

Universidade Federal do Pampa

Ezequiel Butzke Pydd

Avaliação de algoritmos de detecção de colisão em jogos

Alegrete

2015

Ezequiel Butzke Pydd

Avaliação de algoritmos de detecção de colisão em jogos

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Jean Felipe Patikowski Cheiran

Alegrete

2015

Ezequiel Butzke Pydd

Avaliação de algoritmos de detecção de colisão em jogos

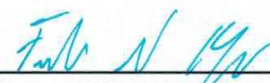
Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal do Pampa como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Trabalho de Conclusão de Curso defendido e aprovado em 30 de novembro de 2015


Banca examinadora:



Jean Felipe Patikowski Cheiran
Orientador



Professor Fábio Natanael Kepler
UNIPAMPA



Professor Marcelo Resende Thielo
UNIPAMPA

*Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.*

*“Ninguém baterá tão forte quanto a vida.
Porém, não se trata de quão forte pode
bater, se trata de quão forte pode ser
atingido e continuar seguindo em frente.
É assim que a vitória é conquistada.”*
Rocky Balboa

Resumo

Detecção de colisão trata do problema de sobreposição de objetos em um ambiente virtual, é uma das principais ferramentas para simulação da física do mundo real em um mundo virtual composto das mais diversas formas geométricas e polígonos. Jogos são aplicações que demandam muito processamento e a detecção de colisão é responsável por boa parte dessa demanda. Esse trabalho tem como objetivo realizar e analisar testes de desempenho e testes de precisão de colisão usando jogadores. Para ambos os testes foram implementados dois jogos em *Java*. Os algoritmos que foram testados são distância euclidiana, distância de *Manhattan*, sobreposição de retângulos, *pixel perfect*, detecção hierárquica e sobreposição de retângulos combinado com *pixel perfect*. Os testes de desempenho foram feitos coletando o tempo que demoram para executar a rotina de detecção de colisão e também quanto tempo demoram para percorrer um percurso pré-programado com diversos números de elementos para colidir. O jogo para os testes de precisão consiste de um jogo questionário onde o jogador, conforme colide com objetos, responde o quão realista a colisão pareceu. Os resultados mostram uma grande eficiência do método de detecção hierárquica em todos os testes realizados, enquanto *pixel perfect* se mostra extremamente oneroso em questão de desempenho. O método de sobreposição de retângulos foi o pior para demonstrar realidade para o jogador.

Palavras-chave: Detecção de colisão. Precisão de colisão. Colisão em jogos.

Abstract

Collision Detection deals with the objects overlap problem in a virtual environment, it is a major tool to simulate real world physics in a virtual world composed of various geometric shapes and polygons. Games are applications that demand a lot of processing and collision detection is responsible for much of this demand. This work aims to conduct and analyze performance tests and collision accuracy tests using players. For both tests were implemented two java games. The algorithms that will be tested are Euclidean distance, Manhattan distance, overlapping rectangles, pixel perfect, hierarchical detection and overlapping rectangles combined with pixel perfect. Performance tests were done collecting the time it takes to perform the collision detection routine and also how long it takes to go through a pre-programmed route with various numbers of elements to collide. The game for precision tests consists of a quiz game where the player, colliding with objects, answers how realistic was the collision. The results show a great efficiency of hierarchical detection in all tests while perfect pixel shown to be extremely costly in performance issue. The overlapping rectangles method has demonstrate the worst accuracy for the player.

Key-words: Collision detection. Collision accuracy. Collision in games.

Lista de ilustrações

Figura 1 – Duas esferas não colidindo e colidindo.	17
Figura 2 – Esqueleto para detector de colisão de personagem de <i>Team Fortress</i> . . .	19
Figura 3 – Falsa colisão.	19
Figura 4 – Total War Shogun 2.	21
Figura 5 – Formas geométricas simples servem como VE's de objetos deformados.	22
Figura 6 – Esquema de etapas do teste de sobreposição.	24
Figura 7 – Esfera extrudada.	25
Figura 8 – (a) Retângulos não colidindo e (b) retângulos colidindo.	26
Figura 9 – Distância euclidiana entre um centro de círculo ao outro.	27
Figura 10 – Comparação entre distância euclidiana (linha reta) e de <i>manhattan</i> (escada).	27
Figura 11 – Formação do losango usando distância de <i>manhattan</i> em um plano cartesiano.	28
Figura 12 – Esquema do método hierárquico utilizando distância euclidiana.	30
Figura 13 – Direções que personagem deve seguir nos testes.	34
Figura 14 – Representação do jogo-questionário.	35
Figura 15 – Representação questionário.	35
Figura 16 – Tempo para calcular detecção de colisão (em nanosegundos).	37
Figura 17 – Tempo para calcular detecção de colisão (em nanosegundos).	38
Figura 18 – Tempo para percorrer percurso (em nanosegundos).	39
Figura 19 – Tempo para percorrer percurso (em nanosegundos).	40
Figura 20 – Tempo para percorrer percurso (em nanosegundos).	40
Figura 21 – Resultado dos testes com usuários.	41

Lista de abreviaturas

VE volume envolvente

Sumário

1	INTRODUÇÃO	17
1.1	Motivação	18
1.2	Objetivos	19
1.3	Estrutura do trabalho	20
2	CONCEITOS BÁSICOS	21
2.1	Jogo	21
2.2	Volumes envolventes (VE)	22
2.3	Laço principal do jogo	22
2.4	Detecção de Colisão	23
2.5	Algoritmos de detecção de colisão	25
2.5.1	Sobreposição de retângulos	25
2.5.2	Distância euclidiana	26
2.5.3	Distância de Manhattan	27
2.5.4	Pixel Perfect	28
2.5.5	Método hierárquico	29
3	TRABALHOS RELACIONADOS	31
4	METODOLOGIA	33
4.1	Medida de desempenho	33
4.1.1	Ambiente de execução	34
4.2	Testes com usuários	34
5	ANÁLISE DE RESULTADOS	37
5.1	Testes de desempenho	37
5.2	Testes com usuários	41
6	CONCLUSÃO	43
	Referências	45

1 Introdução

Detecção de colisão refere-se ao problema computacional de verificar a sobreposição entre dois ou mais objetos, como por exemplo, dois modelos de caixas tridimensionais que estão se tocando em um mundo virtual. Ambientes virtuais fazem uso de conceitos de física (gravidade, atrito, substancialidade) através da detecção de colisão e resolução de colisão (MCSHAFFRY; GRAHAM, 2012), esses são problemas conhecidos de aplicações gráficas interativas como jogos eletrônicos e simulações de física. A resolução de detecção trata do que acontece após a detecção, como definir nova velocidade e direção para um determinado objeto colidente. A detecção e resolução de colisão são problemas intimamente ligados, não existe maneira de tratar uma colisão se não houver a detecção, por outro lado detecção sem o tratamento da colisão seria inútil. Este trabalho não irá tratar sobre resolução de colisão, no entanto é importante enfatizar a ligação dos problemas.

Sem o conceito de substancialidade, os objetos atravessariam uns aos outros, como se fossem fantasmas (RABIN, 2010). É quase impossível desenvolver um jogo eletrônico ou ambiente virtual sem detecção de colisão (COURSE, 2013), esta é responsável por ativar eventos do jogo/ambiente virtual para que a física ocorra, como uma bola que bate na parede e volta, efeitos de luz ou efeitos sonoro. O conceito de substancialidade nesses ambientes é implementado através da detecção de colisão, assim é possível saber quando um objeto A colide com um objeto B.

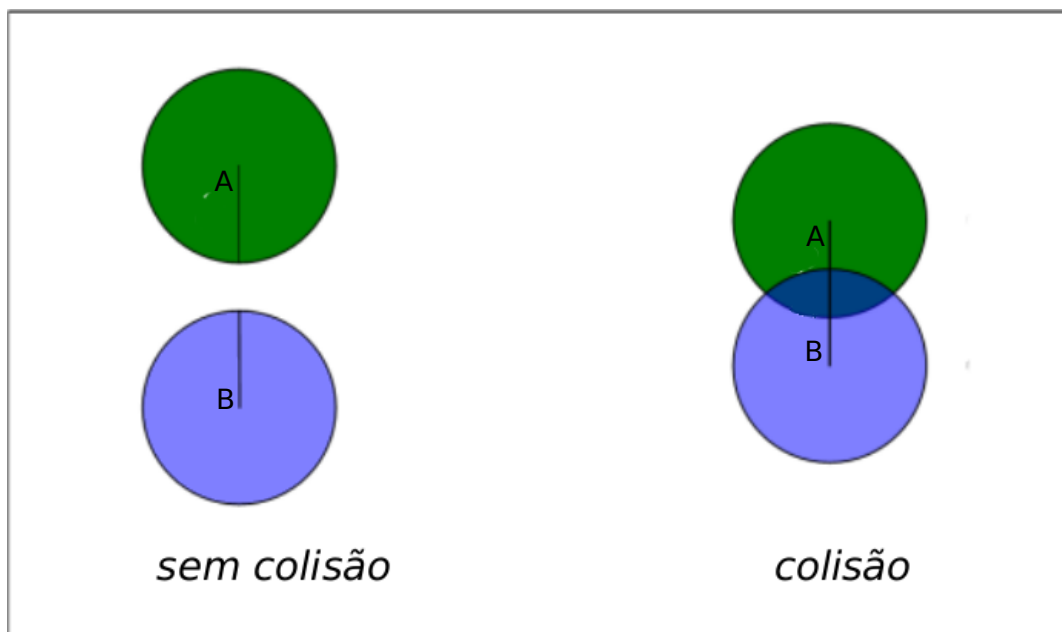


Figura 1 – Duas esferas não colidindo e colidindo.

Existem vários algoritmos para a solução do problema, que variam muito em ques-

tão de custo de processamento e precisão. Aplicações como simulações físicas necessitam de algoritmos mais robustos que garantem exatidão na detecção de colisão, no entanto consumindo mais processamento, por outro lado, jogos priorizam o desempenho utilizando algoritmos mais rápidos. Considerando um cenário onde existem vários objetos/personagens em movimento por trajetória desconhecidas, torna-se custoso identificar tantas colisões ocorrendo ao mesmo tempo. Isso deixa claro um dos desafios de desenvolver jogos, adquirir o máximo de desempenho possível utilizando algoritmos de detecção de colisão simples.

