UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CORNÉLIO PROCÓPIO
DIRETORIA DE GRADUAÇÃO E EDUCAÇÃO PROFISSIONAL
DEPARTAMENTO DE COMPUTAÇÃO
ENGENHARIA DE SOFTWARE

RODRIGO MORETTO

ANÁLISE COMPARATIVA ENTRE DIFERENTES MÉTODOS DE COLISÃO 2D NA UNITY

TRABALHO DE CONCLUSÃO DE CURSO

CORNÉLIO PROCÓPIO

2017

RODRIGO MORETTO

ANÁLISE COMPARATIVA ENTRE DIFERENTES MÉTODOS DE COLISÃO 2D NA UNITY

Trabalho de Conclusão de Curso apresentada à disciplina de Trabalho de Conclusão de Curso 1 da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Bacharel em Engenharia de Software.

Orientador: Paulo Augusto Nardi

CORNÉLIO PROCÓPIO

2017

RESUMO

MORETTO, Rodrigo. Análise comparativa entre diferentes métodos de colisão 2D na Unity. 26 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2017.

As colisões em jogos apresentam um fator importante durante seu desenvolvimento, pois elas irão reger no funcionamento da física e manipulação de eventos. Devido a isso, a escolha de qual o tipo de colisão deverá ser aplicada para determinado objeto ou evento deve ser bem organizada, com o intuito de causar o mínimo ou nenhuma falha durante a execução do jogo. Este trabalho apresentará as vantagens e desvantagens que cada colisão presente na plataforma de desenvolvimento de jogos (game engine) Unity oferece.

Palavras-chave: Jogos Digitais, Unity, Game Engine, Colisão, 2D, Desenvolvimento

ABSTRACT

MORETTO, Rodrigo. Comparative analisis between different 2D collision methods on Unity. 26 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2017.

Collisions in games present an important factor during its development, because they will conduct the operation of physics and event manipulation. Due to that, the choice of which collision type must be applied to a specific object or event should be well organized, with the intention to cause the minimum or no fail during the game execution. This work will show the advantages and disadvantages that each collision available on the game development platform (game engine) Unity can offer.

Keywords: Digital Games, Unity, Game Engine, Collision, 2D, Development

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A indústria de jogos no Brasil cresceu cerca de 600% entre 2008 e 2016 (SILVEIRA, 2017), gerando crescimento na quantidade de empresas de desenvolvimento de jogos. Muitas dessas empresas são pequenas, com funcionários pouco experientes e possuem baixo orçamento para o desenvolvimento de jogos. Tais empresas são conhecidas como *Indie* (MORRIS, 2017). Até o final de 2017 essa indústria está estimada em arrecadar 100 bilhões de dólares de acordo com a Digi-Capital (MORRIS, 2017).

Esse nome é dado a elas pois desenvolvem seus jogos com pouco investimento e pouca ajuda de distribuidoras. Um dos fatores para o crescimento dessas empresas é o aumento de formas de distribuição de jogos, principalmente a distribuição online.

De acordo com o Censo da Indústria Brasileira de Jogos Digitais, a média de funcionários por empresa no Brasil é de 8,5 pessoas (de 133 empresas que participaram da pesquisa de 2014) (FLEURY et al., 2014).

1.1 PROBLEMATIZAÇÃO

Como a maioria dos desenvolvedores iniciantes trabalham com um investimento inicial baixo, além de problemas com publicidade, que é um dos principais fatores que impede o sucesso desses desenvolvedores, o projeto pode encontrar falhas técnicas (REICHERT, 2012) (MOROMISATO, 2013). Essas falhas no planejamento e no plano de negócio da empresa podem levar a uma má reputação do jogo ou da empresa, pois o mesmo pode sofrer problemas de otimização, que afetam o desempenho do jogo e o conforto do jogador.

1.2 JUSTIFICATIVA

Com o crescimento das empresas indie (FLEURY et al., 2014) (MORRIS, 2017), tem aumentado o número de jogos executados em baixa performance, ou seja, os jogos rodam de maneira lenta ou pouco otimizada. Uma possível causa da baixa performance é a inexperiência

dos desenvolvedores. Uma alternativa para suprir a falta de experiência é recorrer a soluções já existentes no mercado, como a plataforma de desenvolvimento de jogos Unity, que será alvo de estudo desse trabalho.

No entanto, as informações sobre a plataforma Unity, mais especificamente sobre os métodos de colisões dessa plataforma são oriundas de sua própria documentação, que se mostra escassa em alguns quesitos, ou de fóruns, podendo trazer informações contraditórias com a documentação. Há uma falta de estudos que gerem soluções fundamentadas em boas práticas.

Este estudo será realizado com o intuito de reduzir essas contradições, trazer dados e informações mais concretas e orientar os desenvolvedores durante a produção do jogo.

1.3 OBJETIVOS

1.3.1 OBJETIVO GERAL

Realizar comparações na plataforma de desenvolvimento de jogos (game engine) Unity, referentes aos métodos de detecção de colisão presentes na engine (por exemplo *BoxCollider2D*, *CircleCollider2D*, que serão explicados e detalhados mais adiante) e sua influência de desempenho, por exemplo quadros por segundo (FPS), consumo de memória, consumo de CPU. Dessas comparações serão gerados relatórios e gráficos comparativos mostrando os resultados, analisando as vantagens e desvantagens de cada colisão descrita, bem como mostrar os cenários onde cada uma se mostra com melhor aproveitamento do desenvolvimento e da plataforma de distribuição (Computadores, Consoles, Dispositivos Móveis, Portáteis).

