ACELERAÇÃO DE ALGORÍTMOS DE DETECÇÃO DE COLISÃO PARA JOGOS DIGITAIS TRIDIMENSIONAIS[1],[2]

Henrique Lorenzi[3]
Vitor Brandi Junior[4]

RESUMO

Detecção de colisão é um aspecto importante para jogos digitais, mas nem sempre é prático utilizar as técnicas em suas formas primitivas: cenas com muitos objetos ou cenários complexos podem tornar a detecção de colisão inviável para jogos em tempo real, porque ela tomaria muito tempo computacional. Este artigo aborda uma estrutura de dados que acelera os algorítmos de detecção de colisão, para torná-los adequados, também, a tais situações problemáticas. Foi feita a descrição de um algoritmo de detecção de colisão – especificamente, a detecção de raio contra triângulos –, uma descrição da estrutura de aceleração utilizada – chamada *octree* –, e uma comparação entre a *performance* do algoritmo primitivo e do acelerado em vários casos-teste. Os resultados mostram que essa técnica é eficaz para cenários complexos, pois o tempo computacional foi reduzido consideravelmente e que sua utilização permite o desenvolvimento de jogos mais elaborados e engajantes.
Palavras-chave: Jogos digitais ; Detecção de colisão ; Otimização

ABSTRACT

Collision detection is an important subject pertaining to digital games, but it is not always practical to use its techniques in their simplest forms: scenes containing large number of objects or a complex terrain model may leave primitive collision detection unsuitable for real-time games, because its computations would take too long. This paper presents a data structure aimed at speeding up these algorithms, in order to make them suitable for such problematic scenarios. Topics addressed by this paper include: a description of a collision detection algorithm – namely, the ray-triangle intersection test –, a description of the acceleration structure – the octree –, and a performance comparison of both algorithms, under several test cases. The results show that this acceleration technique is effective for complex game models, as the computational time was reduced considerably, and that its inclusion in a project allows the development of richer and more enjoyable games.
***Keywords**: Digital games ; Collision detection ; Optimization*

INTRODUÇÃO

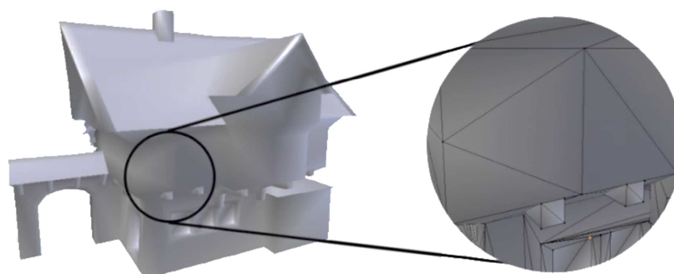
A indústria de jogos digitais vem sendo, desde seus primórdios, uma área altamente lucrativa que, em 2013, movimentou 21,53 bilhões de dólares mundialmente (THE ENTERTAINMENT SOFTWARE ASSOCIATION, 2014). É um mercado de processo criativo inovador e a atual tendência se mostra por jogos tridimensionais. Isto significa que os jogadores experimentam um universo com as mesmas propriedades geométricas da realidade, observado por uma câmera virtual, comparável a um filme de cinema, onde se reproduzem numa tela os atores e o cenário.

A busca por realismo nos jogos digitais, cujas imagens são geradas por computador, tem se mostrado atraente pela imersão que promove aos jogadores, quesito muito desejável para produtos na área de entretenimento. Com a atual tecnologia, a maioria das cenas nos jogos é construída a partir de formas geométricas simples, como triângulos, que generalizam e aceleram a geração de uma imagem pelo computador.

---

[1] Artigo baseado em Trabalho e Conclusão de Curso (TCC) desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais, depositado no 2º semestre de 2014.
[2] Depositado na Biblioteca da Fatec Americana em 29/12/2014.
[3] Tecnólogo em Tecnologia em Jogos Digitais – Fatec Americana – Centro Estadual de Educação Tecnológica Paula Souza ; Contato: hlorenzi12@hotmail.com
[4] Prof. da.Fatec Americana - Mestre em Gerenciamento de Sistemas de Informação pela Pontifícia Universidade Católica de Campinas; Contato: vitor.brandi@gmail.com

Figura 1 – Detalhe de um modelo tridimensional digital



Fonte: Próprio autor

Outra importante aplicação de triângulos em jogos digitais é na área de detecção de colisão. Atores em um jogo utilizam detecção de colisão para reagir ao cenário e a outros atores, por exemplo, ao não atravessar paredes ou cair através do chão. Muitas vezes, o modelo de colisão do cenário é composto por triângulos, tanto pela estrita correspondência com sua representação gráfica, quanto para a eficiência dos algorítmos envolvidos. Mas, ao utilizar esses algorítmos, rapidamente nota-se um problema quando os modelos de colisão em questão possuem um número significativo de triângulos, e o número de atores cresce – o tempo necessário para executar um algoritmo de colisão tradicional, nesses casos, é alto e, portanto inadequado a jogos. No entanto, existem técnicas para amenizar esse problema e permitir uma aplicação eficaz. Este trabalho tem o objetivo de descrever e discutir sobre uma técnica para tal objetivo.

Objetivos

Especificamente, este trabalho tem o objetivo de apresentar um algoritmo de detecção de colisão entre raio e triângulo, apresentar uma estrutura de dados que acelera os cálculos – a *octree* –, e fazer um levantamento do impacto que sua utilização proporciona na eficiência de um jogo. Serão utilizados conhecimentos em teoria de algorítmos, e, para o entendimento da formulação matemática, geometria analítica e álgebra vetorial.

Justificativa

Quando se trata de imersão em jogos digitais, estão intrinsicamente atrelados à qualidade dos sons, gráficos e *interfaces*, e a fidelidade que o jogador espera da simulação virtual nesses quesitos, com base em seu repertório adquirido no mundo real. Suas interações prévias com o ambiente real o fazem construir expectativas quanto aos comportamentos futuros dos objetos, sob interação semelhante. Quando essas expectativas não são atingidas, o jogador pode sentir descrença e desinteresse pela simulação e, consequentemente, pelo jogo (WELLER, 2013).

Um fato semelhante é descrito em Huizinga (2000), que se aplica até a jogos tradicionais, quando ele escreve: “a desobediência às regras implica a derrocada do mundo do jogo. O jogo acaba: o apito do árbitro quebra o feitiço e a vida "real" recomeça”. Se o jogador espera determinadas reações do jogo a suas interações, por exemplo, como fariam as leis da Física, e elas não são efetivadas, dá-se que o mundo do jogo se quebra, e nossa atenção se dissipa. Daí a necessidade de estudar e tratar esses assuntos com mais atenção.

Metodologia

A metodologia utilizada neste artigo é empírica, pois realiza testes práticos para verificar e comprovar a tese; e quantitativa, porque usa - se dados numéricos concretos para tirar conclusões.

Estrutura do trabalho

O artigo foi estruturado em quatro capítulos: o primeiro é reservado à revisão bibliográfica: estudos sobre o algoritmo de detecção de colisão raio-triângulo – juntamente à sua formulação matemática –, a estrutura de aceleração *octree*, e a metodologia utilizada; o segundo capítulo discorre sobre o projeto do trabalho e os testes realizados; o terceiro apresenta os resultados dos testes; e, por fim, o quarto capítulo reserva-se às conclusões.