Neste trabalho é proposto a implementação de algoritmos de detecção de utilizando a linguagem de programação *Java*, os algoritmos são distância euclidiana (subseção 2.5.2), distância de *Manhattan* (subseção 2.5.3), sobreposição de retângulos (subseção 2.5.1), *Pixel Perfect* (subseção 2.5.4) e detecção hierárquica (subseção 2.5.5).

1.1 Motivação

Atualmente a detecção de colisão aparece como um gargalo em aplicações gráficas interativas, pois estas precisam manter taxas de *frames* altas e constantes (ROCHA, 2010), limitando muito o tempo para cálculos entre um *frame* e outro. Se tratando de uma aplicação com poucos objetos em um cenário, como os dois círculos mostrados na Figura 1 não é comum que a aplicação consuma muito processamento. No entanto, vários objetos se movendo, colidindo, ativando eventos, farão com que o poder de processamento necessário seja muito maior e assim tornar a aplicação onerosa (RABIN, 2010). Jogos eletrônicos são aplicações que necessitam constantemente da detecção de colisão, portanto é importante que as rotinas responsáveis por essa tenham bom desempenho. No esforço de ganhar desempenho em detecção de colisão em jogos a precisão do algoritmo usado pode deixar a desejar, métodos robustos que atingem um nível alto de precisão como uma malha de triângulos cobrindo o objeto em questão são inviáveis em razão de tempo, são muito lentos.

Como pode ser observado na Figura 2 a detecção de colisão é realizada usando formas geométricas simples. Apesar do ganho de desempenho, a precisão da colisão pode apresentar imperfeições como uma falsa colisão, isso acontece devido a forma geométrica não ser da mesma forma que o corpo do objeto ou personagem. É possível observar na Figura 3 que os retângulos usados para detectar colisão entre os aviões A e B estão se tocando, a detecção dessa colisão irá ativar eventos pré-programados como por exemplo o efeito de uma explosão, no entanto os aviões nunca se tocaram. Colisões falsas podem causar certa estranheza por parte do jogador e até deixa-lo frustrado.

A detecção de colisão é uma das principais partes de um jogo, pois para desenvolver um bom jogo é necessário possuir um sistema que responde bem a colisão (HARBOUR,



Figura 2 – Esqueleto para detector de colisão de personagem de *Team Fortress*.

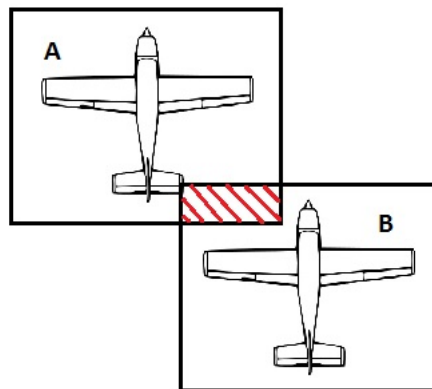


Figura 3 – Falsa colisão.

2014). Para isso acontecer o sistema precisa passar ao jogador uma boa percepção de realidade da colisão.

1.2 Objetivos

Como visto na [seção 1.1](#), dois problemas de detecção de colisão são apontados, o alto custo de processamento que um sistema de detecção de colisão muito robusto pode causar, e também o impacto negativo que as falsas colisões tem sobre um jogo.

Este trabalho tem como objetivo principal medir o desempenho dos algoritmos citados no [Capítulo 1](#), e analisar os resultados coletados. A análise será sobre o tempo que os métodos tomam para executar a detecção de colisão e então determinar qual desses têm melhor desempenho. Além de saber qual método possui melhor desempenho, deseja-se saber o impacto desses algoritmos conforme o crescimento do número de objetos no jogo. O segundo objetivo deste trabalho é identificar qual dos algoritmos transmite uma melhor percepção de realidade, para conseguir tais resultados pretende-se aplicar um questionário em usuários. O usuário irá observar o quão real a colisão parece em um pequeno jogo que será implementado para esse propósito e não terá conhecimento de que método de detecção de colisão está sendo usado. No fim do jogo ele irá julgar o quanto de realidade aquela colisão lhe passou atribuindo uma nota. O [Capítulo 4](#) explica de forma mais detalhada como os testes serão implementados.

Para a implementação dos métodos necessários para os testes decidiu-se utilizar a linguagem de programação *Java*, devido a maior facilidade de implementação que essa oferece em relação com linguagens como C ou C++. Java possui classes e métodos já implementados para trabalhar com imagens e *sprites* em jogos além de uma boa documentação ([DAVISON, 2005](#)).

1.3 Estrutura do trabalho

Este trabalho está estruturado da seguinte forma:

[Capítulo 2](#) introduz conceitos básicos necessários para o entendimento do problema de detecção de colisão em jogos, assim como também explica os algoritmos usados neste trabalho.

[Capítulo 3](#) fala sobre alguns trabalho que são semelhantes a este.

[Capítulo 4](#) descreve o que e como será usado para fazer a coleta de dados proposta pelo trabalho.

[Capítulo 5](#) demonstra e explica os resultados obtidos.

[Capítulo 6](#) descreve as considerações finais.

2 Conceitos Básicos

Este capítulo apresenta conhecimentos básicos necessários para o entendimento desse trabalho, como conceitos usados ao longo do texto e a explicação dos métodos que servirão para realizar o testes citados na [seção 1.2](#)

2.1 Jogo

Jogos não são apenas aqueles jogados em computadores e consoles, os jogos fazem parte da experiência humana muito tempo antes dos primeiros eletrônicos surgirem. Jogos são de grande fascínio para o ser humano, estão presentes por todo o planeta, existem em grande quantidade e em todas as culturas, alguns tão antigos quanto civilizações, um ótimo exemplo disso são os antigos jogos olímpicos gregos. Com tanta variação e quantidade, como se pode definir um jogo? Segundo (CHANDLER; CHANDLER, 2011) um jogo é uma atividade definida por desafios interativos, regras discerníveis e objetivos alcançáveis, como o jogo de xadrez. Jogos eletrônicos também se aplicam a esses três princípios, a [Figura 4](#) mostra um exemplo de jogo eletrônico, Total War shogun 2 desenvolvido por *The Creative Assembly* e publicado pela *SEGA*, um jogo de estratégia baseado em turnos e em tempo real que se passa no Japão entre os séculos quinze e dezessete.



Figura 4 – Total War Shogun 2.

Devido as possibilidades que a virtualização oferece, jogos eletrônicos podem ser muito mais complexos do que um jogo de tabuleiro, assim como Shogun 2 onde é possí-

vel controlar exércitos com milhares de soldados, frotas de barcos, administrar cidades, usar diplomacia para com os adversários e possui vários objetivos. Muito semelhante a mecânica do xadrez, no entanto com uma escala muito maior.

2.2 Volumes envolventes (VE)

Testar diretamente a geometria de objetos para detecção de colisão é geralmente muito caro (ERICSON, 2004). Os VE's são usados para fazer o teste de colisão com formas geométricas muito mais simples, como um círculo, retângulo, losango entre outros, do que os muitos polígonos que formam um corpo de um mundo virtual. VE's podem ser usados combinados com outras técnicas de colisão, comumente essa combinação acontece com um VE simples que cobre todo ou parte de um objeto, com um método mais complexo e preciso. O VE mais externo é usado como um teste mais simples que custa menos processamento, desse forma o método mais robusto só é usado caso o VE externo detectar a colisão.

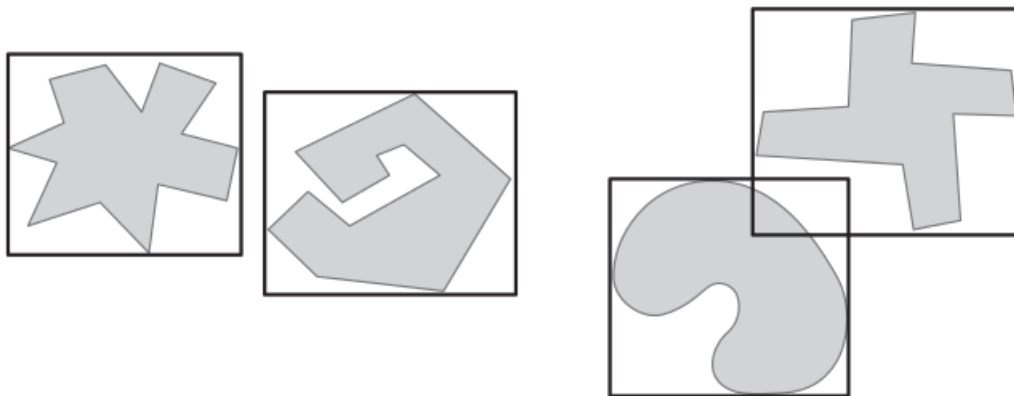


Figura 5 – Formas geométricas simples servem como VE's de objetos deformados.

Fonte: Ericson (2004, p. 76)

Outra maneira de utilizar um VE é usa-lo como um único detector de colisão para um determinado objeto, sendo isso prova suficiente para a colisão. É possível observar na Figura 5 figuras geométricas deformadas usando VE's, essas podem ser utilizadas como testes prévios para então usar testes mais precisos para evitar falsas colisões. Também pode ser usado como um único teste, no entanto isso deixa margem a erros de precisão.

