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

2019

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 da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do grau de Bacharel em Engenharia de Software.

Orientador: Doutor Paulo Augusto Nardi

CORNÉLIO PROCÓPIO

2019

RESUMO

MORETTO, Rodrigo. Análise comparativa entre diferentes métodos de colisão 2D na Unity. 77 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

As colisões em jogos apresentam um fator importante durante seu desenvolvimento, pois elas irão influenciar 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. 77 f. Trabalho de Conclusão de Curso – Engenharia de Software, Universidade Tecnológica Federal do Paraná. Cornélio Procópio, 2019.

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

FIGURA 1 – Exemplo de interação de eventos 13
FIGURA 2 – Exemplo de interação de eventos 13
FIGURA 3 – Colisão de Círculos: Círculos não estão colidindo. 14
FIGURA 4 – Colisão de Círculos: Círculos estão colidindo e interceptando. 15
FIGURA 5 – Colisão de Círculos: Círculos estão colidindo. 15
FIGURA 6 – Código colisão de esferas 16
FIGURA 7 – Exemplo de CircleCollider2D 17
FIGURA 8 – Exemplo de BoxCollider2D 17
FIGURA 9 – Exemplo de PolygonCollider2D 18
FIGURA 10 – Exemplo de EdgeCollider2D 18
FIGURA 11 – Exemplo de colisão AABB 24
FIGURA 12 – Exemplo de colisão OBB e outros 25
FIGURA 13 – Profiler da Unity 31
FIGURA 14 – Profiler Data Exporter 32
FIGURA 15 – Exemplo Cenário 33
FIGURA 16 – Frequência de uso de Colliders 2D 35
FIGURA 17 – Frequência de uso de mensagens de Colliders 2D 35
FIGURA 18 – Média de objetos com Colliders 2D presentes em uma cena 36
FIGURA 19 – Resultado 10 Objetos Box Pre-Runtime 36
FIGURA 20 – Resultado 500 Objetos Box Pre-Runtime 37
FIGURA 21 – Resultado 1000 Objetos Box Pre-Runtime 37
FIGURA 22 – Resultado 10 Objetos Box Runtime 38
FIGURA 23 – Resultado 100 Objetos Box Runtime 38
FIGURA 24 – Resultado 0050 Objetos Circle Pre-Runtime 39
FIGURA 25 – Resultado 1000 Objetos Circle Pre-Runtime 39
FIGURA 26 – Resultado 50 Objetos Circle Runtime 40
FIGURA 27 – Resultado 100 Objetos Circle Runtime 40
FIGURA 28 – Resultado 1000 Objetos Edge Pre-Runtime 41
FIGURA 29 – Problemas 1000 Objetos Edge Pre-Runtime 41
FIGURA 30 – Resultado 1000 Objetos Edge Runtime 42
FIGURA 31 – Resultado 10 Objetos Polygon Pre-Runtime 43
FIGURA 32 – Resultado 50 Objetos Polygon Pre-Runtime 43
FIGURA 33 – Resultado 100 Objetos Polygon Pre-Runtime 43
FIGURA 34 – Resultado 500 Objetos Polygon Pre-Runtime 44
FIGURA 35 – Quadros por segundo 1000 Objetos Polygon Pre-Runtime 44
FIGURA 36 – Resultado 1 Objeto Polygon Runtime 45
FIGURA 37 – Resultado 10 Objetos Polygon Runtime 45
FIGURA 38 – Resultado 50 Objetos Polygon Runtime 46
FIGURA 39 – Quadros por segundo 50 Objetos Polygon Runtime 46
FIGURA 40 – Gráfico de porcentagem de uso da CPU com 50 objetos Box Collider em Pre-Runtime 48
FIGURA 41 – Gráfico de porcentagem de uso da CPU com 50 objetos Circle Collider

em Pre-Runtime 48
FIGURA 42 – Gráfico de porcentagem de uso da CPU com 50 objetos Polygon Collider em Pre-Runtime 49
FIGURA 43 – Gráfico de número de vezes que o método foi chamado com 50 objetos Box Collider em Pre-Runtime 49
FIGURA 44 – Gráfico de número de vezes que o método foi chamado com 50 objetos Circle Collider em Pre-Runtime 50
FIGURA 45 – Gráfico de número de vezes que o método foi chamado com 50 objetos Polygon Collider em Pre-Runtime 50
FIGURA 46 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos Box Collider em Pre-Runtime 51
FIGURA 47 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos Circle Collider em Pre-Runtime 51
FIGURA 48 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos Polygon Collider em Pre-Runtime 52
FIGURA 49 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos Polygon Collider em Pre-Runtime 52
FIGURA 50 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos Box Collider em Pre-Runtime com física 53
FIGURA 51 – Gráfico dequantidade de Garbage Collection em bytes com 50 objetos Circle Collider em Pre-Runtime com física 53
FIGURA 52 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos Polygon Collider em Pre-Runtime com física 54
FIGURA 53 – Gráfico de porcentagem de uso da CPU com 50 objetos por segundo Box Collider em Runtime 54
FIGURA 54 – Gráfico de porcentagem de uso da CPU com 50 objetos Circle Collider em Runtime 55
FIGURA 55 – Gráfico de porcentagem de uso da CPU com 50 objetos po segundo Polygon Collider em Runtime 55
FIGURA 56 – Gráfico de número de vezes que o método foi chamado com 50 objetos Box Collider em Runtime 56
FIGURA 57 – Gráfico de número de vezes que o método foi chamado com 50 objetos por segundo Circle Collider em Runtime 56
FIGURA 58 – Gráfico de número de vezes que o método foi chamado com 50 objetos por segundo Polygon Collider em Runtime 57
FIGURA 59 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo Box Collider em Runtime 57
FIGURA 60 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo Circle Collider em Pre-Runtime 58
FIGURA 61 – Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo Polygon Collider em Runtime 58
FIGURA 62 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos por segundo Box Collider em Runtime 59
FIGURA 63 – Gráfico dequantidade de Garbage Collection em bytes com 50 objetos por segundo Circle Collider em Runtime 59
FIGURA 64 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos por segundo Polygon Collider em Runtime 60
FIGURA 65 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos

por segundo Box Collider em Runtime com física 60
FIGURA 66 – Gráfico dequantidade de Garbage Collection em bytes com 50 objetos por segundo Circle Collider em Runtime com física 61
FIGURA 67 – Gráfico de quantidade de Garbage Collection em bytes com 50 objetos por segundo Polygon Collider em Runtime com física 61
FIGURA 68 – Pergunta 1 do questionário 66
FIGURA 69 – Pergunta 2 do questionário 66
FIGURA 70 – Pergunta 3 do questionário 67
FIGURA 71 – Pergunta 4 do questionário 67
FIGURA 72 – Pergunta 5 do questionário 68
FIGURA 73 – Pergunta 6 do questionário 68
FIGURA 74 – Pergunta 7 do questionário 69
FIGURA 75 – Pergunta 8 do questionário 69
FIGURA 76 – Pergunta 9 do questionário 70
FIGURA 77 – Respostas Pergunta 1 do questionário 71
FIGURA 78 – Respostas Pergunta 2 do questionário 71
FIGURA 79 – Respostas Pergunta 3 do questionário 72
FIGURA 80 – Respostas Pergunta 4 do questionário 72
FIGURA 81 – Respostas Pergunta 5 do questionário 73
FIGURA 82 – Respostas Pergunta 6 do questionário 73
FIGURA 83 – Respostas Pergunta 7 do questionário 74
FIGURA 84 – Respostas Pergunta 8 do questionário 74
FIGURA 85 – Respostas Pergunta 9 do questionário 75

LISTA DE TABELAS

TABELA 1 – Quando mensagens de *trigger* são enviadas de acordo com o tipo de colisão 21
TABELA 2 – Quando mensagens de *trigger* são enviadas de acordo com o tipo de colisão 21
TABELA 3 – Tabela de Engines mais utilizadas no Brasil 26
TABELA 4 – Cronograma de progresso estimado do trabalho 28
TABELA 5 – Exemplo das tabelas presentes no apêndice remoto 77

SUMÁRIO

1 INTRODUÇÃO 10
1.1 PROBLEMATIZAÇÃO 10
1.2 JUSTIFICATIVA 10
1.3 OBJETIVOS 11
1.3.1 Objetivo Geral 11
1.3.2 Objetivos Específicos 11
1.4 ORGANIZAÇÃO DO TRABALHO 11
2 FUNDAMENTAÇÃO TEÓRICA 13
2.1 CONCEITOS BÁSICOS DE COLISÃO 13
2.2 FÍSICA NA UNITY 16
2.2.1 *Colliders* 2D 17
2.2.2 Mensagens dos Colliders 19
2.2.3 Mensagens dos Triggers 20
2.2.4 Interações de Colliders 20
2.2.5 Funções da Physics2D 22
2.2.6 Subdivisões das funções da Physics2D 22
2.2.7 Subdivisões dos Overlaps 22
2.3 TRABALHOS RELACIONADOS 23
2.3.1 AABB (Axis Alligned Bounding Box) 23
2.3.2 OBB (Oriented Bounding Box) 23
2.3.3 Bounding Sphere 24
2.3.4 Convex Hull 25
2.3.5 Otimização de Colisões na *Unity* 25
3 PROPOSTA 26
3.1 PRE-RUNTIME 27
3.2 RUNTIME 27
3.3 ENFATIZAÇÃO DAS COMPARAÇÕES 27
3.4 CRONOGRAMA 28
4 METODOLOGIAS 29
4.1 PESQUISA DE CAMPO 29
4.2 *PROFILER* DA *UNITY* 30
4.3 OBTENÇÃO E FILTRAGEM DE DADOS 32
4.4 CENÁRIOS DE COMPARAÇÃO 32
5 RESULTADOS E DISCUSSÕES 34
5.1 PESQUISA DE CAMPO 34
5.2 RESULTADOS COM *BOX COLLIDER* 35
5.2.1 *Box Collider - Pre-Runtime* 35
5.2.2 *Box Collider - Runtime* 37
5.3 RESULTADOS COM *CIRCLE COLLIDER* 38
5.3.1 *Circle Collider - Pre-Runtime* 38
5.3.2 *Circle Collider - Runtime* 39

5.4 RESULTADOS COM *EDGE COLLIDER* 40
5.4.1 *Edge Collider - Pre-Runtime* 40
5.4.2 *Edge Collider - Runtime* 41
5.5 RESULTADO COM *POLYGON COLLIDER* 42
5.5.1 *Polygon Collider - Pre-Runtime* 42
5.5.2 *Polygon Collider - Runtime* 44
5.6 VISÃO GERAL DOS RESULTADOS 47
5.6.1 Comparações em Pre-Runtime 48
5.6.2 Comparações em *Runtime* 51
6 CONSIDERAÇÕES FINAIS 62
6.1 IMPORTÂNCIA 62
6.2 RESULTADOS 62
6.3 TRABALHOS FUTUROS 63
REFERÊNCIAS 64
Apêndice A – QUESTIONÁRIO - PESQUISA DE CAMPO 66
Apêndice B – RESULTADOS DO QUESTIONÁRIO 71
Apêndice C – TABELAS DE RESULTADOS DAS COMPARAÇÕES 76
C.1 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *BOX COLLIDER* 77
C.2 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *CIRCLE COLLIDER* 77
C.3 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *EDGE COLLIDER* 77
C.4 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *POLYGON COLLIDER* 77

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 é o 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.

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* e *OnCollision*.

1.4 ORGANIZAÇÃO DO TRABALHO

No Capítulo 2, é apresentada 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, é apresentada a proposta de como o trabalho será realizado, junto do cronograma estimado de seu progresso. No Capítulo 4, são apresentadas as metodologias utilizadas para a realização das comparações do projeto. No Capítulo 5, são apresentados os resultados das comparações,

bem como a discussão perante as mesmos. No Capítulo 6, é apresentada a conclusão e as considerações finais do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Os jogos digitais são programas 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).

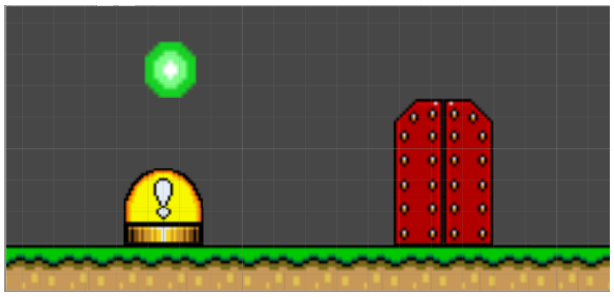

Figura 1: Exemplo de evento de colisão.

