UNIVERSIDADE FEDERAL DO PAMPA

Cristiano Daitx Ribeiro

Avaliação de algoritmos de detecção de colisão em jogos 2D para *Android*

Alegrete
2018

Cristiano Daitx Ribeiro

Avaliação de algoritmos de detecção de colisão em jogos 2D para *Android*

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: Prof. Mestre Jean Felipe Patikowski Cheiran

Coorientador: Prof. Doutora Aline Vieira de Mello

Alegrete
2018

Cristiano Daitx Ribeiro

Avaliação de algoritmos de detecção de colisão em jogos 2D para *Android*

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 27 de junho de 2018
Banca examinadora:

Prof. Mestre Jean Felipe Patikowski Cheiran
Oricntador
UNIPAMPA

Prof. Doutora Aline Vieira de Mello
Coorientadora
UNIPAMPA

Prof. Mestre Vinícius Bittencourt da Silva
QI Faculdade & Escola Técnica

Prof. Doutor Marcelo Resende Thielo
UNIPAMPA

AGRADECIMENTOS

A minha mãe por todo o incentivo desde o início da graduação.

A minha namorada pelo apoio e compreensão.

A minha psicóloga que me deu todo o apoio e me mostrou a cada desafio que eu era capaz de superar.

A Unipampa, em especial para os meus orientadores, que me orientaram e acreditaram neste trabalho desde o inicio.

Aos desenvolvedores do *GameBench* que foram muito atenciosos estendendo o período de acesso ao aplicativo.

Por fim a todos os meus amigos, que me acompanharam durante a graduação.

“ Do not pray for an easy life, pray for the strength to endure a difficult one.
(Bruce Lee)

RESUMO

Os jogos eletrônicos tornaram-se populares na década de 70, formando um mercado bilionário. Com o avanço tecnológico, esta forma de entretenimento chegou aos dispositivos móveis. Devido ao aperfeiçoamento da arquitetura dos dispositivos móveis e a criação de novos sistemas operacionais, o desenvolvimento de jogos para esta plataforma tornou-se mais atraente. Jogos são compostos por diversos módulos, destacando-se entre eles a física que é responsável pela simulação do movimento dos objetos. Neste contexto a detecção de colisão trata do problema de sobreposições de objetos em um ambiente virtual e tem um papel essencial para o desenvolvimento dos jogos, porém ela pode ser muito onerosa. Neste sentido, este trabalho tem por objetivo avaliar diferentes algoritmos de detecção de colisão em jogos 2D para dispositivos Android. Os algoritmos implementados e avaliados são distância euclidiana, distância de Manhattan, sobreposição de retângulos, teorema do eixo de separação (SAT) e *pixel perfect*. Dessa forma, realizou-se uma pesquisa bibliográfica para identificar quais recursos dos dispositivos móveis deveriam ser analisados e como avaliar diferentes algoritmos de detecção de colisão. Efetuou-se a avaliação dos algoritmos por meio da coleta e análise do tempo de execução e também com o auxílio do aplicativo GameBench, que registra os quadros por segundo, a utilização do CPU e GPU, o uso de memória e o consumo da bateria do dispositivo. Considerando diferentes formas de *sprites* e rotas os resultados mostram que o algoritmo SAT é o algoritmo baseado em volumes envolventes que possui a melhor precisão. Dos recursos analisados o SAT teve um elevado uso de CPU o que causou impacto no seu desempenho, mas ainda sendo a melhor alternativa para jogos com poucos objetos que necessitam detecção de colisão. Para uma grande quantidade de objetos, o algoritmo de sobreposição de retângulos teve um desempenho superior ao SAT e sua precisão (na maioria das vezes) similar. O algoritmo distância de Manhattan teve a pior precisão com todas as formas de *sprites*, e o algoritmo distância euclidiana apresentou precisão e desempenho intermediários.

Palavras-chave: Jogos eletrônicos. Dispositivos móveis. Android. Detecção de colisão. Jogos 2D.

ABSTRACT

Electronic games became popular in the 70's, forming a billionaire market. With the technological advance, this form of entertainment has reached the mobile devices. Due to the improved architecture of mobile devices and the creation of new operating systems, the development of games for this platform has become more attractive. Games are composed of several modules, among them the physics that is responsible for the simulation of the movement of objects. In this context collision detection addresses the problem of overlapping objects in a virtual environment and plays a key role in game development, but it can be very costly. In this sense, this work aims to evaluate different collision detection algorithms in 2D games for Android devices. The algorithms implemented and evaluated are euclidean distance, Manhattan distance, overlapping retangles, separation axis theorem (SAT) and pixel perfect. A literature search was performed to identify which mobile device features should be analyzed and how to evaluate different collision detection algorithms. The algorithms evaluation included to collect and to analyze the execution time of the algorithms and also, with the aid of the Gamebench app, to record frames per second, CPU and GPU usage, memory usage and device battery consumption. Considering different forms of sprites and routes, the results show that SAT is the algorithm based on bounding volumes that has the best accuracy. From resources analyzed SAT had a high CPU usage which impacted on its performance, but still being the best alternative for games with few objects that need collision detection. For a large number of objects, the overlapping retangles algorithm performed better than SAT, and its accuracy is (most of the time) similar. The algorithm Manhattan distance had the worst accuracy with all sprites forms, and euclidean distance algorithm presented intermediate accuracy and performance.

Key-words: Electronic games. Mobile devices. Android. Collision detection. 2D Games.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CPU unidade central de processamento

FPS quadros por segundo

GPU unidade de processamento gráfico

RV realidade virtual

SAT teorema do eixo de separação

VE volume envolvente

SUMÁRIO

1 INTRODUÇÃO

Os jogos eletrônicos como uma forma de entretenimento se popularizaram com a criação do jogo Pong na década de 70 e também com o lançamento do console Atari 2600. Desde então a indústria de jogos eletrônicos seguiu em crescimento, tendo um importante papel na evolução dos computadores e movimentando um mercado bilionário (ASSOCIATION, 2017). Nos primeiros dispositivos móveis, os jogos vinham pré-instalados nos aparelhos da Nokia. No final da década de 90, com a facilidade de criar jogos para celulares utilizando o J2ME (Java), mais jogos foram desenvolvidos. A indústria de jogos para celulares tem um ciclo de vida reduzido, comparado com os vídeo-games e computadores pessoais, com cerca de um ano e seguindo a lei de Moore dobrando o seu poder computacional em cerca de 18 meses.

Um jogo contém diversos módulos que são utilizados para sua execução, como conjunto de gráficos, física, inteligência artificial e sons. No desenvolvimento destes módulos, destaca-se a física que é responsável pela simulação do movimento dos objetos. Em um ambiente virtual, dois corpos podem ocupar o mesmo lugar ao mesmo tempo, entretanto com algoritmos de detecção de colisão podemos evitar que isso ocorra. A detecção de colisão identifica quando ocorre a intersecção de dois ou mais objetos e é essencial para o desenvolvimento de um jogo onde há movimentação de objetos e também em outras circunstâncias como ativar eventos, iniciar animações, efeitos de luz ou sonoros.

Vários jogos são desenvolvidos sobre módulos prontos, também conhecidos como *engines* gráficas (KLEINA, 2011). A ideia principal é permitir que os recursos comuns a quase todos os jogos sejam reutilizados. Neste caso, a cada novo jogo, se implementa apenas seus requisitos particulares. Porém os desenvolvedores acabam dependendo das escolhas e implementações dos algoritmos que são utilizados pelas *engines* gráficas.

Há diferentes algoritmos para realizar a detecção de colisão, variando quanto à precisão e o custo de processamento. Em simulações físicas onde a precisão é mais importante são utilizados algoritmos mais robustos, exigindo um maior consumo de processamento. Entretanto, em jogos onde o desempenho é mais importante pode ser utilizado algoritmos de detecção de colisão mais simples, porém menos precisos, com um custo de processamento menor. Um dos desafios do desenvolvimento de jogos é conciliar desempenho e a detecção de uma quantidade elevada de objetos em movimento colidindo simultaneamente. Para vencer este desafio é importante realizar a escolha do algoritmo mais adequado para cada jogo.

Jogos para *smartphones* devem levar em consideração algumas limitações dos recursos disponíveis e também o consumo de bateria. Segundo HOSSEINI; PETERS; SHIRMOHAMMADI, a lei de Moore também se aplica aos celulares, tendo sua capacidade computacional dobrada a cada 18 meses, porém não se aplica à vida da bateria, que tem uma melhora somente de 5% a 8% a cada ano. Neste sentido uma avaliação destes algoritmos se faz necessária em dispositivos móveis devido aos seus recursos limitados.

1.1 Objetivo geral

Neste trabalho é proposta a avaliação de algoritmos de detecção de colisão em um plano bidimensional (2D) em dispositivos móveis com Android. Os algoritmos são distância euclidiana (seção 2.2), distância de *Manhattan*(seção 2.3), sobreposição de retângulos (seção 2.4), teorema do eixo de separação (seção 2.5) e *pixel perfect*(seção 2.6).

1.2 Objetivos específicos

- Realizar pesquisa bibliográfica sobre medição de desempenho, consumo de energia e implementação de algoritmos de detecção de colisão em jogos;
- Implementar distância euclidiana, distância de *Manhattan*, sobreposição de retângulos, teorema do eixo de separação e *pixel perfect* em Java (*Android Studio*);
- Verificar a precisão dos algoritmos para diferentes formatos de *sprite*[1];
- Verificar o desempenho através da medição de tempo de execução;
- Verificar o consumo dos recursos como memória, processamento e energia, através do aplicativo *GameBench* presente no *GooglePlay*;
- Comparar e discutir os resultados, indicando particularidades de cada implementação.

1.3 Organização deste trabalho

Este trabalho tem a seguinte estruturação:

No Capítulo 2, os conceitos de detecção de colisão, os algoritmos propostos e as principais *engines* são apresentados.

O Capítulo 3 contém o processo de pesquisa e alguns trabalhos relacionados.

No Capítulo 4 são descritos como e o que foi usado para fazer a coleta de dados proposta pelo trabalho.

No Capítulo 5, os resultados obtidos são apresentados e discutidos.

No Capítulo 6, as considerações finais e sugestões de trabalhos futuros são apresentados.

[1] Em computação gráfica, *sprite* é uma imagem ou conjunto de imagens bidimensionais que representam um objeto do jogo.

2 FUNDAMENTAÇÃO TEÓRICA

O presente capítulo apresentará os algoritmos de detecção de colisão que serão avaliados neste trabalho e as principais *engines*, para fins de conhecimento do que é utilizado atualmente no desenvolvimento de jogos.

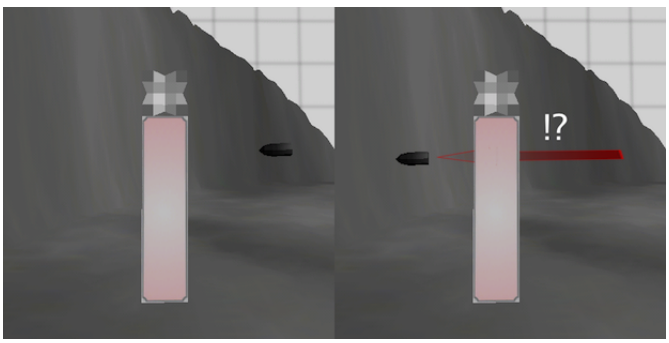
2.1 Detecção de Colisão

De acordo com a proposta de um jogo, a física pode ser um componente muito importante no cenário de mercado atual, onde "[...] consumidores são inflexíveis e querem mais realismo físico." (EBERLY; SHOEMAKE, 2004).

A detecção de colisão informa quando há interseção entre objetos do mundo virtual, aumentando o realismo do ambiente. Em algumas aplicações de Realidade Virtual (RV), existem propriedades dos avatares e dos objetos na cena que faz com que não ocorra sobreposição entre os objetos, realçando a sensação de imersão dos usuários. Em outras aplicações de RV onde simulações físicas são executadas, a detecção de colisão é imprescindível, e bastante complexa para lidar com um grande número de polígonos nas mais diversas posições, devendo ser rápida e precisa em alguns casos. Existem formas diferentes de detectar colisões e deve ser considerada aquela que melhor atende às características de um jogo. Uma das formas mais simples de detectar colisão entre personagens ou objetos em jogos é utilizar polígonos associados às *sprites*[1] destes personagens ou objetos, que os representam para fins de detecção de colisão.

A detecção de colisão é uma tarefa constante na simulação física e em algoritmos de planejamento de movimento, podendo ser dividida em 2 tipos básicos de testes de colisão: discreto e contínuo. Com a primeira opção, a posição dos objetos é atualizada iterativamente com um intervalo de tempo discreto e fixo. Feito isso, é verificado se existe interseção entre qualquer par de objetos. Este processo é simples e eficiente, mas apresenta desvantagens. A principal delas é o efeito chamado de *tunneling*.

Figura 1 – Exemplo de tunneling.



Fonte: NIELSEN (2007).

[1] Em computação gráfica, um *sprite* é um objeto gráfico que representa um personagem, objeto, ou parte do cenário(FERNANDES, 2009).

Ao realizar uma iteração da simulação, é possível que um objeto que se move rapidamente atravesse completamente um obstáculo (PAWASKAR, 2007) como notado na Figura 1.

A detecção contínua por outro lado faz uso de um intervalo de tempo variável. Este intervalo é ajustado para cada par de objetos que tem potencial para colidir. Podendo realizar uma projeção do movimento entre o intervalo de tempo estipulado e com isso verificar se há colisão pela projeção, e assim tratar esta colisão evitando assim o efeito de *tunneling*. A desvantagem deste método é que ele tem um custo computacional maior e sua implementação é mais complexa (SATHE; SHARLET, 2008).

Apesar da imprecisão, a detecção discreta é a mais utilizada juntamente com um algoritmo de volume envolvente (VE). VE é um volume fechado que contém completamente ou parcialmente um determinado objeto na simulação. Normalmente, objetos em jogos digitais têm forma complexa como o corpo de uma pessoa com braços e pernas, portanto testar a colisão buscando um nível de precisão elevado em tais objetos demanda bastante tempo de processamento (ERICSON, 2004). VEs na área de detecção de colisão são comumente usados para acelerar os testes de colisão entre os objetos em questão, em decorrência da sua geometria simples.

VEs simples como um círculo ou um retângulo (considerando um jogo 2D) são usados no lugar dos *sprites* originais para testes de colisão menos precisos e mais rápidos que podem, ou não, ser precedidos por algoritmos mais complexos de confirmação. Os VEs também podem ser usados como único método para julgar se houve colisão, em situações que a precisão de identificação de colisões não é algo crítico para o *software* e que pequenas imprecisões são toleráveis.

2.2 Distância Euclidiana

A forma mais usual para calcular a distância entre dois pontos em um espaço n-dimensional é a distância euclidiana (NETO; MOITA, 1998). Como um objeto possui uma área maior que um ponto, vamos utilizar um círculo para envolver os dois objetos que serão testados. A partir disso será calculada a distância entre os dois centros de círculos e seus respectivos raios, que será de acordo com o tamanho do *sprite* a ser utilizado.

Nesta situação o centro do círculo 1 está nas coordenadas x1 e y1 e o círculo 2 está nas coordenadas x2 e y2, logo para calcular a distância entre os centros dos círculos será utilizada a distância euclidiana:

$$\sqrt{(x1 - x2)^2 + (y1 - y2)^2} \tag{1}$$

Após este cálculo deve-se comparar se o resultado da distância euclidiana é maior que a soma do raio dos dois círculos. Só haverá colisão se a distância entre os círculos for menor ou igual que a soma dos raios, caso contrário não haverá colisão, conforme ilustra a Figura 2.

Figura 2 – Distância euclidiana entre um centro de círculo ao outro.

Fonte: autor.

Pode-se verificar abaixo um exemplo de implementação, em pseudocódigo, para o algoritmo de distância euclidiana, construido com o objetivo de ilustrar este algoritmo.[2]

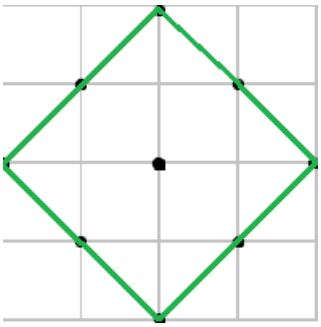
```
1  classe Circulo
2  {
3       publico real X; //centro do circulo em x
4       publico real Y; //centro do circulo em y
5       publico real raio; //distancia do centro a borda do circulo
6  }
7
8  publico booleano Funcao Colisao(Circulo C1, Circulo C2)
9  {
10          distancia = raizquadrada((C1.x - C2.x)^2 + (C1.y - C2.y)^2)
11          se (distancia > (C1.raio + C2.raio))
12                  //nao houve colisao
13          senao
14                  //houve colisao
15 }
```

2.3 Distância de *Manhattan*

A distância de *Manhattan* é bem semelhante a distância euclidiana, também calcula à distância entre dois pontos, porém não é com uma linha reta entre eles, como pode-se observar na Figura 4. Neste algoritmo a distância é calculada seguindo uma trajetória como de um táxi[3], contornando quarteirões até chegar no outro ponto, também conhecida como distância retilínea. Devido a este movimento puramente horizontal e vertical do centro para todos os lados até uma determinada distância, gera um losango, que é a figura geométrica que envolve os *sprites* neste algoritmo, conforme podemos verificar na Figura 3

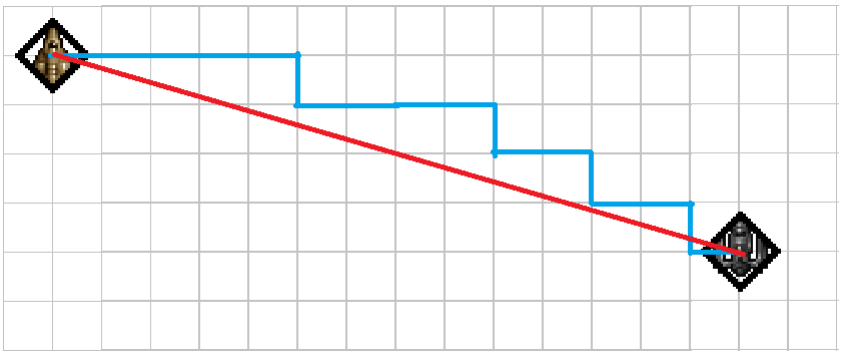
[2] Todos os pseudocódigos utilizam o esquema tradicional de eixos para representação de uma tela, ou seja, diferente do plano cartesiano onde o eixo Y cresce para cima, neste esquema o eixo Y cresce para baixo.