2.3 Laço principal do jogo

Este conceito é importante para compreender este trabalho, pois explica porque a detecção de colisão acaba se tornando um gargalo de processamento. Isso se deve ao

fato que existe um tempo consideravelmente pequeno para que as funções de detecção de colisão ocorram. Assim como um vídeo, o jogo funciona com *frames* que são repintados na tela do dispositivo a cada determinado intervalo de tempo, para que o jogo possua uma boa taxa de *frames* aos olhos humanos, essa deve ser de trinta a sessenta *frames* por segundo. O laço principal é o laço que faz com que o jogo seja contínuo e atualizado a cada iteração, dentro dele estão todas funções que compõe o jogo. Abaixo é exemplificado um pseudocódigo de um laço principal retirado de [Rabin \(2010, p. 238\)](#).

```
1      1.Inicializacao de jogo
2      2.Loop de jogo principal
3          a.Inicializacao front_end
4          b.Loop front_end
5              i.Coletar dados de entrada (inputs)
6              ii.Renderizar a tela
7              iii.Atualizar estado front_end
8              iv.Ativar quaisquer mudancas de estado
9          c.Desligamento front_end
10         d.Inicializacao de nivel
11         e.Loop de jogo de nivel
12             i.Coletar dados de entrada (inputs)
13             ii.Executar IA
14             iii.Executar simulacoes fisicas
15             iv.Atualizar entidades de jogo
16             v.Enviar/receber mensagens de rede
17             vi.Atualizar etapa de tempo
18             vii.Atualizar estado de jogo
19         f.Desligamento de nivel
20     3.Desligamento de jogo
```

Como mencionado anteriormente a detecção de colisão deve ocorrer em um tempo muito restrito, isso pode parecer simples com poucos objetos em um cenário consideravelmente vazio, no entanto, com um grande número de objetos e personagens se movendo no cenário essa tarefa se torna muito mais difícil. Sendo pequeno o tempo para as funções de detecção de colisão, na maioria das vezes os jogos acabam por não usarem métodos complexos como uma malha de triângulos cobrindo o corpo do objeto, mas sim formas simples. Isto faz com que a detecção de colisão se torne muito mais simples e consequentemente rápida.

2.4 Detecção de Colisão

No mundo material não é possível que dois corpos diferentes ocupem o mesmo espaço ao mesmo tempo, em jogos isso é possível devido ao fato de que não existe matéria, o choque de dois corpos conforme as leis da física vai acontecer determinado por algoritmos

de detecção de colisão (SILVA; FEIJO; CLUA, 2010). Também segundo (RABIN, 2010) o conceito de continuidade temporal do jogo não existe, como a detecção de colisão só pode ser aplicada em períodos de tempo como visto na seção 2.3, a colisão só pode ser detectada quando já aconteceu (teste de sobreposição) ou quando irá acontecer (teste de interseção) As imagens abaixo exemplificam os dois métodos.

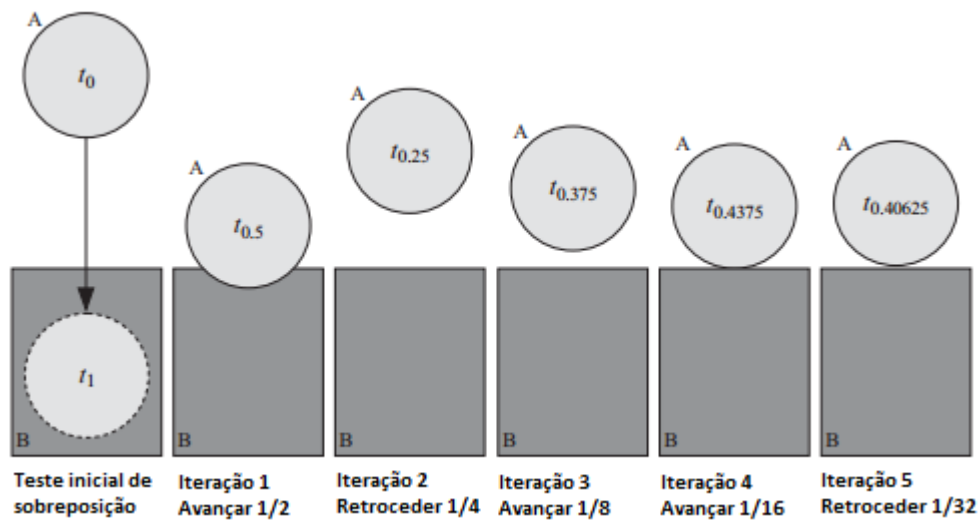


Figura 6 – Esquema de etapas do teste de sobreposição.

Fonte: Rabin (2010, p. 357)

A Figura 6 demonstra como o teste de sobreposição é realizado, neste teste a cada instante da simulação do cenário virtual os objetos são testados uns com os outros para determinar se houve sobreposição. Como é possível observar o objeto (a) no instante t_0 não toca o objeto (b), no entanto, com o passar de um instante da simulação o objeto (a) está tocando o objeto (b). É válido sustentar que na simulação de um jogo, o tempo não é contínuo, mas composto de instantes. Então o objeto (a) não irá andar até tocar no objeto (b) e parar, mas irá andar para frente a distância ordenada pela física da simulação e então a simulação é interrompida para testes de colisão, isso é basicamente o que será feito com um objeto até que a colisão seja detectada. Quando isso acontece a simulação é retrocedida um instante de tempo para determinar quando e onde foi o momento exato da colisão. Conforme a Figura 6 demonstra nas iterações, o objeto (a) é avançado e retrocedido no tempo até que o momento exato da colisão seja encontrado.

O teste de interseção testa se a colisão irá acontecer instantes antes da colisão realmente acontecer, isso é feito com a extrusão dos objetos da simulação. A extrusão de um objeto é grosseiramente falando, “esticar” o objeto, a Figura 7 mostra uma esfera extrudada, a tornando em um cilindro. Após a extrusão, os testes de colisão ocorrem da mesma maneira do que os teste de sobreposição, então no momento que a esfera da frente do cilindro detectar a colisão, significa que a esfera da parte traseira do cilindro irá

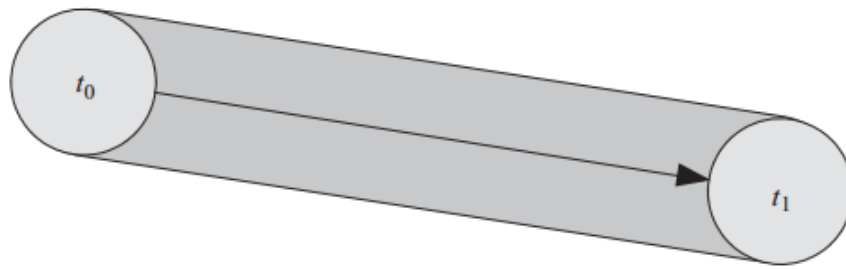


Figura 7 – Esfera extrudada.

Fonte: [Rabin \(2010, p. 358\)](#)

sofrer colisão. Deste instante em diante a simulação pode ser avançada de maneira mais cuidadosa para achar o ponto exato da colisão.

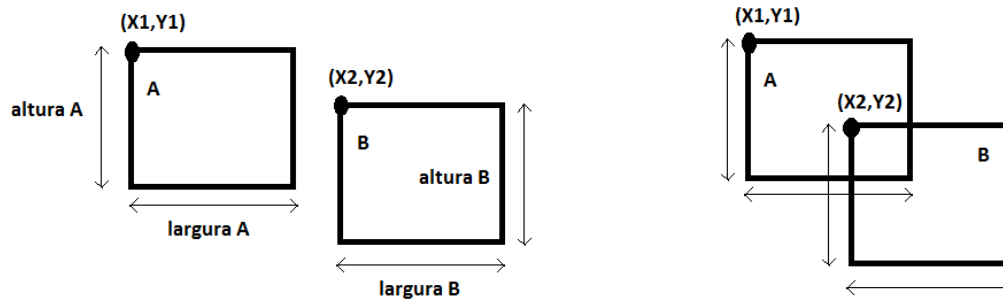
2.5 Algoritmos de detecção de colisão

2.5.1 Sobreposição de retângulos

Esse é um dos métodos mais elementares de detecção de colisão, o algoritmo faz uso das coordenadas dos eixos x e y de dois volumes envolventes em forma de retângulos para verificar se eles se tocam, testando se um dos cantos de um retângulo está dentro do outro. Para essa verificação é necessário testar se o x do primeiro retângulo está entre o x do segundo retângulo e o x + largura do segundo retângulo e também se o y do primeiro retângulo está entre do y do segundo retângulo e o y + altura do segundo retângulo ([AMATO, 1999](#)). Os retângulos que serão utilizados neste trabalho não são orientados, ou seja, não existe rotação do **VE** conforme o personagem do jogo se move.

Abaixo o pseudocódigo da sobreposição de retângulos.

```
1 Funcao Colisao{
2
3   retangulo1 , retangulo2
4
5   se ((retangulo1.X + retangulo1.Largura > retangulo2.X) &&
6     (retangulo1.X < retangulo2.X) &&
7     (retangulo1.Y + retangulo1.Altura > retangulo2.Y) &&
8     (retangulo1.Y < retangulo2.Y))
9     //houve colisao
10  senao
11    //nao houve colisao
12 }
```



(a) Y2 está entre Y1 e Y1 + altura A mas X2 não está entre X1 e X1 + largura A. (b) Y2 está entre Y1 e Y1 + altura A e X2 está entre X1 e X1 + largura A.

Figura 8 – (a) Retângulos não colidindo e (b) retângulos colidindo.

2.5.2 Distância euclidiana

Para a implementação desse algoritmo, os dois objetos em questão serão envoltos por um círculo de tamanho predefinido, e então dois elementos precisam ser calculados: a distância entre os dois centros de círculos que serão testados e os raios dos mesmos. Considerando dois círculos (Figura 9) e que o círculo 1 está nas coordenadas x_1 e y_1 e o círculo 2 está nas coordenadas x_2 e y_2 , a distância entre os centros dos círculos pode ser medida usando a distância euclidiana: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. O método consiste de uma simples comparação, é preciso saber se a distância euclidiana entre os dois centros de círculos é maior que a soma do raio dos mesmos. A imagem abaixo ilustra essa comparação.

Pseudocódigo da distância euclidiana.

```

1
2 Funcao Colisao{
3
4   circulo c1, circulo c2
5   raio1, raio2
6
7   distancia = raizquadrada((c1.x - c2.x)^2 + (c1.y - c2.y)^2)
8   se (distancia > (raio1 + raio2))
9     //nao houve colisao
10  senao
11    //houve colisao
12 }
```

Caso a distância entre os círculos for menor do que a soma dos raios, significa que houve colisão, caso contrário a colisão não aconteceu.