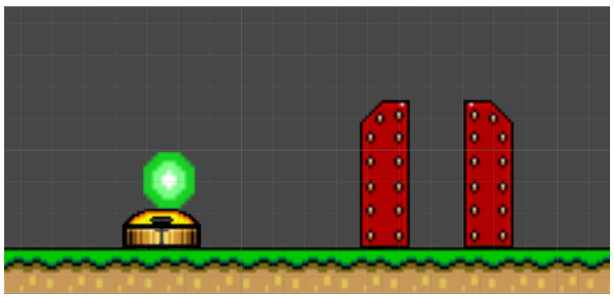

Figura 2: Exemplo de evento de colisão.

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 atributo 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 por meio 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.

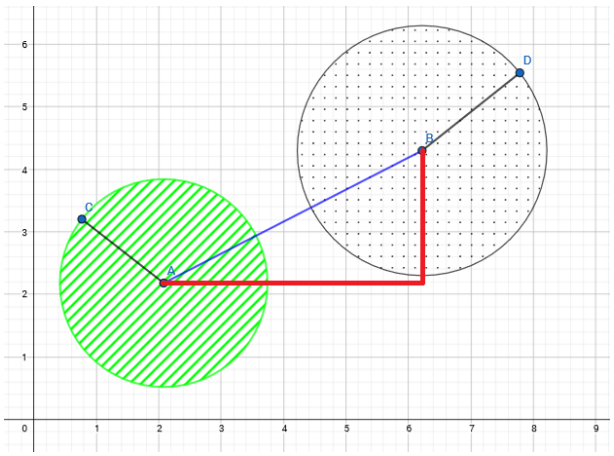

Figura 3: Colisão de Círculos: Círculos não estão colidindo, diferença das distâncias entre os centros é maior que a soma dos raios.

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 $\overline{AB}$ 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).

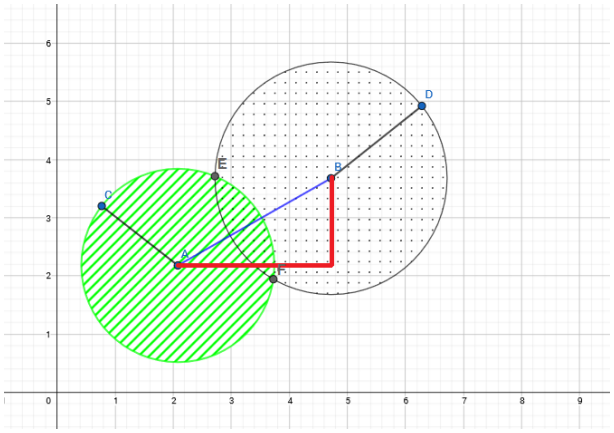

Figura 4: Colisão de Círculos: Círculos estão colidindo e interceptando, diferença das distâncias entre os centros é menor que a soma dos raios.

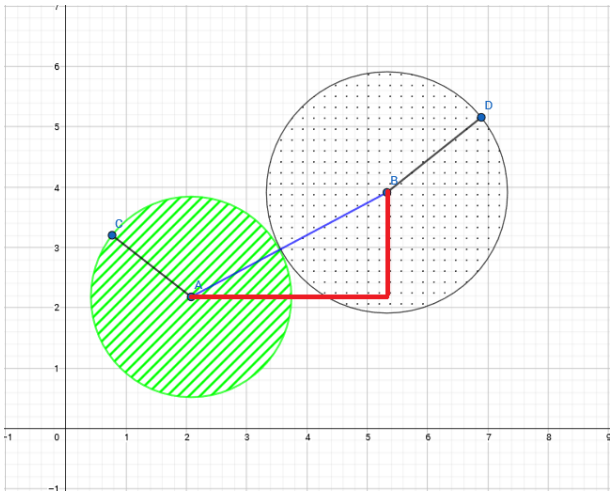

Figura 5: Colisão de Círculos: Círculos estão colidindo, diferença das distâncias entre os centros é igual que a soma dos raios.

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 que 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*.

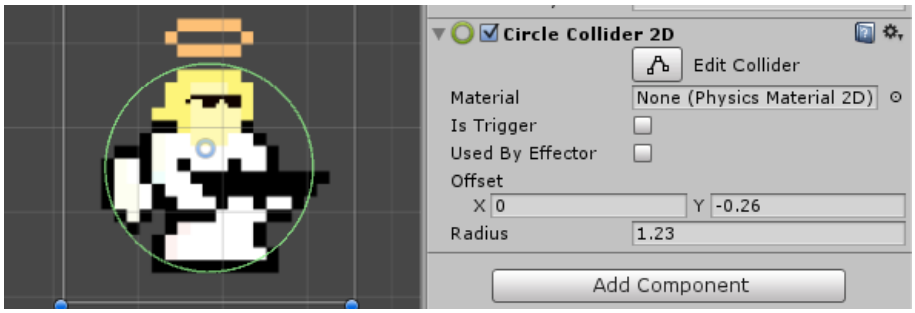

Figura 7: Exemplo de uso do *CircleCollider2D*.
Fonte: (BASICS, 2017)

- *BoxCollider2D*

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

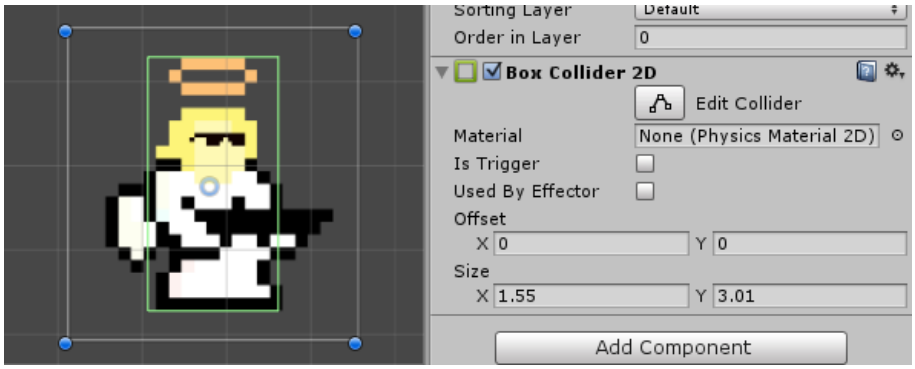

Figura 8: Exemplo de uso do *BoxCollider2D*.
Fonte: (BASICS, 2017)

- *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.

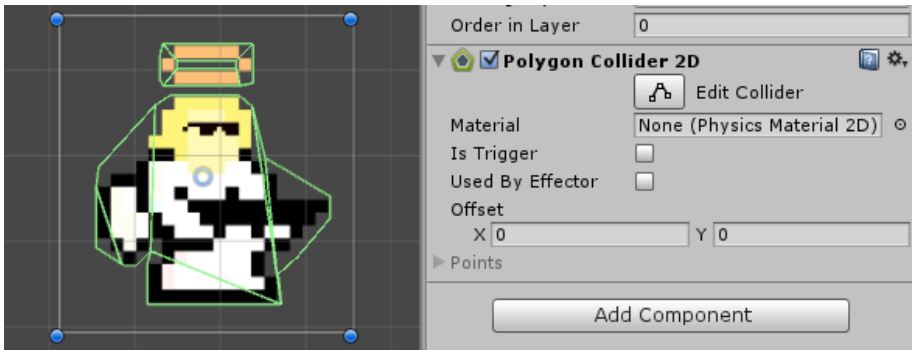

Figura 9: Exemplo de uso do *PolygonCollider2D*.

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.

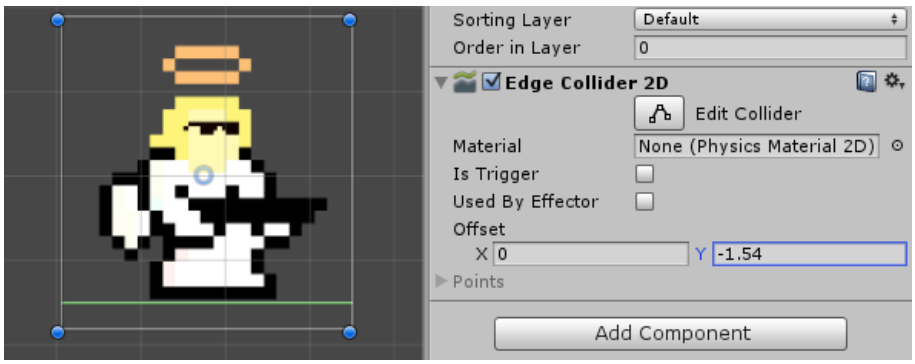

Figura 10: Exemplo de uso do *EdgeCollider2D*.

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.

- *Composite Collider 2D*

  Não é consistido por uma forma, mas sim pelo conjunto de *Box Colliders* e/ou *Polygon Colliders* para construir uma colisão mais complexa.

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.

2.2.3 MENSAGENS DOS TRIGGERS

- *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.

2.2.4 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. Já a Tabela 2, mostra quais interações de colisões geram mensagens para os *colliders*.

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

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				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

Tabela 2: Quando mensagens de *trigger* são enviadas de acordo com o tipo de colisão

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 Collider com Rigidbody Collider, Rigidbody Collider com Kinematic Rigidbody Collider, Static Collider com Rigidbody Collider, Kinematic Rigidbody Collider com Rigidbody Collider*. Assim como mostrado na Tabela 2.

2.2.5 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).

2.2.6 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.

2.2.7 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.

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)

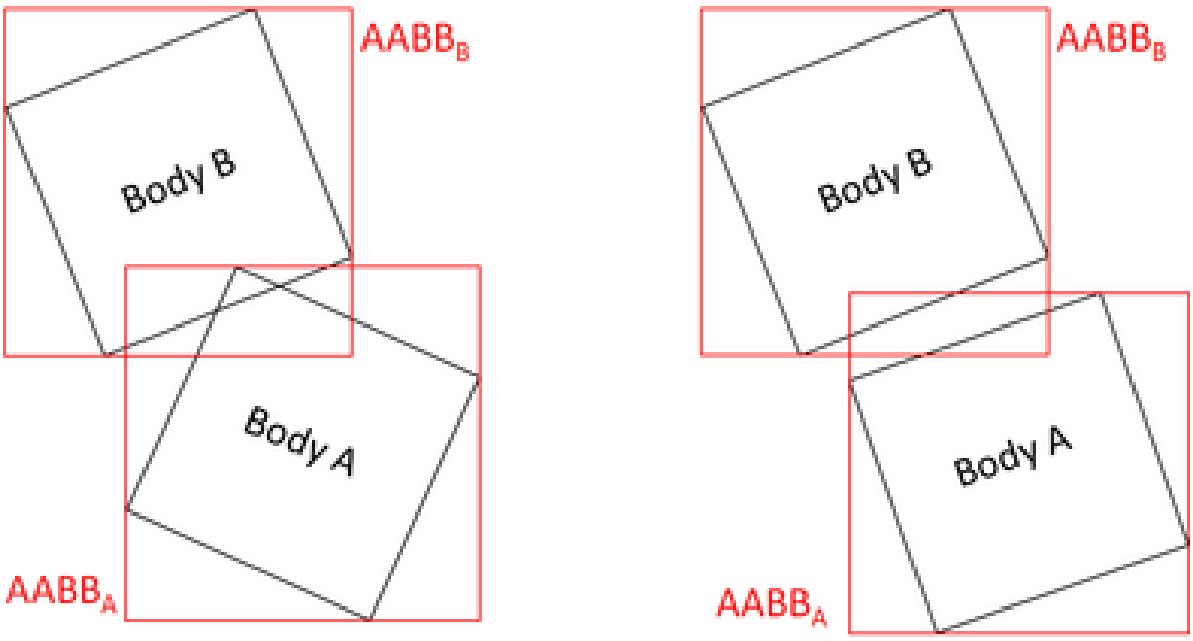

Figura 11: Exemplo de colisão AABB.

Fonte: (EICHNER, 2014)

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 fazelos de maneira mais customizáveis. (CHAZELLE, 1993), (BERG et al., 2008). Um exemplo de volume *Convex Hull* é apresentado na Figura 12 D.