1.3.2 OBJETIVOS ESPECÍFICOS

Análise dos métodos de detecção de colisão mais comumente utilizados (*BoxCollider2D, CircleCollider2D, PolygonCollider2D e EdgeCollider2D*) e que são oferecidos pela Unity individualmente em diferentes cenários e comparar os resultados entre eles. Serão também comparados os métodos de identificação de colisão que serão analisados são *OnTrigger*, *OnCollision* e *OnOverlap*.

1.4 ORGANIZAÇÃO DO TRABALHO

No Capítulo 2 será apresentado a fundamentação teórica que irá cobrir alguns conceitos de colisão 2D, além de explicitar alguns métodos presentes na Unity. No mesmo capítulo são apresentados alguns trabalhos relacionados sobre métodos de detecção de colisão.

No Capítulo 3 será apresentado a proposta de como o trabalho será realizado, junto do cronograma estimado de seu progresso.

2 FUNDAMENTAÇÃO TEÓRICA

Os jogos digitais são softwares que permitem a interação lúdica entre o jogador e o ambiente em que o jogo é executado, como computadores e videogames, afim de proporcionar entretenimento ao jogador (LUCCHESE; RIBEIRO, 2009).

Nos jogos, as colisões são utilizadas para realizar interações entre objetos. Fisicamente elas são usadas para detectar quando um objeto está em colisão com outro, que pode ser utilizado para criar interações entre os objetos. Por exemplo uma bola cai em cima de um botão (Figura 1), e esta interação realiza um evento de abrir uma porta (Figura 2).

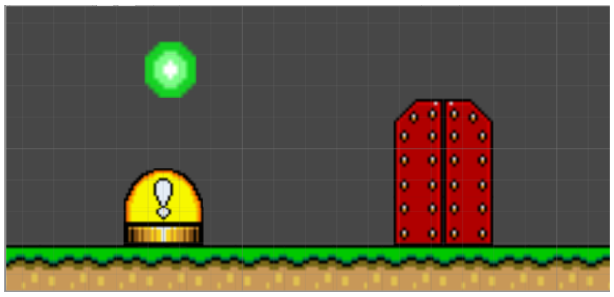

Figura 1: Exemplo de evento de colisão.

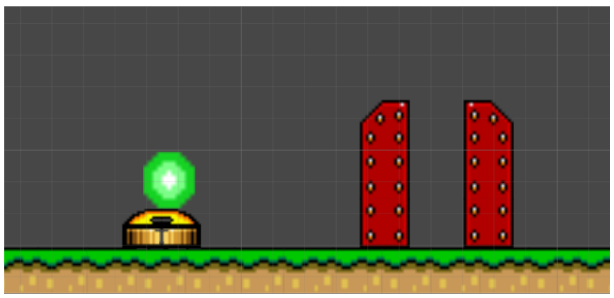

Figura 2: Exemplo de evento de colisão.

2.1 CONCEITOS BÁSICOS DE COLISÃO

Colisões funcionam por cálculos matemáticos, como Teorema de Pitágoras e diferença de pontos, que detectam intersecção entre objetos baseada em coordenadas de suas posições e

rotações, caso produzam alguma, no espaço. Um exemplo é a colisão de círculos, onde cada círculo tem como atributos o seu centro, que irá determinar a sua posição no espaço e o raio. Se a diferença da distância entre os centros dos círculos for menor ou igual a soma dos raios significa que os círculos estão colidindo. Isso é ilustrado nas Figuras 3, 4 e 5.

Na Figura 3 os pontos A e B representam os centros de cada círculo, sendo que o círculo verde tem o raio de comprimento $\overline{AC}$ e o círculo pontilhado tem raio $\overline{BD}$. Então pode ser calculada a distância entre os centros dos círculos através de um Teorema de Pitágoras, tendo o segmento de reta $\overline{AB}$ como sua hipotenusa. Logo em seguida será feita a comparação entre o comprimento da hipotenusa e a soma dos raios dos círculos para verificar estão colidindo. No caso da Figura 3 eles não estão em colisão, pois o comprimento da hipotenusa A-B é maior que a soma dos raios $\overline{AC}$ e $\overline{BD}$ dos círculos, não satisfazendo a condição para que ocorra a colisão.

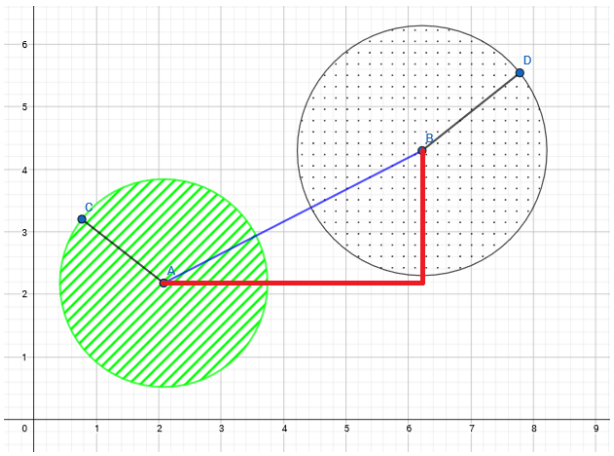

Figura 3: Colisão de Círculos: Círculos não estão colidindo, diferença das distâncias entre os centros é maior que a soma dos raios.

Na Figura 4 o procedimento é o mesmo para verificação. Nesse caso há a colisão e ainda há a intersecção entre os círculos. Isso pode ser confirmado ao comparar o comprimento da hipotenusa A-B com a soma dos raios $\overline{AC}$ e $\overline{BD}$ e verificar que o tamanho da hipotenusa é menor que a soma dos raios. Dependendo dos requisitos e do algoritmo, os pontos onde ocorre a intersecção podem ser armazenados, no caso da Figura 4 são os pontos E e F.