1 REVISÃO BIBLIOGRÁFICA

A detecção de colisão em jogos digitais se preocupa em decidir se dois objetos se intersectam ou, ainda, se dois objetos estão em rota de colisão. É uma área importante, pois, situações como essas observadas no dia-a-dia traduzem-se em expectativas para com o jogo, e, se não forem atingidas, destroem a convicção com que o mundo virtual se apresenta ao jogador. Os seguintes itens discorrem, com mais detalhes, o repertório aplicado no projeto.

1.1 Detecção de colisão

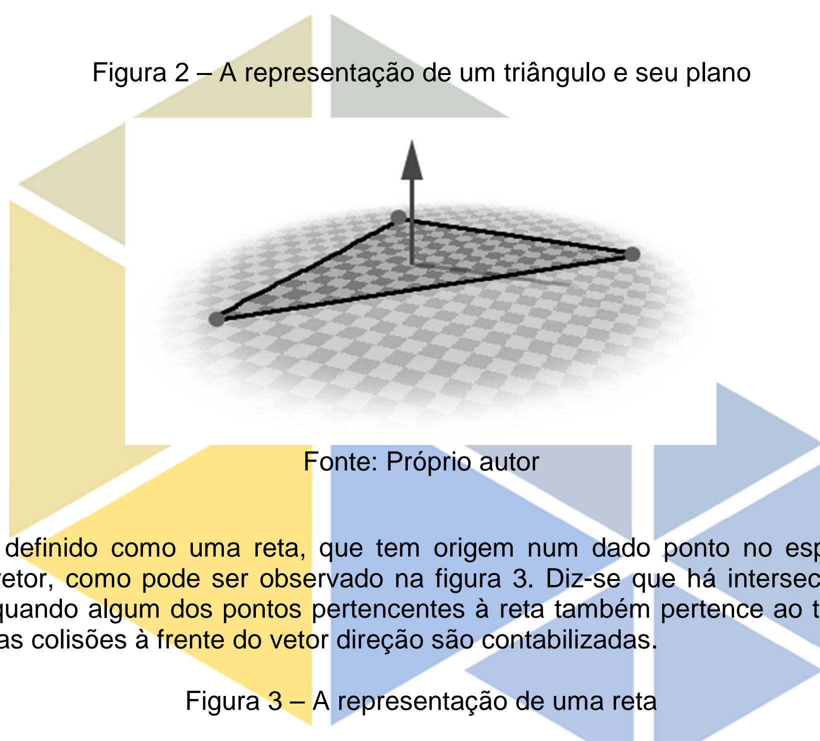
Existem inúmeras técnicas e modelos de representação computacional para a finalidade de detectar colisões. Por exemplo, um dos métodos para se calcular a interseção entre triângulos, como pode ser visto em Möller (2004), é adequado para decidir se dois modelos de objetos complexos colidem, como um personagem humano e um cenário urbano. No entanto, o presente artigo reserva-se apenas à detecção de colisão raio-triângulo, que pode ser usado, por exemplo, para decidir se um personagem está na linha de visão de outro, ou ainda, se um tiro atinge o alvo, sem a obstrução de paredes. Também se reserva apenas a um dos caminhos matemáticos possíveis, e outras formulações podem ser encontradas, por exemplo, em Segura e Feito (2001).

Modelo de representação

A unidade básica escolhida para representar um objeto no ambiente virtual é o triângulo, por dois motivos: a sua representação é a que mais se aproxima do modelo visual, que também usa triângulos devido aos *hardwares* gráficos atuais; e a simplificação dos cálculos, pois há fórmulas matemáticas rápidas para tais formas.

Um triângulo é definido por três pontos no espaço distintos, chamados vértices. Um cenário, num jogo, é definido por um conjunto de triângulos. Um triângulo, por ser formado de três pontos distintos, sempre pertence a um plano, que, por sua vez, possui um vetor que é perpendicular à sua superfície, como pode ser visto na figura 2.

Figura 2 – A representação de um triângulo e seu plano



Fonte: Próprio autor

Um raio é definido como uma reta, que tem origem num dado ponto no espaço, e uma direção expressa por um vetor, como pode ser observado na figura 3. Diz-se que há intersecção (colisão) do raio com um triângulo quando algum dos pontos pertencentes à reta também pertence ao triângulo. No entanto, para um raio, apenas colisões à frente do vetor direção são contabilizadas.

Figura 3 – A representação de uma reta

Fonte: Próprio autor

A formulação matemática para decidir-se se um raio intersecta um triângulo será discutida a seguir.

Formulação matemática

Este tópico reserva-se à teoria matemática da detecção de colisão entre raio e triângulo, como discutido em Möller e Trumbore (1997).

Um raio é definido pela equação vetorial da reta:

$$R = O + t\vec{D} \quad (1)$$

Um triângulo é definido por seus três vértices: V_0, V_1 e V_2. Um ponto pertencente a esse triângulo pode ser dado por

$$T = V_0 + u(\overrightarrow{V_0V_1}) + v(\overrightarrow{V_0V_2}) \quad (2)$$

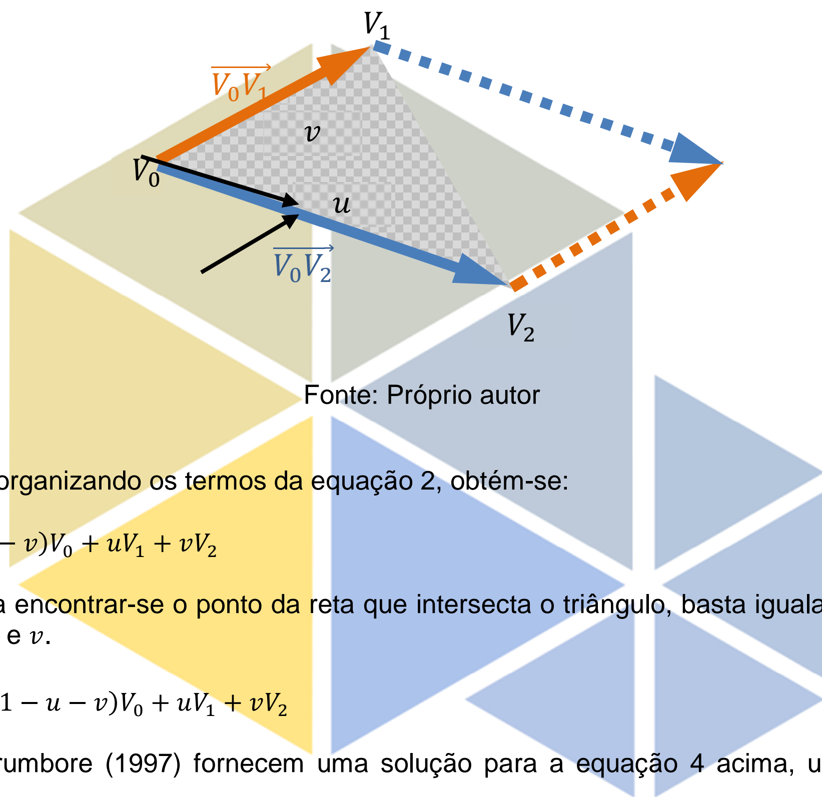
onde u e v são os coeficientes de uma combinação linear dos seus vetores-arestas, como pode ser visto na figura 4. É como se os vetores-arestas formassem a base de um sistema cartesiano com origem em V_0, e (u, v) fossem as coordenadas de um ponto nesse sistema.