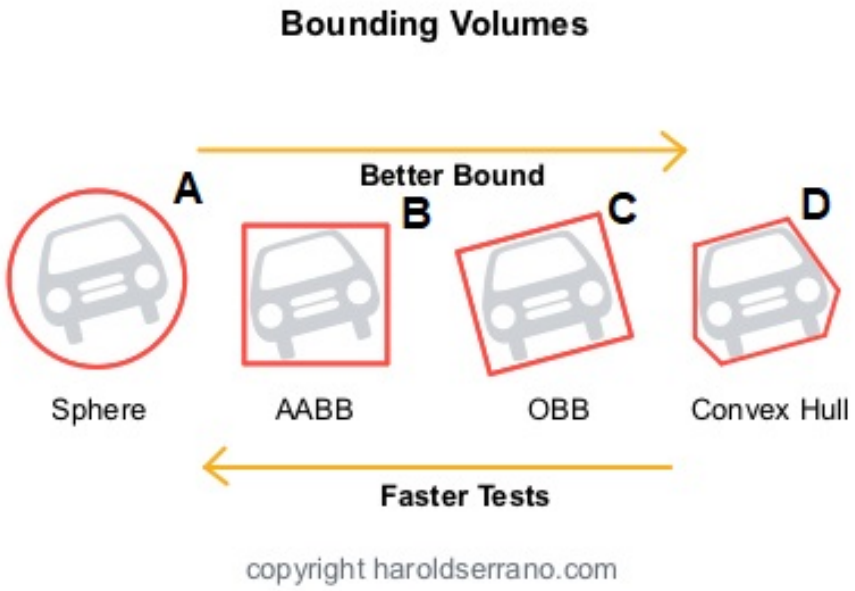

Figura 12: Exemplo de colisão OBB e outros.
Fonte: (SERRANO, 2016)

2.3.5 OTIMIZAÇÃO DE COLISÕES NA *UNITY*

Há uma série de otimizações que podem ser realizadas para obter melhor performance nas colisões, algumas dessas otimizações são apresentadas por Armstrong (2017). Um exemplo de otimização que pode ser realizada se a performance está prejudicada é de filtrar as colisões apenas com os tipos necessários que cada uma deve se conectar. O site também fala da importância de ter o planejamento do projeto antes de começar a execução para evitar possíveis problemas durante o desenvolvimento.

3 PROPOSTA

Este trabalho de conclusão de curso se especifica 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 foi de 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.

Foi 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 3) (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 3: Tabela de Engines mais utilizadas no Brasil
(FLEURY et al., 2014)

Foram realizadas comparações dos métodos de colisão oferecidos gratuitamente pela *game engine* Unity. Essas, foram divididas em 2 tipos: *Pre-runtime* e *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 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 foram feitas usando a interface de duas dimensões da Unity para obter resultados mais simples com esse estudo. Os tipos de *colliders* que tiveram mais ênfase nas comparações foram *Box, Circle, Polygon e Edge*, pelo fato de serem mais utilizados.

As medições incluem taxa de quadros por segundo (FPS), taxa de consumo do processador, taxa de consumo de memória total durante a execução, coleta de lixo de memória, tempo total em milissegundos.

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

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

- Processador: Intel Core i5 4210U @2.4GHz
- Placa de Vídeo: Intel HD Graphics 4400
- Memória RAM: 8GB
- Disco Interno: SSD 120GB
- Disco Externo: HDD 1TB

- Sistema Operacional: Windows 10 Pro

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

3.4 CRONOGRAMA

Na tabela 4 é apresentado um cronograma com as datas estimadas 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 4: Cronograma de progresso estimado do trabalho

4 METODOLOGIAS

Neste capítulo é explicado como as comparações foram feitas e o que foi retirado delas para a abordagem de resultados.

Uma pesquisa de campo foi realizada para saber mais dos tipos de colisões utilizadas através de um formulário que foi divulgado em grupos de comunidades de desenvolvedores de jogos de várias categorias (desenvolvedor independente, estudante, etc.).

Durante os testes foi usada a ferramenta *Profiler* da *Unity*. Essa ferramenta permite analisar em tempo real vários detalhes da performance do jogo em execução, como tempo em milissegundos, *scripts* chamados, memória RAM utilizada, etc.

Para a obtenção dessas comparações foi usado um *script* que grava os resultados em formato *JSON*, criado por PALMA(2017). Em seguida foram extraídos do arquivo apenas os dados que serão usados para essa análise comparativa, que serão detalhados mais a seguir. Após isso os arquivos foram convertidos para valores separados por vírgulas (*CSVs*), para que pudessem ser lidos e manipulados por softwares de planilha.

Planilhas foram usadas para armazenar os resultados das pesquisas e dos testes onde podem ser exibidos em gráficos comparativos para melhor visualização e compreensão dos dados.

4.1 PESQUISA DE CAMPO

A pesquisa de campo foi criada usando o sistema de formulários da *Google* (*Google Forms*), em que é possível criar e visualizar os resultados em tempo real.

O formulário foi feito tanto em português, como em inglês e foi divulgado em grupos de comunidade de desenvolvedores.

O formulário é separado em quatro setores:

- Quem é a pessoa respondendo

Em que situação a pessoa se encontra na área de desenvolvimento de jogos. Se é um desenvolvedor independente, desenvolvedor de grande empresa (AAA), desenvolvedor por passatempo (*hobby*) ou estudante. Isso ajuda a ter uma visão de quem usa e como usa tais tecnologias.

- Perguntas generalizadas sobre as cenas produzidas

  Uma cena ou nível em um jogo é onde a interação entre o jogador, o cenário e seus eventos acontece. Nela podem ter vários tipos de objetos de jogo, com diferentes propriedades, mas o foco nesse caso é em relação às colisões usadas nelas.

  As perguntas nessa seção consistem em saber qual o tipo de colisão predominante na maioria das cenas que produzem, que podem ser *Static Collider*, *Rigidbody Collider* e *Kinematic Rigiddbody Collider*. E qual a média de objetos com a propriedade *trigger* ligada usados em cada cena, que pode ter múltiplos valores atribuídos.

- Perguntas relacionadas à colisões 2D

  Quais os tipos de colisões mais utilizados em um cenário de jogo 2D, bem como as mensagens mais utilizadas para a manipulação de eventos e a média total de objetos com qualquer tipo de colisão em uma cena.

  Os tipos de colisões apresentadas no formulário são as mesmas apresentadas na subseção 2.2.1. E as mensagens apresentadas são as mesmas já explicadas na subseção 2.2.2.

- Perguntas relacionadas à colisões 3D

  Segue exatamente o mesmo padrão do item anterior sobre colisões 2D, apenas alteradas para 3D. Feitas para ter uma visão á respeito do outro setor da *Unity*.

Enquanto o formulário se encontra aberto para ser respondido, os resultados podem ser acompanhados em tempo real. O que permite ter uma visão desde o início dos resultados e como eles vão progredindo com o passar do tempo.

4.2 *PROFILER* DA *UNITY*

Profiler é uma ferramenta que permite a análise em tempo real dos dados e detalhes de uma cena durante sua execução.

Os dados coletados são divididos em vários setores, no entanto, apenas quatro atributos em uso de CPU são o foco deste projeto, que são porcentagem de uso total, chamadas,

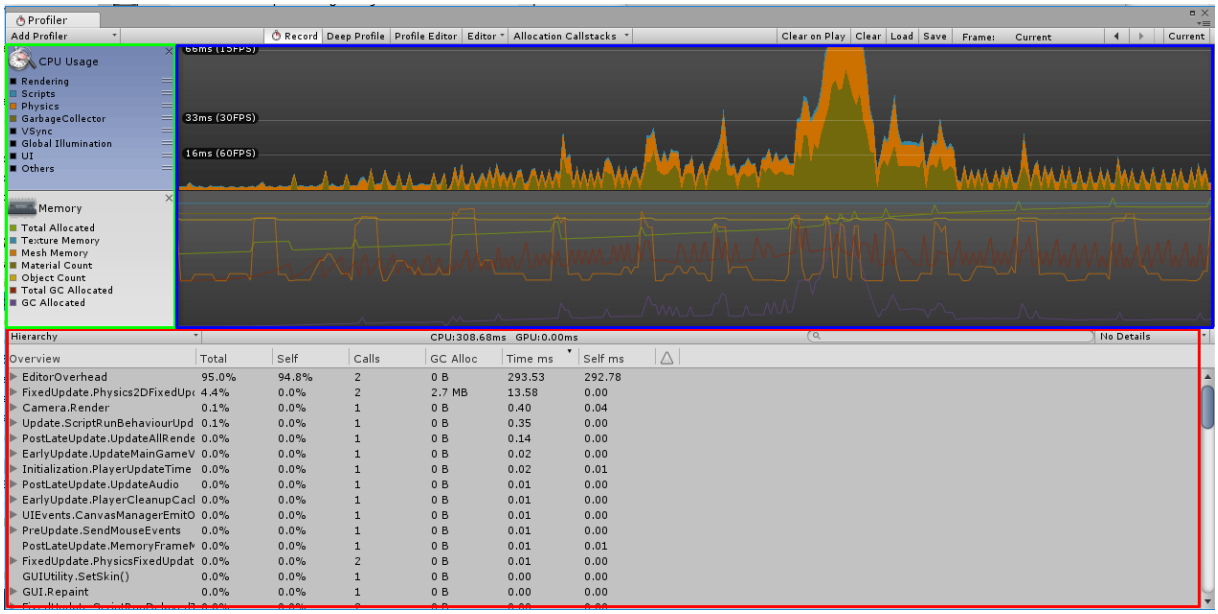

Figura 13: *Profiler* da *Unity*. Em verde, estão as categorias dos dados que são medidos. Em azul, o gráfico porcentual do uso das categorias. Em vermelho, as funções chamadas em dado quadro e seus respectivos dados de consumo.

Fonte: (Unity Technologies, 2017b)

memória alocada pelo *Garbage Collector* e tempo total em milissegundos(ms). O *Profiler* permite também que os dados sejam coletados por métodos. Serão usados como coleta de dados os atributos já mencionados: *Physics2D.Simulate*, as mensagens *OnCollision2D* e *OnTrigger2D*.

Physics2D.Simulate é o método responsável por toda a simulação de física presente na cena, qualquer objeto estático, que tenha massa, corpo rígido ou *trigger* é simulado por ele.

As mensagens de *OnCollision2D* analisadas são as mesmas apresentadas na subseção 2.2.2. No mesmo padrão seguem as mensagens de *OnTrigger2D* na subseção 2.2.3.

- Porcentagem do uso total de CPU: Porcentagem do uso total de CPU de um método em. determinado quadro.
- Chamadas (*Calls*): O número de vezes que o método foi chamado em determinado quadro.
- Memória alocada pelo *Garbage Collector* (*GCMemory*): A quantidade de memória alocada em *bytes* no *Garbage Collector* para a execução do método em determinado quadro.
- Tempo total em milissegundos (ms): A quantidade de tempo em milissegundos requerida para executar o método em determinado quadro.

4.3 OBTENÇÃO E FILTRAGEM DE DADOS

Por padrão, o *Profiler* da *Unity*, não é capaz de salvar os dados coletados para visualização, comparação e manipulação em outras plataformas. No entanto, é possível fazer isso graças a um *script* criado por Palma (2017). Ele permite exportar os dados de uso de CPU em formato *JSON*, o que permite extrair e comparar apenas os dados que serão usados. Sua interface é mostrada na Figura 14.