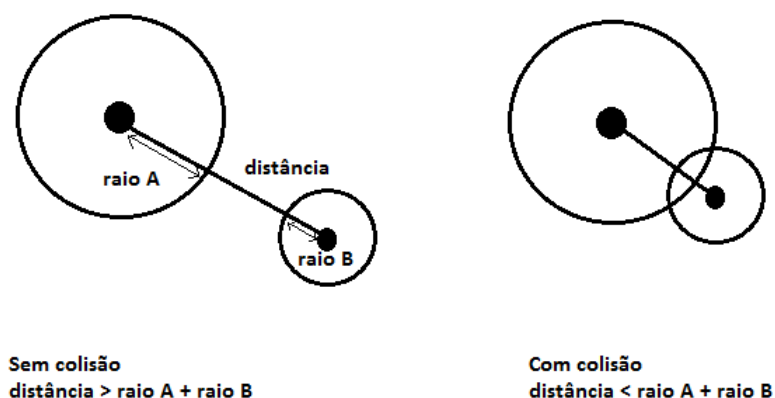


Figura 9 – Distância euclidiana entre um centro de círculo ao outro.

2.5.3 Distância de Manhattan

Esta técnica tem um funcionamento bem parecido com a distância euclidiana, a diferença é que distância de *manhattan* não calcula a distância de um ponto a outro com uma linha reta. A distância de *manhattan* é calculada como a trajetória de um táxi, percorrendo quarteirões de uma cidade para chegar até seu destino.

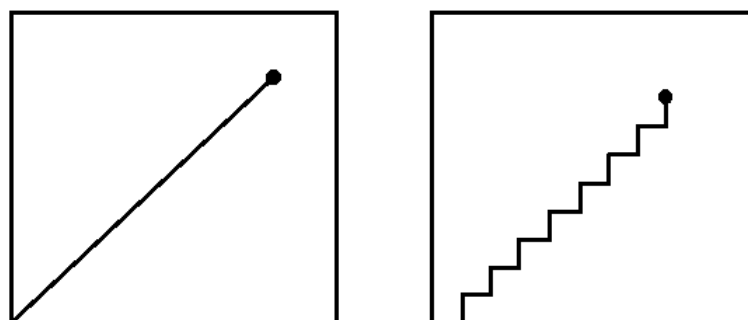


Figura 10 – Comparação entre distância euclidiana (linha reta) e de *manhattan* (escada).

Na [Figura 10](#) é visto a comparação da distância euclidiana com a de *manhattan*, a distância euclidiana tem uma trajetória uniforme em todas as direções, como um pássaro que voa por cima dos quarteirões de uma cidade. A distância de *manhattan* percorre a mesma distância para todos os lados assim como a euclidiana, no entanto obedecendo o

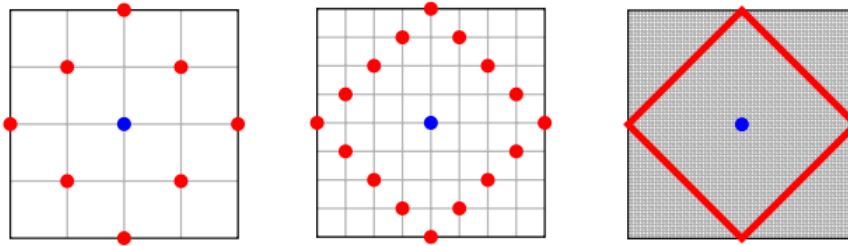


Figura 11 – Formação do losango usando distância de *manhattan* em um plano cartesiano.

esquema de quadras como um motorista de um táxi. Se o pássaro andar 5 quilômetros em todas as direções a partir de um mesmo ponto, essa trajetória iria formar um círculo perfeito. Se o taxista fazer o mesmo que o pássaro, traçando sua trajetória entre as quadras da cidade iria formar um losango.

A [Figura 11](#) demonstra como o percurso das quadras acontece, é possível chegar a um mesmo ponto por vários caminhos usando o mesmo número de quadras percorridas. A maneira de detectar colisão usando distância de *manhattan* é similar a distância euclidiana, a fórmula para descobrir a distância entre objeto e outro é $|x_1 - x_2| + |y_1 - y_2|$ ([KRAUSE, 1986](#)).

Pseudocódigo para distância de *Manhattan*

```

1 Funcao Colisao{
2
3   losango L1, losango L2
4
5   //distancia do centro do lasango para sua borda
6   movimentos1, movimentos2
7
8   distancia = |L1.x - L2.x| + |L1.y - L2.y|
9   se (distancia > (movimentos1 + movimentos2))
10    //houve colisao
11   senao
12    //nao houve colisao
13 }
```

2.5.4 Pixel Perfect

Como o nome do algoritmo sugere, a detecção feita por ele é perfeita, no sentido que não existe uma falsa colisão. Um dos problemas discutidos por este trabalho é justamente a falta de precisão da colisão, com o *Pixel Perfect* é possível atingir máxima precisão. No entanto esse método é bastante custoso ([NETO, 2013](#)), a imagem do objeto é considerada

como uma matriz de números inteiros onde a cor transparente ou branco é considerada "0", o algoritmo percorre pixel por pixel das imagens para determinar se alguma coordenada possui duas cores diferente de transparente. Caso esse teste seja verdadeiro significa que existe um mesmo pixel ocupado por duas imagens. Como já dito, esse processo é bastante oneroso, então é mais comum usar outra técnica combinada com o *Pixel Perfect*. Primeiro, outro método (sobreposição de retângulos, distância euclidiana entre outros) é usado para verificar se houve colisão, quando o teste é verdadeiro, é calculada a intersecção entre os objetos. O segundo passo consiste em utilizar o *Pixel Perfect* normalmente somente na área de intersecção calculada anteriormente. Neste trabalho o *Pixel perfect* é implementado sem a combinação com outras técnicas, para saber o real impacto no processamento desse algoritmo e também uma versão com a combinação com sobreposição de retângulos.

Pseudocódigo do *Pixel perfect*.

```
1 Funcao Colisao{
2
3   imagem1, imagem2
4   para i = 0; i diferente de tamanho da imagem; i++
5     para j = 0; j diferente de tamanho da imagem; j++
6       se (matriz.pixel[i,j] = imagem1 && matriz.pixel[i,j] = imagem2)
7         //houve colisao
8       senao
9         //nao houve colisao
10 }
```

2.5.5 Método hierárquico

Esse método utiliza outros algoritmos de detecção de colisão, a ideia é usar algoritmos em etapas e granularidade diferentes. Na prática um objeto é geralmente envolvido por um VE que o cobre totalmente, então quando a colisão é detectada outra etapa é iniciada, novos VE's menores são criados para envolver partes do objeto. O método hierárquico pode combinar bom desempenho com precisão pois pode aplicar formas geométrica simples, como um círculo ou quadrado, em objetos com formas complexas e complicadas (ERICSON, 2004). Como visto na Figura 3 usar formas simples para detectar colisão pode gerar colisões falsas, o método hierárquico contorna esse problema aplicando mais testes usando formas menores, como poder ser vista na Figura 12. Atualmente a hierarquia é amplamente utilizada em aplicações gráficas (LARSSON; AKENINE-MÖLLER, 2006) para resolução de detecção de colisão justamente pelo fato de que esse método une boa eficiência com precisão.

A Figura 12 mostra três etapas do método hierárquico usando distância euclidiana, a esquerda da imagem mostra a primeira etapa onde a nave (objeto) é envolvida por um

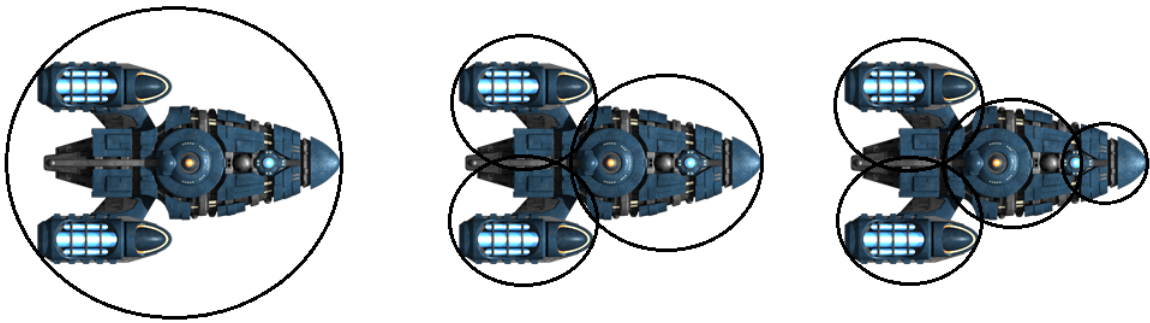


Figura 12 – Esquema do método hierárquico utilizando distância euclidiana.

único grande círculo, assim que a colisão for detectada esse círculo é descartado. O centro da imagem mostra a segunda etapa onde círculos menores são colocados sobre partes da nave, nota-se que esses círculos menores juntos cobrem o corpo da nave de forma mais eficiente do que acontecia na primeira etapa, a direita da imagem mostra o mesmo processo acontecendo novamente. Esse método pode possuir várias etapas sendo que sempre a última vai determinar se houve colisão ou não.

Pseudocódigo do Método hierárquico.

```
1 Funcao Colisao{
2
3   imagem 1
4   se (imagem1.metodoDeDeteccao1 = verdadeiro)
5     se (imagem1.metodoDeDeteccao2 = verdadeiro)
6       se (imagem1.metodoDeDeteccao3 = verdadeiro)
7         //houve colisao
8       senao
9         //nao houve colisao
10 }
```


3 Trabalhos relacionados

Este capítulo fala sobre alguns trabalhos semelhantes a esse. Os trabalhos foram pesquisados em bases de artigos como IEEE, ACM, *google scholar* e periódicos CAPES, dentre essas bases foram pesquisados termos como: detecção de colisão, colisão 2d, colisão 3d, *collision detection*, *collision detection for games*. Dentre os resultados das quatro bases foram escolhidos os 4 primeiros artigos de cada, as bases que ofereceram resultados mais parecidos com este trabalho foram o *google scholar* e periódico CAPES (*google scholar* na maioria). Mais artigos foram então pesquisados nas bases de artigos que mais retornaram resultados.