No entanto, para que o ponto T pertença efetivamente ao triângulo é necessário que se tenha:

(A) $u \geq 0$ e $v \geq 0$, para que o ponto esteja no interior da área do paralelogramo formado pelos vetores-arestas, e

(B) $u + v \leq 1$, para que o ponto esteja na metade do paralelogramo mais próxima a V_0.

Figura 4 – Os vetores-arestas de um triângulo, e o paralelogramo que formam.



Fonte: Próprio autor

Reorganizando os termos da equação 2, obtém-se:

$$T = (1 - u - v)V_0 + uV_1 + vV_2 \quad (3)$$

Agora, para encontrar-se o ponto da reta que intersecta o triângulo, basta igualar as equações 1 e 3, e resolver para t, u e v.

$$O + t\vec{D} = (1 - u - v)V_0 + uV_1 + vV_2 \quad (4)$$

Möller e Trumbore (1997) fornecem uma solução para a equação 4 acima, usando uma equação matricial:

$$\begin{pmatrix} -\vec{D}_x & V_{1_x} - V_{0_x} & V_{2_x} - V_{0_x} \\ -\vec{D}_y & V_{1_y} - V_{0_y} & V_{2_y} - V_{0_y} \\ -\vec{D}_z & V_{1_z} - V_{0_z} & V_{2_z} - V_{0_z} \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} O_x - V_{0_x} \\ O_y - V_{0_y} \\ O_z - V_{0_z} \end{pmatrix} \quad (5)$$

em que denotam a solução como:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{P \cdot E_1} \begin{pmatrix} Q \cdot E_2 \\ P \cdot R \\ Q \cdot D \end{pmatrix} \quad (6)$$

Com

$$E_1 = V_1 - V_0 \quad (7)$$
$$E_2 = V_2 - V_0 \quad (8)$$
$$P = D \times E_2 \quad (9)$$

$$R = O - V_0 \quad (10)$$
$$Q = T \times E_1 \quad (11)$$

onde (ponto centralizado) denota o produto escalar, e × (cruz centralizada) denota o produto vetorial. Depois de calculados os coeficientes u e v, é necessário sinalizar uma colisão apenas quando eles estão de acordo com as condições A e B, acima.

Como alguns termos da equação 6 se repetem – por exemplo, P e Q –, o algoritmo os computa apenas uma vez e reusa seus valores. Além disso, o algoritmo é capaz de descartar uma colisão antes de completar a solução, quando detecta que u ou v não podem mais se enquadrar nas condições. No próximo tópico, será discutida a implementação na linguagem de programação C++.

Formato algoritmico

A partir dos conhecimentos matemáticos do tópico anterior, pode ser criado um algoritmo que implementará as fórmulas em um computador. A seguir, é apresentado o código de intersecção em linguagem C++, adaptado de Möller e Trumbore (1997).

O código leva, como argumentos, a definição de um raio e uma lista de triângulos. A função faz o teste de colisão contra todos os triângulos da lista, sem restrição, e retorna a menor distância em que houve uma colisão, caso aconteça alguma.

```
01 Colisao DeteccaoRaioTriangulo
02     (Raio raio, std::vector<Triangulo>& listaDeTriangulos)
03 {
04     double margemDeErro = 0.000001;
05
06     Colisao resultado;
07     resultado.houveColisao = false;
08
09     for (int i = 0; i < listaDeTriangulos.size(); i++)
10     {
11         Triangulo tri = listaDeTriangulos[i];
12         Vetor e1      = tri.vertice[1] - tri.vertice[0];
13         Vetor e2      = tri.vertice[2] - tri.vertice[0];
14         Vetor p       = raio.direcao.ProdutoVetorial(e2);
15
16         double div    = p.ProdutoEscalar(e1);
17         if (div < margemDeErro)    continue;
18
19         Vetor r       = raio.origem - tri.vertice[0];
20         double u      = r.ProdutoEscalar(p);
21         if (u < 0 || u > div)      continue;
22
23         Vetor q       = r.ProdutoVetorial(e1);
24         double v      = raio.direcao.ProdutoEscalar(q);
25         if (v < 0 || u + v > div) continue;
26
27         double t      = fabs(e2.ProdutoEscalar(q) / div);
28         if (t < resultado.distancia || !resultado.houveColisao)
29         {
30             resultado.houveColisao = true;
31             resultado.distancia = t;
32         }
33     }
34
35     return resultado;
36 }
```

Nas linhas 6 e 7, uma estrutura “Colisao” é inicializada como não havendo colisão. Ela guarda sempre o estado atual da colisão durante o programa, e a menor distância encontrada. No teste na linha 28, a estrutura é atualizada, caso seja detectada uma colisão mais próxima àquela que já estiver guardada, ou se ocorrer a primeira colisão.

Pode-se observar, nas linhas 12, 13, 14, 19 e 23, o cálculo das equações de 7 a 11, respectivamente. Na linha 16, é computado o valor do divisor da fração da equação 6, que, se for muito próximo de zero, significa que o raio pertence ao plano do triângulo: não é considerada colisão. Usa-se uma margem de erro para tal decisão devido à precisão limitada dos números de ponto flutuante.

Nas linhas 20, 24 e 27, pode-se observar o cálculo da solução da equação 6, denotado por t, u e v. O programa descarta a colisão logo que detecta que algum desses fatores não está de acordo com as condições A e B, acima. No entanto, para a condição B, é usado o teste contra o fator div, ao invés do valor um, por se tratar de uma otimização: adia-se o cálculo de divisão para a linha 27, mas isso não afeta o resultado.

1.2 Estrutura de aceleração

Ainda que um algoritmo seja útil para se resolver um problema, ele pode não ser eficiente, nem, talvez, prático. Um algoritmo escrito em sua forma mais básica pode se tornar muito custoso, em termos de tempo ou memória do computador, para ser usado com grandes quantidades de dados de entrada, inviabilizando sua aplicação. Ziviani (1999) mostra que determinadas classes de algorítmos, mesmo com um tamanho de entrada pequeno, podem tomar bastante tempo de computação.

Muitas vezes, porém, há uma alternativa que consiste em organizar os dados previamente, e reutilizar esta forma ordenada em todas as execuções do algoritmo. A intenção é de que isso acelere a solução ao se utilizar resultados parciais previamente calculados, e, assim, omitir os cálculos que os gerariam na hora. É especialmente eficaz quando um mesmo cálculo é efetuado repetidamente, apenas para gerar os mesmos resultados todas as vezes, e, portanto, seria possível usar-se da resposta diretamente, caso ela estivesse disponível. É uma troca entre tempo e espaço, já que os cálculos são acelerados, ao custo de se preservar tais resultados em memória.