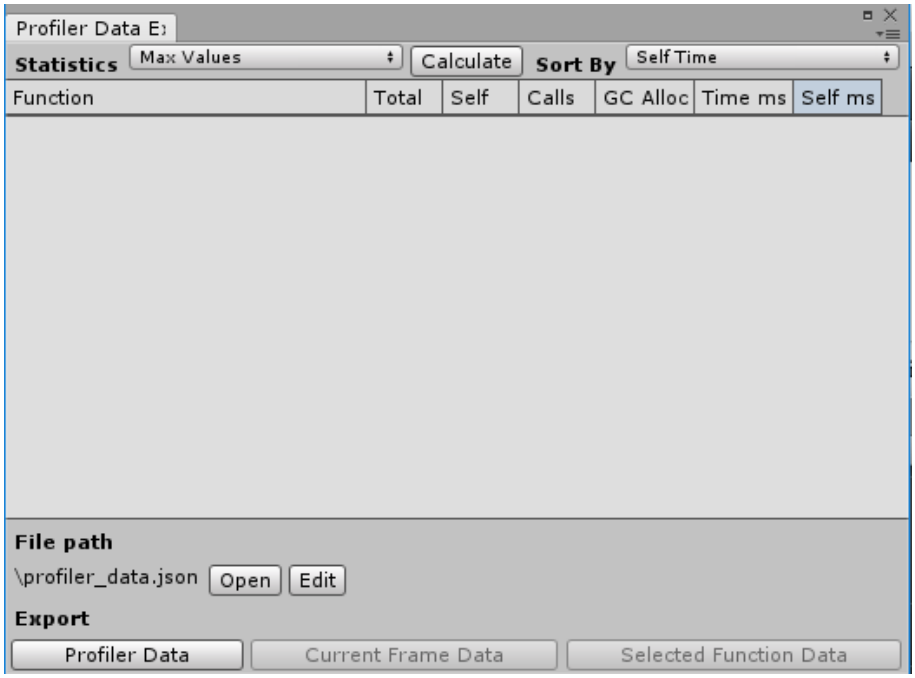

Figura 14: *Profiler Data Exporter*.
Fonte: (PALMA, 2017)

Para extrair esses dados, foi usado um *script* em *Javascript* que os atribui para o formato certo para que possam ser convertidos em valores separados por vírgulas por meio do site *http://convertcsv.com/json-to-csv.htm* e depois lidos em um *software* de planilhas que permita a leitura desse formato de valor.

4.4 CENÁRIOS DE COMPARAÇÃO

Cada cenário foi feito usando um dos quatro *colliders* que foram mencionados antes (subseção 3.3), cada um em *Pre-Runtime* e *Runtime*.

Para obter resultados mais relevantes, cada cena em *Pre-Runtime* foi renderizada com um, dez, cinquenta, cem, quinhentos e mil objetos inseridos na cena antes de sua execução.

Já em cenas em que os objetos seriam invocados, o *script* gerava um, dez, cinquenta, cem, quinhentos e mil objetos por segundo durante a execução do programa.

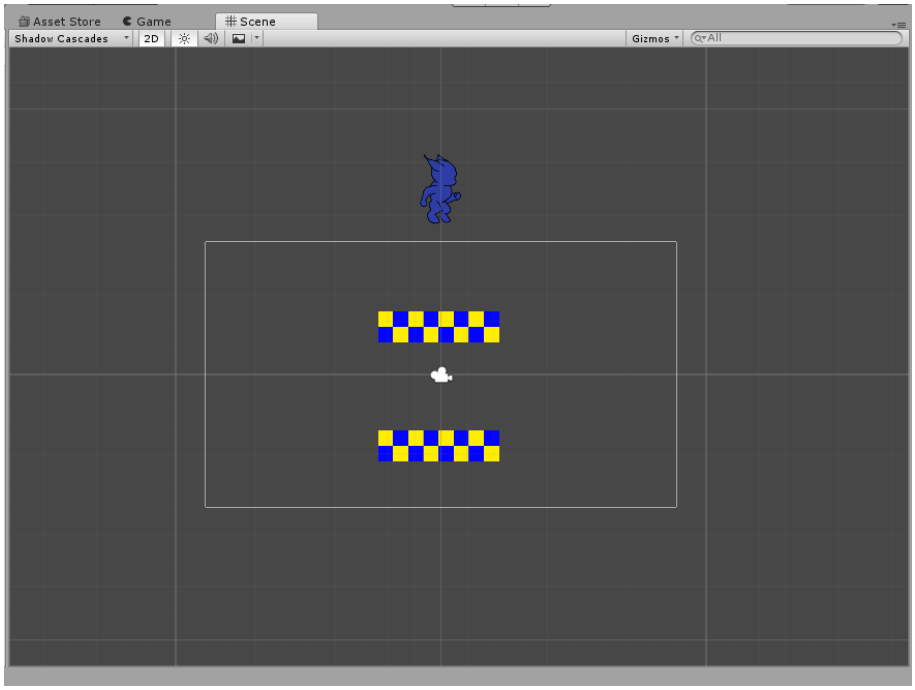

Figura 15: Exemplo de cenário de comparação.

O cenário *Pre-Runtime* consiste de três elementos que são exibidos na Figura 15, que é detalhada logo a seguir.

- O objeto com um tipo de *collider* e *rigid body* atribuídos a ele, sem ser do tipo *kinematic*. Esse objeto é representado pelo personagem em azul.
- Um objeto que permanece parado, com *box collider* em todas as comparações, porém, não é atribuído *rigid body* a ele, e sim *trigger*, ou seja, objetos podem atravessá-lo, mas a detecção de colisão ainda ocorre. Este é representado pela plataforma que se encontra logo abaixo do personagem.
- A última plataforma em baixo, é um objeto estático, porém com um *box collider*, ou seja, os objetos não podem atravessá-lo.

Conforme a quantidade de objetos do tipo do personagem aumentar (parâmetro de medição de número de objetos da cena), os próprios são posicionados aleatoriamente na cena, e as plataformas são alongadas para ter uma área de contato maior para os objetos dinâmicos.

O cenário *Runtime* consiste dos mesmos elementos que o cenário *Pre-Runtime*, porém no lugar do personagem já inserido, há um invocador que se mover na cena e invocar o objeto do personagem dentro de um período de tempo.

5 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados mais relevantes das comparações. É categorizado por tipo de *collider* e em seguida por tipo de cenário (*Pre-Runtime* e *Runtime*).

A maioria dos resultados com poucos objetos no cenário, ou seja, com um e dez objetos ou objetos gerados por segundo, não apresentaram grandes diferenças entre si em relação a *performance* que pudessem intervir na jogabilidade. Em todas as comparações, os cálculos da física dos objetos foram os que mais consumiram recursos, que foram aumentando conforme o aumento do número de objetos na cena dependendo do tipo de *collider* usado. Mais detalhes podem ser encontrados no apêndice.

A pesquisa de campo foi realizada com vinte e sete pessoas até o dia dezenove de outro de 2019 (19/10/2019), sendo que dezessete delas responderam através do formulário em português e as dez restantes, no formulário em inglês

5.1 PESQUISA DE CAMPO

De acordo com a pesquisa de campo realizada, cerca de 70% do uso das colisões são de *Box* e *Circle Collider* para suas cenas em 2D como mostra a Figura 16.

Em relação as mensagens usadas, há uma distribuição equilibrada quando o uso é de início e final de contato com a colisão ou trigger, como mostra a Figura 17. Já enquanto o contato ainda está ocorrendo é pouco utilizado.

O número médio de objetos numa cena podem ser encontrados na Figura 18. Nesse resultado apenas dois participantes não responderam à pergunta.

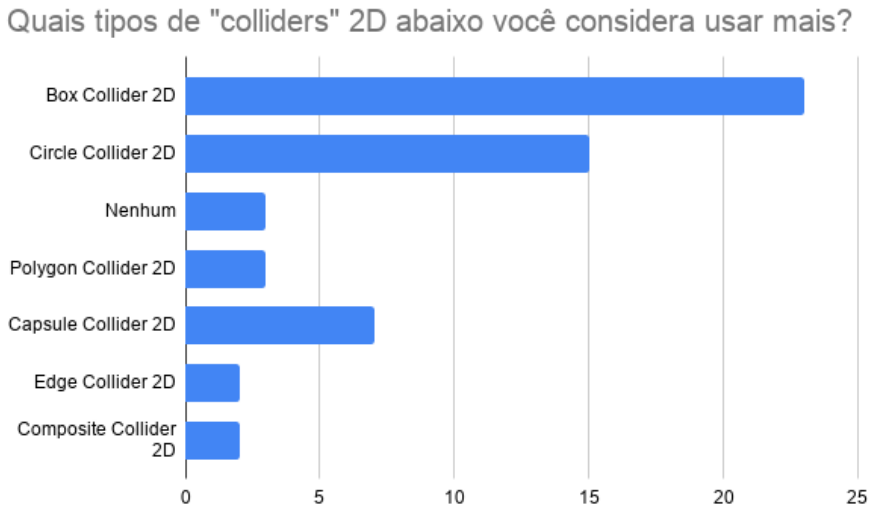

Figura 16: Frequência de uso de *Colliders* 2D.

Figura 17: Frequência de uso de mensagens de *Colliders* 2D.

5.2 RESULTADOS COM *BOX COLLIDER*

5.2.1 *BOX COLLIDER - PRE-RUNTIME*

Os resultados apresentaram uma progressão linear conforme o número de objetos foi aumentando com pequenas variações. Nos três primeiros testes com um, dez e cinquenta objetos, os resultados se mostram com performance estável, sem queda bruscas de quadros e sem uso intensivo de CPU e de memória RAM.

A partir de cem objetos, apesar de ainda não haver uso intenso de memória e CPU que afetem a jogabilidade, é possível perceber picos de *Garbage Collection*. Esses picos começam

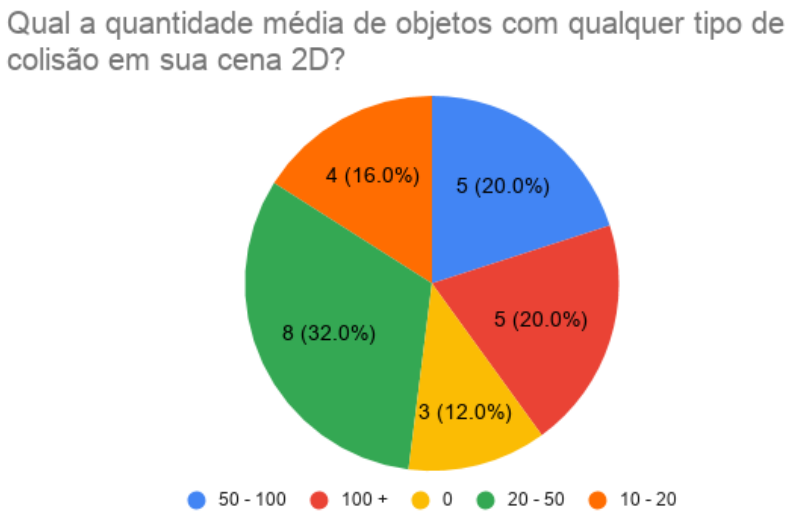

Figura 18: Média de objetos come *Colliders* 2D presentes em uma cena.

a afetar a performance a partir dos quinhentos objetos, fazendo com que o uso total de CPU para calcular os quadros chega a 16 ms como mostrado na Figura 19, onde antes era de 0.25 ms a 1 ms, mostrado na Figura 20.

Figura 19: Resultado de performance de dez objetos com *box collider* em *Pre-Runtime*.

O resultado com mil objetos foi o mais instável, apresentando um aumento de consumo apenas certo tempo depois do início da execução, chegando a fazer o tempo de consumo total de CPU chegar a 66 ms e 0.65GB de uso total de memória RAM (em comparação com a margem de 200MB de RAM nos resultados anteriores) como mostrado na Figura 21.

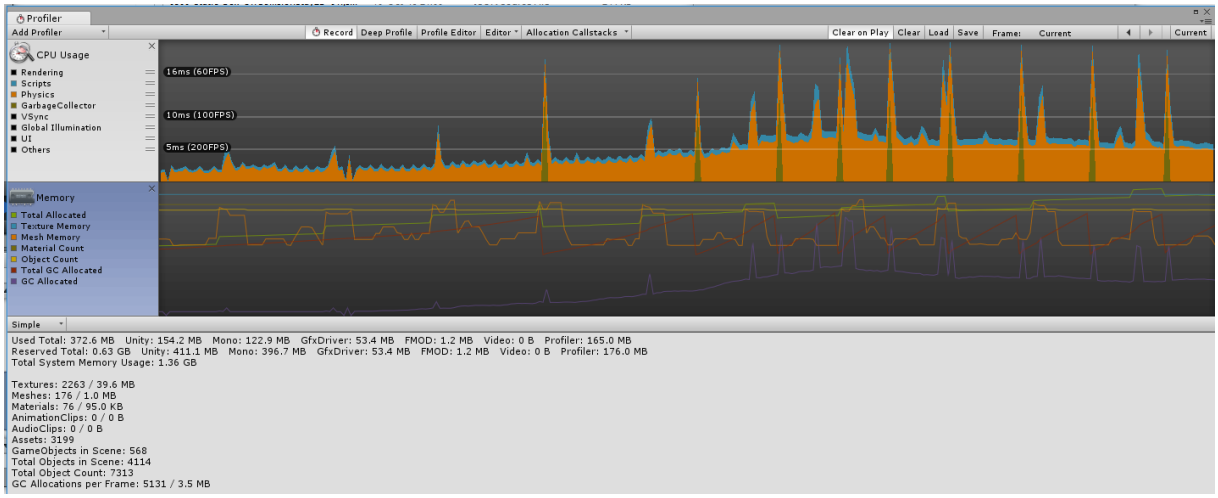

Figura 20: Resultado de performance de quinhentos objetos com *box collider* em *Pre-Runtime*.