Na Figura 5 o procedimento também é o mesmo e nesse existe a colisão, porém não há intersecção entre os círculos, isso é verificado através do cálculo da diferença entre a hipotenusa A-B e a soma dos raios $\overline{AC}$ e $\overline{BD}$, que nesse caso são do mesmo comprimento, portanto há colisão.

Um código semelhante é explicado no livro Programação de Jogos com C++ e DirectX de André Santee, porém com uma esfera ao invés de um círculo, no entanto o conceito se aplica de forma semelhante (SANTEE, 2005).

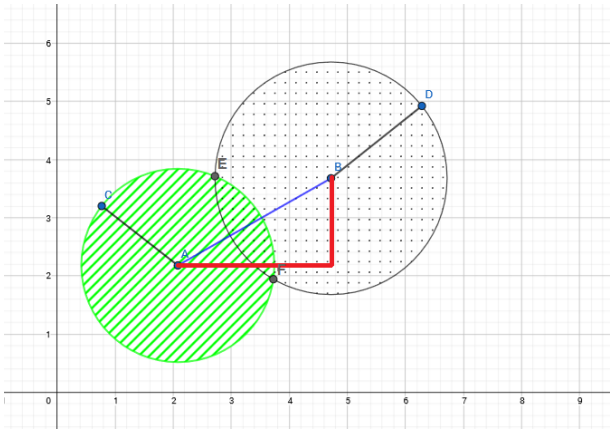

Figura 4: Colisão de Círculos: Círculos estão colidindo e interceptando, diferença das distâncias entre os centros é menor que a soma dos raios.

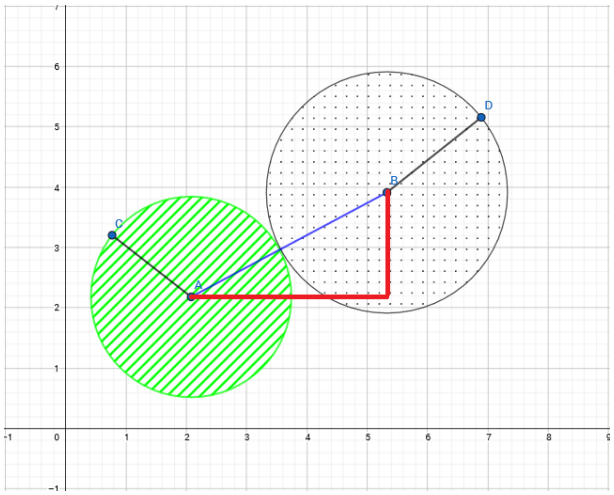

Figura 5: Colisão de Círculos: Círculos estão colidindo, diferença das distâncias entre os centros é igual que a soma dos raios.

O código representado na Figura 6 é basicamente dividido em duas etapas: encontrar a diferença na distância do centro das duas esferas utilizando o teorema de Pitágoras. Essa diferença será então verificada se ela é menor ou igual à soma dos raios das esferas.

No código, é criada uma *struct* Sphere (linha 1), que contém os atributos *pos* (linha 3), recebendo a posição do centro da esfera; e *radius* (linha 4), que recebe o valor do raio da esfera, sendo elas as estruturas presentes na simulação.

A função *SphereCollision* (linha 7) do tipo booleana retorna falso se as esferas não estiverem colidindo e verdadeiro se estiverem. Ela recebe como parâmetro o endereço de duas estruturas do tipo esfera que serão usadas na detecção, já o último parâmetro recebe o endereço da distância entre os centros das esferas, no entanto esse parâmetro é opcional.

Dentro da função é criado um vetor *diff* (linha 9), que será usado para armazenar a

```
1   struct Sphere
2   {
3       float pos[3];
4       float radius;
5   };
6
7   bool SphereCollision (const Sphere *ps1, const Sphere *ps2, float *poutDist = NULL)
8   {
9       float diff[];
10      for (int t = 0; t < 3; t++)
11      {
12          diff[t] = ps1->pos[t] - ps2->pos[t];
13      }
14      const float dist = sqrt(diff[0]*diff[0] + diff[1]*diff[1] + diff[2]*diff[2]);
15      if (poutDist)
16      {
17          *poutDist = dist;
18      }
19      if (dist <= ps1->radius + ps2->radius)
20      {
21          return true;
22      }
23      return false;
24  }
25
```

Figura 6: Código sobre colisão de esferas

Fonte: (SANTEE, 2005).

diferença das distâncias dos centros das esferas em cada eixo através de uma iteração num *for* (linhas 10 a 13).

É criada então uma variável *dist* que recebe a raiz da soma dos quadrados das distâncias entre os centros das esferas (*diff*) (linha 14), ou seja, o teorema de Pitágoras.

Nas linhas 15 a 18 verifica se o *poutDist* é válido, se for, *dist* é atribuído ao local da memória indicado.

As linhas 19 a 22 verifica se há colisão entre as esferas calculando se a distância delas é menor ou igual a soma dos raios das mesmas.

2.2 FÍSICA NA UNITY

Nesta seção são apresentados os métodos e classes dos colisores mais comumente utilizados nos *GameObjects* na plataforma de desenvolvimento de jogos Unity.

GameObjects são as classes básicas para todas as entidades compõem uma cena na Unity, e a elas podem ser atribuídos componentes que alteram suas propriedades.