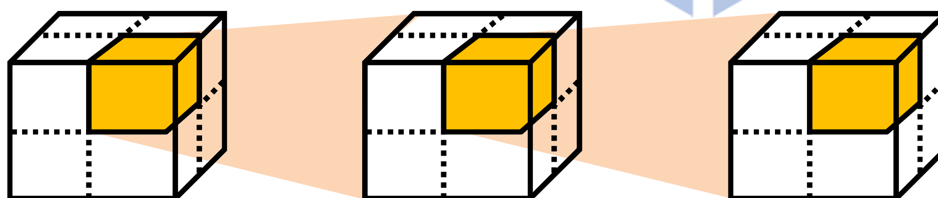
Outras vezes, o algoritmo pode se beneficiar quando os dados de entrada podem ser ordenados segundo algum critério, facilitando, por exemplo, a busca de uma informação específica. Nesse caso, chama-se, aqui, a forma como esses dados foram ordenados, de estrutura de aceleração.

Um estudo conduzido por Ziviani (1999) apresenta um exemplo de algoritmo, que tem por objetivo encontrar o menor e o maior valor presentes em uma sequência, em que ele estrutura previamente a lista de dados de entrada em duas outras listas especiais, a fim de reduzir a quantidade de cálculos no corpo do algoritmo. Demonstrou que, utilizando-se deste método, o algoritmo acelerado, em qualquer caso de entrada, tomava menos tempo que outros algorítmos com o mesmo objetivo, mas que não utilizavam a estrutura de aceleração.

Octree

Uma das formas de se acelerar um algoritmo que lida com dados espaciais é usar uma estrutura de particionamento de espaço. Aqui, foi escolhida a estrutura *octree* (PHARR e FERNANDO, 2005). Esta estrutura é formada por uma hierarquia de células cúbicas, cada uma com exatamente oito subcélulas, daí o nome. As subcélulas são oito cubos, de arestas da metade do tamanho da aresta da célula-mãe, dispostos em cada canto dela, a fim de preencherem o mesmo volume, como pode ser visto na figura 5. As subcélulas são chamadas octantes, por dividirem o espaço em oito partes iguais.

Figura 5 – Visualização de uma *octree*.



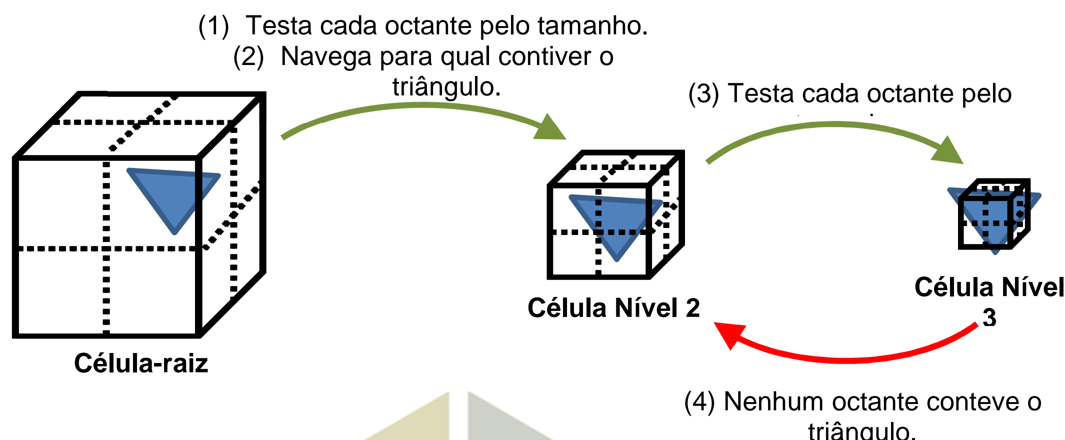
Fonte: Próprio autor

A construção da hierarquia da *octree* se inicia com uma célula-raiz cúbica de um tamanho de aresta que envolva todo o modelo tridimensional ao qual será aplicada. A seguir, para cada triângulo do modelo, verifica-se se ele cabe totalmente em um dos seus octantes. Caso isso se verifique, o algoritmo navega para dentro de tal octante, e parte para verificar, recursivamente, se o triângulo cabe em alguma de suas subcélulas. O tratamento desse triângulo termina quando ele não cabe em nenhum octante, e, nesse caso, ele é adicionado à lista de triângulos pertencentes à célula atual.

Também é possível parar o algoritmo quando a profundidade se torna maior que certo número de células, como válvula de escape da recursividade, adicionando o triângulo à célula de profundidade anterior.

O algoritmo termina quando todos os triângulos do modelo foram contabilizados. Um exemplo visual é mostrado pela figura 6.

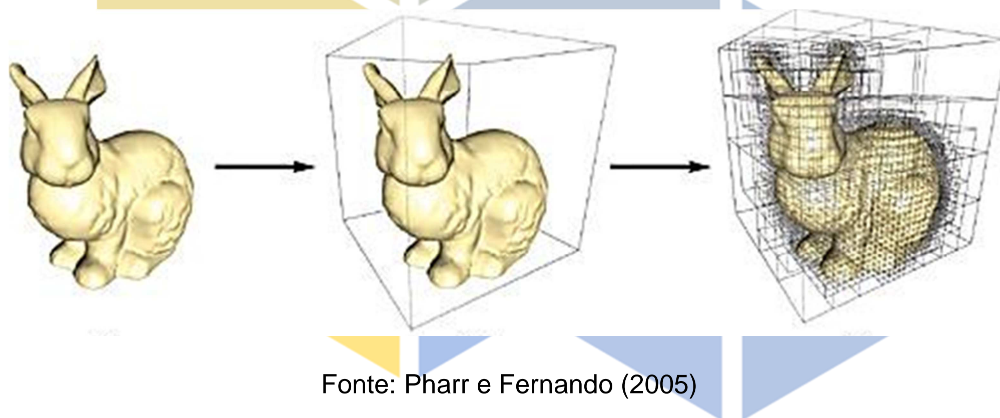
Figura 6 – Exemplo de construção de uma *octree*



Fonte: Próprio autor

Um exemplo de *octree* totalmente construída ao redor de um modelo é mostrado pela figura 7.

Figura 7 – Exemplo de aplicação de uma *octree* a um modelo.



Fonte: Pharr e Fernando (2005)

Detecção de colisão com *OCTREE*

Com a estrutura construída, ela pode ser reservada na memória e ser usada em todas sessões subsequentes de detecção de colisão. O algoritmo final de colisão raio-triângulo usa os mesmos passos que o algoritmo básico. A *octree*, por sua vez, permite acelerar esse cálculo porque descarta certos triângulos numa fase mais ampla, antes que a etapa mais fina e complexa seja executada.

O algoritmo acelerado é semelhante a uma busca *n*-ária, com *n* igual a 8. Ele é recursivo e começa com a célula-raiz. O raio é testado com cada octante cúbico da célula atual, e, se for constatada colisão raio-cubo, como discutido em Owen (1998), o algoritmo navega para dentro do octante e repete o teste. Por fim, ele executa o teste de colisão fino raio-triângulo com todos os triângulos pertencentes às células navegadas.

O algoritmo permite ganhos de velocidade, como será visto adiante, porque se um dado octante for descartado, todas as suas subcélulas também serão, por definição de hierarquia.

Profundidade e afinidade da OCTREE

A profundidade de uma célula é determinada pelo número de células-mãe que estão acima dela. A célula-raiz está no nível de profundidade um, seus octantes estão no nível dois, os octantes dos octantes estão no nível três, e assim por diante.