Figura 3 – Losango com distância de 2 entre o centro e a borda, gerado utilizando a distância de Manhattan.



Fonte: autor.

Figura 4 – Utilizando losango como volume envolvente, e comparando a distância entre os sprites na geometria do táxi (distância de *Manhattan* em azul) e geometria euclidiana (distância euclidiana em vermelho)



Fonte: autor.

A fórmula para calcular a distância é:

$$|x1 - x2| + |y1 - y2| \tag{2}$$

Essa estratégia é mais simples que a distância euclidiana, tornando a sua utilização mais eficiente (KUGLER; JÚNIOR; LOPES, 2003). Nessa estratégia também é necessário verificar se a distância é maior que a soma das distâncias do centro até a borda dos losangos.

[3] Essa geometria também é conhecida como geometria pombalina ou geometria do táxi, devido a essa propriedade.

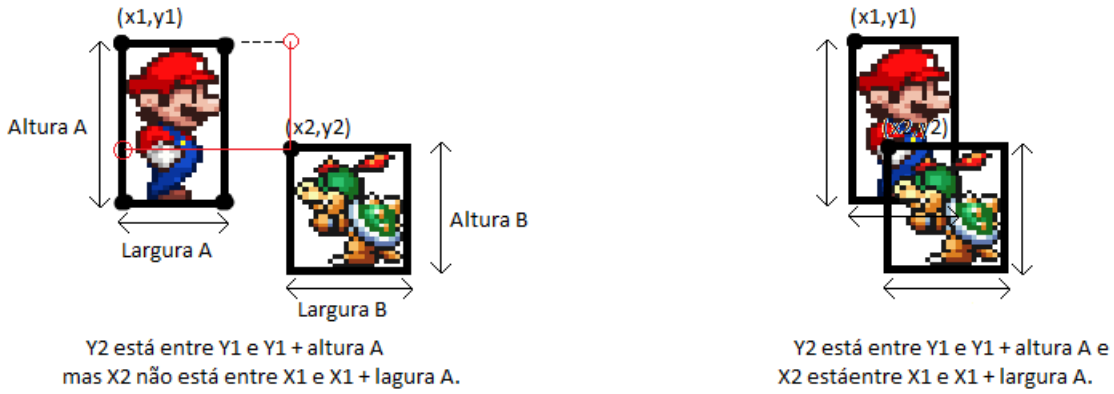
Para ilustrar a implementação do algoritmo distância de *Manhattan* desenvolveu-se o pseudocódigo abaixo.

```
1  classe Losango
2  {
3       publico real X; //centro do losango em x
4       publico real Y; //centro do losango em y
5       publico real raiodeManhattan; //distancia do centro a borda do losango
6  }
7  publico boleano Funcao colisao(losango L1,losango L2)
8  {
9       distancia = | L1.x - L2.x | + | L1.y - L2.y |
10      se (distancia > (L1.raiodeManhattan + L2.raiodeManhattan ))
11      // nao houve colisao
12      senao
13      // houve colisao
14
15 }
```

2.4 Sobreposição de retângulos

Utilizar retângulos como VE facilita a detecção de colisões uma vez que a associação entre retângulos e imagens é óbvia, pois por padrão a representação de imagens utiliza a altura e a largura para definir os seus limites.

Figura 5 – Na esquerda retângulos não colidindo e na direita retângulos colidindo.



Fonte: autor.

No algoritmo de sobreposições de retângulos, é possível verificar se houve detecção de uma forma bastante simples: testando se as áreas de dois retângulos estão se tocando ou se sobrepondo (AMATO, 1999). Os testes de sobreposição são na vertical e na horizontal das extremidades dos dois retângulos testados. A partir do ponto (x, y) do canto superior esquerdo dos retângulos é possivel obter as coordenadas das extremidades e com isso

verificar se um retângulo está sobreposto ao outro retângulo, conforme está presente na Figura 5.

Para isso, testa-se a coordenada (x, y) do primeiro retângulo está localizada entre o intervalo das coordenadas do canto superior esquerdo e do canto superior direito, repetindo o teste para os outros intervalos e, dessa forma testando todos os cantos do retângulo.

Para ilustrar a implementação do algoritmo sobreposição de retângulos foi criado o pseudocódigo abaixo.

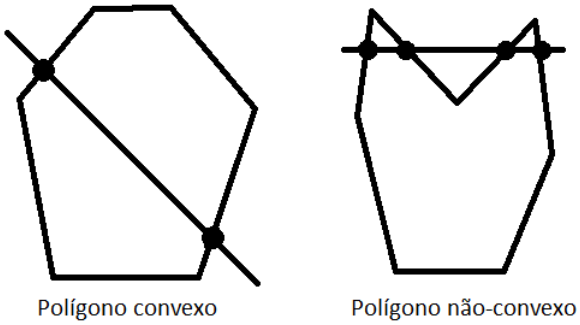
```
1  classe Retangulo
2  {
3       publico real X; //posicao em x
4       publico real Y; //posicao em y
5       publico real L; //largura
6       publico real A; //altura
7  }
8
9  publico boleano Colisao(Retangulo retangulo1, Retangulo retangulo2)
10 {
11         // testa se houve colisao
12         se ((retangulo1.X + retangulo1.L > retangulo2.X) E
13         (retangulo1.X < retangulo2.X + retangulo2.L) E
14         (retangulo1.Y + retangulo1.A > retangulo2.Y) E
15         (retangulo1.Y < retangulo2.Y + retangulo2.A))
16         //houve colisao
17         senao
18         //nao houve colisao
19 }
```

2.5 Teorema do eixo de separação

O teorema do eixo de separação, ou SAT (do inglês *Separating Axis Theorem*), é um método para determinar se dois polígonos convexos estão sobrepostos. O SAT é um algoritmo genérico rápido que pode remover a necessidade de ter código de detecção de colisão para cada par de tipo de forma, reduzindo assim o código e a manutenção. Um polígono é convexo se uma linha que atravesse este polígono é cruzada apenas duas vezes como pode ser visto na Figura 6.

Projeção é outro conceito que o SAT utiliza. Por exemplo, uma fonte de luz em cima de um polígono com os raios paralelos entre si, isso irá criar uma sombra na superficie. A sombra é a projeção de um objeto (BITTLE, 2010). A premissa deste algoritmo é: se dois objetos convexos não estão sobrepostos, portanto, existe um eixo no qual a projeção destes objetos não estão sobrepostas, ou simplesmente, se é possível traçar uma linha entre dois polígonos então eles não estão colidindo (CHONG, 2012). Os eixos a serem testados serão as normais de cada vértice do par de objetos. A principal vantagem deste

Figura 6 – Diferença entre polígonos convexos e não-convexos.



Fonte: autor.

algoritmo é que se em ao menos um eixo não houver sobreposição das projeções logo estes dois objetos não estão colidindo. A Figura 7 contém os polígonos convexos que foram associados às *sprites*.

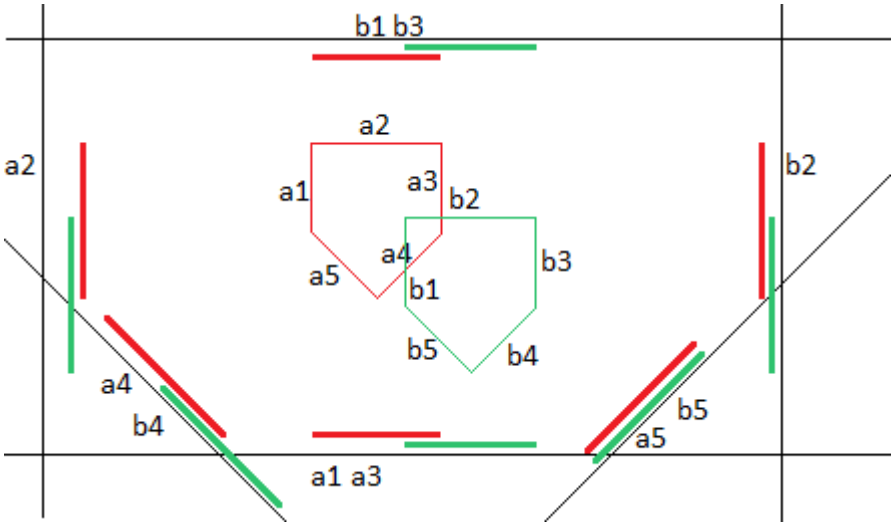
Figura 7 – Polígonos convexos utilizados nos *sprites*.

Fonte: autor.

Para calcular a projeção primeiro foi necessário tratar os vértices dos objetos como vetores 2D, desta forma é possível realizar o produto escalar, entre os vetores a e b:

$$a \cdot b = a_x b_x + a_y b_y \tag{3}$$

Figura 8 – Duas formas convexas colidindo, há sobreposição em todas as projeções.



Fonte: autor.

A fórmula para realizar a projeção entre os vetores a e b (que também será um vetor):

$$proj_x = \frac{a \cdot b}{(b_x^2 + b_y^2)} b_x \tag{4}$$

$$proj_y = \frac{a \cdot b}{(b_x^2 + b_y^2)} b_y \tag{5}$$

Onde proj será o vetor projeção (BURNS; SHEPPARD, 2011).

A Figura 8 deixa claro como é o funcionamento deste algoritmo, só existe colisão caso tenha sobreposição das projeções em todos os eixos.

Abaixo criou-se o pseudocódigo de uma possível implementação do SAT:

```
1  classe p1
2  {
3          publico real Vertices[];
4          (...)
5          publico Lista<Vetor2D> GetVertices()
6      {
7                  return Vertices;
8      }
9
10 }
11
12 classe p2
13 {
14         publico real Vertices[];
15         (...)
16         publico Lista<Vetor2D> GetVertices()
17         {
18                 return Vertices;
19         }
20
21 }
22
23
24 publico booleano temSobreposicao(p1 jogador, p2 inimigo)
25 {
26         Lista<Vetor2D> eixos1;
27         Lista<Vetor2D> eixos2;
28         //GetNormais retorna a lista dos eixos normais
29     //para cada lado do poligono
30         eixos1 = GetNormais(jogador.GetVertices());
31         eixos2 = GetNormais(inimigo.GetVertices());
32
33         //concatenamos as duas listas
34         Lista<Vetor2D> eixos;
35         eixos.addAll(eixos1);
36         eixos.addAll(eixos2);
```

```
37
38          para(i=0;i<eixos; i++)
39          {
40                  //Devemos verificar o ponto minimo e maximo
41          //de cada vertice em cada eixo.
42                  jmin = projecao_min(jogador.GetVertices(), eixos(i));
43                  imin = projecao_min(inimigo.GetVertices(), eixos(i));
44                  jmax = projecao_max(jogador.GetVertices(), eixos(i));
45                  imax = projecao_max(inimigo.GetVertices(), eixos(i));
46
47                  //Caso nao tenha sobreposicao neste eixo,
48          //nao e necessario verificar os demais, pois nao ha colisao
49                  se( jmin > imax ou imin > jmax)
50                  {
51                          //nao ha colisao
52                  retorna falso;
53                  }
54          }
55
56          //ha colisao
57          retorna verdadeiro;
58  }
```

2.6 Pixel perfect

Este é o método mais preciso, pois verifica se algum pixel da imagem de um objeto está sobreposto a um pixel da imagem de outro objeto. Portanto se algum pixel de um objeto colidir com algum pixel do outro objeto a colisão é detectada.

No entanto este algoritmo apresenta um maior custo computacional (NETO, 2013), pois cada imagem passa a ser considerada como uma matriz de números inteiros, com as áreas transparentes recebem o valor "0". Então o algoritmo percorre pixel a pixel de ambas imagens para determinar se alguma coordenada possui dois valores diferentes de zero para um mesmo pixel na tela. Ocorrendo isso fica determinado que houve uma colisão. A principal vantagem deste algoritmo é de permitir a detecção de colisão entre formas arbitrárias.

Figura 9 – Representação da matriz do *sprite* para detectar a colisão com *pixel perfect*

Fonte: autor

Como pode ser verificado na Figura 9, a única área que pode ocorrer a colisão é a área na cor branca.

Para ilustrar melhor a implementação do *pixel perfect* abaixo é apresentado a criação do pseudocódigo do algoritmo.

```
1  classe Imagem{
2  publico real X; // origem da imagem1 em X
3  publico real Y; // origem da imagem1 em Y
4  publico real L; //largura
5  publico real A; //altura
6  //Matriz com valores 0 para preto e 1 para branco
7  publico real matriz[L,A];
8  }
9  publico booleano Funcao Colisao(Imagem imagem1, Imagem imagem2){
10
11 para(x1 = 0; x1< imagem1.L x1++){
12     para(y1 = 0; y1 < imagem1.A; y1++){
13         para(x2 = 0; x2 < imagem2.L;x2++){
14             para(y2 = 0; y2< imagem2.A; y2++){
15                 se (x1+imagem1.X == x2+imagem2.X
16                    e y1+imagem1.Y == y2+imagem2.Y){
17                    se (imagem1.matriz[x1,y1] != 0
18                       e imagem2.matriz[x2,y2] != 0){
19                              //houve colisao
20                        }
21                    Senao{
22                      //Nao houve colisao
23                      }
24                }
25             }
26         }
27     }
28 }
```

2.7 Principais Engines

Unity é uma *engine* de jogo completo, provido de uma renderização poderosa e uma ferramenta de *workflows* intuitivos (UNITY, 2017), sendo por origem uma ferramenta de jogo proprietário, desenvolvido e mantido pela *Unity Technologies*. A *engine* textitUnity pertence ao grupo *Framework Module Engine*, que possui um número distinto de módulos, fornecendo subsistemas, o que apresenta uma ótima relação entre customização e complexidade (ANDERSON et al., 2013). A *Unity* permite o desenvolvimento de jogos para mais de dez plataformas, dentre elas *Windows*, Linux, Mac OS, iOS, *Android*, consoles e navegadores *web*. Existem sete módulos comuns nas *engines* de jogos atuais: Gráfico, Física, Detecção de colisão, Entrada/Saída, Som, Inteligência Artificial e Rede

(PETRIDIS et al., 2010). *Unity* faz uso de volumes envolventes com formas geométricas como circulos, caixas, poligonos e suas variações para realizar a detecção de colisão tanto em ambientes 2D como em 3D.

Unreal é uma *engine* gráfica que traz um *framework* de desenvolvimento completo que fornece uma vasta gama de tecnologias, ferramentas de criação de conteúdo, suporte de infraestrutura e conteúdo, e possui licença proprietária liberada para uso não-comercial(UNREAL, 2017). Foi escrita em C++ e é utilizada para o desenvolvimento de diversos jogos AAA[4]. Também roda em diversas plataformas como aparelhos *Apple* iOS, Google *Android*, Mac OS, *Microsoft XboxOne*, *Sony PlayStation* 4 e PSVita e *Windows* PC, com suporte a versões 32/64 bits e DirectX 9 e 11. O sistema de física da *engine* é provido pela *PhysX* da NVIDIA e propõe duas possibilidades para tratar de colisões: simples com formatos geométricos mais primitivos como caixas, esferas, *convex-hull*[5]e complexa com malhas triangulares que cobrem o objeto tridimensional com mais detalhes.

Como jogos para celulares também podem ser criados com JavaScript e HTML5 existem algumas *engines* que auxiliam na criação destes jogos. Dentre estas engines há a *Phaser* que é gratuita e foi desenvolvida no MIT. Segundo DAVEY, essa *engine* é projetada para criar jogos que serão executados em navegadores de *desktops* e dispositivos móveis. Além de ser focada para navegadores de dispositivos móveis, a *Phaser* possui algumas outras características, como: facilidade em carregar componentes de um jogo como imagens, áudio, texturas, scripts, entre outros; renderização através de canvas e WebGL (ambas tecnologias introduzidas com o HTML5). A *Phaser* possui três sistemas de física, que são responsáveis pela gravidade e tratamento de colisões já integrados com a *engine* para o desenvolvimento de jogos que necessitem de física. O sistema mais simples no qual todo objeto possuirá um corpo retangular sem rotação, que é denominado Arcade, é geralmente utilizado para colisões rápidas ou colisões sem complexidade. Já para detectar colisões entre corpos circulares ou com rotação é necessário utilizar o sistema Ninja, mas seu funcionamento é semelhante ao Arcade. A Phaser também conta com um sistema de física mais completo denominado de P2, no qual é possível detectar colisão em objetos de qualquer forma, utilizando uma máscara sobre o *sprite* (semelhante ao *pixel perfect*) e materiais (elementos que irão afetar o modo que um corpo reage a outro corpo).