2.2.1 COLLIDERS 2D

São detalhados com figuras os *colliders* dos tipos *CircleCollider2D*, *BoxCollider2D*, *PolygonCollider2D* e *EdgeColider2D*, pois são os tipos que serão o alvo de estudos desse trabalho, pelo fato de serem mais utilizados comumente e não necessitarem de interações entre outros *colliders* para existirem.

- *CircleCollider2D*

  Componente que adiciona a propriedade de colisão do tipo círculo. É restrito ao raio do *collider*.

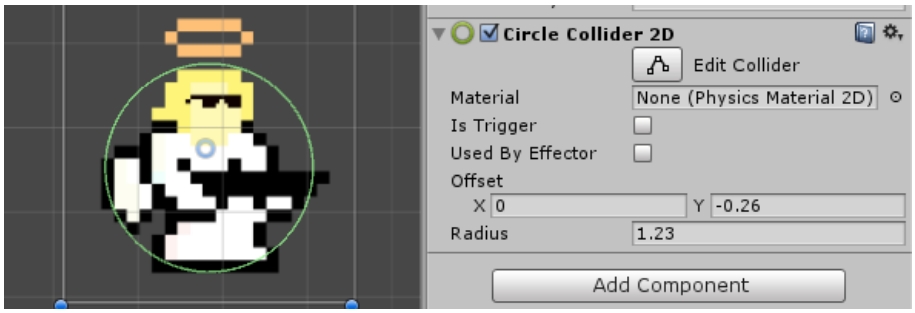

Figura 7: Exemplo de uso do *CircleCollider2D*.
Fonte: (BASICS, 2017)

- *BoxCollider2D*

  Componente que adiciona a propriedade de colisão do tipo retangular. É restrito a altura e largura do *collider*.

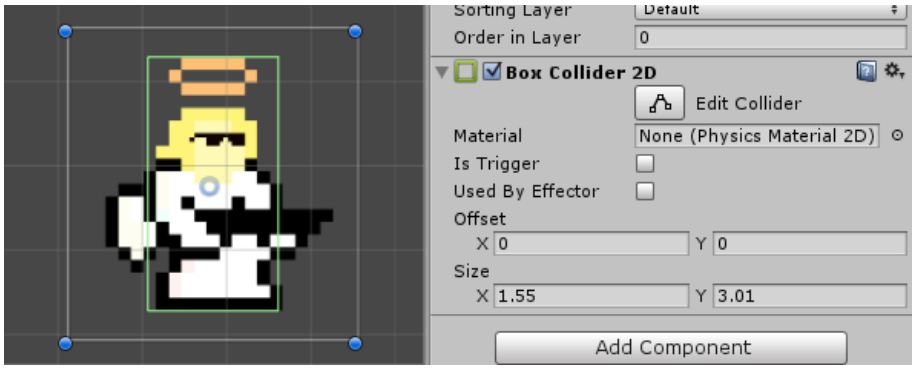

Figura 8: Exemplo de uso do *BoxCollider2D*.
Fonte: (BASICS, 2017)

- *PolygonCollider2D*

  Componente que adiciona a propriedade de colisão do tipo poligonal, ou seja qualquer forma poligonal convexa ou não-convexa fechada. O *PolygonCollider2D* não tem restrição de forma, porém é comumente utilizado quando algum objeto tem uma forma complexa que precisa ser utilizar todo o formato do objeto. A forma final do seu *collider* representa um polígono fechado côncavo ou convexo.

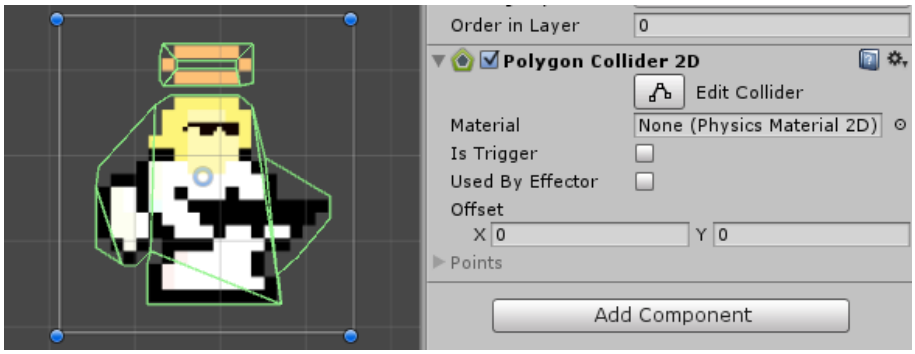

Figura 9: Exemplo de uso do *PolygonCollider2D*.

Fonte: (BASICS, 2017)

- *Edge Collider 2D*

  Componente que adiciona a propriedade de colisão do tipo aresta, ou seja qualquer forma poligonal convexa ou não-convexa aberta. Não tem restrição de forma, porém é comumente utilizado para terrenos e plataformas, onde apenas parte da superfície do objeto é utilizada.

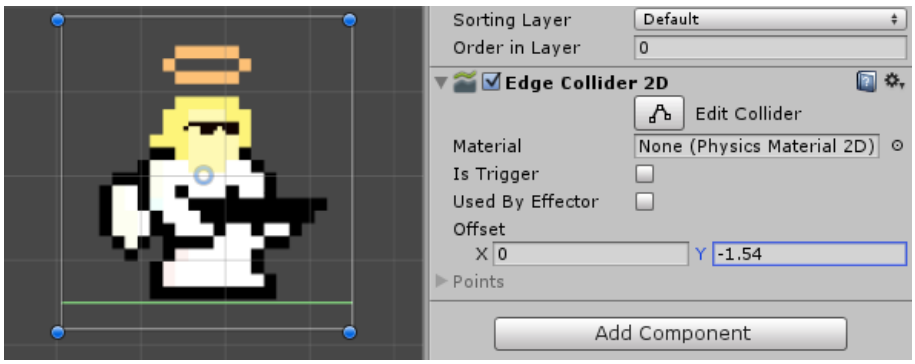

Figura 10: Exemplo de uso do *EdgeCollider2D*.