Pode-se chamar certo conjunto de triângulos afim da *octree* quando grande parte deles se acomoda em células de níveis profundos, proporcionando, assim, grandes omissões de galhos da estrutura quando o

algoritmo de colisão estiver executando. Pode-se, ainda, considerar a afinidade como: baixa, quando quase a totalidade dos triângulos permanece no primeiro nível – ou seja, na raiz –; média, quando a maioria dos triângulos fica nos primeiros três níveis; e, alta quando a maioria dos triângulos se encaixa do nível quatro para frente.

2 PROPOSTA DO TRABALHO

Este artigo se propõe a avaliar a diferença de *performance* temporal com relação aos algorítmos de detecção de colisão básico e detecção de colisão com *octree*, com a finalidade de observar sua aplicabilidade em um jogo digital. Este artigo não tem o objetivo de avaliar a *performance* de construção das estruturas de dados, nem do consumo de memória que podem ocasionar.

O método de avaliação será uma medição, de acordo com técnicas revisadas em Jain (1991). O objetivo é analisar a diferença entre os tempos necessários de cada algoritmo para concluir o mesmo resultado, com os mesmos dados de entrada, de vários casos-teste. O sistema em que a medição será feita é um computador executando o sistema operacional *Windows* 7 de 64 *bits*, com um processador Intel Core i5-3210M de 2,5 GHz, utilizando o compilador G++, versão 4.8.1, no nível de otimização O3, com medida de tempo de alta resolução fornecida pela função *QueryPerformanceCounter* do *Windows*. A métrica para a avaliação será o tempo necessário para executar-se cada algoritmo dez mil vezes em cada caso-teste, com um raio diferente em cada teste, de posição e direção aleatórias.

2.1 Parâmetros da avaliação

A seguir, serão discutidos os parâmetros que podem afetar o desempenho da avaliação.

Velocidade do processador

Um processador mais rápido executará os mesmos testes em menos tempo, mas esta avaliação se preocupa apenas com a diferença entre as medições. Dada as complexidades computacionais dos algorítmos, de acordo com Ziviani (1999), a razão entre as medições deve permanecer a mesma, independentemente da velocidade do sistema.

Nível de otimização e qualidade do compilador

Do mesmo modo ao item anterior, a razão entre as medições deve permancer a mesma, independentemente da qualidade do código gerado.

Prioridade do processo no sistema operacional

Dependendo das condições de uso do sistema operacional, o programa do algoritmo pode ser interrompido momentaneamente, levando a parecer que toma mais tempo do que realmente precisa. Pode ser amenizado executando-se o programa em prioridade de tempo real, ou realizando diversos testes repetidos, a fim de que neutralize erros ocasionais.

Casos-teste

Certos casos-teste podem não apresentar diferença suficiente na medição para julgar um algoritmo como melhor em detrimento a outro. Não há uma solução que satisfaça completamente qualquer caso-teste; logo, serão avaliados diversos tipos de casos-teste para demonstrar as vantagens e desvantagens de cada.

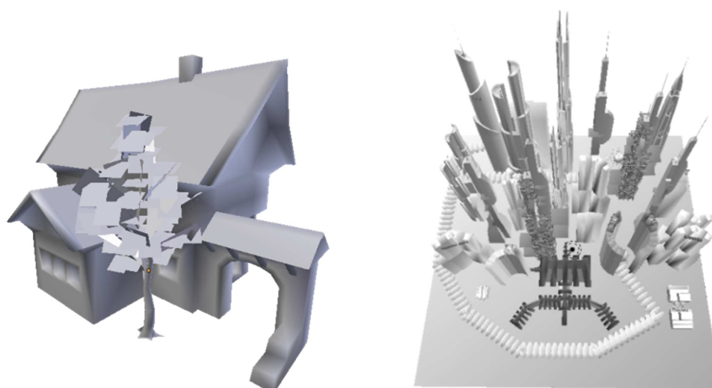
2.2 Fatores de estudo

Os casos reais aos quais os algorítmos podem ser submetidos variam em diversos fatores, como o tamanho da entrada, ou a afinidade dos dados com a estrutura de aceleração. Estes dois critérios serão avaliados por meio de casos-teste aleatórios com variação nesses fatores: a cada etapa, serão feitas medições com um número de triângulos gradualmente maior, e um tamanho de triângulo gradualmente menor. A variação no número de triângulos aplica-se ao teste de tempo relacionado ao tamanho de entrada, enquanto a variação no tamanho dos triângulos torna possível observar o comportamento do algoritmo relacionado à afinidade dos dados com a estrutura *octree.*

Serão avaliados, também, casos-teste com dados típicos de jogos digitais, por meio de modelos criados por artistas, disponíveis em TF3DM (2014), para verificar a aplicabilidade em casos reais de uso. Os dois modelos usados são chamados de *Building*, uma casa com 1.567 triângulos, e *SciFiCity*, uma cidade com 118.026 triângulos, como pode-se ver na figura 8.

Todos os casos-teste, incluindo os aleatórios, foram esticados ou encolhidos para caberem completamente num cubo de cem unidades de aresta, para padronização e melhor aproveitamento da estrutura *octree*.

Figura 8 – Os modelos *Building* e *SciFiCity.*



Fonte: TF3DM (2014)

3 DESENVOLVIMENTO DO TRABALHO

A avaliação foi desenvolvida a partir de classes de objetos representando conceitos geométricos, e uma série de funções que efetuam o algoritmo e executam casos-teste.

Os resultados são tabulados automaticamente pelo programa, e escritos para um arquivo em disco. A seguir, são discutidos os resultados obtidos.

3.1 Resultados

Os resultados dos testes mostram que o algorítmo com *octree* pode diminuir o tempo de execução consideravelmente, quando o número de triângulos é alto, e a afinidade dos dados com a estrutura também. No entanto, para casos-teste com entradas pequenas, o algoritmo com *octree* mostrou-se pior do que o algoritmo básico. Isso porque está incorporada, ao tempo de execução, a navegação pela estrutura, além do teste de intersecção.

Com um volume suficiente de triângulos, como pode ser visto nos próximos tópicos, o tempo de navegação torna-se ínfimo comparado ao teste de intersecção. A partir disso, conclui-se que o algoritmo com *octree* é mais adequado a casos com modelos complexos, e quando se pretende realizar grandes quantidades de testes.

Em todos os testes realizados, a detecção de colisão sinalizou exatamente o mesmo resultado para ambos os algorítmos: não houve erro ou imprecisão ao utilizar-se o algoritmo com *octree*.

Caso-teste aleatório de baixa afinidade

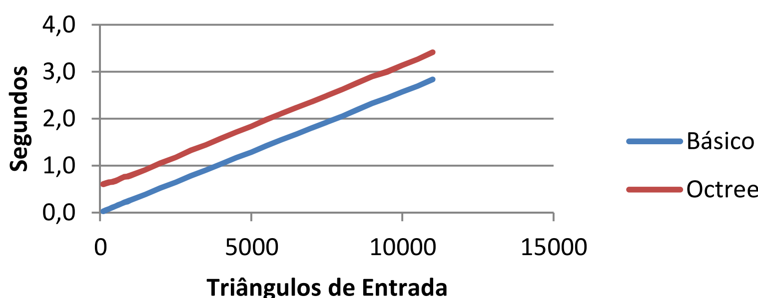
O primeiro caso-teste avaliado gerou triângulos aleatórios de qualquer tamanho que coubessem em um cubo de cem unidades de aresta. A figura 9 mostra a afinidade dos dados do caso-teste com a *octree*. A afinidade é baixa, como pretendido. No teste de intersecção fino, em média, 98,95% de todos os triângulos foram considerados pelo algoritmo acelerado – quase a mesma quantidade do algoritmo básico.