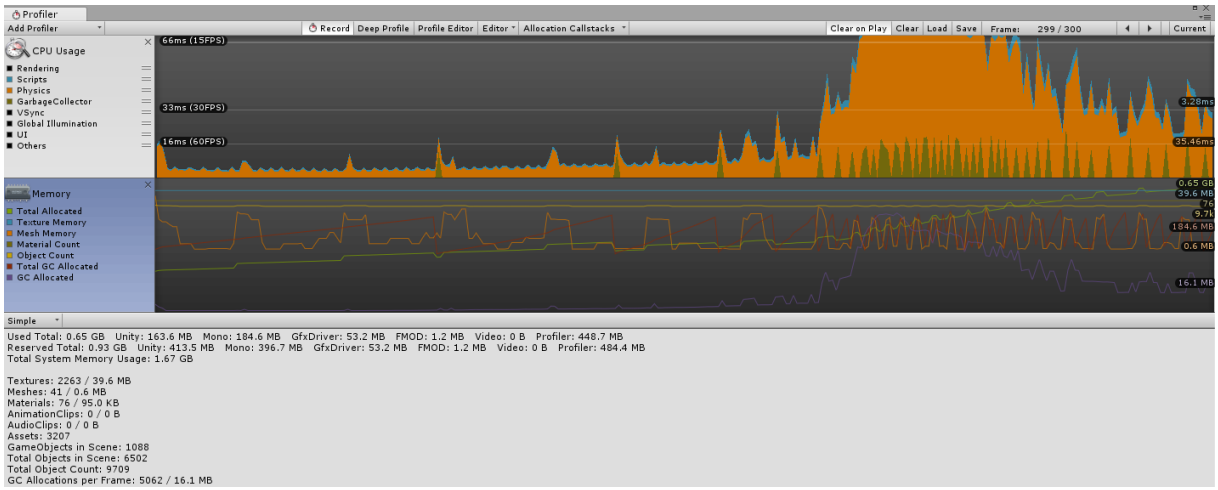

Figura 21: Resultado de performance de mil objetos com *box collider* em *Pre-Runtime*.

5.2.2 *BOX COLLIDER - RUNTIME*

Nos três primeiros testes com um, dez e cinquenta objetos por segundo, os resultados se mostram com performance estável, porém com picos de *Garbage Collection* contantes conforme os objetos eram invocados pelo script de invocação. O consumo total de memória RAM mostra ser menor do que os resultados em *Pre-Runtime*, como mostra a Figura 22.

A partir de cem objetos por segundo, há uma queda brusca de performance. Foi gerada uma alta quantidade de *Garbage Collection*, taxa de uso de CPU passou o limite de 66 ms e 1.64GB de RAM total, como pode ser mostrado na Figura 23, o que afeta bastante a performance, a visualização e estabilidade do jogo. As demais comparações seguintes dessa subseção apresentaram resultados semelhantes, com o aumento de consumo após pouco tempo de execução.

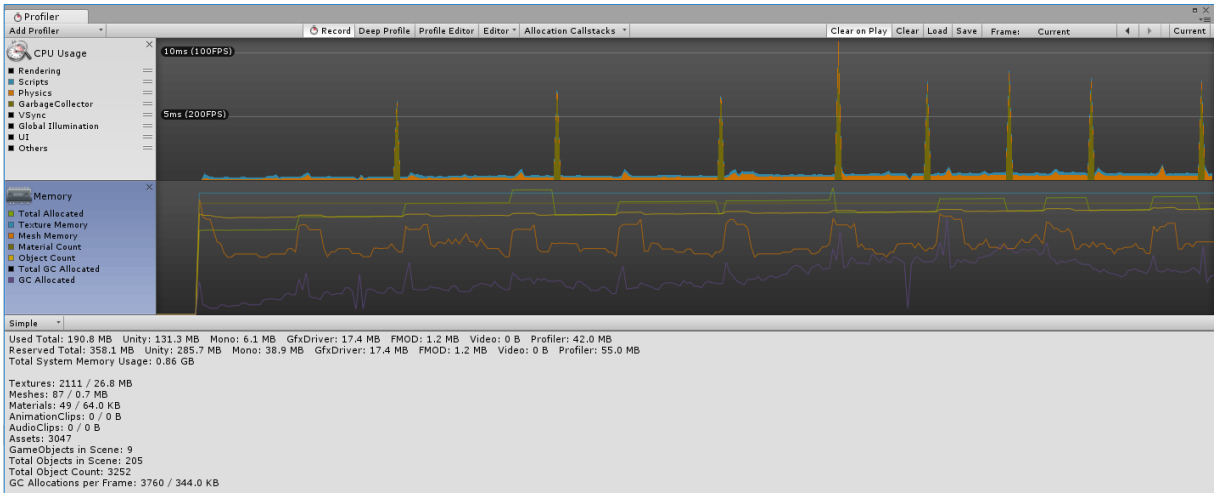

Figura 22: Resultado de performance de dez objetos com *box collider* em *Runtime*.

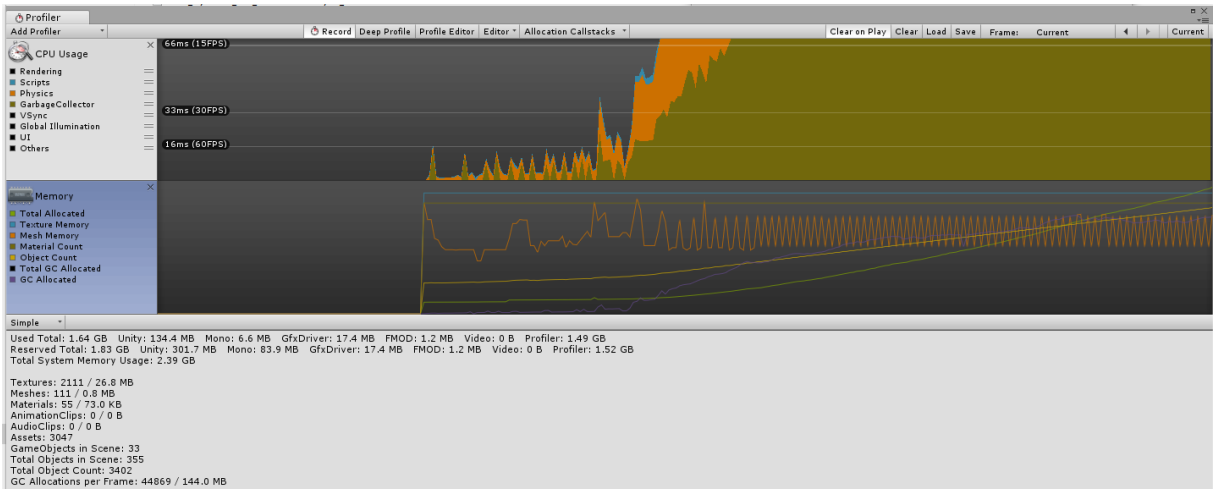

Figura 23: Resultado de performance de quinhentos objetos com *box collider* em *Runtime*.

5.3 RESULTADOS COM *CIRCLE COLLIDER*

5.3.1 *CIRCLE COLLIDER - PRE-RUNTIME*

Assim como nos resultados de *box collider*, as três primeiras comparações se mostraram estáveis nas suas medidas, com tempo de uso de CPU entre 0.25 ms e 1 ms e consumo total de memória RAM em cerca de 250 MB como mostrado na Figura 24.

A partir de 100 objetos em diante, o resultado mostra o início de picos de *Garbage Collection*, que aumentam drasticamente de acordo com o número de objetos na cena. No entanto, com mil objetos, o resultado se mostra estável no início, porém com o aumento de colisões entre os objetos, a quantidade de *Garbage Collection* aumenta, como mostrado na Figura 25, deixando a performance relativamente baixa. Ela mostra-se melhor do que a mesma situação com *box collider*. Logo em seguida, o *Garbage Collection* diminui, e a performance

Figura 24: Resultado de performance de cinquenta objetos com *circle collider* em *Pre-Runtime*.

se estabiliza. O *circle collider* apresentou mais *Garbage Collection* que o *box collider*.

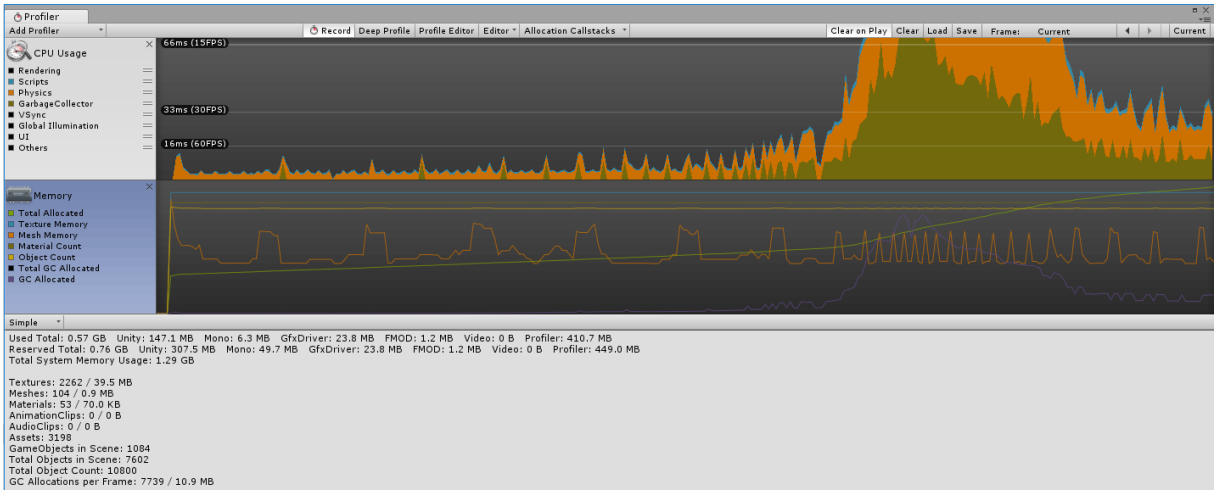

Figura 25: Resultado de performance de mil objetos com *circle collider* em *Pre-Runtime*.

5.3.2 *CIRCLE COLLIDER - RUNTIME*

Segue os mesmos padrões de resultado de *box collider*. porém, os picos de *Garbage Collection* só se tornam mais presentes a partir de 50 objetos por segundo na cena. E até esse ponto, o resultado ainda se mostra estável, com consumo de CPU variando de 1 ms a 5 ms e memória RAM total em cerca de 240 MB, como mostrado na Figura 26.

A partir de cem objetos por segundo, os resultados se assemelham aos cenários com *box collider*, com exceção do consumo total de memória RAM, que teve um aumento significativo de 240 MB com cinquenta objetos, para 1.5 GB, como mostrado na Figura 27. No entanto, os resultados seguintes mostram consumo de total memória RAM semelhante ao próprio,

Figura 26: Resultado de performance de cinquenta objetos com *circle collider* em *Runtime*.

mesmo com o aumento do número de objetos.