Fonte: (BASICS, 2017)

- *SpringJoint2D*

  Componente que adiciona a propriedade de colisão do tipo mola.

- *DistanceJoint2D*

  Articulação 2D que une dois *GameObjects* controlados pela física de *RigidBody2D* e os mantém a uma certa distância.

- *HingeJoint2D*

  Componente que adiciona a propriedade do tipo restrição angular, permitindo restringir rotação do objeto em seu próprio eixo (que pode ser personalizado) ou ser ligado (e ter o angulo de rotação restringido) por outro objeto.

- *SliderJoint2D*

  Articulação que permite o *GameObject* deslizar ao longo de uma linha no espaço.

- *WheelJoint2D*

  Usado para simular uma roda rolante. A roda usa uma suspensão de "mola" para manter a distância do corpo principal do veículo.

2.2.2 MENSAGENS DOS COLLIDERS

- *OnCollisionEnter2D*

  É chamado quando o *collider/rigidbody* do objeto ao qual ele pertence começou a tocar outro *collider/rigidbody*.

- *OnCollisionExit2D*

  É chamado quando o *collider/rigidbody* do objeto ao qual ele pertence parou de tocar outro *collider/rigidbody*.

- *OnCollisionStay2D*

  É chamado uma vez por quadro para cada *collider/rigidbody* que está tocando o *collider/rigidbody* do objeto ao qual ele pertence.

- *OnTriggerEnter2D*

  É chamado quando o *collider* entra no *trigger*(gatilho) do objeto ao qual ele pertence.

- *OnTriggerExit2D*

  É chamado quando o *collider* parou de tocar no *trigger* do objeto ao qual ele pertence.

- *OnTriggerStay2D*

  É chamado em quase todos os quadros para cada *collider* que esteja tocando o *trigger* do objeto ao qual ele pertence.

2.2.3 INTERAÇÕES DE COLLIDERS

Colliders interagem uns com os outros diferentemente dependendo de como seus componentes *Rigidbody* são configurados. As três configurações importantes são *Static Collider, Rigidbody Collider e Kinematic Rigidbody Collider* (Unity Technologies, 2017b).

- Static Collider

  Trata-se de um *GameObject* que tem um *Collider* mas não contém *Rigidbody*. *Static Colliders* são utilizados para desenvolvimento do cenário, já que eles se mantém no mesmo lugar e nunca se mexem. Objetos com *Rigidbody* irão colidir com o *static collider* mas não o moverão.

- *Rigidbody Collider*

  Este é um *GameObject* com um *collider* e um *Rigidbody* não-cinemático atribuído a ele. *Colliders Rigidbody* são totalmente simulados pela *engine* de física e podem reagir a colisões e forças aplicadas por *script*. Eles podem colidir com outros objetos (incluindo *static colliders*) e é a configuração mais usada de *Collider* em jogos que usam física.

- *Kinematic Rigidbody Collider*

  Este é um *GameObject* com um *Collider* e um *Rigidbody* cinemático atribuído a ele (a propriedade *IsKinematic* do *RigidBody* é habilitada). Você pode mover um *rigidbody* cinemático a partir de um *script* modificando o componente *Transform* dele, porém não irá responder a colisões e forças como um *rigidbody* não-cinemático. *Rigidbodies* cinemáticos devem ser usados em objetos que podem ser movidos ou habilitados/desabilitados quando necessário, porém devem se comportar como *static colliders*. Diferente de um *static collider*, um *rigidbody* cinemático que se mova irá aplicar fricção para outros objetos e irá "acordar" outros *rigidbodies* quando fazem contato.

A Tabela 1 mostra quais interações de colisões geram mensagens para os *triggers* enviarem.

Alguns exemplos de quando mensagens de *trigger* são enviadas sobre uma colisão podem ser comparados seguindo a Tabela 1. Comparando a colisão entre um *Static Collider* e um *Rigidbody Trigger Collider*, ele envia uma mensagem de *trigger*. Porém se forem comparados *Kinematic Rigidbody Collider* com um *Rigidbody Collider*, ele não envia mensagem de *trigger*.

Quando ocorre detecção de colisão e mensagens *não-triggers* são enviadas ao ocorrer a colisão, esse envio acontece apenas entre *Rigidbody Collider com Static Collider, Rigidbody*

Trigger messages are sent upon collision						
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Tabela 1: Quando mensagens de *trigger* são enviadas de acordo com o tipo de colisão

Collider com Rigidbody Collider, Rigidbody Collider com Kinematic Rigidbody Collider, Static Collider com Rigidbody Collider, Kinematic Rigidbody Collider com Rigidbody Collider.

2.2.4 FUNÇÕES DA PHYSICS2D

- *Raycast*

  Emite um raio contra *colliders* na cena.

- *Linecast*

  Emite um segmento de linha contra *colliders* na cena.

- *Overlap*

  Checa se um *collider* intercepta com uma área.

Em um *Raycast* é configurado o ponto inicial e a direção desse ponto para detectar se algo se encontra nesse raio (KACER, 2011), (DIGISCOT, 2015). Esse raio emitido pode ser infinito no espaço.

Em um *Linecast* é configurado o ponto inicial e o ponto final. Se houver algum *collider* entre esses dois pontos ele irá retornar verdadeiro (KACER, 2011), (DIGISCOT, 2015).