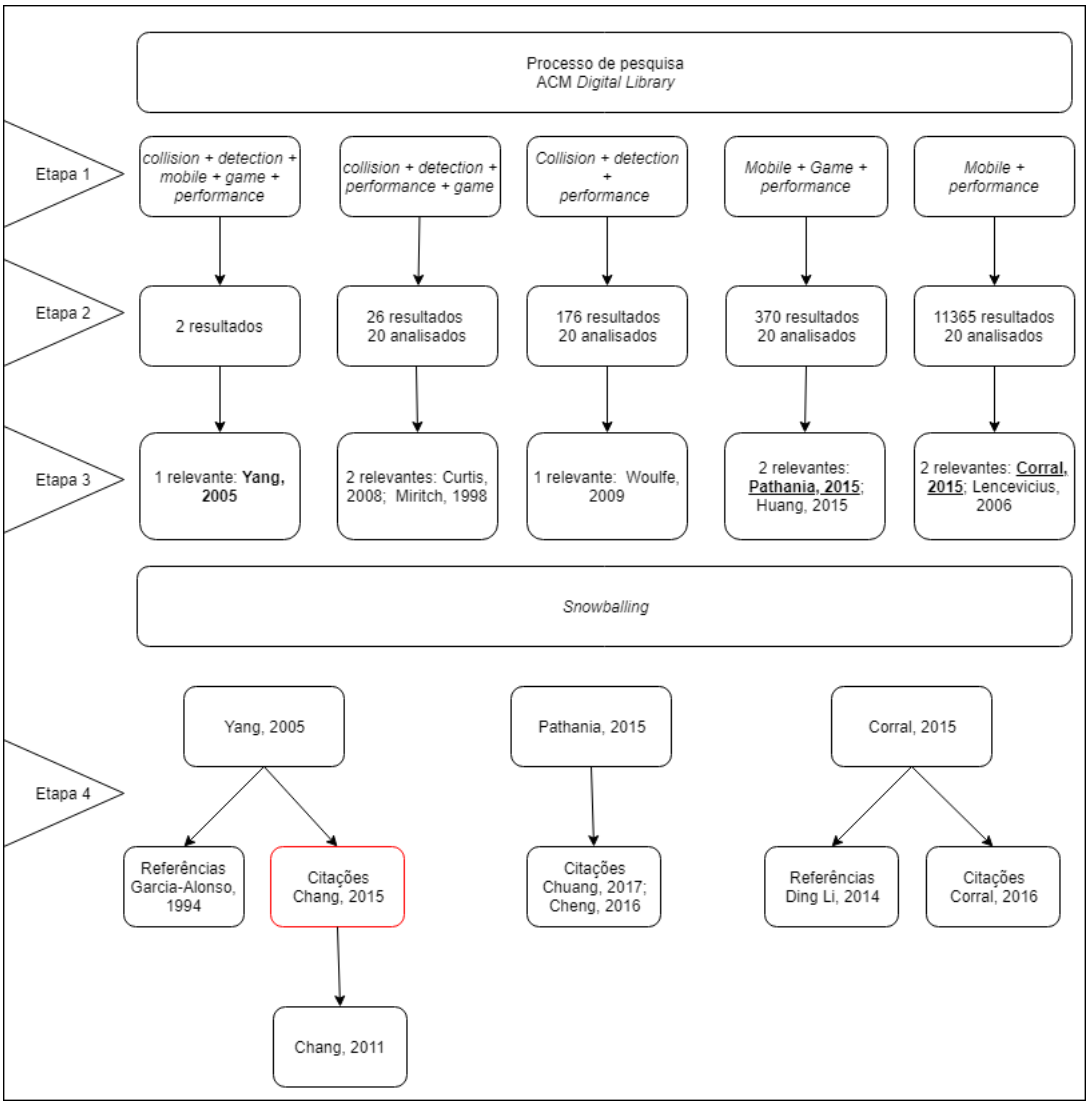
[5] Jogos AAA ou triplo A, como são pronunciados trata-se de uma classificação informal para jogos com grandes orçamentos e niveis de divulgação.

[5] *convex-hull* se trata do menor polígono convexo que envolve um determinado objeto.

3 TRABALHOS RELACIONADOS

A base de dados bibliográficos ACM *Digital Library*[1] com o auxílio da ferramenta de pesquisa avançada foi utilizada na execução deste trabalho porque apresentou resultados mais próximos com a proposta deste trabalho em comparação com as bases Google *Scholar* e IEEExplore. O processo de busca foi composto por quatro etapas, conforme pode ser visto na Figura 10. Essas etapas são descritas a seguir.

Figura 10 – Diagrama do processo de pesquisa.



Fonte: autor.

A etapa 1 consistiu na busca de trabalhos relacionados utilizando 5 conjuntos de palavras chaves. O número total de trabalhos encontrados para cada conjunto de palavras na primeira etapa é apresentado na etapa 2, onde foram analisados os primeiros 20 resultados ordenados por relevância. O critério temporal não foi adotado porque a

[1] http://dl.acm.org/

detecção de colisão é um tema presente na área de jogos digitais desde os primeiros videogames da década de 80.

Na etapa 3 é realizada uma seleção tendo como critério o título e o resumo do artigo. Nessa etapa são selecionados apenas os artigos que contribuem para este trabalho com maneiras de verificar o uso dos recursos em dispositivos móveis com sistema operacional Android e/ou algoritmos de detecção de colisão. Adicionalmente, são descartados os trabalhos que foram selecionados por conjuntos de palavras anteriores. Ao final do processo de pesquisa através do ACM *Digital Library* foram selecionados 8 trabalhos relacionados.

Na etapa 4 foi adotada outra técnica de pesquisa conhecida como "*snowballing*" ou bola de neve, que consiste em, a partir de um artigo, verificar tanto os trabalhos citados pelo artigo como também os trabalhos que o citaram (VOGT, 1999). Esta técnica pode ser aplicada quando se tem dois propósitos. Primeiro, como um método informal de se alcançar novos artigos. Em segundo lugar, a amostragem da bola de neve pode ser aplicada como uma metodologia formal, quando se busca uma pesquisa exploratória qualitativa. Através dessa técnica foi possível obter mais 6 trabalhos.

Todos os trabalhos relacionados estão divididos em duas tabelas: a Tabela 1 contém os trabalhos que utilizam alguma medida para analisar o desempenho e o consumo dos recursos em dispositivos móveis, e a Tabela 2 contém os trabalhos com propostas para detecção de colisão.

3.1 Desempenho e consumo de recursos

Nesta seção são apresentados os trabalhos que se relacionam com os objetivos de coletar os recursos do dispositivo móvel e também verificar o desempenho. Apesar do objetivo deste trabalho ser em jogos 2D, também foram considerados trabalhos que utilizavam outros processos como carga de trabalho, pois identificam limitações em dispositivos móveis e maneiras de coletar estas informações.

Uma alternativa para controlar o consumo de energia é realizar uma escalada dinâmica da voltagem e frequência do *chip* heterogêneo o qual contém CPU e GPU juntos. PATHANIA et al.(2015) sugere esta alternativa como uma maneira de consumir menos energia e manter um bom desempenho nos jogos. Através de um gerenciador de energia que foi implementado no *Android* OS da plataforma Odroid-XU+E, que é atualmente presente no *smartphone* Samsung Galaxy S4, alcançando um acréscimo de 20% em desempenho por *watt* comparado com o estado da arte. Foi utilizado um conjunto de 10 jogos para realizar os testes e aprendizagem do comportamento do sistema, e também os sensores *on-board* do Texas *Instrument* INA231 para obter as medições do CPU e GPU independentemente. Devido a falta de programas que possibilitam verificar o desempenho, foi necessário se basear pelo nível de utilização do CPU para uma representação aproximada da carga de trabalho como fator de atividade. Utilizando esta técnica de

gerenciamento de energia é possível rodar jogos em dispositivos móveis por mais tempo. Este trabalho se relaciona com a nossa proposta pois evidencia algumas barreiras quando se trata em verificação de desempenho e consumo de recursos no *Android*.

CORRAL et al.(2015), em sua pesquisa sobre a otimização do Kernel do sistema operacional *Android*, define cenários de testes para avaliar o consumo de bateria.

Para realizar a análise de uso da bateria foi utilizado o aplicativo *Battery Drain Analyzer (Pro)*, pois é possível obter o quanto de energia foi gasto por cada aplicativo ou componente de *hardware*. Além disso para se ter uma maneira de rastrear o uso da bateria de uma maneira mais automática foi utilizado um outro aplicativo chamado *Battery Monitor Widget*. Com este aplicativo é possível adquirir relatórios de minuto a minuto do uso da bateria sendo este relatório salvo em um arquivo na memória do aparelho. A principal contribuição deste trabalho é a utilização de aplicativos para monitorar o consumo da bateria.

Quando se fala em jogos para dispositivos móveis, é possível utilizar a nuvem para tentar diminuir o consumo de bateria e ganhar mais desempenho. Com este propósito HUANG et al.(2015), tendo por base a plataforma *open source GamingAnywhere*, quantificou o desempenho de diversos jogos para celular utilizando a nuvem e comparou com o desempenho desses jogos quando rodando nativamente no aparelho. Para os testes foram considerados dois dispositivos, um *tablet* e um *smartphone*, ambos com o *Android* 4.4.2. As ferramentas *UseMon* e *Current Widget* foram utilizadas para medir e coletar dados de uso da CPU e consumo de bateria. O estudo de Huang identificou outros componentes são consumidores de energia, como: o uso da rede 3G que pode aumentar de 30% a 40% o consumo da bateria se comparado com a rede *WiFi*; e o processo do toque na tela que consome recursos de CPU e energia. Este trabalho contribui para a importância de se realizar os testes desativando todos os meios de comunicação do dispositivo móvel e também com aplicativos que permitem verificar o uso do CPU e o consumo da bateria.

LENCEVICIUS; METZ(2006) introduz um método de verificação de desempenho que foi utilizado por muitos pesquisadores em diversas áreas da computação. Lancevicius apresenta a implementação e aplicação de asserções de desempenho, ou seja, técnicas para coletar informações quanto o uso dos recursos em programas para dispositivos móveis, conseguindo resolver problemas de especificações, avaliações que estavam em aberto em pesquisas anteriores. Este trabalho propõem um *framework* simples e eficaz de realizar verificações de desempenho que consiste em contabilizar o tempo de execução em determinados métodos e funções. A principal contribuição deste trabalho é de explorar uma alternativa para a verificação de desempenho em dispositivos móveis.

Criar um aplicativo que é eficiente no consumo de energia é um dos objetivos dos desenvolvedores de aplicativos para dispositivos móveis, porém segundo LI; HALFOND(2014) falta aos desenvolvedores um guia de como consumir menos energia nos aplicativos. Li realizou uma análise empírica de melhoramento do código seguindo algu-

mas sugestões comumente associadas ao consumo de energia, verificando que o uso da memória não afeta de uma maneira significativa o consumo de energia. Para a avaliação foi comparado o grau de economia de energia em relação à versão original do código. Servindo assim como um guia para desenvolvedores de aplicativos, com boas práticas para economizar energia em dispositivos móveis. A principal contribuição deste trabalho foi em relação a otimização do código para um menor consumo de energia.

Eficiência energética é um problema crítico quando se trata de dispositivos móveis movidos a bateria. Com a alta popularidade dos jogos para celulares com gráficos cada vez mais sofisticados cresce a necessidade por um regulador de potência tanto para CPU e GPU on-line. CHUANG; CHEN; HUANG(2017) propôs um regulador de potência de CPU-GPU adaptativo online para jogos em dispositivos móveis para minimizar o consumo de energia. O conceito foi implementado em um Google Nexus 7, avaliado usando 3 jogos. Cada jogo foi testado 5 vezes, para minimizar erros de medições. Foi utilizado uma média de quadros por segundo, e a média do consumo de energia como métrica. Para testar a eficiência energética Chuang compara o regulador adaptativo com um regulador de desempenho que sempre ajusta as frequências para o valor máximo. Apesar de ter algumas quedas de quadros por segundo, o regulador adaptativo consegue resultados semelhantes ao regulador de desempenho, consumindo mais energia. A principal contribuição desse trabalho é com relação aos diferentes perfis existentes que possibilitam a economia da bateria.

Com o grande avanço dos jogos nos celulares mais modernos equipados com SoCs multiprocessadores, destacou-se os problemas de consumo de energia e a vida da bateria, reduzindo assim o poder dos jogos nos dispositivos móveis. O *design* de gerenciadores independente de energia para CPU e GPU nas plataformas modernas resulta em um desperdício de energia. Devido a desconsideração da interação entre CPU-GPU em um comportamento do jogo como carga de trabalho. CHENG et al.(2016) realizou uma análise através das chamadas do sistema e mais a API *OpenGL* informativas, para caracterizar a carga de trabalho do jogo de uma forma menos complexa. Baseado na identificação do comportamento do jogo como carga de trabalho, foi proposto um gerenciador de energia integrado de CPU-GPU considerando o comportamento e inclusive implementando uma política de economia de energia. Cheng utilizou como métrica de desempenho tanto o número de quadros por segundo quanto o nível de detalhamento do jogo. Foram selecionados seis jogos populares para *Android*, com jogos mais complexos em 3D a jogos mais simples em 2D com diferentes frequências de interação com o usuário para realizar os testes. Para analisar o consumo de energia foi utilizado o dispositivo mais comum, o monitor de energia. Os testes foram realizados na plataforma Odroid-PC rodando o sistema operacional *Android* 4.0.4. Como eficiência no consumo de energia foi utilizado a métrica de quadros por segundo/Watt (FPS/Watt), conseguindo uma eficiência-energética em média de 5% maior do que o estado da arte. A principal contribuição deste trabalho é com as métricas

utilizadas para analisar o consumo de energia e o desempenho dos jogos.

CORRAL et al.(2016) apresenta um novo experimento e teste para economizar energia nos *smartphones*, implementada no kernel do sistema operacional *Android*. A abordagem adotada por Corral foi de implementar extensões do Kernel que avaliem o status do dispositivo e habilitem perfis econômicos sem a intervenção do usuário. Desta maneira, Corral conseguiu agir no nível do kernel, evitando a sobrecarga associada ao envio de mensagens do espaço do usuário até os módulos que fazem interface com o *hardware*. Os testes foram realizados em 3 aparelhos LG Nexus 5 novos. A observação do desempenho foi realizada através do aplicativo *Geekbench*. A principal contribuição desse trabalho é utilizar um aplicativo capaz de analisar o desempenho em dispositivos *Android*.

Tabela 1 – Desempenho e consumo de recursos.

Autor	**Objetivo**	**critério**	**SO**	**Medida**
Pathania	Escalada dinâmica de voltagem e frequência	Consumo de Energia Desempenho Utilização de CPU/GPU Frequência	Android	Nível de utilização do CPU
Corral	Otimizar o kernel do Sistema Operacional	Consumo de energia Desempenho	*Android*	*Battery Drain Analyzer (Pro) Battery Monitor Widget Antutu Benchmark*
Huang	Quantificar o desempenho de um jogo	Desempenho e bateria	*Android*	*Usemon* e *Current Widget*
Lencevicius	Especificações e validações de desempenho para celulares.	Asserções de desempenho	-	Identificadores de inicio e fim de eventos.
Ding	Reduzir o consumo de bateria	Desempenho e bateria	*Android*	Tamanho de pacotes de redes, uso de memória, tamanho do vetor de acesso e tipos de invocação.
Chuang	Melhorar a eficiência da bateria	Regular a potência do do CPU e GPU para reduzir o consumo de bateria	*Android*	Tempo de execução, consumo de energia e quadros por segundo.
Cheng	Gerenciamento de energia integrado de CPU-GPU com consciência do comportamento da carga de trabalho	Desempenho e consumo de bateria	*Android*	Quadros por segundo, nivel de detalhamento, quadros por segundo por watt.
Corral	Economizar bateria	Alterações no kernel	*Android*	*Geekbench* para avaliar o desempenho, aplicativo padrão para verificar o consumo da bateria.

Esta tabela visa identificar o autor, os objetivos, os critérios, o sistema operacional e por fim como foram mensurados os resultados.

3.2 Algoritmos de detecção de colisão

Nesta seção são apresentados os trabalhos relacionados que possuem diferentes algoritmos de detecção de colisão, mas que também identificam a importância e desafios destes algoritmos.

CURTIS; TAMSTORF; MANOCHA(2008) apresenta uma nova maneira de acelerar a detecção de colisão para modelos deformáveis. Esta técnica pode ser aplicada a todos modelos triangulares e reduz significamente o número de testes entre recursos da malha, como vértices, arestas e faces. Na proposta são apresentadas duas contribuições principais: primeiro na eliminação de testes duplicados que naturalmente surgem em formulações baseadas em triângulos e segundo providenciando uma maior eficiência na redução da área de testes. Estas melhorias se tornam possíveis devido à utilização do conceito de triângulos representativos obtendo assim os benefícios de hierarquias de volumes envolventes, e a redução da área de testes se dá excluindo dos testes os pares de triângulos que claramente não estão se sobrepondo. A eficiência se dá pelo número de falso positivos. O equipamento utilizado para estes testes foi um computador com processador Intel Xeon 3 Ghz e 3 GB de memória RAM com o sistema operacional de 32 bits *Windows* XP. Para verificar o desempenho desse algoritmo foi feito uma comparação com outros 3 algoritmos que utilizam a mesma estrutura de hierarquia de volumes envolventes. O primeiro é a forma mais básica que executa todos os 15 testes elementares para cada par de triângulos, o segundo algoritmo verifica os pares de triângulos não adjacentes primeiro e utiliza esse resultado para definir os testes para os pares adjacentes. Esse método também elimina testes duplicados e utiliza características de volumes envolventes e por último usa um conceito de representação para testes duplicados mas não utiliza características de volumes envolventes para melhorar o escopo. Os resultados do algoritmo proposto neste artigo tem uma melhoria nos testes de 7.8% a 34%, porém uma queda considerável no número de quadros por segundo. Este trabalho se relaciona com a nossa proposta pois apresenta um algoritmo de detecção de colisão, testa o desempenho e compara com outras abordagens.