Figura 9 – Caso-teste aleatório de baixa afinidade: gráfico de triângulos por profundidade

Fonte: Próprio autor

A figura 10 demonstra o tempo de execução dos algorítmos durante dez mil testes. Nota-se que, para este caso-teste de baixa afinidade, o tempo de execução do algoritmo com *octree* permanece sempre pior do que o algoritmo básico, e a porcentagem de triângulos descartados é baixíssima. Portanto, a *octree* não é adequada para esta situação.

Figura 10 – Caso-teste aleatório de baixa afinidade: gráfico de tempo de execução

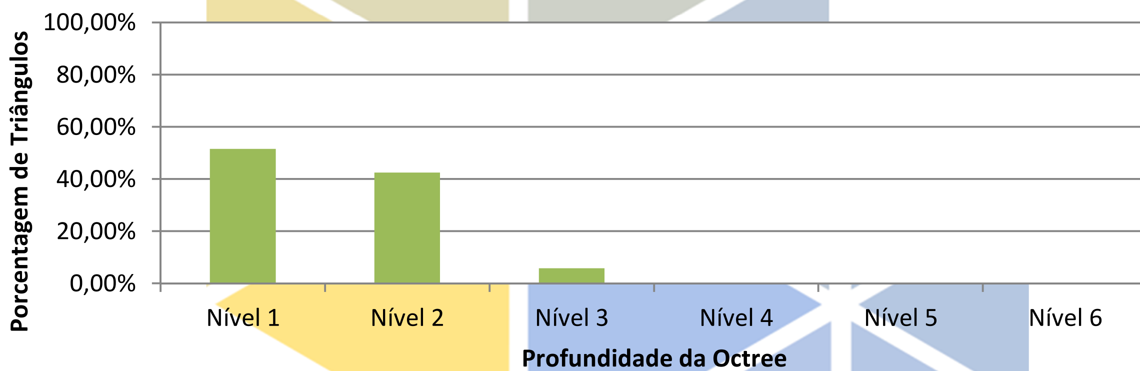


Fonte: Próprio autor

Caso-teste aleatório de média afinidade

O segundo caso-teste avaliado gerou triângulos aleatórios de qualquer tamanho que coubesse em um cubo de trinta unidades de aresta. A seguir, foram deslocados para posições aleatórias, para que preenchessem um cubo de cem unidades de aresta. A intenção é que, com triângulos menores e mais espalhados, se conseguisse obter uma afinidade maior com a *octree*. A figura 11 mostra a afinidade dos dados: ela pode ser considerada média, em relação aos outros casos-teste. No teste de intersecção fino, em média, o algoritmo acelerado considerou 66,45% de todos os triângulos – dois terços da quantidade do algoritmo básico. Nota-se uma melhoria em relação ao caso-teste de baixa afinidade.

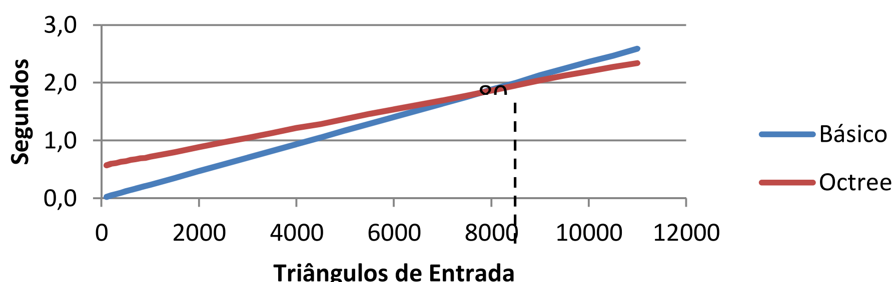
Figura 11 – Caso-teste aleatório de média afinidade: gráfico de triângulos por profundidade



Fonte: Próprio autor

A figura 12 mostra o tempo de execução dos algorítmos durante dez mil testes. Verifica-se que o algoritmo com *octree* é pior que o algoritmo básico para entradas com menos de cerca de oito mil triângulos. No entanto, a partir desta quantidade, o algoritmo com *octree* começa a se tornar, conforme aumenta a entrada, cada vez melhor. Com dez mil triângulos, a aceleração é de 1,07 vez.

Figura 12 – Caso-teste aleatório de média afinidade: gráfico de tempo de execução.

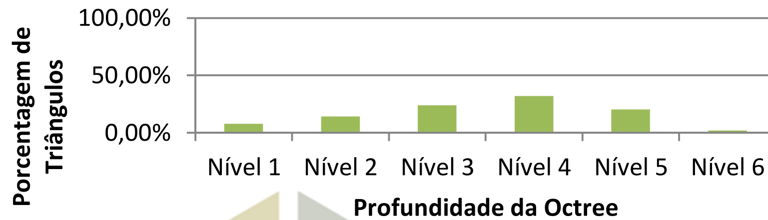


Fonte: Próprio autor

Caso-teste aleatório de alta afinidade

O terceiro caso-teste avaliado gerou triângulos aleatórios de um tamanho que coubesse em um cubo de cinco unidades de aresta. Novamente, a seguir, os triângulos foram deslocados aleatoriamente para preencher um cubo de cem unidades de aresta, com as mesmas intenções do caso-teste de média afinidade do tópico anterior. A figura 13 mostra a afinidade dos dados: ela é alta, pois a maioria dos triângulos se encaixou em níveis mais profundos da *octree*, promovendo, assim, o descarte de grandes volumes, de uma só vez, durante o algoritmo. Em média, apenas 15,91% de todos os triângulos foram considerados no teste de intersecção fino: uma grande melhoria comparada aos casos-teste anteriores.

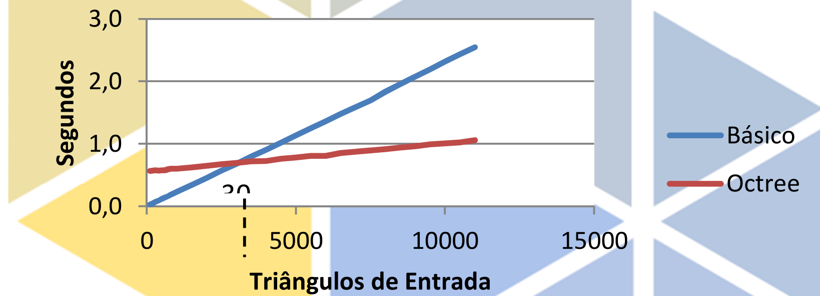
Figura 13 – Caso-teste aleatório de alta afinidade: gráfico de triângulos por profundidade



Fonte: Próprio autor

A Figura 14 mostra o tempo de execução dos algorítmos durante dez mil testes. Verifica-se que o algoritmo com *octree* é pior que o algoritmo básico nos casos com menos de três mil triângulos de entrada, mas proporciona uma aceleração considerável quando esse número aumenta. Com dez mil triângulos, a aceleração é de 2,3 vezes.