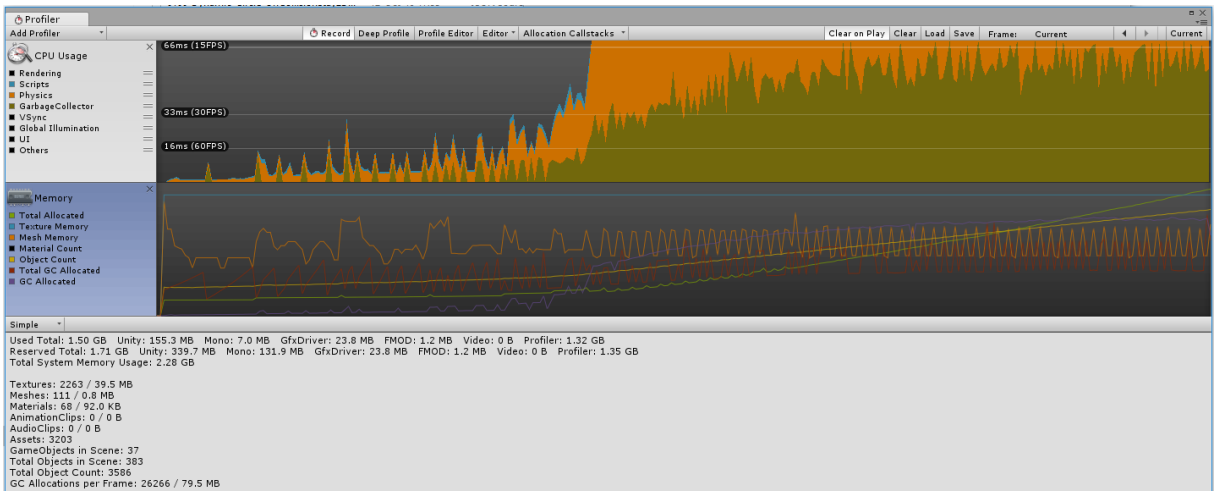

Figura 27: Resultado de performance de cem objetos com *circle collider* em *Runtime*.

5.4 RESULTADOS COM *EDGE COLLIDER*

5.4.1 *EDGE COLLIDER - PRE-RUNTIME*

Os resultados com esse tipo de *collider* mostraram-se estáveis em todos os cenários, tendo baixo consumo de processamento, que com mil objetos chegou a 1 ms, e uso total de memória RAM com pouca variação em comparação com os outros *colliders*, variando entre 250 MB e 350 MB para até 500 objetos, e cerca de 450 MB para mil objetos como demonstrado pela Figura 28.

No entanto foi o que demonstrou a menor precisão em como as colisões ocorrem. Os objetos, mesmo com *rigid body*, se sobrepunham uns entre os outros. Além disso, como esse

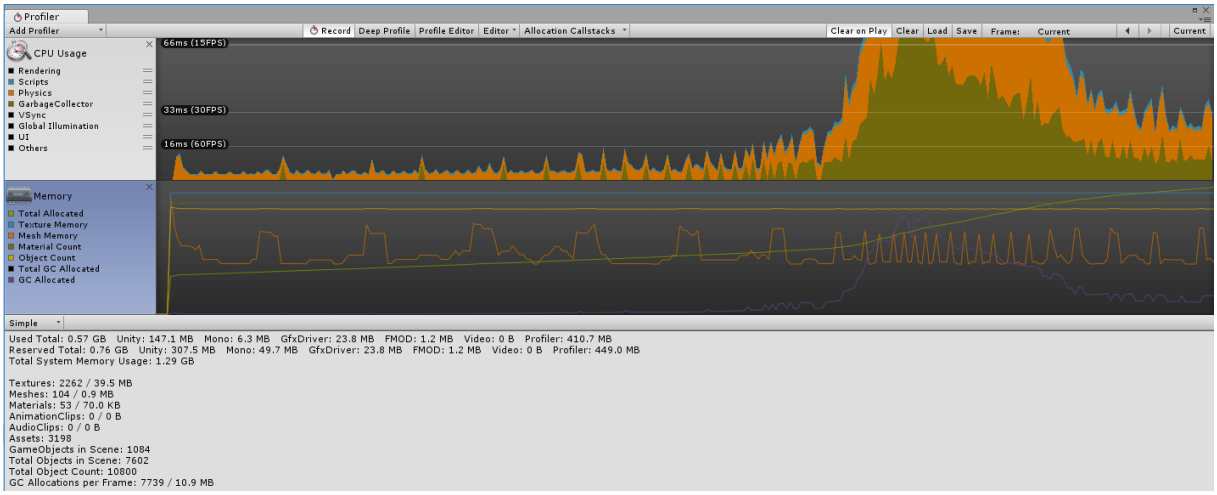

Figura 28: Resultado de performance de mil objetos com *edge collider* em *Pre-Runtime*.

tipo de *collider* consiste de uma linha contínua, não é apropriado para o uso de personagens ou objetos que requerem uma interação mais dinâmica. Esse tipo de *collider* é mais apropriado para terrenos e plataformas que não requerem uma interação além de uma parte da superfície. Outro problema observado pelo uso de muitos objetos com *edge collider* na mesma cena é que alguns deles podem atravessar outros objetos, mesmo estando com *rigidbody*. Esses problemas podem ser vistos por meio da Figura 29

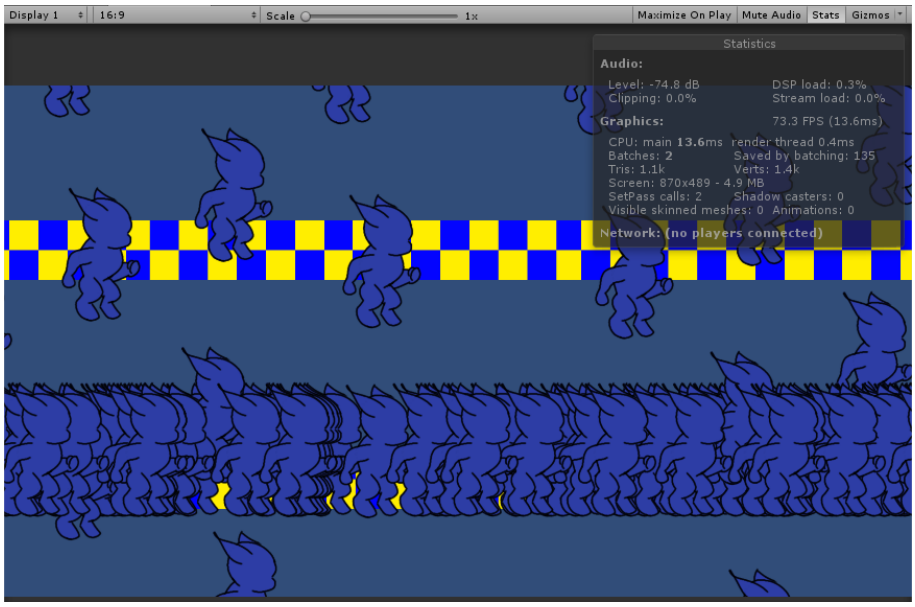

Figura 29: Problemas de colisão causados pelo uso do tipo *edge collider* em *Pre-Runtime*.

5.4.2 *EDGE COLLIDER - RUNTIME*

Os resultados nessa categoria mostraram-se estáveis em seu percurso, tendo mais performance do que os resultados em *Pre-Runtime*. Com mil objetos por segundo, o tempo de uso

de CPU ficou entre 5 ms e 10 ms, e consumo total de memória RAM em cerca de 320 MB, como mostra a Figura 30. Picos de *Garbage Collection* começam a ser mais frequentes a partir de cem objetos por segundo.

Figura 30: Resultado de performance de mil objetos por segundo com *edge collider* em *Runtime*.

5.5 RESULTADO COM *POLYGON COLLIDER*

O *Polygon Collider* é o tipo de *collider* que oferece a melhor precisão de colisão em comparação aos outros. No entanto, é o que requer mais uso de recursos, o que resulta em uma performance baixa se usado com muita frequência.

Devido a isso, os resultados em Runtime a partir de 50 objetos já demonstraram uma queda brusca na performance. Com 100 objetos ou mais não foi possível realizar as medições, pois o programa não respondeu mais ao iniciar a execução.

5.5.1 *POLYGON COLLIDER - PRE-RUNTIME*

Os resultados com um e cem objetos, demonstram boa estabilidade de consumo de recursos, mantendo o tempo de consumo total da CPU entre 0.1 ms e 1 ms, como mostra a Figura 31. O consumo total de memória RAM variou entre 270 MB e 180 MB.

A partir de 50 objetos, houve um consumo maior de CPU, fazendo o tempo total de uso da mesma ser entre 5 ms a 16 ms. No entanto, apesar do consumo total de memória RAM se manteve estável como mostra a Figura 32, a performance começa a cair gradativamente.

Com cem objetos a performance diminui pouco, relativamente ao resultado anterior. O consumo total de memória RAM se mantém, e o tempo de consumo da CPU teve um pico de

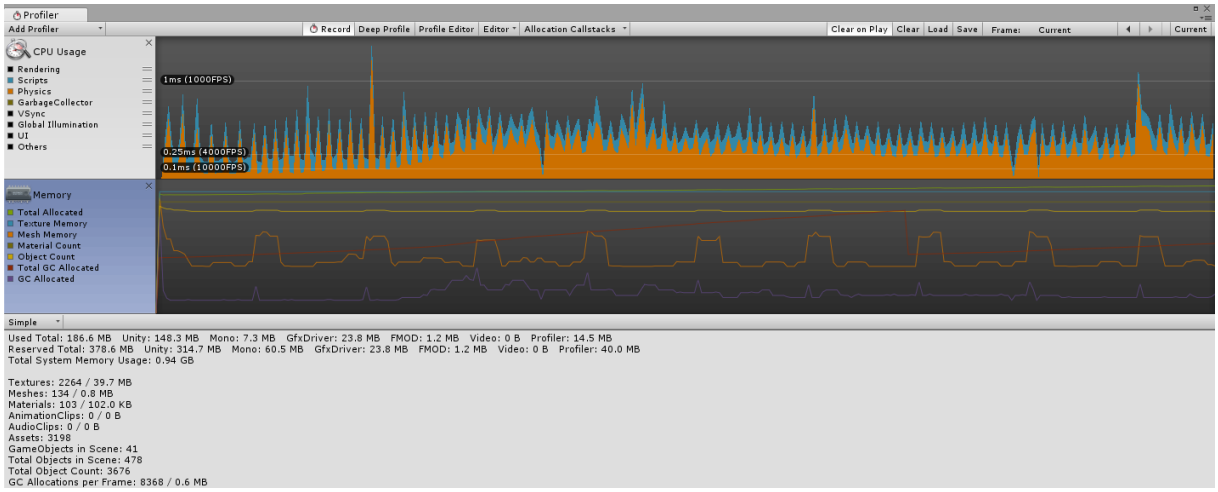

Figura 31: Resultado de performance de dez objetos com *polygon collider* em *Pre-Runtime*.

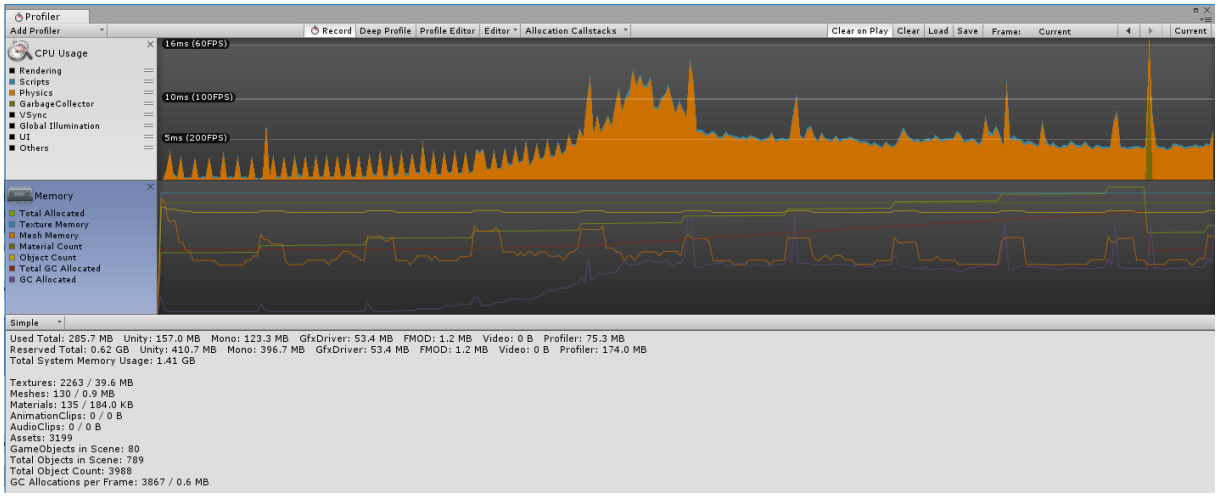

Figura 32: Resultado de performance de cinquenta objetos com *polygon collider* em *Pre-Runtime*.

66 ms, porém, em sua maioria, manteve estável entre 33 ms e 16 ms como mostra a Figura 33.

Figura 33: Resultado de performance de cem objetos com *polygon collider* em *Pre-Runtime*.