MIRTICH(1998) apresenta um algoritmo de detecção de colisão para objetos poliédricos especificados por uma representação de limites, conhecido como Voronoi-clip ou V-Clip. Este algoritmo rastreia os pares de recursos mais próximos entre um poliédro convexo. Este trabalho é importante pois apresenta um algoritmo de detecção de colisão que obtém ganhos de desempenho e robustez comparado com outros algoritmos. Mirtich também mostra que com este algoritmo é possível reduzir a complexidade no código e que é capaz de lidar com poliedros penetrantes. A implementação não tem tolerâncias numéricas e não exibe problemas cíclicos. Os recursos de um poliedro são os vértices, bordas, e faces formando assim os limites do objeto. Além do V-Clip Mirtich também compara com outros dois algoritmos como o Lin-Canny e o GJK Melhorado, o cenário de teste foi feito de uma maneira simplificada até porque os algoritmos não são completos para lidar

com um ambiente com muitos objetos colidindo. Dessa forma, foi testada a coerência da colisão entre apenas dois objetos. Após diversos testes, Mirtich conclui que o V-Clip tem uma significativa melhora sobre o Lin-Canny por três razões: o V-Clip consegue lidar com penetrações, e bem como objetos não convexos; é mais simples de codificar; e é mais robusto. Entretanto, o V-Clip e o GJK melhorado não demonstraram uma vantagem de robustez um sobre o outro. No geral, V-Clip obteve a menor contagem de operações de ponto flutuante dos três algoritmos. Este trabalho contribui apresentando um novo algoritmo de detecção de colisão que consegue aumentar o desempenho.

Segundo WOULFE; MANZKE(2009) a detecção de colisão é um componente vital de aplicações de diversos campos, mas ainda não existia uma maneira de desenvolvedores analisar em quão adequado o algoritmo de detecção de colisão está dentro do espectro de cenários que podem ser encontrados. Este trabalho propõe um *framework* para poder se verificar o desempenho de detecção de colisões interativas, o qual consiste em um *benchmark* genérico que pode ser adaptado usando um número de parâmetros para criar assim uma grande variedade de *benchmarks* práticos. O *framework* proposto por Woulfe permite aos desenvolvedores testarem a validade dos seus aplicativos interativos em recriar os cenários e rapidamente determinar qual o algoritmo mais adequado. Para demonstrar a utilidade deste *framework*, foi adaptado para funcionar com três algoritmos de detecção de colisão, utilizando o *Bullet Physics* SDK. O cenário de testes contou com uma *engine* de física (*Bullet Physics* SDK), pois as aplicações interativas modernas utilizam este recurso para controlar o movimento dos objetos. O cenário de testes consiste em um cubo com tamanho determinado pelo número de objetos multiplicado por 50. Alguns parâmetros também foram considerados para adequar aos diferentes casos de colisão, como ciclos, ou seja, a frequência de testes se há colisão. Por padrão este parâmetro tem o valor 1000, a quantidade de objetos que serão testados, os tipos de objetos, como esferas, cubos, cilindros ou cones. Foram utilizados esferas a localização dos objetos, a massa dos objetos, aceleração dos objetos, velocidade linear e velocidade angular. Há também um parâmetro de observações o que faz com que os testes possam ser repetidos. Para os testes foram gravados o tempo de execução em duas fases diferentes de execução do algoritmo, a fase onde os objetos estão mais distantes e ficam fora das comparações de detecção de colisão, e a outra fase onde são considerados os objetos que estão mais próximos, onde é realizado os testes mais minuciosos para determinar se há colisões. Este trabalho contribui com cenários de testes que indicam as variáveis que podem existir em algoritmos de detecção de colisão.

Para YANG; CHENG; PAN(2005) a detecção de colisão em jogos para dispositivos móveis não deve ser feita da mesma maneira de como é feita nos computadores, pois não dispõem dos mesmos recursos computacionais e nem de um mesmo tamanho de tela, eles propõem um novo algoritmo de detecção de colisão com multinível de resolução. Yang verifica o desempenho desta nova proposta através de um jogo de bilhar produzido

pela *Bird Company* e também comparando seu desempenho com o desempenho de um algoritmo comum. Também é verificado o desempenho através de testes com até 20 objetos e outro teste com até 2000 objetos, onde o algoritmo demonstra possuir um melhor desempenho. A principal contribuição deste trabalho é um algoritmo de detecção de colisão voltado para dispositivos móveis.

Contando com o avanço computacional proporcionado pelo computador *Hewlett-Packard* 350 SRX em Julho de 1987, GARCIA-ALONSO; SERRANO; FLAQUER(1994) desenvolveu um algoritmo capaz de detectar colisões em tempo real, através de cálculos entre os limites dos recipientes dos corpos, utilizando uma estrutura de dados do tipo *voxel*, com ponteiros para elementos da superfície de cada objeto. O programa se divide em 3 módulos: 1. Colocar os objetos em uma referência global através das matrizes dos objetos ou a inversa destas matrizes, em um instante fixo de tempo ou quadro. 2. Renderização em 3D com uma interface com o usuário. 3. Módulo de análise de interferência que verifica se há interferência entre qualquer par de objetos no sistema de referência global. Este último módulo faz as análises em 4 passos, primeiro verifica se há colisão entre as matrizes de interesse (matrizes que contém os limites de cada objeto), calcula as interferências entre pares de caixas, calcula as interferências entre voxels e por último calcula as interferências entre pares de facetas. As matrizes de interesse contém também uma "*flag*" que indica se um objeto necessita ou não de análise, eliminando assim os objetos que não colidem entre si. Com base nos testes realizados foi possível concluir que um modelo geométrico hierárquico que representa polígonos espacialmente utilizando *voxels* é uma maneira eficiente de se obter a detecção de colisão em tempo real. É possível aumentar o desempenho ao utilizar facetas equilaterais. Esse trabalho se relaciona com o nosso por se tratar de um dos primeiros algoritmos para a realização da detecção de colisão.

Um algoritmo para gerar uma caixa delimitadora ou (*bounding box*) orientada em 3D foi publicado em 1985 por Joseph O'Rourke, mas é lenta e extremamente difícil de implementar. Então CHANG; GORISSEN; MELCHIOR(2011) propõem uma nova abordagem, onde a computação do volume mínimo da caixa delimitadora orientada ou (OBB) é formulada como um problema de otimização sem restrições no grupo rotacional SO (3,R). Sendo resolvido utilizando um método híbrido combinando algoritmo genético e o algoritmo Nelder-Mead. Este método é analisado e então comparado com técnicas do estado da arte. Foram usados 300 casos de teste. Os resultados comprovaram que este novo método é tão rápido quanto o algoritmo mais rápido (O'Rourke's) ou mais preciso que a heurística mais rápida no principal componente de análise. A principal contribuição deste trabalho é a possibilidade de utilizar outros algoritmos com a finalidade de otimizar a detecção de colisão.

Tabela 2 – Detecção de colisão.

Autor	**Objetivo**	**cenário de teste**	**Comparações**	**Medidas**
Curtis	Acelerar a detecção de colisão	Modelos deformáveis	3 algoritmos com mesma estrutura hierárquica de volumes envolventes	Quantidades de testes realizados, quadros por segundo
Mirtich	Novo algoritmo de detecção de colisão para objetos poliédricos	Variações entre coerência da colisão entre somente dois objetos	Comparado com outros 2 algoritmos semelhantes	Contagem de operações com ponto flutuante
Woulfe	A criação de um *framework* que funciona como um avaliador de detecção de colisão	Objetos dentro de um espaço cúbico colidindo entre si ou com o chão.	Foram utilizados 3 algoritmos diferentes, e uma engine de física para recriar uma grande quantidade de cenários variados	Tempo de execução em diferentes fases de execução dos algoritmos
Yang	Desenvolvimento de um algoritmo de detecção de colisão voltado para celulares	2 casos diferentes um com poucos objetos e outro com muitos objetos, calculando as variações de velocidade, distribuições e direção dos objetos	Foi realizado o mesmo os mesmos testes no algoritmo de detecção de colisão padrão ou mais comum.	Número de comparações entre objetos.
Chang	Otimizar a geração de caixas de limites orientadas para objetos	Aproximadamente 300 conjuntos de pontos, formando diferentes formas geométricas desde formas simples até objetos anatômicos formados por milhões de pontos.	Outros 6 algoritmos considerando a categoria, a acurácia, a complexidade e implementação.	Tempo computacional, desempenho, taxa de falha e erro relativo.
Garcia-Alonso	Detectar colisões entre dois objetos em movimentos e analisar em tempo real	Testes com 2 objetos, dividindo os objetos em várias facetas, verificando interferências entre eles. Simulação da Antena Daisy	Lista de facetas de um objeto contra a lista de arestas do outro objeto, intersecção entre matrizes.	Tempo de execução para computar as interferências entre objetos volumétricos com matrizes de ordem M.
Chang	Otimizar a geração de caixas de limites orientadas para objetos	Aproximadamente 300 conjuntos de pontos, formando um grande número de diferentes formas geométricas desde formas simples até objetos anatômicos formados por milhões de pontos	Com outros 6 algoritmos considerando a categoria, a acurácia, a complexidade e implementação.	Tempo computacional, desempenho, taxa de falha e erro relativo.

Esta tabela visa identificar o autor, os objetivos, os cenários de testes utilizados, as comparações e por fim como foram mensurados os resultados.

4 METODOLOGIA

Este trabalho se caracteriza como uma pesquisa experimental, pois segundo PRODANOV; FREITAS(2013) pesquisas experimentais manipulam diretamente as variáveis relacionadas com o objeto de estudo, proporcionando o estudo da relação entre as causas e os efeitos de determinado fenômeno.

4.1 Ambiente de execução

Os testes de desempenho e consumo dos recursos foram executados em um *smartphone Asus Go* modelo X00ADA com o sistema operacional *Android* versão 6.0.1 e com a seguinte configuração: processador Arm Qualcomm 8916 Quad Core com 1GHz, memória RAM com 2 GB DDR, memória interna de 16GB, bateria de lítio com 2600 mAh e tela com resolução de 1280x720 *pixel*. Para obter os resultados dos testes foi utilizado o aplicativo *GameBench* versão 5.01. Todos os testes são com a bateria com carga inicial de 98%, o volume em mudo e com modo avião ativo, deixando o dispositivo sem nenhuma forma de comunicação, isolando assim a utilização de seus recursos. Os algoritmos foram desenvolvidos com o *Android Studio* versão 3.0.1.

4.2 Casos de teste

Os algoritmos de detecção de colisão testados são: distância euclidiana, distância de *Manhattan*, sobreposição de retângulos e SAT (detalhes de funcionamento e implementação desses algoritmos pode ser encontrados no Capítulo 2). O algoritmo *pixel perfect* foi usado como referência para cálculo da precisão na detecção de colisões dos demais algoritmos, mas não faz parte da avaliação.

Os *sprites* foram escolhido com base nas formas geométricas que podem ser utilizadas para a detecção de colisão, sendo eles: Pac-Man[1], Mario[2] e Tyrian[3], conforme apresentado na Figura 11.

Figura 11 – Sprites utilizados

Fonte: Dazz e Petie (2017)

[1] Pac-Man é um jogo da Namco lançado em 1980 ao mercado norte americano(ENTERTAINMENT, 2018), os *sprites* do Pac-Man e do inimigo Blinky foram obtidos através do site *The spriters Resource* que mantém e distribui *sprites* para fins não comerciais. (DAZZ; PETIE, 2017).

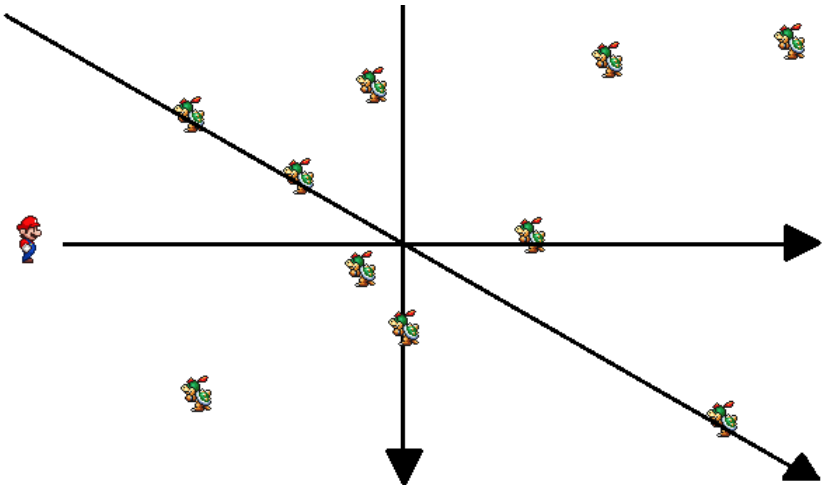
[2] O jogo Super Mario *World* foi lançado em 1990 pela Nintendo, porém o personagem teve sua estreia como *Jump Man* em 1981 no jogo *Donkey Kong* (POT, 2012) os *sprites* do Mario e do inimigo Koopajr foram inspiradas no Super Mario World e também obtidas através do site *The Sprites Resource*(DAZZ; PETIE, 2017).

Os algoritmos sobreposição de retângulos, distância euclidiana, distância de *Manhattan* e SAT são testados com 3 quantidades diferentes de objetos, que aqui são chamados de "inimigos".

Para uma correta análise dos algoritmos algumas regras precisam ser estabelecidas:

- O personagem percorre uma mesma rota, sendo as rotas horizontal, vertical e diagonal ilustradas na Figura 12;
- As rotas vão do início ao fim do cenário;
- Quando o personagem chegar ao fim do cenário, o personagem retorna para a posição inicial;
- A posição dos inimigos é definida aleatóriamente no inicio de cada teste;
- As colisões são testadas apenas em relação ao personagem principal com todos os inimigos, não havendo teste de colisões entre os inimigos;
- Cada teste é executado com as quantidades de inimigos predefinidas nas tabelas 3 e 4;
- Quando uma colisão acontecer, a colisão é contabilizada e o inimigo é marcado para que seja ignorado por novos testes de colisão naquela rodada de teste;

Figura 12 – Rotas que os personagens deverão percorrer nos testes.



Fonte: autor.

[3] O jogo Tyrian foi desenvolvido pela empresa *World Tree Games Productions* em 1995 porém em 1999 foi re-lançado com o nome de Tyrian 2000 (WELLER, 2011), os *sprites* para o personagem principal e o inimigo neste caso são as naves jogáveis USP e Gencore rotacionadas para seguirem o padrão dos demais *sprites*, os *sprites* foram inspirados no Tyrian 2000 e também foram obtidos através do site *The Sprites Resource*(DAZZ; PETIE, 2017).

Tabela 3 – Universo de análise: consumo de recursos e desempenho

Algoritmos	Distância Euclidiana Distância de *Manhattan* Sobreposição de retângulos SAT
Sprites	Mario Pac-Man Tyrian
Rotas	Horizontal Vertical Diagonal
Quantidades de inimigos	1 10 100

A Tabela 3 reúne os algoritmos, os *sprites*, as rotas e as quantidades de inimigos que foram utilizados nos testes, totalizando 108 casos de teste diferentes.

Para cada caso de teste são executados testes com o aplicativo *GameBench* ativo para obter o uso dos recursos do dispositivo e com um tempo total de execução de 15 minutos, devido a uma limitação no aplicativo *GameBench* que requer este período mínimo para obter informações precisas sobre o consumo da bateria do *smartphone*.

Tabela 4 – Universo de análise: comparações estatísticas

Algoritmos	Distância Euclidiana Distância de *Manhattan* Sobreposição de retângulos SAT *Pixel Perfect* (otimizado)
Sprites	Mario Pac-Man Tyrian
Rotas	Horizontal Vertical Diagonal
Quantidades de inimigos	100

Além dos resultados com o *GameBench*, na Tabela 4 os algoritmos são testados com 100 inimigos nas rotas horizontal, vertical e diagonal com cada *sprite* sendo gerado um arquivo de texto contendo o tempo de execução e a quantidade de colisões. São analisadas 42 amostras. É considerado uma amostra a execução do teste com o personagem no inicio até o final da rota, ou seja, são coletados a quantidade de colisões e o tempo necessário para completar a rota de uma extremidade a outra da tela.

O algoritmo *pixel perfect* também é testado com a quantidade de 100 inimigos nas direções horizontal, vertical e diagonal para comparar os resultados com os demais algorit-

mos quanto à quantidade de colisões. Por motivos de desempenho também foi realizada uma alteração no algoritmo do *pixel perfect* que foi a combinação com a sobreposição de retângulos, otimizando assim o tempo necessário para realizar os testes. Com isso a verificação com o *pixel perfect* só ocorre após ser detectada uma colisão por sobreposição de retângulos. Para estes testes foram analisados 5 amostras.