(MOURA; MACHADO, 2008) usam a técnica de volumes envolventes com os métodos de AABB (*axis aligned bounding box*) estáticos e OBB (*oriented bounding box*) orientado, para fazer uma comparação. OS autores utilizam estes métodos como volumes envolventes de pré-deteção, ou seja para um objeto dois envolventes são usados para detecção de colisão, um interno e um externo. Conforme o decorrer do artigo, são comentados alguma vantagens e desvantagens dos dois métodos como fato de AABB ser mais simples de calcular, no entanto surgem problemas como o aumento de tamanho quando é feito a rotação. Quanto a OBB, o método se diz mais custoso de calcular mas a rotação da caixa não altera o seu tamanho. No artigo são comparados dois objetos (AABB e OBB) com crescentes números de pontos (o que são pontos não fica claro) e o tempo de criação dos envolventes dos objetos é coletado. Os autores observam que os tempos de criação das caixas não tem impacto de forma significativa sobre a execução, pois apenas dois objetos foram testados e salienta que a adição de mais objetos pode gerar um custo de processamento impactante.

(TAVARES; LEMOS, 2008) também fazem comparações de algoritmos utilizando volumes envolventes assim como no primeiro trabalho relacionado mencionado, com o diferencial que esse volumes são usados na forma de hierarquia. Três algoritmos são usados, hierarquia de volumes envolvente para esferas, caixas alinhadas no eixo e caixas orientadas. Neste trabalho os autores implementam não apenas a detecção de colisão mas como o tratamento de colisão em uma simulação física, nessa simulação são feitas três formas (torus, pedra e cubo) para testes de sobreposição de volumes, sobreposição de triângulos e a equação de custo de processamento que segundo os autores é $V.C_v + T.C_t$ (V = número de teste de sobreposições de volume, C_v = custo em termos de números de operações aritméticas de um teste de sobreposições de volume, T = número de teste de sobreposições de triângulos, C_t = custo em termos de números de operações aritméticas de um teste de sobreposições de triângulos), como o autor chega ao custos das operações aritméticas não é explicado. O número de objetos na cena variam de 2,4,8,16,32 e 64.

Os autores perceberam que a taxa de quadros por segundo permanece aceitável para a execução da cena de até 32 objetos sendo que dos três algoritmos, o algoritmo que utiliza esferas obteve mais desempenho em termos de processamento. E observado também que caixas orientadas utilizam mais espaço.

(NAKAMURA; CELES, 2005) trabalham na implementação de uma biblioteca para detecção de colisão. A biblioteca permite a detecção de colisão entre volumes envolventes, detecção de colisão entre volumes envolventes das primitivas que compõem os objetos, detecção de colisão entre os primitivas que compõem os objetos e detecção entre elementos externos como um raio ou plano e objetos da cena. Os autores fazem testes de desempenho e precisão com a biblioteca fazendo três testes, dois para desempenho e um para precisão de colisão. Os testes são feitos comparando algoritmos de detecção de colisão da biblioteca com algoritmos implementados com força bruta em um cenário virtual, a taxa de *frames* é usada como medidor de desempenho. Os testes demonstram que a taxa de *frames* cai drasticamente quando a simulação é testado com os algoritmos de força bruta, o que não acontece nos teste utilizando os algoritmos da biblioteca. Os testes de precisão de são realizados com o *spectator mode* disponível na aplicação. Um personagem é usando para andar pelo cenário, os próprios autores identificam os erros de precisão, e enfatizam que isso acontece devido ao uso de algoritmos com baixo grau de precisão.

(LORBIESKI; BRUN; BRACARENSE, 2009) implementam e comparam três técnicas de detecção de colisão de objetos tridimensionais. Os métodos são, Método das áreas, método dos cossenos e métodos das coordenadas baricêntricas, os autores analisam a complexidade assintótica e o tempo de execução dos métodos. Para aplicar os testes foram criados arquivos com malhas de triângulos de tamanhos que variam. A análise assintótica dos três métodos apresentam uma complexidade de $O(n)$, onde n é número de triângulos das malhas, apresentando diferenças muito pequenas. Os testes temporais também demonstram uma diferença muito pequena para criar as malhas de triângulos. Os autores concluem que todos os métodos tem tempo de execução satisfatório e podem envolver a criação de vários objetos sem comprometer a taxa de *frames*.

Todos os trabalhos apresentados, assim como esse próprio fazem testes de desempenho e se preocupam com o impacto de processamento que os algoritmos de detecção de colisão implementados podem ter sobre a simulação de um jogo e a taxa de *frames*. O diferencial desse trabalho sobre os outros é a medição de precisão do métodos de detecção de colisão, (NAKAMURA; CELES, 2005) em seu trabalho demonstra algum resultado em relação a precisão dos algoritmos implementados em seu trabalho, no entanto, não existe uma medição apropriada de precisão.

4 Metodologia

Este capítulo explica como os testes serão implementados. Os teste se dividem em duas partes principais, os testes de desempenho e os testes com usuários.

4.1 Medida de desempenho

Para medir o desempenho dos métodos é necessário saber o tempo que a função de detecção de colisão requer desde o seu início até o seu fim. Para obter esse tempo será utilizada a função *nanoTime* do *Java* que calcula precisamente o tempo de execução que se passou de um ponto do código até outro, e retorna esse tempo medido em nanosegundos. Como cenário de testes será implementado um pequeno jogo, onde devem existir um personagem principal que será o objeto onde as colisões serão testadas e outros vários personagens que aqui serão denominados de “inimigos”. Cada método será testado com quatro números de inimigos diferentes: 1,10,100,1000.

Em cada teste é necessário estabelecer algumas regras para testar o desempenho, são elas:

- O personagem deve traçar sempre a mesma rota em todos os testes.
- As rotas serão uma vez na horizontal, uma na vertical e uma na diagonal, todas do início ao fim do cenário.
- Os inimigos serão posicionados aleatoriamente.
- A colisão será testada apenas no personagem principal com todos os inimigos.
- Cada teste será repetido vinte vezes para no fim fazer a média dos tempos coletados.

A [Figura 13](#) mostra um esquema das direções que o personagem deve percorrer em cada teste, o personagem percorre o seu caminho por apenas uma direção e não desvia sua trajetória. O jogo é feito em uma tela 800 x 600 e o personagem principal começa na posição 0 do eixo x e 300 do eixo y, os inimigos podem estar posicionados em qualquer lugar. Conforme o personagem principal percorre o trajeto e colidir com um inimigo esse inimigo é marcado como morto e retirado do jogo, o personagem principal segue sua trajetória normal.

Tempos que serão coletados:

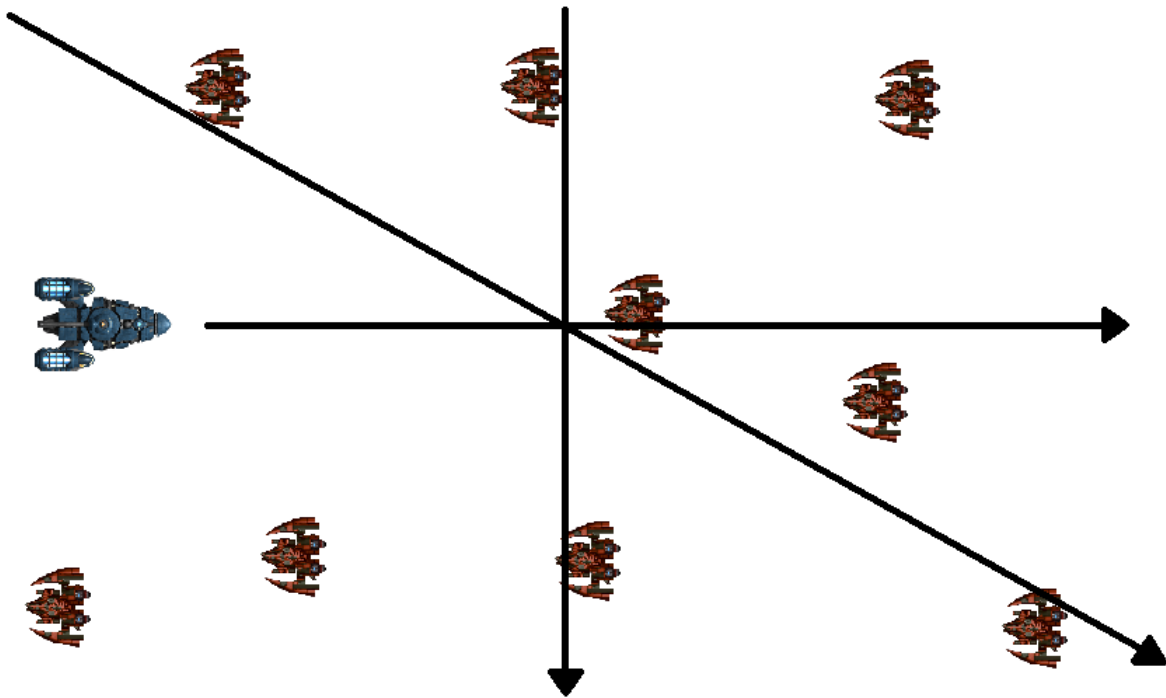


Figura 13 – Direções que personagem deve seguir nos testes.

- Tempo total gasto para percorrer todo o percurso (não importando se há colisão ou não).
- Média de tempo necessário para o cálculo de colisão quando há colisão.

Os tempos serão usados para montar gráficos de desempenho, que mostrarão qual método necessita de mais tempo para executar uma única iteração de sua função de detecção de colisão como também para atravessar o cenário executando essa função.

4.1.1 Ambiente de execução

Os testes de desempenho serão realizados em uma máquina com a seguinte configuração: Processador intel core i5-3337U 1.8GHz, 4 núcleos divididos em 2 núcleos físicos e 2 núcleos lógicos, Cache L1 com 32 kB, Cache L2 com 256 kB, Cache L3 com 3 MB. Memória RAM com 6 GB 1600MHz, Hard disk com 720 GB, Nvidia Geforce 735 M.

4.2 Testes com usuários

Para realizar os testes com usuários também é necessária a implementação de um pequeno jogo (diferente do jogo que é usado para os teste de desempenho). O jogo tem forma de um questionário onde o usuário julga o quão real a colisão lhe pareceu. O jogo consiste de um personagem que é controlado pelo usuário para desviar de inimigos que

andam em direção contrária do personagem pelo cenário, o objetivo do jogo é manter o personagem desviando dos inimigos o máximo de tempo possível. Quando o personagem colidir com algum inimigo o questionário aparece com opções para atribuir uma nota para o realismo da colisão. A [Figura 14](#) mostra uma representação do jogo para testes com usuários.