Os resultados com quinhentos e mil objetos demonstraram uma queda drástica na performance, além de uso elevado dos recursos, com consumo total de memória RAM sendo de 0.99 GB e 1.88 GB respectivamente, e em ambos o tempo de consumo de CPU passou de 66 ms, como mostra a Figura 34. Com mil objetos, a taxa de quadros por segundo se manteve em 0.6 (1638 ms), como mostra a Figura .

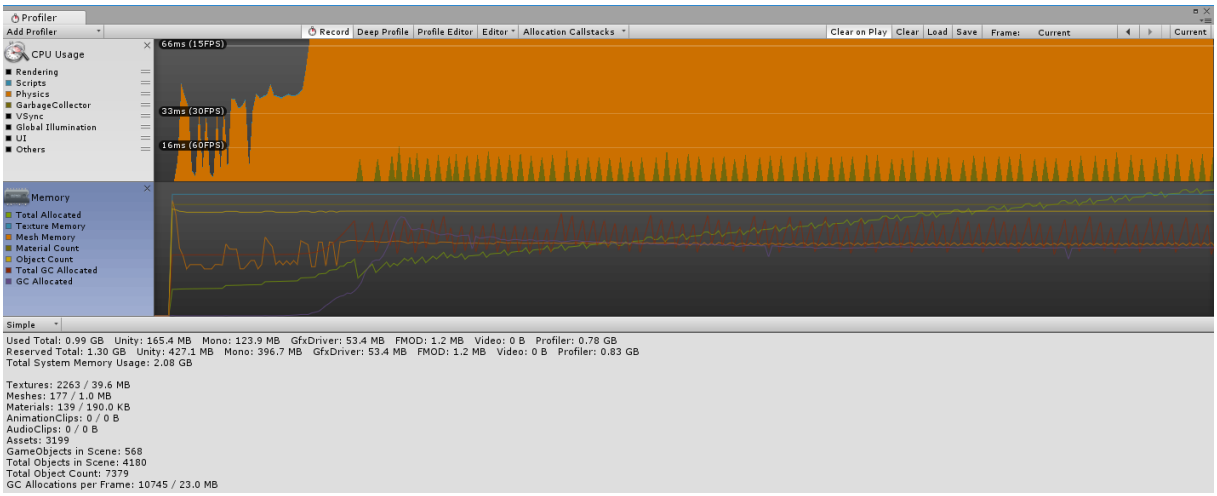

Figura 34: Resultado de performance de quinhentos objetos com *polygon collider* em *Pre-Runtime*.

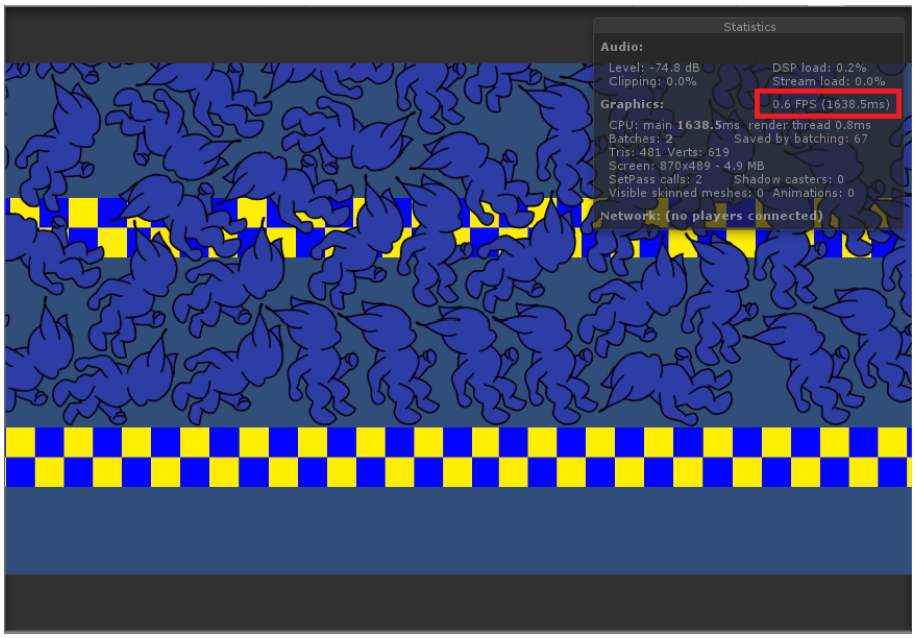

Figura 35: Captura de quadros por segundo média aproximada com *polygon collider* em *Pre-Runtime* e mil objetos.

5.5.2 *POLYGON COLLIDER - RUNTIME*

O resultado com um objeto por segundo foi relativamente estável, apresentando tempo de consumo de CPU com média de 1 ms e consumo total de memória RAM com 230 MB, como

mostra a Figura 36.

Figura 36: Resultado de performance de um objeto com *polygon collider* em *Runtime*.

A partir de 10 objetos por segundo a performance começa a cair conforme o tempo de execução passa e os objetos são inseridos. Inicialmente o tempo de consumo de CPU era de cerca de 16 ms, porém rapidamente passou de 66 ms como mostra a Figura 37. O consumo de memória RAM total foi de cerca de 500 MB.

Figura 37: Resultado de performance de dez objetos com *polygon collider* em *Runtime*.

Com 50 Objetos, a performance é comprometida desde o início da execução, fazendo com que o programa travasse com frequência. O tempo de consumo de CPU que foi possível medir passou de 66 ms e consumo total de memória RAM de 335 MB como mostra a Figura 38 até ser encerrada a execução pelo programa por travar frequentemente. O programa indicou 0.8 quadros por segundo (1209 ms) como mostra a Figura e se manteve assim durante o resto da execução

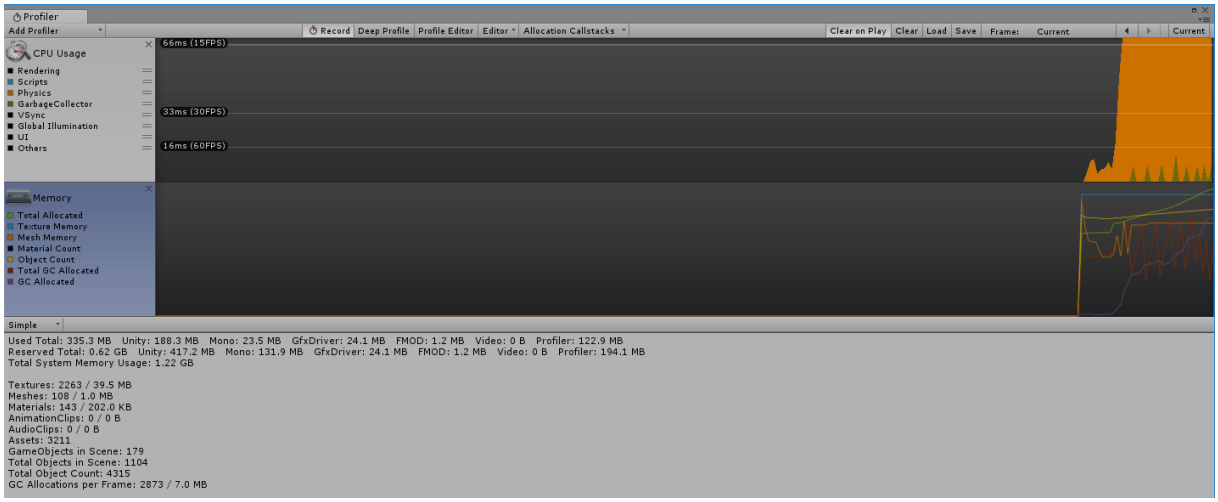

Figura 38: Resultado de performance de cinquenta objetos com *polygon collider* em *Runtime*.

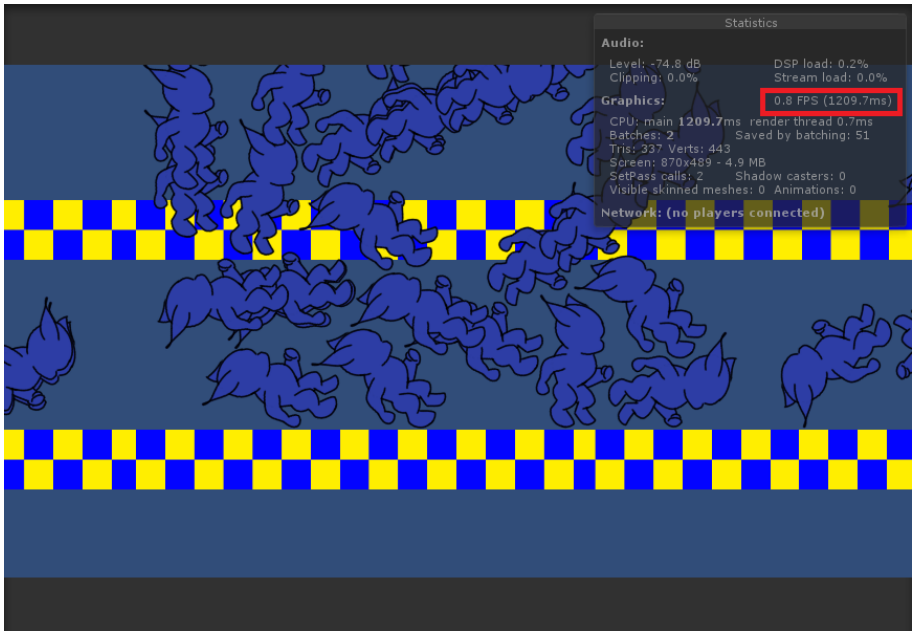

Figura 39: Captura de quadros por segundo média aproximada com *polygon collider* em *Runtime* e cinquenta objetos.

Com cem objetos por segundo, o programa inicia, porém não prossegue com a execução. Por manter-se parado dessa forma por um longo período de tempo, essa comparação foi cancelada e não foi possível obter resultados.

As comparações com quinhentos e mil objetos por segundo apresentaram o mesmo efeito e também foram cancelados.

5.6 VISÃO GERAL DOS RESULTADOS

Como foi possível observar pela análise dos resultados, a maioria das comparações com um alto número de objetos já inseridos na cena ou objetos gerados por segundo, exigiram mais dos recursos da máquina, causando degradação da performance durante a execução. E nos casos do *Polygon collider* com cem objetos por segundo em diante o programa travou e não foi possível obter resultados.

As comparações com poucos objetos não demonstraram grandes diferenças entre si.

O tipo de *collider* que requisitou maior quantidade de recursos na maioria de suas comparações certamente foi o *Polygon Collider*, pelo fato de ser um tipo de colisão que tenta se encaixar melhor com o formato do objeto original. Isso gera uma precisão melhor na colisão, porém em excesso, causa degradação da performance. No entanto, se o não usado com muita frequência e o escopo exigir precisão, é possível usá-lo sem perder muita performance.

Em comparação com os *colliders* mais utilizados (*Box collider 2D* e *Circle Collider 2D*), de acordo com a pesquisa de campo na Figura 16, eles mostram resultados de performance melhores e, muitas vezes, se encaixam no escopo do que é necessário para ser realizado nas colisões de um devido jogo. Se o jogo ou alguns objetos da cena não necessitarem muita precisão da colisão, as formas básicas podem ser o suficiente para o funcionamento e escopo do jogo.

A seguir são mostrados os principais gráficos dos resultados com cinquenta objetos inseridos antes de executar o programa e cinquenta objetos por segundo durante a execução. Cinquenta objetos por ser um número médio mais próximo da quantidade de objetos usados por cena de acordo com a pesquisa de campo na Figura 18.

O atributo física foi removido da maioria dos gráficos por obstruir a visualização dos outros atributos que, na maioria dos casos apresentam valores muito menores que os da física, mais informações dos dados brutos podem ser encontradas no apêndice C. E todos os métodos apresentados nos gráficos são referentes aos métodos da interface 2D da *Unity*, por exemplo: *TriggerStay* = *TriggerStay2D*. Métodos *Collision* foram abreviados, por exemplo: *ColEnter* = *CollisionEnter2D*.

Os resultados com *Edge Collider* não foram incluídos por serem os mais estáveis, como mencionado na seção 5.4.