4.3 Medidas de desempenho e consumo

Para medir o desempenho e consumo dos recursos do celular foi utilizado o aplicativo *GameBench*[4]. Este aplicativo é capaz de determinar o desempenho de cada caso de teste através da identificação do uso do CPU, GPU do dispositivo e a taxa de quadros por segundo. O *GameBench* também permite a verificação do uso médio de memória e o consumo de bateria. Adicionalmente para coletar os tempos de execução para completar a rota e para detectar a colisão foi utilizado a função *nanoTime* da classe *System* de *Java*. Para completar a rota foi descartada a primeira execução devido ao tempo necessário ao carregamento das imagens e iniciação do aplicativo.

4.4 Análise estatística

Para verificar se há diferença significativa entre os algoritmos, quanto ao tempo de execução e quantidades de colisões, foi utilizado o *Real Statistics* que é um suplemento para o Excel que contém funções mais avançadas de estatística (ZAIONTZ, 2013). Foram utilizadas três funções deste suplemento: Shapiro-Wilk, para verificar se os resultados tem distribuição normal; ANOVA de um fator para a quantidade de colisões para dados com distribuição normal seguido do teste Tukey para comparar a quantidade de colisões; e Kruskal-Wallis para dados com distribuição não normal seguido do teste Nemenyi para comparar os tempos de execução. Para todas as análises foi considerado o valor α de 0.05, indicando 95% de confiabilidade.

Para estas análises a Tabela 4 apresenta os algoritmos, os *sprites*, as rotas e a quantidade de inimigos avaliadas, com exceção do algoritmo *pixel perfect*.

[4] Mais informações sobre o *GameBench* podem ser obtidas no site www.gamebench.net e no GooglePlay play.google.com/store/apps/details?id=com.gamebench.metricscollector.

5 ANÁLISE DE RESULTADOS

Neste capítulo são apresentados e explicados os resultados obtidos[1]. Os resultados foram divididos em quatro seções: corretude, desempenho, consumo de recursos e resultados estatísticos. Para facilitar a leitura das tabelas presentes foi necessário abreviar o nome dos algoritmos, seguindo esta nomenclatura: distância euclidiana (DE), distância de *Manhattan* (DM), sobreposição de retângulos (SR) e *pixel perfect* (PP).

5.1 Corretude

Os resultados apresentados nesta seção foram obtidos através dos testes realizados conforme as variáveis presentes na Tabela 4. Para validar a corretude dos algoritmos de detecção de colisão é necessário comparar a quantidade de colisões detectadas mantendo o mesmo cenário de teste, ou seja, para esta verificação a posição dos 100 inimigos, gerada de forma aleatória, foi a mesma para os cinco algoritmos. Portanto as colisões detectadas pelo algoritmo *pixel perfect* (PP), são consideradas as colisões reais, os valores entre parênteses representam os falsos positivos que ocorreram nos demais algoritmos. Para estes dados foram considerados cinco execuções nas três rotas para cada um dos três *sprites*. Os resultados desta comparação foram mostrados nas Tabelas 5, 6 e 7.

Na Tabela 5 é possível ver os resultados utilizando os *sprites* do Mario. Tendo como parâmetro os resultados com o algoritmo *Pixel Perfect*, é possível observar que o algoritmo

[1] A implementação dos algoritmos que geraram estes resultados estão disponíveis em https://cc.alegrete.unipampa.edu.br/pampagd/publicacoes/CodigoFonteTCCCristianoRibeiro20181.zip e adicionalmente https://goo.gl/QqvuxM

Tabela 5 – Comparação dos algoritmos quanto a precisão usando *sprites* do Mario

MARIO					
ALGORITMO	**P.P**	D.E	D.M	S.R	SAT
QUANTIDADE DE COLISÕES HORIZONTAL	**22**	22(0)	23(1)	22(0)	22(0)
	13	14(1)	15(2)	13(0)	13(0)
	11	11(0)	12(1)	11(0)	12(1)
	19	20(1)	21(2)	19(0)	19(0)
	17	18(1)	20(3)	17(0)	18(1)
QUANTIDADE DE COLISÕES VERTICAL	**7**	8(1)	8(1)	7(0)	8(1)
	9	13(4)	13(4)	9(0)	10(1)
	2	4(2)	5(3)	2(0)	2(0)
	8	10(2)	10(2)	8(0)	9(1)
	2	4(2)	5(3)	2(0)	2(0)
QUANTIDADE DE COLISÕES DIAGONAL	**22**	22(0)	22(0)	24(2)	22(0)
	21	22(1)	21(0)	25(4)	24(3)
	12	12(0)	12(0)	15(3)	12(0)
	13	13(0)	13(0)	14(1)	13(0)
	25	25(0)	25(0)	27(2)	26(1)
FALSOS POSITIVOS	**(0)**	(15)	(23)	(12)	(9)

Tabela 6 – Comparação dos algoritmos quanto a precisão usando *sprites* do Pac-Man

PAC-MAN					
ALGORITMO	**P.P**	D.E	D.M	S.R	SAT
QUANTIDADE DE COLISÕES HORIZONTAL	**6**	7(1)	7(1)	7(1)	7(1)
	12	13(1)	13(1)	12(0)	12(0)
	9	9(0)	10(1)	9(0)	9(0)
	7	7(0)	10(3)	7(0)	7(0)
	13	15(2)	15(2)	14(1)	15(2)
QUANTIDADE DE COLISÕES VERTICAL	**3**	4(1)	5(2)	3(0)	3(0)
	4	4(0)	4(0)	4(0)	4(0)
	1	1(0)	1(0)	1(0)	1(0)
	6	7(1)	9(3)	6(0)	6(0)
	4	5(1)	5(1)	4(0)	4(0)
QUANTIDADE DE COLISÕES DIAGONAL	**6**	6(0)	6(0)	6(0)	8(2)
	14	14(0)	14(0)	14(0)	14(0)
	3	3(0)	3(0)	3(0)	3(0)
	8	8(0)	8(0)	8(0)	9(1)
	13	13(0)	13(0)	13(0)	14(1)
FALSOS POSITIVOS	**(0)**	(7)	(14)	(2)	(7)

distância euclidiana teve 15 falsos positivos. Já algoritmo distância de *Manhattan* obteve 23 falsos positivos, porém na rota em diagonal obteve a mesma precisão do *Pixel Perfect*. Já o algoritmo de sobreposição de retângulos teve 12 falsos positivos somente na rota em diagonal. O algoritmo SAT teve apenas 9 falsos positivos, sendo a melhor alternativa em termos de precisão para *sprites* similares ao Mario.

Na Tabela 6 são apresentados os resultados utilizando os *sprites* do Pac-Man, que tem a forma geral mais arredondada. O algoritmo distância euclidiana pode ser verificado 7 falsos positivos, porém na rota diagonal teve a mesma precisão do *pixel perfect*. O algoritmo distância de *Manhattan* teve 14 falsos positivos, sendo 8 na rota horizontal devido a forma envolvente (um losango) que possui as extremidades muito distante do *sprite*. O SAT teve 7 falsos positivos, sendo 4 na rota diagonal. A sobreposição de retângulos teve a melhor precisão tendo apenas dois falsos positivos. A utilização do retângulo da imagem se mostrou mais eficiente neste caso.

Na Tabela 7 temos os resultados com os *sprites* do jogo Tyrian. A distância de *Manhattan* apesar de ter um formato mais próximo dos *sprites* teve 19 falsos positivos, sendo 13 na horizontal. O algorimo de sobreposição de retângulos, também não foi satisfatório para este caso, pois foram detectados 12 falsos positivos. A distância euclidiana teve uma precisão mais próxima ao *pixel perfect* com 8 falsos positivos, sendo 5 na rota horizontal e 3 na diagonal. No entando o algoritmo SAT utilizando o polígono convexo presente na Figura 7 obteve apenas 6 falsos positivos.

Com base nos resultados, podemos considerar que quanto à precisão a melhor escolha de algoritmo é o SAT. Embora existam falsos positivos, este algoritmo possui

Tabela 7 – Comparação dos algoritmos quanto a precisão usando *sprites* do Tyrian

TYRIAN					
ALGORITMO	**P.P**	D.E	D.M	S.R	SAT
QUANTIDADE DE COLISÕES HORIZONTAL	**10**	12(2)	14(4)	10(0)	12(2)
	8	8(0)	12(4)	8(0)	9(1)
	17	17(0)	18(1)	17(0)	17(0)
	15	17(2)	17(2)	15(0)	15(0)
	11	12(1)	13(2)	11(0)	11(0)
QUANTIDADE DE COLISÕES VERTICAL	**7**	7(0)	7(0)	7(0)	7
	7	7(0)	8(1)	7(0)	8(1)
	6	6(0)	6(0)	6(0)	6(0)
	10	10(0)	10(0)	10(0)	10(0)
	9	9(0)	10(1)	9(0)	10(1)
QUANTIDADE DE COLISÕES DIAGONAL	**16**	16(0)	17(1)	19(3)	16(0)
	8	9(1)	9(1)	11(3)	8(0)
	15	15(0)	15(0)	16(1)	15(0)
	15	17(2)	17(2)	18(3)	15(0)
	8	8(0)	8(0)	10(2)	9(1)
FALSOS POSITIVOS	**(0)**	(8)	(19)	(12)	(6)

uma boa precisão nas três rotas e possui a vantagem de ter a sua forma que se adapta melhor aos *sprites*.

5.2 Desempenho

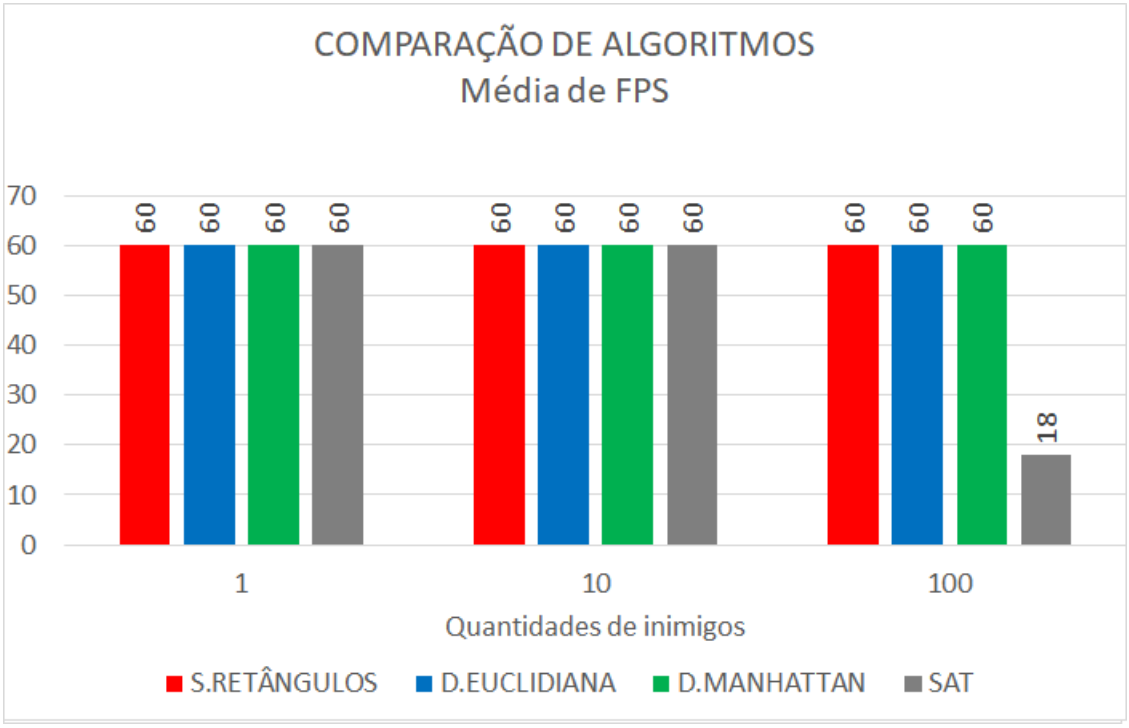
O desempenho foi determinado pela média de quadros por segundos, também conhecido como *frames per second* (FPS). Além disso também foram coletados os tempos para completar a rota e para determinar quando houve colisão.

Quanto à quantidade de quadros por segundo, sabendo que o olho humano é capaz de captar em média 30 quadros por segundo e como pode ser observado na Figura 13, os métodos de sobreposição de retângulos, distância euclidiana e distância de *Manhattan* apresentaram um excelente desempenho mantendo a média de 60 quadros por segundo com as três quantidades de inimigos. O algoritmo SAT teve uma queda no desempenho com 100 inimigos, mantendo uma média de 18 FPS. As médias de FPS foram obtidas através do aplicativo *GameBench*, seguindo os casos de teste presentes na Tabela 3

Para determinar o desempenho de cada algoritmo para detectar a colisão foi coletado o tempo necessário para executar as instruções do algoritmo, sem considerar a quantidade de inimigos, pois o tempo foi coletado dentro do laço de repetição retornando apenas quando há colisão. Eliminando portanto a coleta de tempo necessário para testar com todos os inimigos, assim o gráfico presente na Figura 14 é exclusivamente da detecção de colisão.

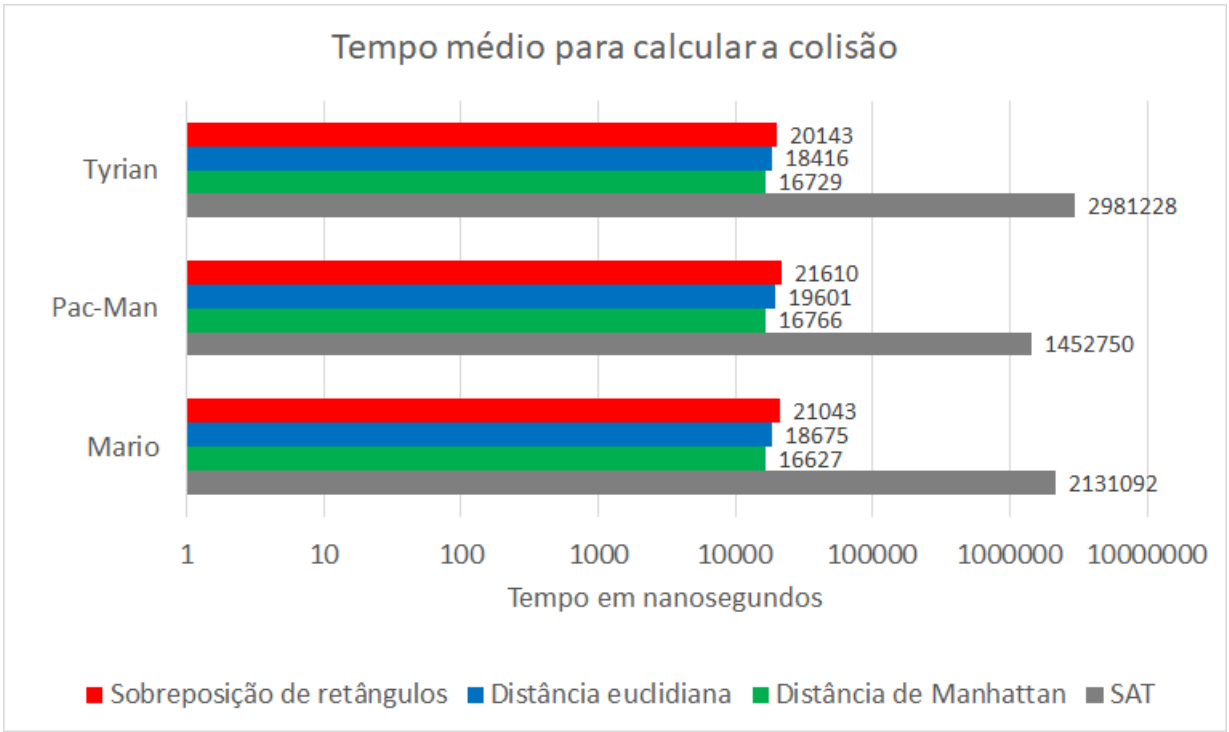
Para obter as médias presentes na Figura 14 foram utilizados os resultados dos testes na diagonal com 100 inimigos que gerou em média 1688 colisões para cada algoritmo

Figura 13 – Média de quadros por segundo (FPS).



Fonte: autor.

Figura 14 – Média de tempo para detectar uma colisão.

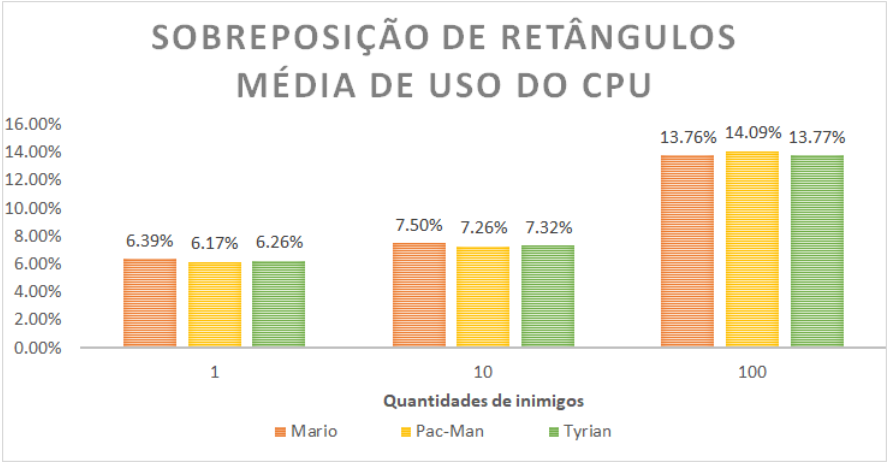


Fonte: autor.

com cada *sprite*.