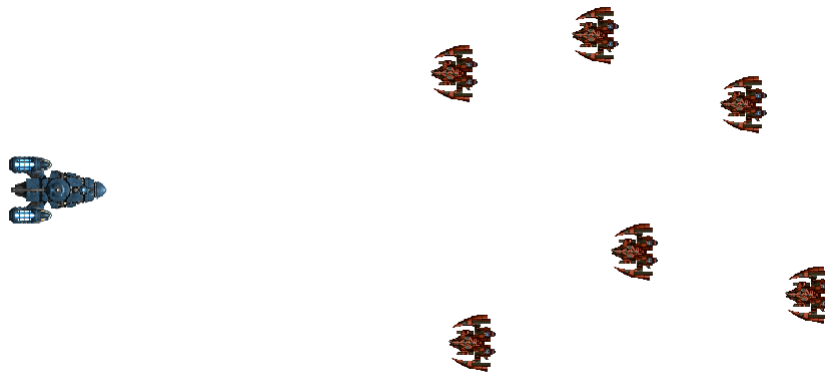


Figura 14 – Representação do jogo-questionário.

No total o jogo terá cinco rodadas, uma para cada método de detecção de colisão, por exemplo, a primeira rodada será usado o método da distância euclidiana, então quando o personagem colidir o questionário aparece, quando o questionário for respondido outra rodada começa com outro método de detecção de colisão. O questionário será em escala *Likert* para possibilitar a coleta de dados mais precisos do que notas por algarismos (LIKERT, 1932). A [Figura 15](#) demonstra o questionário utilizado nos testes com usuários.

A imagem mostra uma janela de software com o título "Questionário". O conteúdo da janela é o seguinte: "Você concorda que a sua nave colidiu?". Abaixo disso, há cinco opções de resposta, cada uma com um botão de opção (radio button) à esquerda: "Concordo plenamente", "Concordo parcialmente", "Não concordo nem discordo", "Discordo parcialmente" e "Discordo totalmente".

Figura 15 – Representação questionário.

Assim como os testes de desempenho esse teste deve seguir algumas regras:

- O usuário não sabe que tipo de método de detecção de colisão está sendo usado em cada rodada do jogo.

- A cada início de jogo a ordem dos métodos utilizados serão sorteados randomicamente.

Estas regras serão aplicadas para que o usuário não tenha uma mesma experiência caso jogue mais de uma vez.

5 Análise de resultados

Neste capítulo os resultados obtidos são apresentados e explicados. Como proposto por este trabalho os resultados são divididos em duas partes: testes de desempenho e testes com usuários, nos testes de desempenho procura-se saber quanto tempo cada método toma para ser executado e quanto tempo leva para percorrer um percurso com variados números de elementos no cenário. Os métodos testados neste estágio são: euclidiana, *manhattan*, sobreposição de retângulos (retângulo), *pixel perfect* (somente para o teste de desempenho), hierárquico e retângulo com *pixel perfect*. Os testes com usuários têm o objetivo de saber através de uma pesquisa qual a precisão em termos de realidade de cada método, os métodos testados com usuários são: euclidiana, *manhattan*, retângulo, hierárquico e retângulo com *pixel perfect*.

5.1 Testes de desempenho

Os tempos para os métodos completarem o processo de verificação de colisão se diferenciam em sua grandeza quando usado os métodos *pixel perfect* e retângulo com *pixel perfect*, portanto o gráfico desses será separados dos outros para não criar discrepância entre os resultados. Os resultados serão apresentados de forma que primeiro mostre os gráficos de desempenho de detecção de colisão do método seguido do gráfico de tempos para percorrer o percurso pré-programado descrito no [Capítulo 4](#).

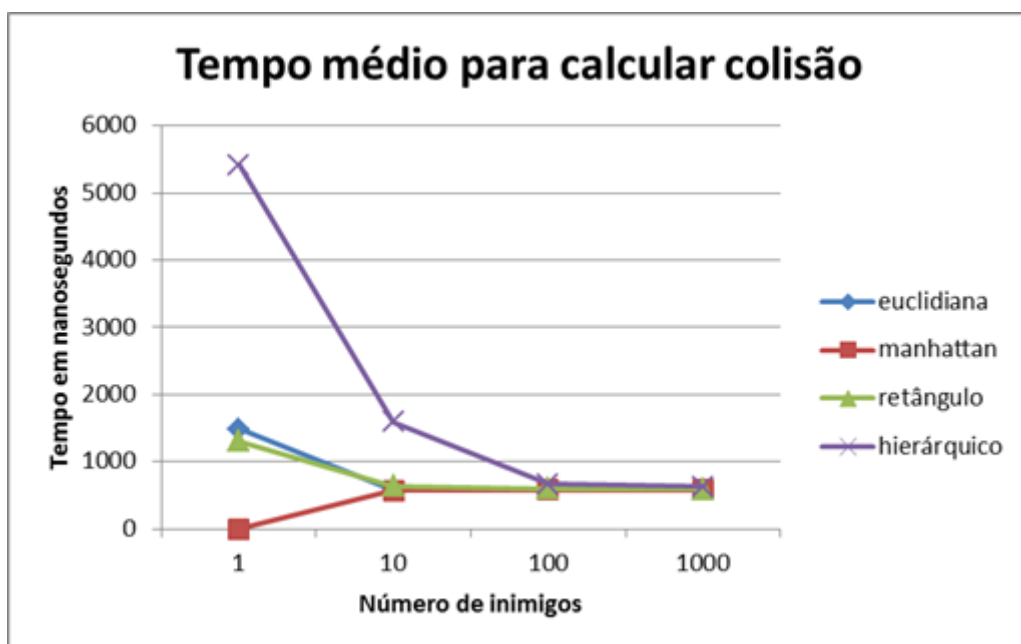


Figura 16 – Tempo para calcular detecção de colisão (em nanosegundos).

Na [Figura 16](#), o tempo do método *manhattan* no teste com um inimigo mostra que o tempo levado para calcular a colisão é zero, isso acontece porque o método *nanotime* da classe *System* do *Java* precisa ter um tempo de resolução maior que o tempo de resolução do trecho de código que está medindo, caso essa rotina retorna 0 como tempo. Como é possível observar no gráfico da [Figura 16](#) conforme os testes são executados os tempos convergem para a mesma região. A detecção hierárquica teve um tempo maior para ser executada pela primeira vez, no entanto todos ficam em torno do tempo de 600 nanosegundos variando minimamente para maior ou menor conforme o número de inimigos aumenta.

É possível comprovar a eficiência do método hierárquico, para implementar esse método foi usado o teste do método euclidiana várias vezes e mesmo assim os tempos são semelhantes ao teste que usou euclidiana somente uma vez. Isso indica a possibilidade de usar um mesmo *VE* várias vezes para o mesmo objeto sem possuir um acréscimo de tempo muito grande e gerar uma precisão do resultado muito maior.

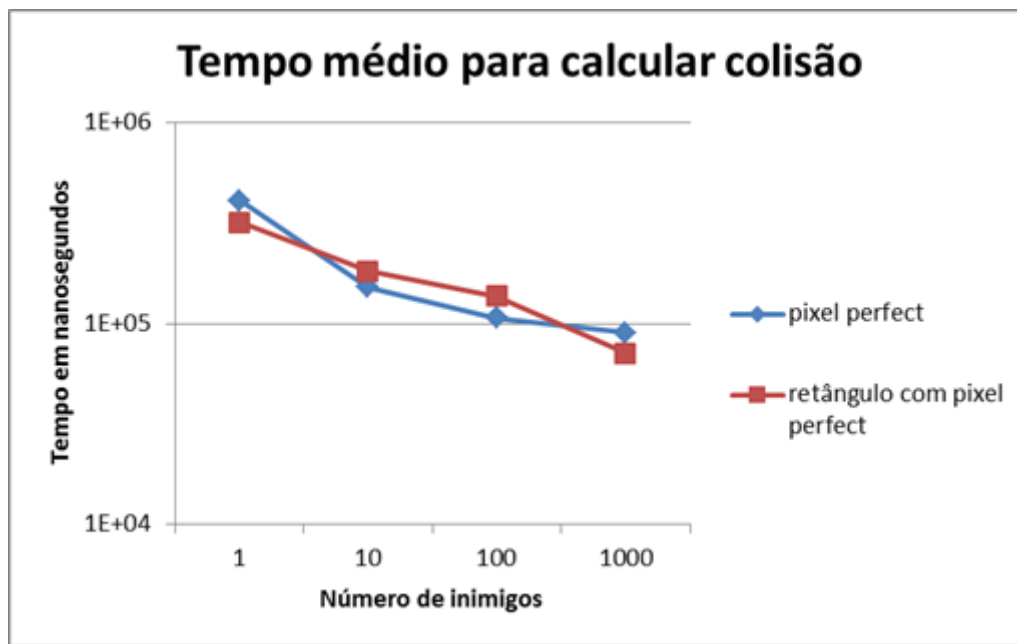


Figura 17 – Tempo para calcular detecção de colisão (em nanosegundos).

A [Figura 17](#) possui o gráfico com os tempos de cálculo de detecção de colisão dos métodos *pixel perfect* e *retângulo com pixel perfect*, nota-se que o tempo consumido para executa-los foi muito maior do que o tempo dos métodos mostrados anteriormente. Isso se deve ao fato de que a técnica de *pixel perfect* se torna muito onerosa conforme o número de objetos testados aumenta, no entanto, existe uma queda considerável do tempo após o número de inimigos aumentar. Esse comportamento é observado em todos os métodos, quando repetidos várias vezes tendem a baixar o tempo de execução e depois ficar dentro de um intervalo de tempo. Para saber o motivo que os tempos se tornam menores conforme é necessária pesquisas mais aprofundadas nesta questão, esse trabalho não tem isso como

objetivo e somente aponta para o acontecimento e enfatizar sua importância nos testes de desempenho.

