDBVT Documentation”. <bulletphysics.org/mediawiki1.5.8/index.php?title=BtDbvt_dynamic_aabb_tree>.

Coumans, E. (2015) “Bullet Physics Library”. Disponível em: <http://bulletphysics.org>. Último acesso em: 24/04/2015.

Coutinho, M. G. (2001) “Dynamic Simulations of Multibody Systems”. Los Angeles, CA: Springer-Verlag.

He, L., Ortiz, R., Enquobahrie, A., e Manocha, D. (2015). “Interactive Continuous Collision Detection for Topology Changing Models Using Dynamic Clustering”. In Proc. of the I3D’15, pages 47–54, NY, USA, ACM.

Kettner, L., Meyer, A., Zomorodian, A. (2015). “Intersecting Sequences of dD Iso-oriented Boxes”. In CGAL User and Reference Manual. CGAL Edit. Board, 4.6 Ed.

Liu, F., Harada, T., Lee, Y., Kim Y. J. (2010). “Real-time Collision Culling of a Million Bodies on Graphics Processing Units”. ACM TOG, 29(6), p. 154:1-154:8.

Lo, S. –H., Lee, C.-R., Chung. I.-H., e Chung. Y.-C. (2013). “Optimizing Pair-Wise Box Intersections Checking on GPUs for Large-scale Simulations”. ACM TOMACS, 23(3), p. 19:1-19:22, ACM Press, NY, USA.

Mirtich, B. (1997) “Efficient Algorithms for Two-Phase Collision Detection”, Technical Report, December.

Oliveira, A.E.; Serpa, Yvens. R.; Serpa, Ygor, R.; Abreu, A.P.; Macedo, D.V.; Rodrigues, M.A.F. (2013). “Visualização Interativa 3D e Simulação de Dinâmica de Acidentes de Trânsito em Forense Utilizando a Blender Game Engine em um Serious Game”. Em Anais do *SBGames’13, Arte & Design*, SP, p. 244-252.

Rocha, R.S. (2010) “Algoritmos Rápidos de Detecção de Colisão *Broad Phase* Utilizando *KD-Trees*”. Dissertação de Mestrado, Pós-Graduação em Informática Aplicada (PPGIA), Universidade de Fortaleza (UNIFOR), Fortaleza-CE.

Rodrigues, M.A.F.; Macedo, D.V.; Serpa, Yvens. R.; Martins, C.; Candolo, P.H.T.; Gobet, T.; Serpa, Ygor, R.; Secundino, L. (2014). “Combatendo a Halitose: Um Serious Game Multiplataforma em Saúde Bucal”. Em Anais do *SBGames’14, Arte & Design*, RGS, p. 210-219.

Rodrigues, M.A.F.; Macedo, D.V.; Serpa, Ygor, R.; Serpa, Yvens. R. (2015). “Beyond Fun: An Interactive and Educational 3D Traffic Rules Game Controlled by Non-traditional Devices”. In Proc. of the *ACM SAC’15*, Salamanca, Spain, v. 1, p. 1-8.

Serpa, Ygor, R.; Rodrigues, M.A.F. (2013) “Estudo Comparativo entre Algoritmos para Detecção de Colisão Broadphase em CPU e GPU na *Bullet*”. Em Anais do *Workshop of Undergraduate Works (WUW)* do *SIBGRAPI’13*, Arequipa, Peru, p. 1-6.

Serpa, Ygor, R.; Rodrigues, M.A.F. (2014). “*Parallelizing Broad Phase Collision Detection for Animation in Games: A Performance Comparison of CPU and GPU Algorithms*”. In Proc. of the *SBGames’14, Computação*, RGS, p. 770-779.

Zomorodian. A., Edelsbrunner, H. (2000). “Fast Software for Box Intersections”. In Proc. of the SCG’00. ACM Press, p. 129-138, NY, USA.