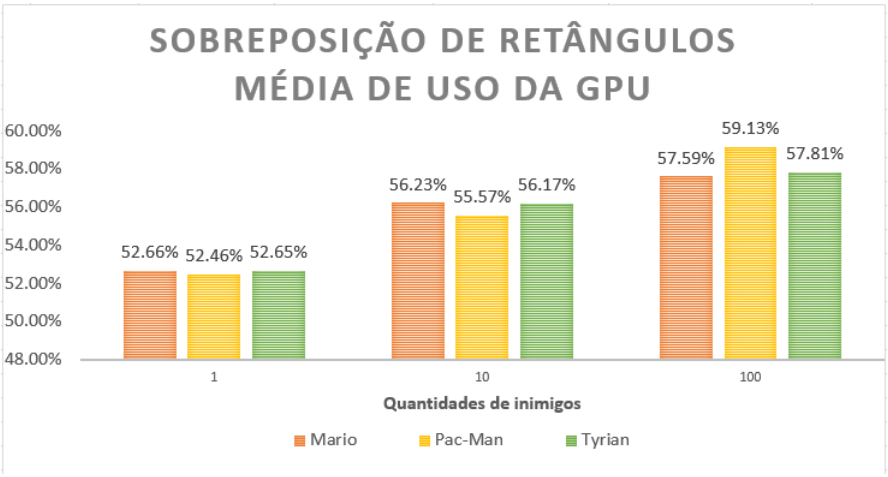
Quanto ao desempenho dos algoritmos analisados o SAT teve o pior desempenho

Figura 15 – Médias de uso de CPU com o algoritmo sobreposição de retângulos.



Fonte: autor.

Figura 16 – Médias de uso de GPU com o algoritmo sobreposição de retângulos.



Fonte: autor.

com 100 inimigos, mas pode ser utilizado, sem afetar muito o desempenho, em jogos com poucos objetos que seja necessário detectar a colisão. O algoritmo que teve o melhor desempenho foi o de sobreposição de retângulos, porém o desempenho dos algoritmos distância euclidiana e distância de *Manhattan* tiveram um desempenho muito próximo.

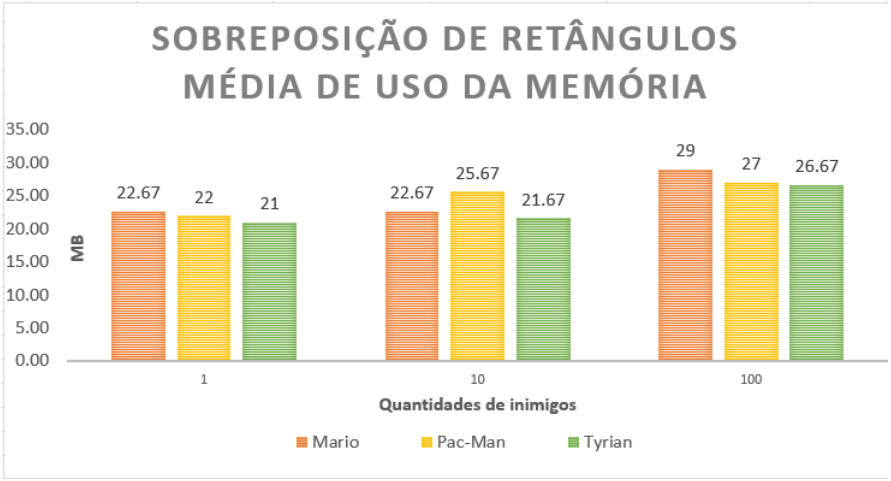
5.3 Consumo de recursos

Os resultados desta seção foram obtidos através do *GameBench* seguindo o ambiente de execução descritos no Capítulo 4. Aqui são apresentados as médias de uso do CPU, GPU, memória e o consumo da bateria do dispositivo. Para facilitar a comparação entre os algoritmos, presentes na Tabela 3, os resultados foram divididos pelos recursos a serem analisados.

Ao comparar o consumo de CPU na Figura 15 e GPU na Figura 16, com o algo-

ritmo de sobreposição de retângulos é possível observar que não há uma variação significativa entre os diferentes *sprites*.

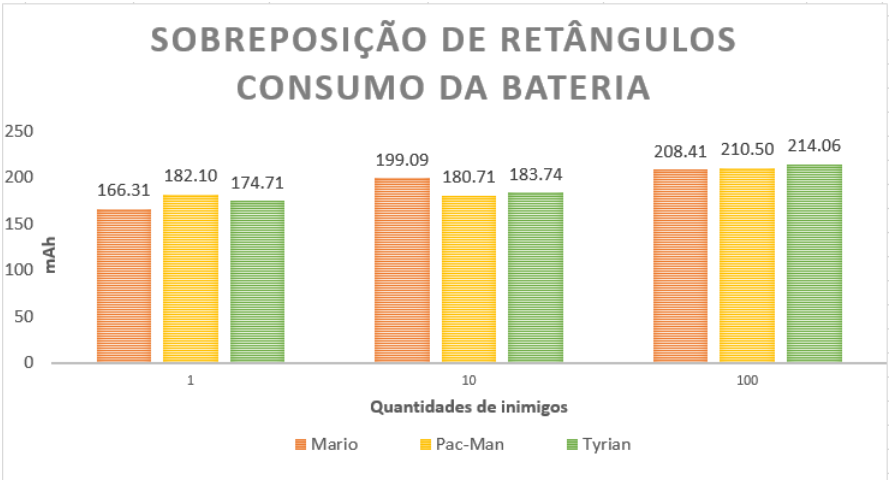
Figura 17 – Médias de uso da memória com o algoritmo sobreposição de retângulos.



Fonte: autor.

Pouca variação entre os diferentes *sprites* também estão presentes no uso médio da memória na Figura 17 e consumo de bateria na Figura 18.Portanto a comparação entre os algoritmos se dará com a média dos resultados dos *sprites*.[2]

Figura 18 – Consumo da bateria com o algoritmo sobreposição de retângulos.

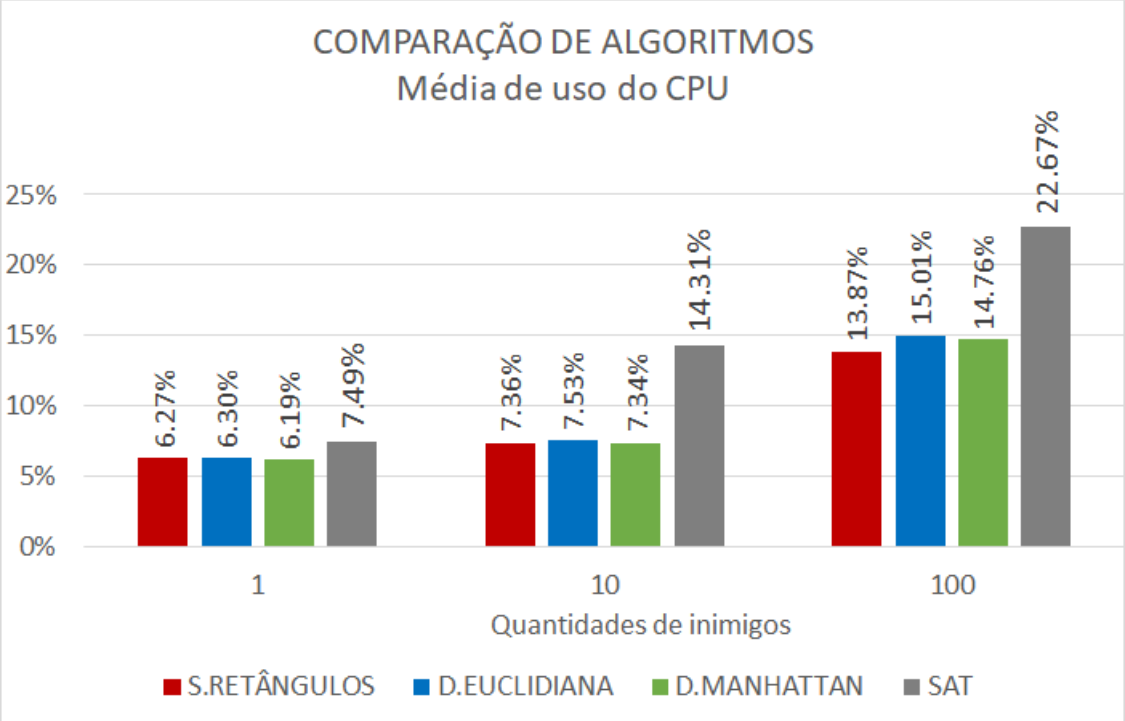


Fonte: autor.

A Figura 19 mostra a média de uso do CPU, como não foi utilizado nenhuma função de paralelismo, os algoritmos utilizaram apenas um dos núcleos disponíveis. Portanto o gráfico que podemos ver na Figura 19 é a média do uso de apenas um dos núcleos.

[2] A comparação entre os diferentes *sprites* com os demais algoritmos pode ser encontrado aqui: https://cc.alegrete.unipampa.edu.br/pampagd/publicacoes/ResultadosTCCCristianiRibeiro20181.xlsx e adicionalmente https://goo.gl/q194m5

Figura 19 – Médias de uso do CPU.



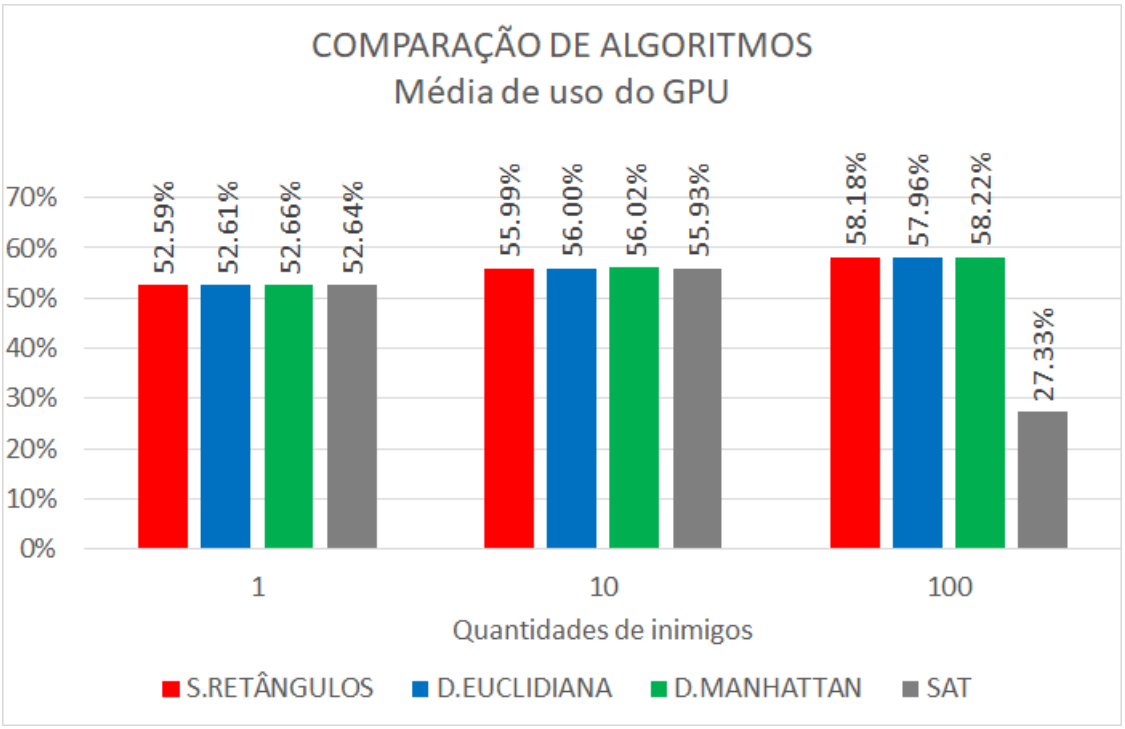
Fonte: autor.

Quanto ao uso do CPU é possível perceber que o método de sobreposição de retângulos é o que menos utiliza o CPU com 100 inimigos e a distância de *Manhattan* teve uma pequena vantagem comparado a distância euclidiana. Já o algoritmo SAT foi o que mais utilizou o CPU, o que era esperado devido aos cálculos que são executados neste algoritmo.

A unidade de processamento gráfico é a responsável por desenhar os elementos gráficos na tela do dispositivo. Como é esperado, jogos utilizam mais da metade do processador gráfico do *smartphone* (Figura 20). Essa utilização está diretamente relacionada com taxa de quadros por segundo em que estes algoritmos são executados, não há diferença significativa entre os algoritmos sobreposição de retângulos, distância euclidiana e distância de *Manhattan*. Contudo, o elevado uso de CPU do algoritmo SAT fez com que tivesse queda de quadros por segundo, diminuindo o uso da GPU.

A média de uso da memória foi relativamente baixa comparada com a memória disponível de 2GB, mas é importante destacar que na implementação em Java Android, a máquina virtual Dalvik (que substitui a JVM) executa uma melhor gerência de memória nas implementações de todos os algoritmos. A baixa utilização da memória também pode estar relacionada a forma como foram implementados os algoritmos, não mantendo objetos referenciados desnecessariamente e fazendo um bom reúso de recursos já instanciados. Entretanto, em um jogo poderá ter uma quantidade maior de elementos o que usaria mais memória, então estas médias de uso da memória podem ser considerados o mínimo

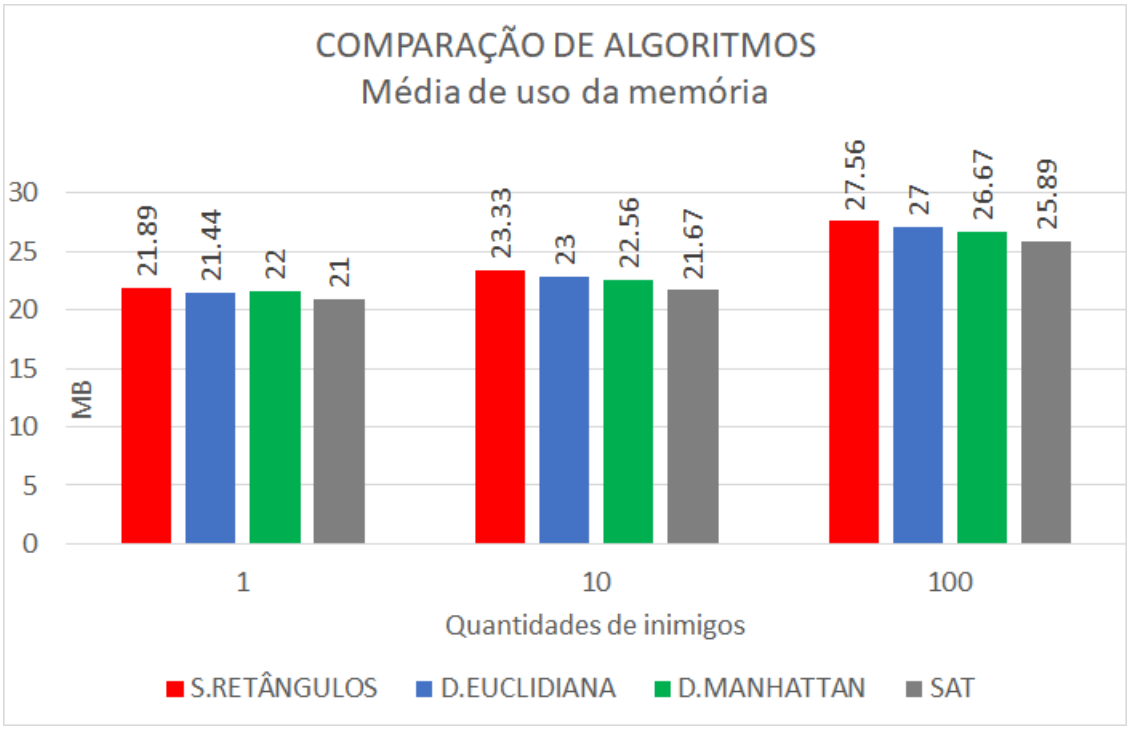
Figura 20 – Médias do uso de GPU.



Fonte: autor.

para um jogo com detecção de colisão.