Figura 14 – Caso-teste aleatório de alta afinidade: gráfico de tempo de execução

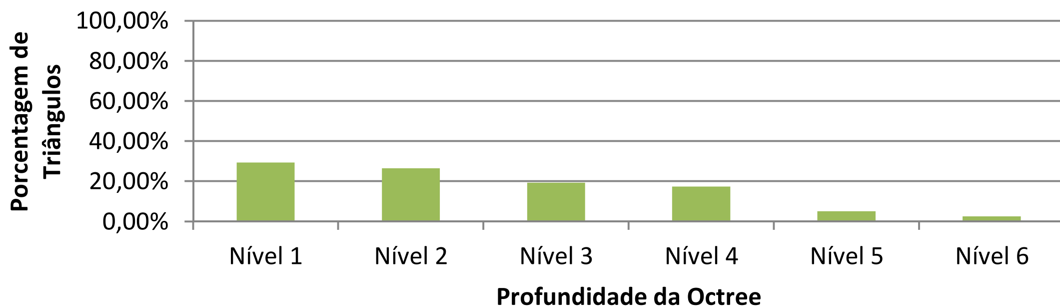


Fonte: Próprio autor

Caso-teste realista do modelo building

Este caso-teste avaliou a *performance* do algoritmo em relação ao modelo *Building*, criado por artista. A figura 15 mostra a afinidade deste caso-teste com a estrutura *octree*: ela pode ser considerada média. O algoritmo acelerado considerou no teste de intersecção fino, 40,91% de todos os triângulos.

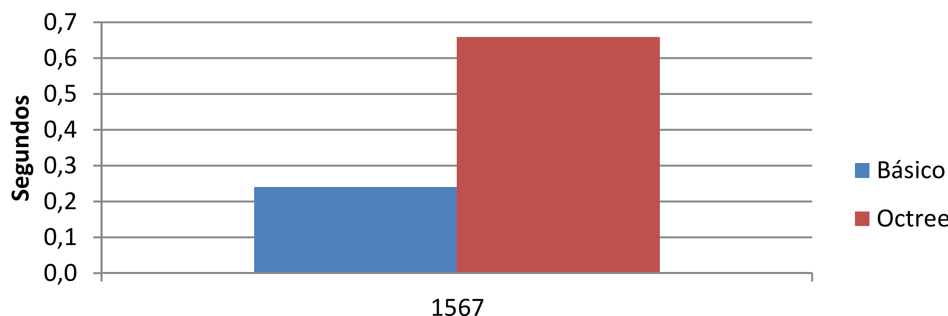
Figura 15 – Caso-teste realista do modelo *Building*: gráfico de triângulos por profundidade.



Fonte: Próprio autor

A figura 16 mostra o tempo de execução dos algorítmos durante dez mil testes. Nota-se que, como este caso-teste tem um baixo número de triângulos, o algoritmo com *octree* se mostrou pior do que o algoritmo básico: ele tomou 2,73 vezes mais tempo.

Figura 16 – Caso-teste realista do modelo *Building*: gráfico de tempo de execução

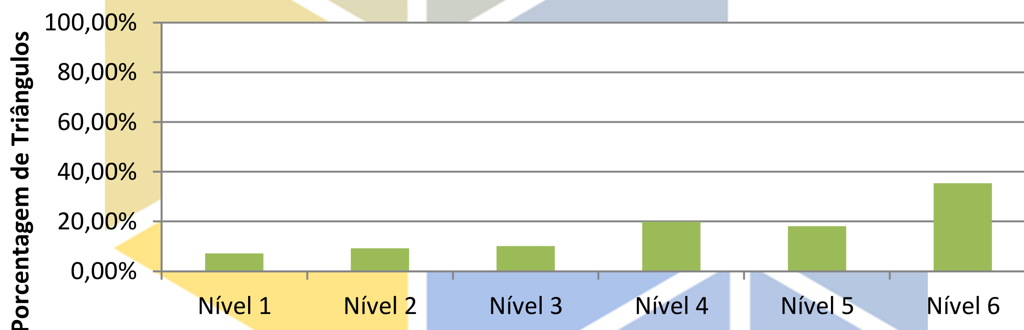


Fonte: Próprio autor

Caso-teste realista do modelo Scificity

Este caso-teste avaliou a performance do algoritmo em relação ao modelo *SciFiCity*, criado por artista. A figura 17 mostra a afinidade do modelo com a estrutura *octree*: ela pode ser considerada altíssima. 12,35% de todos os triângulos foram considerados no teste de intersecção fino.

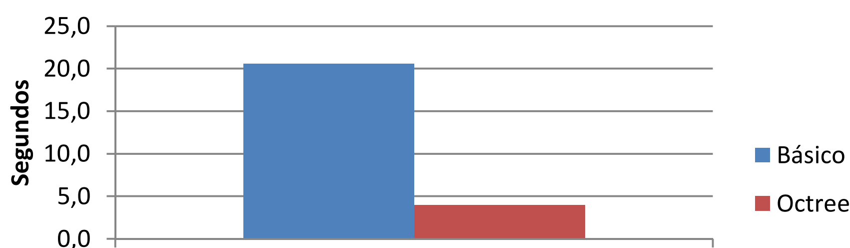
Figura 17 – Caso-teste realista do modelo *SciFiCity*: gráfico de triângulos por profundidade



Fonte: Próprio autor

A figura 18 mostra o tempo de execução dos algorítmos durante dez mil testes. Verifica-se que, devido à grande quantidade de triângulos e à alta afinidade com a estrutura, o algoritmo com *octree* se mostrou consideravelmente melhor do que o algoritmo básico: ocorreu uma aceleração de 5,2 vezes.

Figura 18 – Caso-teste realista do modelo *SciFiCity*: gráfico de tempo de execução