2.2.5 SUBDIVISÕES DAS FUNÇÕES DA PHYSICS2D

- *All*

  Adquire uma lista de todos os colliders que interceptam em uma área.

- *NonAloc*

  Adquire uma lista de todos os colliders que interceptam em uma área específica.

2.2.6 SUBDIVISÕES DOS OVERLAPS

- *Area*

  Verifica se um collider está dentro de uma área retangular.

- *Box*

  Verifica se um collider está dentro de uma área de caixa.

- *Capsule*

  Verifica se um collider está dentro de uma área de capsula.

- *Circle*

  Verifica se um collider está dentro de uma área de círculo.

- *Collider*

  Obtém uma lista de todos os colliders que se sobrepõem ao collider.

- *Point*

  Verifica se um collider se sobrepõe a um ponto no espaço.

(Unity Technologies, 2017b)

2.3 TRABALHOS RELACIONADOS

Existem muitos métodos de implementação de detecção de colisão, e alguns deles podem ser implementados na Unity além dos já existentes. Nesta seção serão mencionados brevemente dois métodos muito utilizados e semelhantes em certos aspectos para detecção de colisão, que podem ser implementados na Unity através de implementação própria, como o AABB (*Axis Alligned Bounding Box*). Em alguns métodos apresentados nessa seção, suas comparações de performance foram realizadas com outros métodos de colisões já existentes, porém nenhum deles feitos na Unity e sim (a maioria) em *engine* própria. Este trabalho tem a intenção de realizar comparações desses métodos ou de métodos que se assemelham aos mesmos, porém na Unity e fundamentar um estudo comparativo para análises futuras.

2.3.1 AABB (AXIS ALLIGNED BOUNDING BOX)

O método de AABB cria uma caixa que envolve todo o volume do objeto, porém, diferente do OBB, as caixas não são alinhadas com o eixo base do próprio objeto, e sim com o eixo do espaço mundial onde o objeto se encontra. Em outras palavras, se o objeto rotacionar, o volume da caixa que será usada para detectar a colisão irá aumentar, pois não é fixo ao eixo base do objeto. Isso quer dizer que se dois objetos, dependendo de como estiverem arranjados no espaço, mesmo que não estejam colidindo, a detecção pode resultar que estão, pois suas caixas que os envolvem estão colidindo, como mostrado na Figura 11. Outra representação do AABB pode ser encontrada na Figura 12 B.

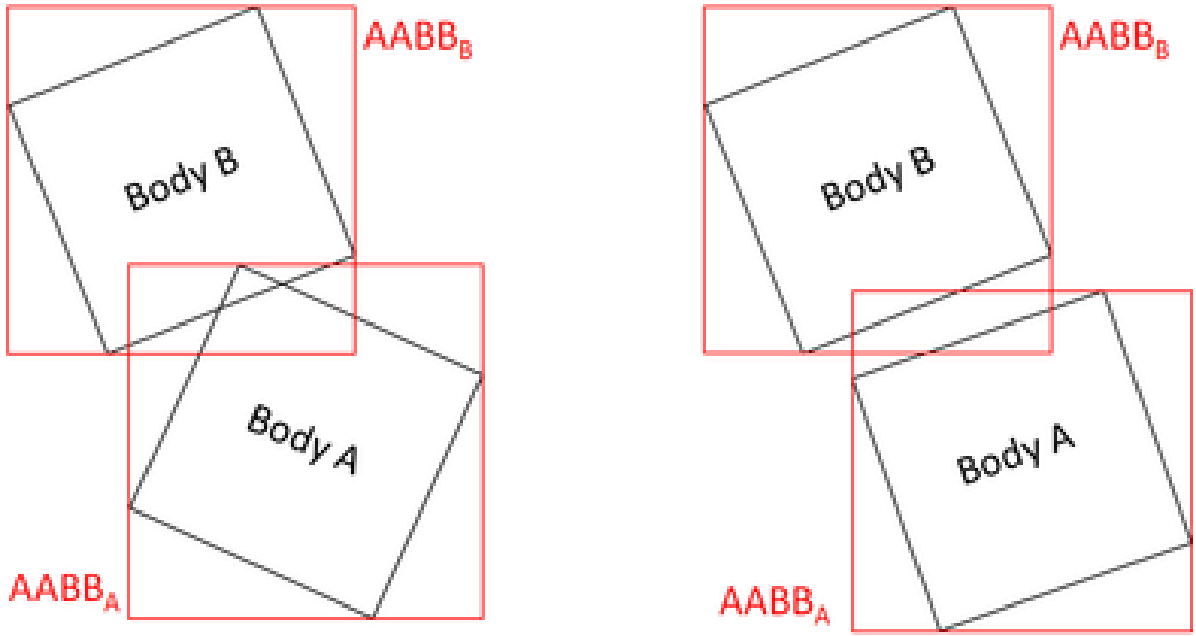

Figura 11: Exemplo de colisão AABB.

Fonte: (EICHNER, 2014)

Basicamente o que o algoritmo faz é adquirir os extremos das caixas, geralmente por arestas, que estão envolvendo os objetos e calcula a diferença entre eles. (EICHNER, 2014).

2.3.2 OBB (ORIENTED BOUNDING BOX)

(EBERLY, 2002), (EICHNER, 2014), (ERICSON, 2004)

O método de OBB funciona criando uma caixa que envolve o volume de cada objeto que estará envolvido no teste de colisão tendo cada um deles como orientação seu eixo base. A detecção pode ser feita de várias formas, a mais simples é checar se os vértices de uma caixa

estão dentro de uma outra e vice-versa. Esse método tende a ser mais preciso que o AABB, pois se o objeto rotaciona, a caixa que o envolve também irá seguir essa rotação. No entanto tem um custo ligeiramente maior na performance. Esse método se assemelha ao componente *BoxCollider2D* na Unity. Um exemplo do OBB está representado na Figura 12 C.