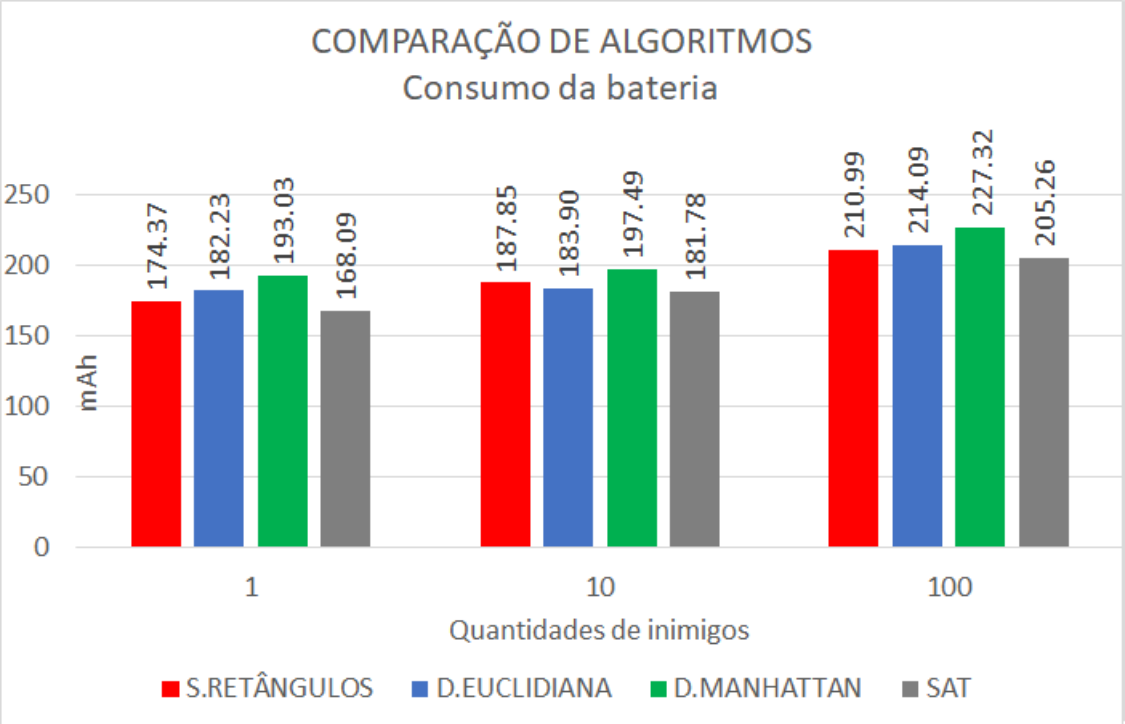
Figura 21 – Médias de uso da memória.



Fonte: autor.

Conforme apresenta a Figura 21 não há uma diferença significativa entre os algoritmos com 1 e 10 inimigos, mas com 100 inimigos a sobreposição de retângulos teve uma utilização um pouco mais elevada do que os demais e o algoritmo SAT teve o menor consumo de memória entre os algoritmos testados, confirmando portanto que a queda de quadros se deu pelo elevado uso do CPU.

Figura 22 – Consumo total da bateria.



Fonte: autor.

Se tratando de dispositivos móveis é importante salientar a limitação da bateria nestes dispositivos. Na Figura 22 temos a média dos três *sprites* do consumo total da bateria em 15 minutos de execução. Observa-se que o consumo da bateria dos algoritmos foram bastante similares. Esse resultado possibilita, de acordo com o *GameBench*, rodar este jogo por aproximadamente 3h sem ser necessário recarregar a bateria. Sendo o algoritmo SAT levemente mais econômico que os demais. Ao compararmos os resultados mostrados nas Figuras 19 e 22 é possível verificar que o consumo da CPU não é proporcional ao consumo da bateria, já o uso da GPU (Figura 20) pode ter contribuido para esta leve vantagem do SAT.

5.4 Resultados estatísticos

Para a análise estatística foram utilizados os resultados na três rotas com 100 inimigos, e 42 amostras para cada algoritmo conforme Tabela 4. Os resultados são separados por orientação das rotas.

Duas grandezas quantitativas são analisadas para os resultados estatísticos: a quantidade de colisões e a média do tempo necessário para completar o percurso, por tanto os resultados são apresentados nestas duas grandezas, separadamente.

5.4.1 Quantidades de colisões

Para ser possível comparar a quantidade de colisões utilizando o teste ANOVA de um fator seguido da aplicação do teste Tukey, os resultados devem ter uma distribuição normal. Portanto, primeiramente, foi feito o teste Shapiro-Wilk. Como as quantidades de colisões testadas tinham uma distribuição normal, pode ser aplicado o teste Tukey, que compara os resultados de dois a dois e o valor-p nos indica qual dos algoritmos é significativamente diferente. Quando o valor-p for menor que ou igual a 0,05 então há diferença significativa entre os algoritmos, já quando o valor-p for maior que 0,05 então não há diferença significativa entre os algoritmos.

Tabela 8 – Médias das quantidades de colisões na rota horizontal

Algoritmo	Mario	Pac-Man	Tyrian
DE	18.0714286	8.634146341	15.4047619047
DM	20.8333333	12.56097561	15.642857142
SR	18.1190476	8.146341463	13.0952380952
SAT	18.0952381	9.219512195	14.2142857142

Comparando as quantidades de colisões com o *sprite* do Mario na rota horizontal (Tabela 8), o teste ANOVA teve valor-p de 0,001666, indicando que há diferença significativa entre dois ou mais algoritmos.

Tabela 9 – Valores-p do teste ANOVA seguido do teste Tukey para cada par de algoritmos na rota horizontal

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	**0.006895399**	$\mathbf{2.35 \times 10^{-07}}$	0.990144
DE	SR	0.999934852	0.890181	**0.018591**
DE	SAT	0.999991849	0.825125	0.42572
DM	SR	**0.008255799**	$\mathbf{6.15 \times 10^{-09}}$	**0.007325**
DM	SAT	**0.007547649**	$\mathbf{1.31 \times 10^{-05}}$	0.263764
SR	SAT	0.999991849	0.394295	0.48113

Conforme pode ser observado na Tabela 9, o algoritmo distância de *Manhattan* é pior que os outros três algoritmos, pois possui mais falsos positivos, reforçando o que verificamos na seção 5.1. Entre os três melhores, não há diferença significativa, ou seja, os algoritmos distância euclidiana, sobreposição de retângulos e SAT tiveram precisão similar.

Comparando as quantidades de colisões com o *sprite* do Pac-Man na rota horizontal, o teste ANOVA teve valor-p de 9.7×10^{-10}, indicando que há diferença significativa entre dois ou mais algoritmos.

Quanto a quantidade de colisões, com os *sprites* do Pac-Man. De acordo com a Tabela 8 o algoritmo distância de *Manhattan* teve uma quantidade maior de falsos positivos, assim como com os *sprites* do Mario. Entre os outros três algoritmos não há diferença significativa, como podemos verificar na Tabela 9.

Comparando as quantidades de colisões com o *sprite* do Tyrian na rota horizontal, o teste ANOVA teve valor-p de 0.004436, indicando que há diferença significativa entre dois ou mais algoritmos.

Com os *sprites* do Tyrian, os resultados presentes na Tabela 9 permitem afirmar que os algoritmos distância de *Manhattan* e distância euclidiana são piores que sobreposição de retângulos, por possuirem mais falsos positivos, conforme podemos verificar na Tabela 8. Há uma tendência de sobreposição de retângulos ser tão preciso quanto SAT, mesmo que SAT não tenha apresentado diferenças significativas entre os algoritmos distância euclidina e distância de *Manhattan*.

Em todos os formatos de *sprites* testados, na rota horizontal a distância de *Manhattan* teve mais falsas colisões (falsos positivos). A distância euclidiana, sobreposição de retângulos e SAT tiveram aproximadamente a mesma quantidade de falsos positivos, exceto no *sprite* mais triangular (Tyrian) no qual sobreposição de retângulos e SAT tiveram ainda menos falsos positivos.

Comparando as quantidades de colisões com o *sprite* do Mario na rota vertical, o teste ANOVA teve valor-p de $2,05^{\times}10-14$, indicando que há diferença significativa entre dois ou mais algoritmos.

Na Tabela 11 é possível verificar que com os *sprites* do Mario na rota vertical, apenas entre os algoritmos de sobreposição de retângulos e SAT não há diferença significativa, ou seja, a quantidade de colisões detectadas por estes algoritmos foi muito similar, conforme foi verificado pelas médias presentes na Tabela 10. Os algoritmos distância euclidiana e distância de *Manhattan* tiveram mais falsos positivos, ficando evidente nesta comparação.

Comparando as quantidades de colisões com o *sprite* do Pac-Man na rota vertical, o teste ANOVA teve valor-p de $2,03 \times 10^{-5}$, indicando que há diferença significativa entre

Tabela 10 – Médias das quantidades de colisões na rota vertical

Algoritmo	Mario	Pac-Man	Tyrian
DE	9.5238095	4.595238095	7.73809523
DM	12.0714285	6.595238095	10.9047619
SR	7.1666666	6.88095238	7.30952380
SAT	7.6666666	6.30952380	7.85714285

Tabela 11 – Valores-p do teste ANOVA seguido do teste Tukey para cada par de algoritmos na rota vertical

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	**0.000212225**	**0.000413**	$\mathbf{3.78 \times 10^{-6}}$
DE	SR	**0.0007225**	$\mathbf{3.88 \times 10^{-5}}$	0.896408
DE	SAT	**0.01229862**	**0.00341**	0.9973699
DM	SR	$\mathbf{4.733 \times 10^{-13}}$	0.93734	$\mathbf{1.31 \times 10^{-7}}$
DM	SAT	$\mathbf{5.661 \times 10^{-11}}$	0.93734	$\mathbf{9.16 \times 10^{-6}}$
SR	SAT	0.8386397	0.65019	0.806815

Tabela 12 – Médias das quantidades de colisões na rota diagonal

Algoritmo	Mario	Pac-Man	Tyrian
DE	20.85714	10.5238095	15.7380952
DM	21.57142	11.7142857	19.2857142
SR	22.40476	11.4285714	18.5238095
SAT	18.738095	11.3095238	16.2619047

dois ou mais algoritmos.

Os resultados presentes na Tabela 11 os algoritmos de sobreposição de retângulos e SAT não possuem diferença significativa, o que é justificável pelo formato do polígono convexo escolhido para os *sprites* do Pac-Man (Figura 7), o algoritmo distância de *Manhattan* foi tão bom quanto estes algoritmos. Já o algoritmo distância euclidiana foi o que teve mais falsos positivos, contrariando os resultados presentes na Tabela 6, porém se faz necessário frisar que para os resultados estatísticos os inimigos foram posicionados aleatóriamente, portanto anomalias como a presente nesta comparação eram esperados.

Comparando as quantidades de colisões com o *sprite* do Tyrian na rota vertical, o teste ANOVA teve valor-p de $1,76 \times 10^{-8}$, indicando que há diferença significativa entre dois ou mais algoritmos.

Com os *sprites* do Tyrian os resultados presentes na Tabela 11 é possível verificar que o algoritmo distância de *Manhattan* teve mais falsos positivos, conforme pode ser verificado na Tabela 10. Os demais algoritmos tiveram resultados similares, não havendo diferença significativas entre eles.

Com estes resultados fica evidente que a trajetória que o personagem faz influencia na detecção de colisão. Os algoritmos de sobreposição de retângulos e SAT foram os melhores nesta trajetória para todos os diferentes *sprites*. Os algoritmos distância euclidiana e distância de *Manhattan* não são indicados para *sprites* retângulares (Mario), mas podem ser utilizados em *sprites* circulares (Pac-Man) ou triângulares (Tyrian) em um jogo com movimento vertical.

Comparando as quantidades de colisões com o *sprite* do Mario na rota diagonal (Tabela 12), o teste ANOVA teve valor-p de 0,001575, indicando que há diferença significativa entre dois ou mais algoritmos.

Tabela 13 – Valores-p do teste ANOVA seguido do teste Tukey para cada par de algoritmos na rota diagonal

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	0.8795592	0.29273703	0.99014394
DE	SR	0.3760365	0.53677663	**0.0185910**
DE	SAT	0.1264401	0.64875621	0.42572396
DM	SR	0.82199547	0.97427149	**0.00732495**
DM	SAT	**0.0190854**	0.93162825	0.26376389
SR	SAT	**0.00110122**	0.998031114	0.48113025

Na Tabela 13 não há diferença significativa entre os algoritmos distância euclidiana e distância de *Manhattan*, ou seja, a quantidade de colisões detectadas com estes algoritmos para esta rota foram muito boas confirmando os resultados presentes na Tabela 5. O algoritmo SAT foi tão bom quanto o algoritmo distância euclidiana, já o algoritmo sobreposição de retângulos foi tão bom quanto o algoritmo distância de *Manhattan*. No entanto ao verificar a Tabela 5 a distância de *Manhattan* teve a mesma precisão que o *pixel perfect* nesta rota, já o algoritmo sobreposição de retângulos teve mais falsos positivos, portanto, é possível afirmar que apesar de haver diferença significativa entre os algoritmos distância de *Manhattan* e o SAT o algoritmo SAT é melhor que sobreposição de retângulos.

Comparando as quantidades de colisões com o *sprite* do Pac-Man na rota diagonal, o teste ANOVA teve valor-p de 0,333174, indicando que não há diferença significativa entre dois ou mais algoritmos. Como podemos ver na Tabela 13. Porém podemos destacar que o algoritmo SAT obteve um número muito próximo de quantidades de colisões que o algoritmo de sobreposição de retângulos.

Comparando as quantidades de colisões com o *sprite* do Tyrian na rota diagonal, o teste ANOVA teve valor-p de 0,004436, indicando que há diferença significativa entre dois ou mais algoritmos.

A Tabela 13 indica que há diferença significativa entre os algoritmos distância euclidiana e sobreposição de retângulos, pois o segundo possui mais falsos positivo conforme os resultados obtidos na Tabela 7. Apesar dos resultados indicarem que o algoritmo SAT ser tão bom quanto os outros três algoritmos, inclusive o sobreposição de retângulos, na Tabela 7 é possível verificar que o SAT possui a mesma precisão que o *pixel perfect* em 4 dos 5 resultados observados.

Para jogos com trajetória em diagonal são indicados os algoritmos distância de *Manhattan* e SAT, para os diferentes *sprites* testados.

Após avaliar a precisão dos algoritmos nas diferentes rotas e *sprites*, o algoritmo SAT se destaca como o melhor, mesmo tendo alguns falsos positivos, pois foi o algoritmo que melhor se adaptou as mudanças de rotas. Em seguida podemos destacar o algoritmo de sobreposição de retângulos, que teve uma precisão pior apenas na rota diagonal, mas

que pode ser uma boa alternativa para jogos com *sprites* mais retângulares, similares ao Mario.

5.4.2 Média de tempo para completar o percurso

Para comparar os resultados dos tempos para completar o percurso, foi utilizado os testes Kruskal-Wallis seguido da aplicação do teste Nemenyi, o qual permite selecionar o par de grupos que se deseja verificar. A análise do valor-p é a mesma do teste Tukey.

Tabela 14 – Médias dos tempos necessário para completar a rota horizontal

Algoritmo	Mario	Pac-Man	Tyrian
DE	20757321829	20949713114	20520301155
DM	20761147641	20951006059	20523590446
SR	20748288056	20956470107	20517239202
SAT	66862936269	62872591479	82159364480

Comparando os tempos para completar o percurso horizontal com o *sprite* do Mario (Tabela 14), o teste Kruskal-Wallis teve valor-p de $2,04 \times 10^{-24}$, indicando que há diferença significativa entre dois ou mais algoritmos.

Tabela 15 – valores-p do teste Kruskal-Wallis seguido do teste Nemenyi para cada par de algoritmos na rota horizontal

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	0.666632	0.979037548	**0.038224**
DE	SR	**0.000144**	**0.000611327**	0.817908
DE	SAT	$\mathbf{1.20 \times 10^{-08}}$	$\mathbf{-3.8858 \times 10^{-14}}$	$\mathbf{-3.7 \times 10^{-14}}$
DM	SR	**0.010235**	**0.002714649**	**0.002357**
DM	SAT	$\mathbf{9.90 \times 10^{-12}}$	$\mathbf{-3.8858 \times 10^{-14}}$	$\mathbf{5.88 \times 10^{-08}}$
SR	SAT	$\mathbf{-3.89 \times 10^{-14}}$	$\mathbf{8.40837 \times 10^{-07}}$	$\mathbf{-3.9 \times 10^{-14}}$

A Tabela 15 possibilita a afirmação que o algoritmo SAT é muito mais lento que os outros algoritmos. Os algoritmos distância euclidiana e distância de *Manhattan* são mais lentos que sobreposição de retângulos.

Comparando os tempos para completar o percurso com o *sprite* do Pac-Man, o teste Kruskal-Wallis teve valor-p de 9.87×10^{-24}, indicando que há diferença significativa entre dois ou mais algoritmos.

Os tempos com os *sprites* do Pac-Man tiveram resultados similares com os resultados observados com os do Mario, como pode ser verificado na Tabela 15.

Comparando os tempos para completar o percurso com o *sprite* do Tyrian, o teste Kruskal-Wallis teve valor-p de 3.5×10^{-23}, indicando que há diferença significativa entre dois ou mais algoritmos.

Com os *sprites* do Tyrian também é observado que o algoritmo SAT é muito mais lento que os outros algoritmos, com uma média de tempo quatro vezes maior do que

Tabela 16 – Médias dos tempos necessário para completar a rota vertical

Algoritmo	Mario	Pac-Man	Tyrian
DE	10885007875	11585237971	11414124359
DM	10881516180	11583466228	11414898734
SR	10881325304	11581149949	11415058847
SAT	36919903035	34777620418	45930634167

os outros algoritmos, como podemos observar na Tabela 14, portanto é o algoritmo mais lento. O algoritmo sobreposição de retângulos é mais rápido que a distância de *Manhattan*, mas não necessariamente mais rápido que a distância euclidiana.

Podemos concluir que o algoritmo SAT é muito mais lento que os outros algoritmos, e sobreposição de retângulos é mais rápido que distância euclidiana e distância de *Manhattan*, exceto no *sprite* mais triangular no qual a distância euclidiana foi tão rápido quanto.