Nos testes de percurso os tempos dos métodos também se mostram indo em direção para a mesma área do gráfico como pode ser visto na [Figura 18](#). Os tempos variam de 14 milissegundos a 18 milissegundos o que pode ser considerado ótimos tempos para completar o percurso mesmo com um número grande de inimigos no cenário, esse tempo é considerado bom pela razão de permitir uma boa taxa de quadros ao jogo.

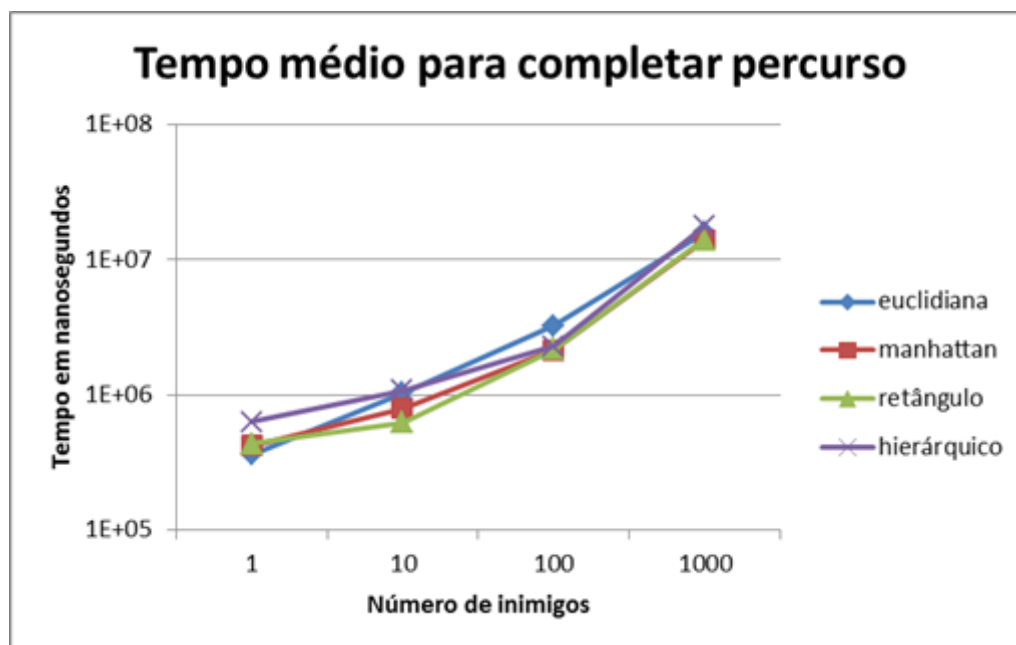


Figura 18 – Tempo para percorrer percurso (em nanosegundos).

Para analisar os resultados do método *pixel perfect* é necessário mais atenção do que os demais. A [Figura 19](#) e a [Figura 20](#) mostram que o tempo necessário para completar o percurso do método *pixel perfect* é extremamente maior com aproximadamente 2 horas e meia no teste com mil inimigos e o tempo do mesmo combinado com retângulo com mil inimigos fica um pouco acima de 300 milissegundos. Entretanto a [Figura 17](#) demonstra que os tempos para calcular a detecção de colisão dos dois métodos é bastante semelhante, isso pode ser explicado na forma como o *pixel perfect* é executado e como os tempos dos testes são coletados. Para realizar um teste de colisão entre dois objetos usando *pixel perfect* um retângulo é criado de tamanho definido pela posição desses objetos. Imaginando um plano cartesiano de duas dimensões quanto maior a distância dos dois objetos no eixo x e eixo y maior será o tamanho do retângulo criado.

Após isso como explicado na [subseção 2.5.4](#) o algoritmo verifica se existe dentro do retângulo um pixel que possui duas cores que não seja branco, o que consome muito tempo de processamento. Caso o tamanho desse retângulo for pequeno o tempo de processamento do algoritmo é relativamente pequeno, no entanto quando o tamanho do retângulo é muito

grande o tempo de processamento é extremamente grande. Fazendo a média aritmética de todos os tempos obtidos com os testes (de retângulos pequenos e grandes) pode fazer com que o tempo resultante seja parecido com o tempo resultante do método retângulo com *pixel perfect* que só faz verificação pixel por pixel quando o retângulo que tem tamanho do personagem detectar algum inimigo. Apesar da média aritmética não ser eficiente para coletar tempos de execução do *pixel perfect* a Figura 19 demonstra como esse algoritmo pode ser muito custoso para ser executado, entretanto quando combinado com outros algoritmos como os testados nesse trabalho pode atingir tempos de execução satisfatórios.

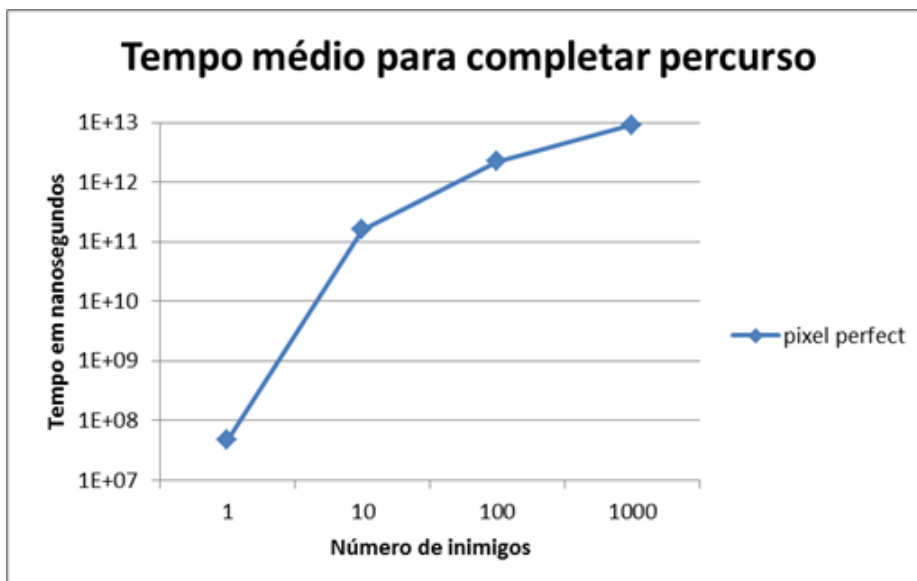


Figura 19 – Tempo para percorrer percurso (em nanosegundos).

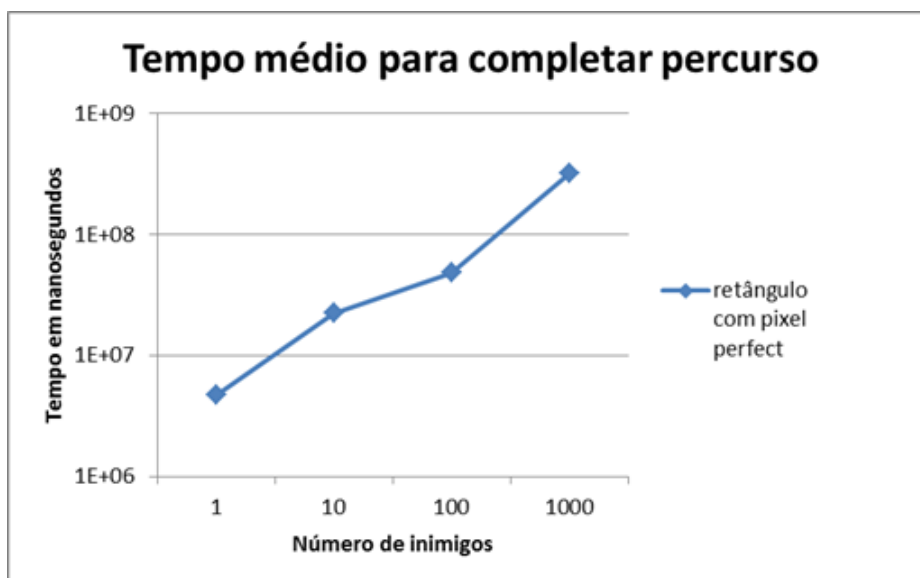


Figura 20 – Tempo para percorrer percurso (em nanosegundos).

5.2 Testes com usuários

Devido aos altos custos de processamento do método *pixel perfect* esse não foi utilizado para realizar os testes com usuários, mesmo simples o jogo implementado não seria jogável. O jogo teria uma taxa de quadros por segundos extremamente baixa deixando o jogador insatisfeito ou impaciente durante o jogo. Por essa razão foi decidido que o algoritmo *pixel perfect* usaria uma hierarquia com a combinação do método de retângulo para tornar o jogo mais eficiente em relação a taxa de quadros.

No total os usuários que responderam a pesquisa somam 31 indivíduos, cada pessoa podia jogar mais de uma vez caso isso fosse desejado e isso não altera o resultado final, pois como informado na [seção 4.2](#) a ordem dos métodos no início do jogo é sorteada aleatoriamente. Os votos de todos os jogadores somam 64.