2.3.3 BOUNDING SPHERE

O *Bounding Sphere* adquire o valor do centro da esfera e raio dos objetos que estarão envolvidos na detecção. Se a diferença da distância entre os centros das esferas for menor que a soma dos raios, então as esferas estão colidindo (um código semelhante foi explicado na seção 2.1 na Figura 6) (SANTEE, 2005), (DOPERTCHOUK, 2000). Um exemplo de *Bounding Sphere* está apresentado na Figura 12 A.

2.3.4 CONVEX HULL

O *Convex Hull* procura atribuir o menor volume convexo para que possa conter o objeto, ou seja, o volume apresenta um sólido que não contém concavidades. Existem vários algoritmos que podem realizar os cálculos para formar esses volumes e/ou que permitem fazê-los de maneira mais customizáveis. (CHAZELLE, 1993), (BERG et al., 2008). Um exemplo de volume *Convex Hull* é apresentado na Figura 12 D.

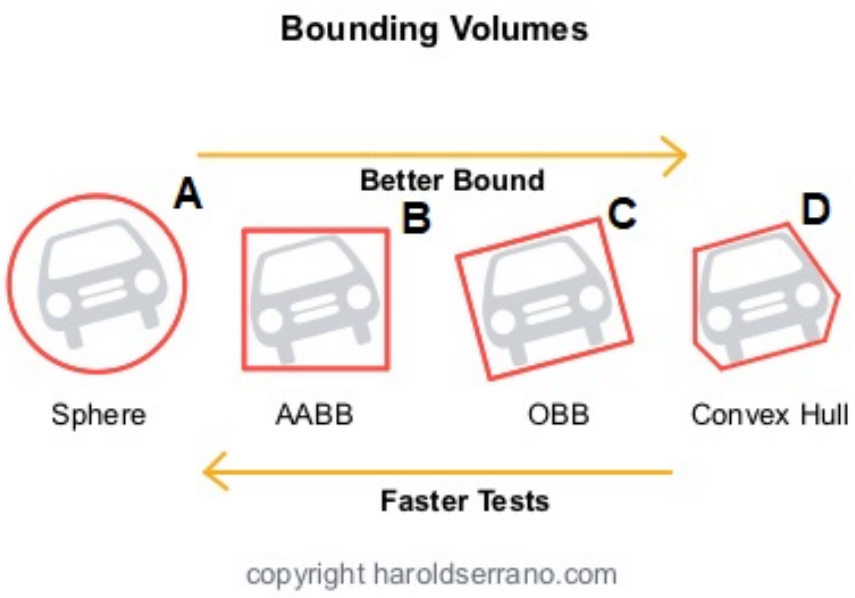

Figura 12: Exemplo de colisão OBB e outros.

Fonte: (SERRANO, 2016)

3 PROPOSTA

Este trabalho de conclusão de curso irá se especificar em um dos problemas de conceito técnico, sendo ele relacionado a detecção de colisões no jogo, ou seja, quando dois objetos se interceptam. A intenção do trabalho é realizar as comparações das colisões e fundamentá-las num estudo para evitar as divergências entre informações devido a documentação da ferramenta pouco completa e dados oriundos de outras fontes.

Será adotada uma das *engines* mais populares entre os desenvolvedores *indie* e algumas grandes empresas, a Unity (Unity Technologies, 2017a) (BANERJEE, 2017), também muito utilizada no Brasil de acordo com o Censo da Indústria Brasileira de Jogos Digitais (106 das 133 empresas pesquisadas em 2014, como mostrado na Tabela 2) (FLEURY et al., 2014) pelo fato de ter uma burocratização fácil.

Engine Utilizada	Quantidade de empresas que utilizam	Porcentagem de empresas que utilizam
Unity	106	79,7%
Tecnologia própria	25	18,8%
Cocos 2D	18	13,53%
Blender	13	9,77%
Corona SDK	11	8,27%
Construct2	11	8,27%
GameMaker	8	6,02%
Flash	8	6,02%
Unreal3 (UDK)	7	5,26%
Marmelade	2	1,5%
CryEngine	1	0,75%
Outros	16	12,03%

Tabela 2: Tabela de Engines mais utilizadas no Brasil
(FLEURY et al., 2014)

Realizar comparações dos métodos de colisão oferecidos gratuitamente pela *game engine* Unity. Essas comparações são divididas em 2 grandes tipos: *Pre-runtime* e *Procedural-runtime*.

3.1 PRE-RUNTIME

Os objetos, seus *colliders*, componentes de física e programação são inseridos na cena antes de serem executados. O programa então é executado, os objetos reagem de acordo com as suas posições na cena e próprias configurações. Os resultados são então coletados.

3.2 PROCEDURAL-RUNTIME

Os objetos são gerados em tempo de execução, ou seja, são invocados durante a execução do programa, de acordo com a programação dos objetos-base, dos controladores e das configurações da cena. O programa é executado, os objetos são gerados em tempo de execução, reagem de acordo com suas configurações de colisão e cena. Os resultados são então coletados.

3.3 ENFATIZAÇÃO DAS COMPARAÇÕES

As comparações serão feitas usando a interface de duas dimensões da Unity para obter resultados mais simples com esse estudo. Os tipos de *colliders* que terão mais ênfase nas comparações serão *Box, Circle, Polygon e Edge*, pelo fato de serem mais utilizados.