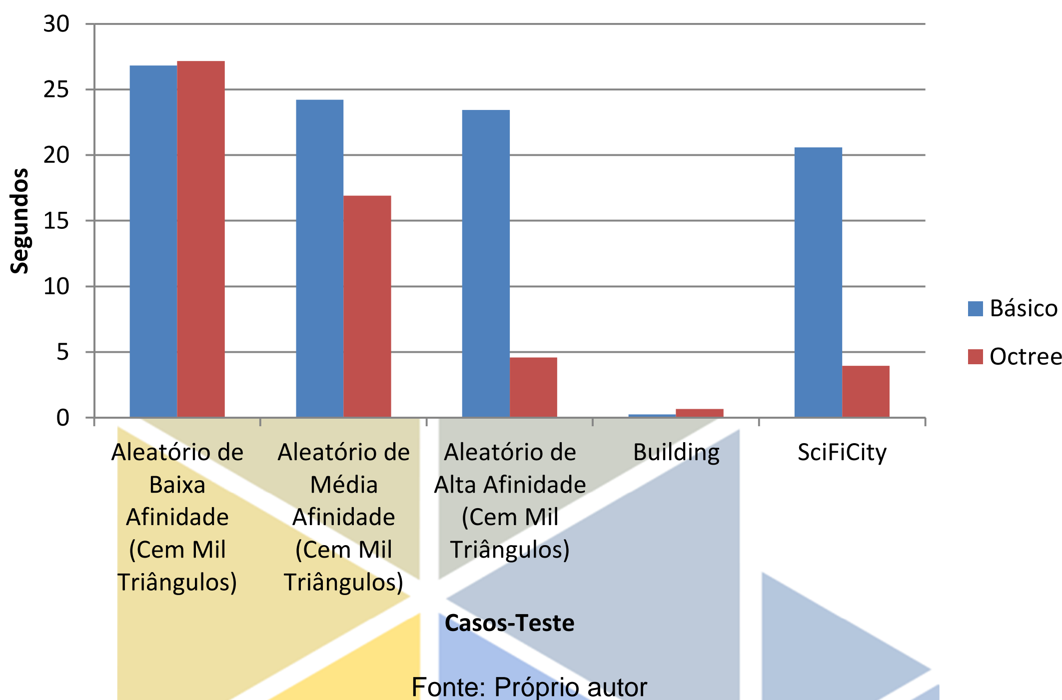
Fonte: Próprio autor

Resumo dos resultados

Os dados dos tópicos anteriores podem gerar informações melhores se comparados lado-a-lado. A

Figura **19** mostra um resumo dos tempos de execução dos algorítmos durante dez mil testes, em todos os casos avaliados. Pode-se notar uma tendência à aceleração quando a afinidade dos modelos com a *octree*, bem como o número de triângulos, aumenta.

Figura 19 – Gráfico resumido do tempo de execução dos casos-teste



Fonte: Próprio autor

4 CONSIDERAÇÕES FINAIS

Os resultados apresentados no capítulo anterior mostram que, dependendo das circunstâncias, o algoritmo com *octree* é muito recomendável para algorítmos de detecção de colisão.

No caso-teste de alta afinidade, o algoritmo com *octree* proporcionou aceleração a partir de 3.000 triângulos de entrada, e a razão de tempo entre o algoritmo básico e este aumenta em uma vez a cada 4.000 triângulos. Portanto, pode-se dizer que, quanto maior a quantidade de triângulos, mais rápido o algoritmo com *octree* solucionará o problema da detecção de colisão, em comparação ao algoritmo básico. A *octree* é uma estrutura muito recomendável para este caso.

Nos casos-teste de média ou baixa afinidade, o algoritmo com *octree* se mostrou menos capaz de acelerar a detecção de colisão, e por isso, não é recomendável nesses casos.

Ainda assim, modelos criados por artistas, que são ultimamente usados nos jogos, tendem a ter alta afinidade com a estrutura e, portanto continua recomendável utilizar-se a *octree*. No caso-teste do modelo *SciFiCity*, por exemplo, nota-se uma aceleração de 5,2 vezes, em comparação ao algoritmo básico: a *octree* foi extremamente eficaz como estrutura aceleradora.

Finalmente, é preciso cautela ao considerar-se a *octree* para modelos com baixo número de triângulos. Essa estrutura precisa de tempo para navegação, e esse tempo só é compensado com um alto número de triângulos de entrada. O algoritmo com *octree* não é recomendável para modelos com menos de poucos milhares de triângulos.

No entanto, ainda há margem para pesquisa neste assunto: a estrutura *octree* pode consumir muita memória do computador – seria interessante projetar-se algum método que diminuísse tal consumo. O algoritmo aqui apresentado, também, apenas trabalha com raios e triângulos – seria necessário desenvolver outras técnicas para colisões entre novas formas geométricas. De qualquer modo, jogos digitais tridimensionais que utilizam algorítmos básicos de colisão poderão enriquecer seus mundos virtuais e

proporcionar experiências mais realistas, precisas e elaboradas, ao implementarem estruturas de aceleração, como a *octree*, quando adequado.

REFERÊNCIAS

HUIZINGA, J. **Homo ludens**. São Paulo: Perspectiva, 2000.

JAIN, R. **The art of computer systems performance analysis**. [S.l.]: John Wiley, 1991.

MÖLLER, T. **A fast triangle-triangle intersection test**,Disponivel em: <http://knight.temple.edu/~lakaemper/courses/cis350_2004/etc/moeller_triangle.pdf>. Acesso em: 12 outubro 2014.

MÖLLER, T. ; TRUMBORE, B. **Fast, minimum storage ray/triangle intersection**, 1997. Disponivel em: <http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf>. Acesso em: 12 outubro 2014.

OWEN, G. S. **Ray-Box intersection**, 1998. Disponivel em: <https://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm>. Acesso em: 12 outubro 2014.

PHARR, M.; FERNANDO, R. **GPU Gems 2**. 2nd. ed. Taunton, Massachusetts: Pearson Education, 2005. Disponivel em: <https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_frontmatter.html>. Acesso em: 27 outubro 2014.

SEGURA, R. J.; FEITO, F. R. WSCG: International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision. **Algorithms to test ray-triangle intersection:** Comparative Study, 7 Fevereiro 2001. Disponivel em: <http://iason.zcu.cz/WSCG2001/Papers_2001/R75.pdf>. Acesso em: 12 outubro 2014.

TF3DM. **3D Models for free**, 2014. Disponivel em: <http://tf3dm.com/>. Acesso em: 12 outubro 2014.

THE ENTERTAINMENT SOFTWARE ASSOCIATION. **Essential facts about the computer and video game industry**, abril 2014. Disponivel em: <http://www.theesa.com/facts/pdfs/ESA_EF_2014.pdf>. Acesso em: 12 outubro 2014.

WELLER, R. **New geometric data structures for collision detection and haptics**. [S.l.]: Springer, 2013.

ZIVIANI, N. **Projeto de algorítmos:** com implementações em Pascal e C. 4.ed. São Paulo: Pioneira, 1999.

Henrique Lorenzi
Cursando Tecnologia em Jogos Digitais na FATEC Americana (previsão de conclusão para junho de 2015). Atua na área de Programação e Engenharia de Software, trabalhando com as Linguagens C, C++, C#, Objective-C, Verilog, Java, HTML, CSS e Javascript. Desde 2013 é sócio da empresa POCKET TRAP Games e Desenvolvimento Ltda., onde já desenvolveu os jogos Ninjin, Lub & Dub, Hell Broker e Indie Speed Run. Atualmente está trabalhando em novos jogos para PC, iOS, Android, Playstation e XBox.
Contato: hlorenzi12@hotmail.com
Fonte: CNPQ – Curriculo Lates

Vitor Brandi Junior
Graduado em Tecnologia em Processamento de Dados pela Universidade Metodista de Piracicaba (1989) e mestre em Gerenciamento de Sistemas de Informação pela Pontifícia Universidade Católica de Campinas (1997). Atualmente é professor do Instituto Federal de São Paulo - Campus Capivari, onde também responde pela coordenação do curso de Tecnologia em Análise e Desenvolvimento de Sistemas. Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: Algorítmos e Estruturas de Dados, Banco de Dados Orientação a Objetos, Java e Metodologias para Desenvolvimento de Sistemas.
Contato: vitor.brandi@gmail.com
Fonte: CNPQ – Curriculo Lates

Justificativa dos autores
"O artigo aborda tecnologias que permitem acelerar o desempenho dos jogos, possibilitando elaborá-los com maior quantidade de detalhes, o que aguça o interesse do jogador e, consequentemente, ajuda no sucesso comercial do produto".