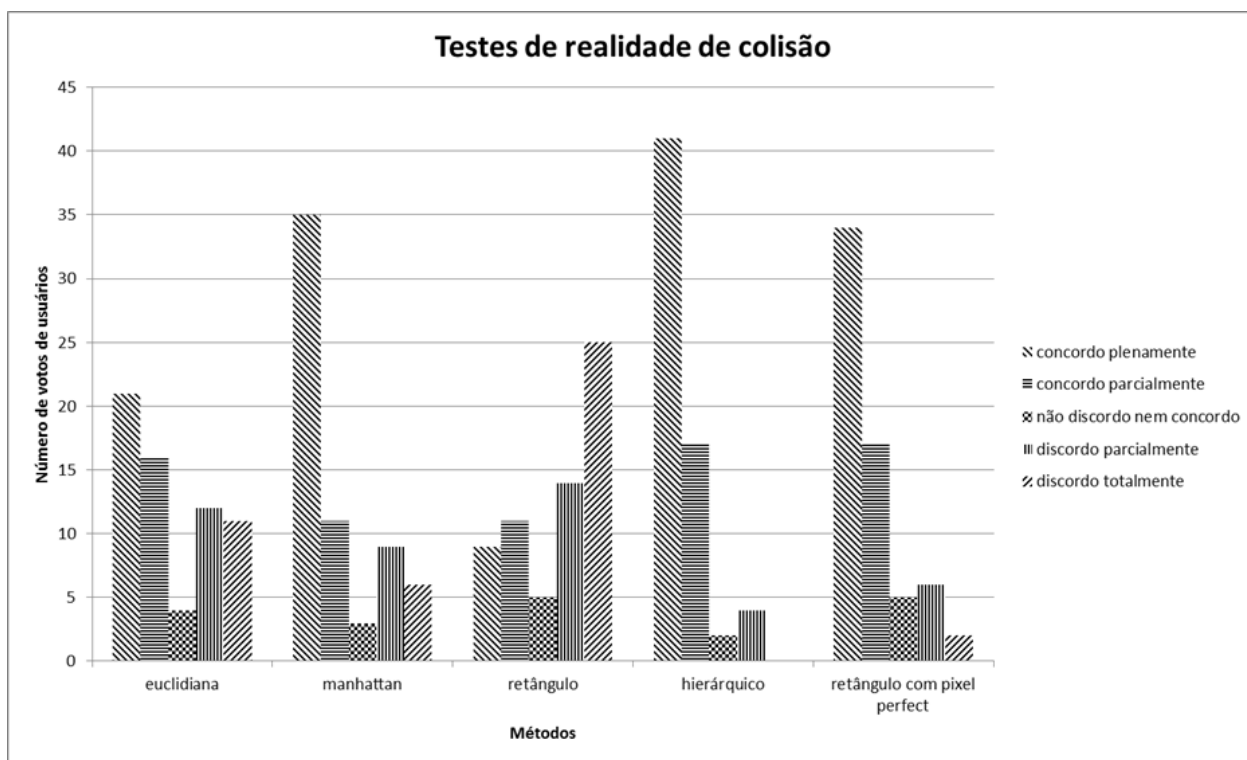


Figura 21 – Resultado dos testes com usuários.

A [Figura 21](#) demonstra os resultados obtidos com os usuários, o pior dos métodos em questão de precisão é o de sobreposição de retângulos, é natural que um retângulo não possa envolver a maioria dos objetos a não ser que esse seja quadrado ou retangular. A detecção de colisão por distância euclidiana mostra que tem precisão para pouco mais da metade dos casos, considerando sua forma circular quase sempre envolve o objeto e alguma área desnecessária. Dos métodos que têm formas geométricas simples como volume envolvente o que obteve melhor resultado foi o *manhattan*, na categoria de voto “concordo plenamente” recebeu uma pontuação até melhor que o método retângulo com *pixel*

perfect. Já na categoria de voto “concordo parcialmente” o contrário acontece indicando que o algoritmo *manhattan* funciona mais de formas específicas e o retângulo com *pixel perfect* funciona geralmente para a maioria dos casos com objetos de formas variadas. O método que obteve melhores resultados entre todos foi o hierárquico, comprovando sua eficiência em desempenho como também em precisão, como pode dividir e diminuir um volume envolvente de forma simples por várias fases de testes consegue o desempenho de um VE simples e a precisão de um polígono complexo.

6 Conclusão

A área de detecção de colisão tem muita importância em aplicações gráficas, sendo usada em áreas como medicina, inteligência artificial e como enfatizado por este trabalho a área de jogos digitais. A detecção de colisão é considerado um gargalo para aplicações gráficas interativas, algumas dessas fazem centenas e até milhares de testes para poder detectar uma possível colisão (KLOSOWSKI et al., 1998).

Nesse trabalho foram analisados algoritmos de detecção de colisão, os algoritmos testados foram distância euclidiana, distância de *manhattan*, sobreposição de retângulos, *pixel perfect*, detecção de colisão hierárquica e sobreposição de retângulos com *pixel perfect*. Os testes foram divididos em duas formas de testes: testes de desempenho e testes de precisão, os testes de desempenho foram feitos coletando os tempos que cada método leva para executar e o tempo que precisam para percorrer um percurso pré-programado. Os testes de desempenho usaram números diferentes de objetos para testar a colisão com um objeto principal, as iterações usaram os números de 1, 10, 100, 1000 objetos. Os testes de precisão foram feitos usando usuários aleatórios em uma pesquisa de percepção onde o indivíduo joga um jogo usando os métodos de detecção de colisão citados nesse trabalho com exceção do *pixel perfect* devido ao seu péssimo desempenho deixando o jogo com taxas de quadros baixíssimos.

Os testes de desempenho mostram um tempo baixo para executar métodos que utilizam VE's simples e que depois de várias iterações executando um método o tempo de execução tende a baixar e variar de forma mínima em torno de um tempo específico. Esse comportamento se repete quando o método hierárquico utiliza um mesmo VE várias vezes, o tempo de execução fica quase igual ao tempo de execução do método que utilizou esse mesmo VE apenas uma vez. Isso demonstra que é possível utilizar um método de forma a fazer vários testes sobre o mesmo objeto sem que o desempenho seja muito afetado. Os algoritmos que usaram *pixel perfect* obtiveram um tempo de execução muito maiores chegando a casa dos milissegundos para verificar se houve colisão. Para percorrer o percurso pré-programado os algoritmos obtiveram um desempenho semelhante com exceção do *pixel perfect* que obteve um tempo muito grande.

Os testes de precisão demonstram a sobreposição de retângulos como o pior método para obter precisão, a maioria dos usuários que jogaram com esse método perceberam falsas colisões ocorrendo. O melhor método demonstrado pelos testes de precisão dentre os que são testados por esse trabalho é o de detecção de colisão hierárquica que obteve bons resultados em ambos os testes de desempenho e precisão.

Referências

AMATO, J. Collision Detection. 1999. Disponível em: <http://www.gamedev.net/page/resources/_/technical/game-programming/collision-detection-r735>. Acesso em: 10 jan. 2015. Citado na página 25.

CHANDLER, H.; CHANDLER, R. *Fundamentals of Game Development*. Jones & Bartlett Learning, 2011. (Foundations of game development). ISBN 9780763778958. Disponível em: <<https://books.google.com.br/books?id=XhrBBeA0-LMC>>. Citado na página 21.

COURSE, M. *Exam 98-374 Gaming Development Fundamentals*. Wiley, 2013. (Microsoft official academic course). ISBN 9781118359891. Disponível em: <<https://books.google.com.br/books?id=tRJ-tgAACAAJ>>. Citado na página 17.

DAVISON, A. *Killer Game Programming in Java*. O'Reilly Media, 2005. ISBN 9780596552909. Disponível em: <http://books.google.com.br/books?id=dOz-UK8F1_UC>. Citado na página 20.

ERICSON, C. *Real-Time Collision Detection*. Elsevier Science, 2004. (The Morgan Kaufmann Series in Interactive 3D Technology). ISBN 9780080474144. Disponível em: <http://books.google.com.br/books?id=0MvuykjoW_IC>. Citado 2 vezes nas páginas 22 e 29.

HARBOUR, J. *Beginning Game Programming, Fourth Edition*. Cengage Learning, 2014. ISBN 9781305259102. Disponível em: <<https://books.google.com.br/books?id=wogDBAAAQBAJ>>. Citado na página 19.

KLOSOWSKI, J. et al. Efficient collision detection using bounding volume hierarchies of k-dops. *Visualization and Computer Graphics, IEEE Transactions on*, v. 4, n. 1, p. 21–36, Jan 1998. ISSN 1077-2626. Citado na página 43.

KRAUSE, E. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, 1986. (Dover Books on Mathematics Series). ISBN 9780486252025. Disponível em: <<http://books.google.com.br/books?id=IW7ICV0QXWwC>>. Citado na página 28.

LARSSON, T.; AKENINE-MÖLLER, T. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, Elsevier, v. 30, n. 3, p. 450–459, 2006. Citado na página 29.

LIKERT, R. *A Technique for the Measurement of Attitudes*. publisher not identified, 1932. (A Technique for the Measurement of Attitudes, N° 136-165). Disponível em: <<http://books.google.com.br/books?id=9rotAAAAYAAJ>>. Citado na página 35.

LORBIESKI, R.; BRUN, A.; BRACARENSE, J. Análise comparativa entre três métodos de detecção de colisão em objetos 3d. *III EPAC - Encontro Paranaense de Computação*, 2009. Citado na página 32.

MCSHAFFRY, M.; GRAHAM, D. *Game Coding Complete*. Course Technology, Cengage Learning, 2012. ISBN 9781133776574. Disponível em: <<http://books.google.com.br/books?id=nmOMpwAACAAJ>>. Citado na página 17.

MOURA, I. F.; MACHADO, L. S. Análise comparativa entre métodos de detecção de colisão broad em ambientes de realidade virtual. *HÍFEN*, v. 1, n. 1, 2008. Citado na página 31.

NAKAMURA, F. I.; CELES, W. Detecção hierárquica de colisão em ambientes 3d. In: *Workshop de Trabalhos de Graduação, Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, Natal*. [S.l.: s.n.], 2005. Citado na página 32.

NETO, D. A. Intelligent 2D Collision and Pixel Perfect Precision. 2013. Disponível em: <http://www.gamedev.net/page/resources/_/technical/game-programming/intelligent-2d-collision-and-pixel-perfect-precision-r3311>. Acesso em: 14 dez. 2014. Citado na página 28.

RABIN, S. *Introduction to Game Development: Second Edition*. Course Technology Cengage Learning, 2010. (Game development series). ISBN 9781584506799. Disponível em: <https://books.google.com.br/books?id=79ud9_8mbgYC>. Citado 5 vezes nas páginas 17, 18, 23, 24 e 25.

ROCHA, R. *Algoritmos Rápidos de Detecção de Colisão Broad Phase utilizando KD-Trees*. Dissertação (Mestrado) — FUNDAÇÃO EDSON QUEIROZ UNIVERSIDADE DE FORTALEZA – UNIFOR, 2010. Citado na página 18.

SILVA, F. D.; FEIJO, B.; CLUA, E. *Introdução À Ciência da Computação Com Jogos*. CAMPUS - RJ, 2010. ISBN 9788535234190. Disponível em: <<https://books.google.com.br/books?id=27WSgzfQiHsC>>. Citado na página 24.

TAVARES, D. L.; LEMOS, R. R. Detecção de colisões para corpos rígidos em tempo real utilizando hierarquias de volumes envoltórios. *HÍFEN*, v. 30, n. 58, 2008. Citado na página 31.