As medições incluirão taxa de quadros por segundo (FPS), taxa de consumo do processador, taxa de consumo de memória RAM. Essas medições serão coletadas e analisadas posteriormente.

As comparações serão casos sintéticos, ou seja, não será um jogo específico que sofrerá a comparação, mas sim cenários de teste elaborados para estressar os recursos e verificar os limites de cada método de colisão.

O hardware que será utilizado para realizar as comparações é um notebook, com as seguintes configurações:

- Processador: Intel Core i5 4210U
- Placa de Vídeo: Intel HD Graphics 4400
- Memória RAM: 8GB
- Disco Interno: SSD 120GB
- Disco Externo: HDD 1TB

- Sistema Operacional: Windows 10 Pro

Também será usado o software gratuito de edição de imagem *Krita* (Krita Foundation, 2017) para desenvolver os *sprites* que serão usados nas comparações.

3.4 CRONOGRAMA

Na tabela 3 é apresentado um cronograma com as datas estimativas da progressão do projeto.

Cronograma	Ago 2017	Set 2017	Out 2017	Nov de 2017	Dez 2017	Jan 2018	Fev 2018	Mar 2018	Abr 2018	Mai 2018
Pesquisas Iniciais	X	X								
Pesquisas sobre colisões		X								
Pesquisas sobre colisões na Unity			X							
Escrita do TCC 1			X							
Apresentação do TCC 1			X							
Proposta de cenários				X	X					
Realização das comparações					X	X	X	X		
Registro de resultados						X	X	X		
Escrita do TCC 2								X	X	X
Apresentação do TCC 2										X

Tabela 3: Cronograma de progresso estimado do trabalho

REFERÊNCIAS

BANERJEE, S. **7 Most Popular Game Development Engines You Should Consider**. 2017. Acessado em 5 de outubro de 2017. Disponível em: <https://www.rswebsols.com/tutorials/software-tutorials/popular-game-development-engines>.

BASICS, . min. **Unity3d collider types**. 2017. Acessado em 5 de outubro de 2017. Disponível em: <http://10minbasics.com/unity3d-collider-types/>.

BERG, M. D. et al. **Computational Geometry: Introduction**. [S.l.]: Springer, 2008.

CHAZELLE, B. An optimal convex hull algorithm in any fixed dimension. **Discrete & Computational Geometry**, Springer, v. 10, n. 1, p. 377–409, 1993.

DIGISCOT. **Difference between Linecast and Raycast - Unity Answers**. 2015. Disponível em: <http://answers.unity3d.com/questions/848189/difference-between-linecast-and-raycast.html>.

DOPERTCHOUK, O. **Simple Bounding-Sphere Collision Detection**. 2000. Disponível em: <https://www.gamedev.net/articles/programming/math-and-physics/simple-bounding-sphere-collision-detection-r1234/>.

EBERLY, D. Dynamic collision detection using oriented bounding boxes. **Geometric Tools, Inc**, 2002.

EICHNER, H. **Game Physics - 2D Collision Detction**. 2014. Acessado em 10 de outubro de 2017. Disponível em: <http://myselph.de/gamePhysics/collisionDetection.html>.

ERICSON, C. **REAL-TIME COLLISION DETECTION**. 1. ed. [S.l.: s.n.], 2004. 593 p.

FLEURY, A.; SAKUDA, L. O.; CORDEIRO, J. H. D. O. I censo da indústria brasileira de jogos digitais. **NPGT-USP e BNDES: São Paulo e Rio de Janeiro**, 2014.

KACER. **Raycast and Linecast - Unity Answers**. 2011. Acessado em 10 de outubro de 2017. Disponível em: <http://answers.unity3d.com/questions/164770/raycast-and-linecast.html>.

Krita Foundation. **Krita**. 2017. Acessado em 8 de outubro de 2017. Disponível em: <https://krita.org/en/>.

LUCCHESE, F.; RIBEIRO, B. Conceituacao de jogos digitais. **Sao Paulo**, 2009.

MOROMISATO, G. **What is the biggest problem you face as a independent game developer?** 2013. Acessado em 5 de outubro de 2017. Disponível em: <https://www.quora.com/What-is-the-biggest-problem-you-face-as-a-independent-game-developer>.

MORRIS, A. **How The Rise of Indie Games Has Revitalized the Video Game Industry**. 2017. Acessado em 5 de outubro de 2017. Disponível em: <https://www.allbusiness.com/indie-games-video-game-industry-101485-1.html>.

REICHERT, K. **Top 5 Problems Faced By Indie Game Developers**. 2012. Acessado em 5 de outubro de 2017. Disponível em: <https://goo.gl/jj6yhp>.

SANTEE, A. **Programação de Jogos com C++ e DirectX**. [S.l.: s.n.], 2005. 400 p.

SERRANO, H. **How does a Physics Engine work? An Overview**. 2016. Acessado em 10 de outubro de 2017. Disponível em: <https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview>.

SILVEIRA, D. **Número de desenvolvedores de games cresce 600% em 8 anos, diz associação**. 2017. Acessado em 8 de outubro de 2017. Disponível em: <https://g1.globo.com/economia/negocios/noticia/numero-de-desenvolvedores-de-games-cresce-600-em-8-anos-diz-associacao.ghtml>.

Unity Technologies. **Unity - Game Engine**. 2017. Acessado em 28 de agosto de 2017. Disponível em: <https://unity3d.com/>.

Unity Technologies. **Unity Manual**. 2017. Acessado em 28 de agosto de 2017. Disponível em: <https://docs.unity3d.com>.