Comparando os tempos para completar o percurso vertical com o *sprite* do Mario (Tabela 16), o teste Kruskal-Wallis teve valor-p de 1.49994×10^{-20}, indicando que há diferença significativa entre dois ou mais algoritmos. Como é possível observar na Tabela 17, o algoritmo SAT é o mais lento, os algoritmos distância de *Manhattan* e sobreposição de retângulos tiveram um tempo muito próximo, como indica o valor-p, nesta comparação. O algoritmo distância euclidiana foi tão rápido quanto.

Comparando os tempos para completar o percurso vertical com o *sprite* do Pac-Man, o teste Kruskal-Wallis teve valor-p de 1.4639×10^{-20}, indicando que há diferença significativa entre dois ou mais algoritmos. O algoritmo SAT é o mais lento, os resultados são semelhantes com os observados com o *sprite* do Mario.

Comparando os tempos para completar o percurso vertical com o *sprite* do Pac-Man, o teste Kruskal-Wallis teve valor-p de 3.03162×10^{-20}, indicando que há diferença significativa entre dois ou mais algoritmos. Novamente o algoritmo SAT é o mais lento, não havendo diferença significativa entre os demais algoritmos. Porém a Tabela 16 indica que o algoritmo distância euclidiana foi em média o mais rápido.

Através destes resultados é possível observar que o algoritmo SAT é muito mais

Tabela 17 – valores-p do teste Kruskal-Wallis seguido do teste Nemenyi para cada par de algoritmos na rota vertical

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	0.71478443	0.920756288	0.996247534
DE	SR	0.721506215	0.604255224	0.997540482
DE	SAT	$\mathbf{1.29551 \times 10^{-11}}$	$\mathbf{7.66764 \times 10^{-12}}$	$\mathbf{5.55112 \times 10^{-16}}$
DM	SR	0.999999493	0.931670346	0.999991083
DM	SAT	$\mathbf{-3.08642 \times 10^{-14}}$	$\mathbf{5.76206 \times 10^{-14}}$	$\mathbf{1.49436 \times 10^{-13}}$
SR	SAT	$\mathbf{-3.01981 \times 10^{-14}}$	$\mathbf{-3.75255 \times 10^{-14}}$	$\mathbf{1.14242 \times 10^{-13}}$

Tabela 18 – Médias dos tempos necessário para completar a rota diagonal

Algoritmo	Mario	Pac-Man	Tyrian
DE	10381734140	10482619477	10264615601
DM	10382014007	10482291128	10265810957
SR	10381289141	10481641640	10265741733
SAT	35002987590	31462739136	41249446322

lento que os demais. O algoritmo de sobreposição de retângulos foi o mais rápido, exceto com os *sprites* do Tyrian no qual, assim como na rota horizontal, o algoritmo distância euclidiana foi o mais rápido.

O teste Kruskal-Wallis retornou valores-p muito semelhantes entre os 3 *sprites* analisados para a rota diagonal (Tabela 18, com o *sprite* do Mario tivemos o valor-p de 2.57966×10^{-20}, com o *sprite* do Pac-Man o valor-p obtido foi de 2.217×10^{-20} e com o *sprite* do Tyrian valor-p foi igual a 1.8808×10^{-20}. Com os 3 *sprites* houve diferença significativa entre dois ou mais algoritmos, no entanto a Tabela 19 mostra que a diferença significativa existe apenas nas comparações com o algoritmo SAT que nesta rota também foi o mais lento.

Tabela 19 – valores-p do teste Kruskal-Wallis seguido do teste Nemenyi para cada par de algoritmos na rota diagonal

grupo 1	grupo 2	valor-p (Mario)	valor-p (Pac-Man)	valor-p (Tyrian)
DE	DM	0.989707086	0.954724052	0.933057146
DE	SR	0.989707086	0.844542812	0.746610165
DE	SAT	$\mathbf{6.57252 \times 10^{-14}}$	$\mathbf{2.29372 \times 10^{-12}}$	$\mathbf{-3.66374 \times 10^{-14}}$
DM	SR	0.925962076	0.990348997	0.977005093
DM	SAT	$\mathbf{8.74412 \times 10^{-13}}$	$\mathbf{2.38698 \times 10^{-14}}$	$\mathbf{1.24345 \times 10^{-13}}$
SR	SAT	$\mathbf{-2.75335 \times 10^{-14}}$	$\mathbf{-3.17524 \times 10^{-14}}$	$\mathbf{2.71949 \times 10^{-12}}$

Como fica claro na Tabela 18 o algoritmo de sobreposição de retângulos é o mais rápido, assim como nas outras rotas, exceto com os *sprites* do Tyrian no qual o algoritmo distância euclidiana é o mais rápido.

Portanto quanto aos tempo necessário para completar a rota, o algoritmo SAT confirmou o que foi observado quanto ao desempenho, sendo cerca de três vezes mais lento que os demais algoritmos em todas as rotas testadas. O algoritmo de sobreposição de retângulos é o mais rápido nas diferentes rotas testadas, exceto com os *sprites* do Tyrian, que assim como nas colisões, o algoritmo distância euclidiana foi o mais rápido.

Para obter os resultados presentes neste trabalho foram necessários aproximadamente 100 horas de testes, sendo 55 horas no GameBench para verificar o uso dos recursos e 45 horas para as quantidades de colisões, tempos para completar o percurso e precisão dos algoritmos.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foram avaliados os algoritmos de detecção de colisão sobreposição de retângulos, distância euclidiana, distância de *Manhattan*, SAT e sobreposição de retângulos com *pixel perfect*. Para avaliar foram escolhidos três conjuntos de *sprites* diferentes para verificar como diferentes formatos de objetos poderiam influenciar na escolha dos algoritmos. Os conjuntos de *sprites* escolhidos foram os do jogo Mario, Pac-Man e Tyrian.

Através dos resultados foi observado que quanto ao uso dos recursos (médias de uso de CPU, GPU, memória e consumo da bateria) os diferentes *sprites* não tiveram uma variação significativa. Já quanto à precisão tiveram algumas variações, ou seja, a importância na escolha do algoritmo de detecção de colisão que será utilizado em jogo deve levar em conta o *sprite* para se ter uma melhor precisão. Os testes quanto ao uso dos recursos foram com diferentes quantidades de objetos. Desta forma foi possível verificar o impacto da quantidade de objetos que necessitavam de detecção de colisão nos recursos disponíveis no dispositivo. Os algoritmos foram avaliados conforme o uso dos recursos do dispositivo móvel, com auxilio do aplicativo *Gamebench*. Dos algoritmos avaliados o que demonstrou melhor desempenho foi a sobreposição de retângulos. Já o SAT teve uma melhor precisão porém com queda de desempenho com 100 inimigos. Quanto a utilização de recursos a sobreposição de retângulos detecta colisões utilizando menos recursos de processamento.

Como limitações desse trabalho, destacamos as limitações de tempo devido a uma licença temporária para uso do aplicativo do *GameBench*. Após contato com o CEO da empresa, ele nos forneceu gratuitamente essa licença para uso nesse trabalho, mas durante algum tempo (em revalidações da licença) o acesso ao aplicativo ficou indisponível. A quantidade de tempo necessário para os testes também foi bastante grande, limitando as combinações nos testes realizados. Isso foi especialmente relevante nos testes que precisavam ser monitorados pelo autor.

Devido a atualizações do aplicativo *GameBench* e do próprio desgaste natural da bateria entre os testes realizados no TCC1 e TCC2, foi necessário refazer alguns testes. Finalmente, não aplicar otimizações nos algoritmos, como a divisão de espaço em grade (como sugerido por RABIN(2010) ou o uso de paralelismo), abre espaço para investigações futuras nas quais alguns dos algoritmos se beneficiem e se tornem mais viáveis.

Como trabalhos futuros dessa pesquisa, pretende-se realizar os mesmos testes em diferentes dispositivos. Isso inclui diferentes dispositivos com sistema operacional *Android* e sistema iOS. Adicionalmente, podem ser incluída avaliação de algoritmos de detecção de colisão não implementados nesse trabalho como detecção hierárquica (em suas diferentes formas - variação de granularidade com o mesmo algoritmo, ou variação de algoritmo). Salienta-se que esses testes podem ser também aplicados para jogos tridimensionais, alterando os volumes envolventes 2D para seu equivalente 3D.

REFERÊNCIAS

AMATO, J. Collision Detection. 1999. Disponível em: <http://www.gamedev.net/page/resources/_/technical/game-programming/collision-detection-r735>. Acesso em: 13 jun. 2017. Citado na página 27.

ANDERSON, E. F. et al. Choosing the infrastructure for entertainment and serious computer games - a whiteroom benchmark for game engine selection. p. 1–8, Sept 2013. Citado na página 32.

ASSOCIATION, E. S. **Two Thirds American Households Regularly Play Video Games**. 2017. Disponível em: <http://www.theesa.com/article/two-thirds-american-households-regularly-play-video-games>. Acesso em: 20 april. 2017. Citado na página 21.

BITTLE, W. **SAT (Separating Axis Theorem)**. 2010. Disponível em: <http://www.dyn4j.org/2010/01/sat/>. Acesso em: 17 feb 2018. Citado na página 28.

BURNS, R.; SHEPPARD, M. **N Tutorial A - Collision Detection and Response**. 2011. Disponível em: <http://www.metanetsoftware.com/technique/tutorialA.html>. Acesso em: 19 feb 2018. Citado na página 30.

CHANG, C.-T.; GORISSEN, B.; MELCHIOR, S. Fast oriented bounding box optimization on the rotation group so(3,ℝ). **ACM Trans. Graph.**, ACM, v. 30, n. 5, p. 122:1–122:16, out. 2011. Disponível em: <http://doi.acm.org/10.1145/2019627.2019641>. Citado na página 42.

CHENG, Z. et al. Behavior-aware integrated cpu-gpu power management for mobile games. p. 439–444, 2016. Disponível em: <http://ieeexplore.ieee.org/document/7774618/>. Citado na página 38.

CHONG, K. S. **Collision Detection Using the Separating Axis Theorem**. 2012. Disponível em: <https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169>. Acesso em: 20 feb 2018. Citado na página 28.

CHUANG, P. K.; CHEN, Y. S.; HUANG, P. H. An adaptive on-line cpu-gpu governor for games on mobile devices. p. 653–658, 2017. Disponível em: <http://ieeexplore.ieee.org/document/7858398/>. Citado na página 38.

CORRAL, L. et al. Preserving energy resources using an android kernel extension: A case study. ACM, p. 23–24, 2016. Disponível em: <http://doi.acm.org/10.1145/2897073.2897124>. Citado na página 39.

CORRAL, L. et al. Energy-aware performance evaluation of android custom kernels. IEEE Press, p. 1–7, 2015. Disponível em: <http://dl.acm.org/citation.cfm?id=2820158.2820160>. Citado na página 37.

CURTIS, S.; TAMSTORF, R.; MANOCHA, D. Fast collision detection for deformable models using representative-triangles. ACM, p. 61–69, 2008. Disponível em: <http://doi.acm.org/10.1145/1342250.1342260>. Citado na página 40.

DAVEY, R. How to Learn the Phaser HTML5 Game Engine. 2013. Disponível em: <https://gamedevelopment.tutsplus.com/articles/how-to-learn-the-phaser-html5-game-engine--gamedev-13643>. Acesso em: 25 May 2017. Citado na página 33.

DAZZ; PETIE. The Spriters Resource. 2017. Disponível em: <https://www.spriters-resource.com>. Acesso em: 20 May 2017. Citado 2 vezes nas páginas 45 e 46.

EBERLY, D.; SHOEMAKE, K. **Game Physics**. Taylor & Francis, 2004. (Interactive 3D technology series). Disponível em: <https://books.google.com.br/books?id=a9SzFHPJ0mwC>. Citado na página 23.

ENTERTAINMENT, B. N. History of Pac-Man. 2018. Disponível em: <http://pacman.com/en/pac-man-history>. Acesso em: 4 May 2018. Citado na página 45.

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 na página 24.

FERNANDES, M. **Sprite - Game Conceito**. 2009. Disponível em: <https://doctorzeroth.wordpress.com/2009/12/25/sprite-game-conceito/>. Acesso em: 2 may 2018. Citado na página 23.

GARCIA-ALONSO, A.; SERRANO, N.; FLAQUER, J. Solving the collision detection problem. **IEEE Comput. Graph. Appl.**, IEEE Computer Society Press, v. 14, n. 3, p. 36–43, maio 1994. Disponível em: <http://dx.doi.org/10.1109/38.279041>. Citado na página 42.

HOSSEINI, M.; PETERS, J.; SHIRMOHAMMADI, S. Energy-efficient 3d texture streaming for mobile games. ACM, New York, NY, USA, p. 5:1–5:6, 2013. Disponível em: <http://doi.acm.org/10.1145/2579465.2579471>. Citado na página 21.

HUANG, C.-Y. et al. To cloud or not to cloud: Measuring the performance of mobile gaming. ACM, p. 19–24, 2015. Disponível em: <http://doi.acm.org/10.1145/2751496.2751497>. Citado na página 37.

KLEINA, N. **O Que é Engine ou Motor Gráfico**. 2011. Disponível em: <https://www.tecmundo.com.br/video-game-e-jogos/9263-o-que-e-engine-ou-motor-grafico-.htm>. Acesso em: 25 apri. 2017. Citado na página 21.

KUGLER, M.; JÚNIOR, J. T.; LOPES, H. S. Desenvolvimento de uma rede neural lvq em linguagem vhdl para aplicações em tempo-real. 2003. Citado na página 26.

LENCEVICIUS, R.; METZ, E. Performance assertions for mobile devices. ACM, New York, NY, USA, p. 225–232, 2006. Disponível em: <http://doi.acm.org/10.1145/1146238.1146264>. Citado na página 37.

LI, D.; HALFOND, W. G. J. An investigation into energy-saving programming practices for android smartphone app development. ACM, p. 46–53, 2014. Disponível em: <http://doi.acm.org/10.1145/2593743.2593750>. Citado na página 37.

MIRTICH, B. V-clip: Fast and robust polyhedral collision detection. **ACM Trans. Graph.**, ACM, v. 17, n. 3, p. 177–208, jul. 1998. Disponível em: <http://doi.acm.org/10.1145/285857.285860>. Citado na página 40.

NETO, D. A. Intelligent 2D Collision and Pixel Perfect Precision. 2013. Disponível em: <https://www.gamedev.net/resources/_/technical/game-programming/intelligent-2d-collision-and-pixel-perfect-precision-r3311>. Acesso em: 01 Jun 2017. Citado na página 31.

NETO, J. M.; MOITA, G. C. Uma introdução à análise exploratória de dados multivariados. **Química nova**, SciELO Brasil, v. 21, n. 4, p. 467–469, 1998. Citado na página 24.

NIELSEN, S. **When Bullets Move Too Fast...** 2007. Disponível em: <https://www.aorensoftware.com/blog/2011/06/01/when-bullets-move-too-fast/>. Acesso em: 31 may. 2017. Citado na página 23.

PATHANIA, A. et al. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. ACM, p. 201:1–201:6, 2015. Disponível em: <http://doi.acm.org/10.1145/2744769.2744894>. Citado na página 36.

PAWASKAR, C. **Continuous collision detection for translating ellipsoid. [S.l.]**. 2007. Disponível em: <https://www.gamedev.net/resources/_/technical/game-programming/continuous-collision-detection-for-translating-ellipsoid-r2426>. Acesso em: 15 may. 2017. Citado na página 24.

PETRIDIS, P. et al. An engine selection methodology for high fidelity serious games. p. 27–34, March 2010. Citado na página 33.

POT, J. The Origins and History of Mario [Geek History Lessons]. 2012. Disponível em: <https://www.makeuseof.com/tag/origins-history-mario-geek-history-lessons/>. Acesso em: 4 May 2018. Citado na página 45.

PRODANOV, C.; FREITAS, E. C. de. **metodologia do trabalho científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico**. [S.l.]: Universidade Feevale, 2013. Citado na página 45.

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 na página 65.

SATHE, R.; SHARLET, D. **Fast Rigid-Body Collision Detection Using Farthest Feature Maps**. [S.l.]: Charles River Media, 2008. 143–151 p. Citado na página 24.

UNITY. Unity Manual. 2017. Disponível em: <https://docs.unity3d.com/Manual/index.html>. Acesso em: 01 Jun 2017. Citado na página 32.

UNREAL. Unreal Documentation. 2017. Disponível em: <https://docs.unrealengine.com/latest/INT/>. Acesso em: 01 Jun 2017. Citado na página 33.

VOGT, W. **Dictionary of Statistics & Methodology: A Nontechnical Guide for the Social Sciences**. SAGE Publications, 1999. ISBN 9780761912743. Disponível em: <https://books.google.com.br/books?id=_Zt6ONmRkocC>. Citado na página 36.

WELLER, T. Forgotten Old Games - Tyrian. 2011. Disponível em: <https://gamecomments.wordpress.com/2011/06/22/forgotten-old-games-tyrian/>. Acesso em: 4 May 2018. Citado na página 46.

WOULFE, M.; MANZKE, M. A framework for benchmarking interactive collision detection. ACM, p. 205–212, 2009. Disponível em: <http://doi.acm.org/10.1145/1980462.1980501>. Citado na página 41.

YANG, B.; CHENG, X.; PAN, Z. A real-time collision detection algorithm for mobile billiards game. ACM, p. 294–297, 2005. Disponível em: <http://doi.acm.org/10.1145/1178477.1178530>. Citado na página 41.

ZAIONTZ, C. **Real Statistics Using Excel**. 2013. Disponível em: <http://www.real-statistics.com/>. Acesso em: 07 Mar 2018. Citado na página 48.