5.6.1 COMPARAÇÕES EM PRE-RUNTIME

É possível observar nos gráficos de porcentagem de uso total da CPU das Figuras 40, 41 e 42 que os *colliders Box* e *Circle* apresentaram resultados bem semelhantes, com picos chegando a 1.4% do uso total da CPU, e média variando entre 0.02% e 0.05%. O *Polygon collider* apresentou resultados semelhantes também, porém com a diferença nos picos, chegando a 2.5% de uso total da CPU.

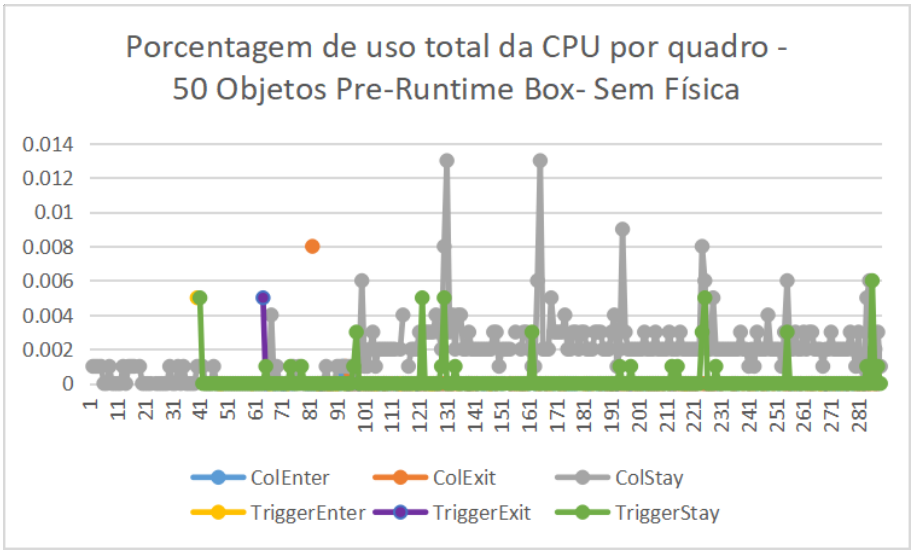

Figura 40: Gráfico de porcentagem de uso da CPU com 50 objetos *Box Collider* em *Pre-Runtime*.

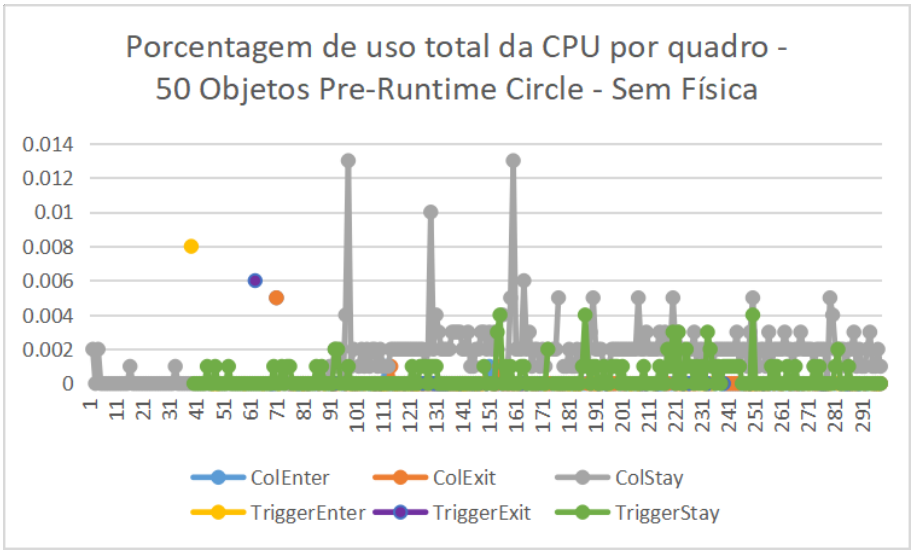

Figura 41: Gráfico de porcentagem de uso da CPU com 50 objetos *Circle Collider* em *Pre-Runtime*.

Em relação ao número de vezes que os métodos que foram chamados durante a execução, são mostrados nos gráficos das Figuras 43, 44 e 45. É possível observar que o método *CollisionStay2D* foi o que mais requisitou chamadas, e o segundo que mais requisitou foi o *TriggerS-*

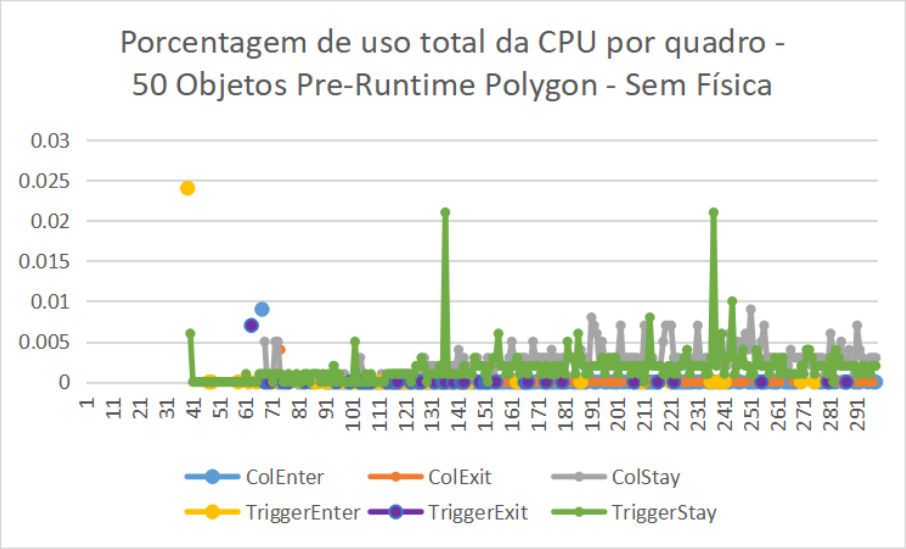

Figura 42: Gráfico de porcentagem de uso da CPU com 50 objetos *Polygon Collider* em *Pre-Runtime*.

tay2D. Também é possível notar que com o *collider Polygon* houve maior número de chamadas durante a execução, que se mantiveram estáveis com o decorrer do tempo.

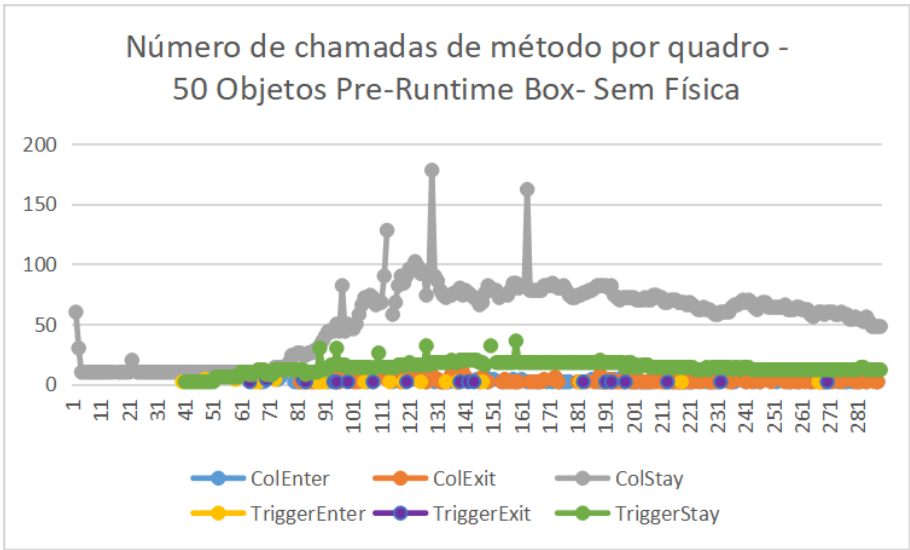

Figura 43: Gráfico de número de vezes que o método foi chamado com 50 objetos *Box Collider* em *Pre-Runtime*.

O tempo em milissegundos exigido para a execução de cada método é exibido nos gráficos das Figuras 46, 47 e 48. E a partir deles é possível observar que novamente os métodos *CollisionStay2D* e *TriggerStay2D* foram os que mais exigiram do recurso. É possível também notar que o *Circle collider* apresentou melhor performance em relação ao *Box collider*. O *Polygon collider* continua sendo o menos performático.

A coleta de lixo (*Garbage Collection*) desses métodos com os *colliders* mencionados

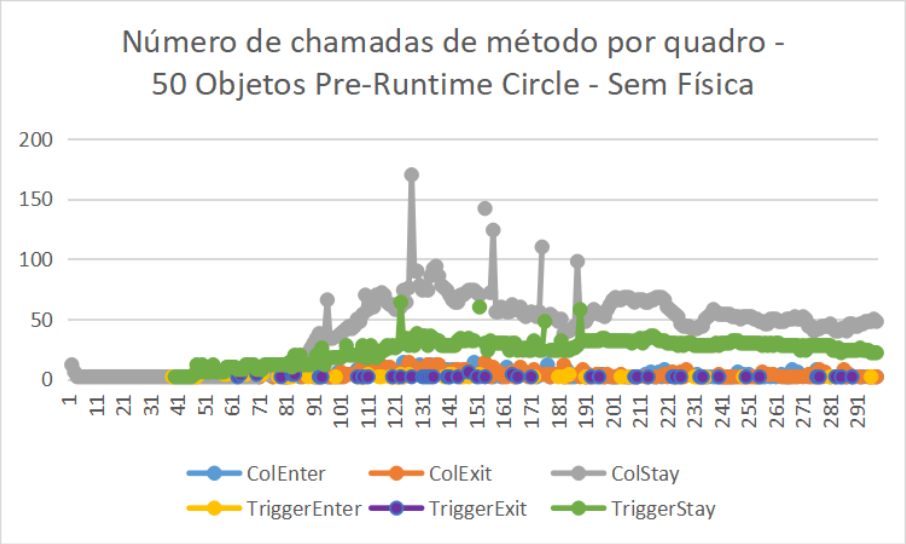

Figura 44: Gráfico de número de vezes que o método foi chamado com 50 objetos *Circle Collider* em *Pre-Runtime*.

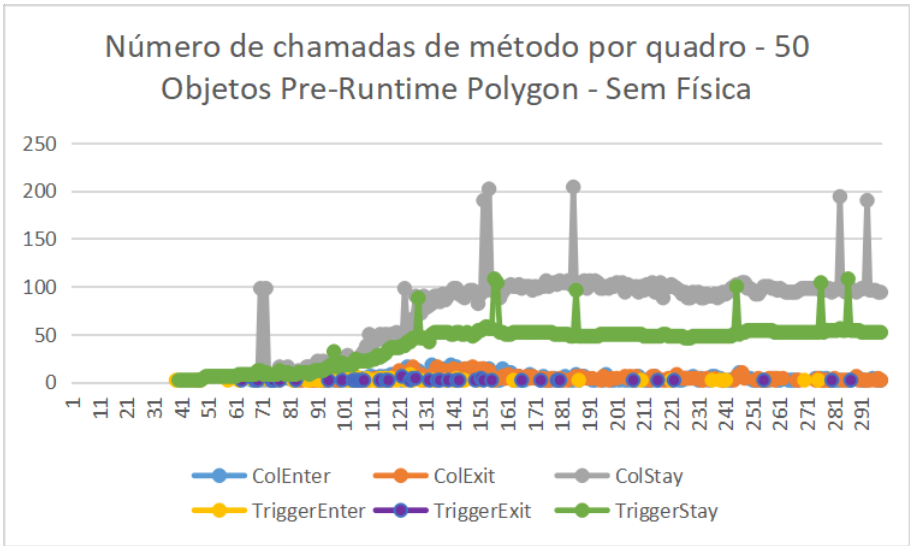

Figura 45: Gráfico de número de vezes que o método foi chamado com 50 objetos *Polygon Collider* em *Pre-Runtime*.

só mostrou variâncias quando o atributo física está presente. Sem ele, os métodos não mostraram alteração de quantidade de coleta de lixo, com exceção do *Polygon collider* que apresentou apenas 32 bytes no quadro 41 com o *TriggerEnter2D* e depois para zero, como mostra o gráfico na Figura 49.

A seguir é exibido nos gráficos das Figuras 50, 51 e 52, o *Garbage Collection* com física. Com isso é possível observar que o *Circle collider* apresentou total estabilidade de 30000 *bytes*, com alguns picos de quase 60000 *bytes*, porém foi o único que já iniciou com *Garbage Collection*. Os *colliders Box* e *Polygon* foram aumentando com o passar do tempo.

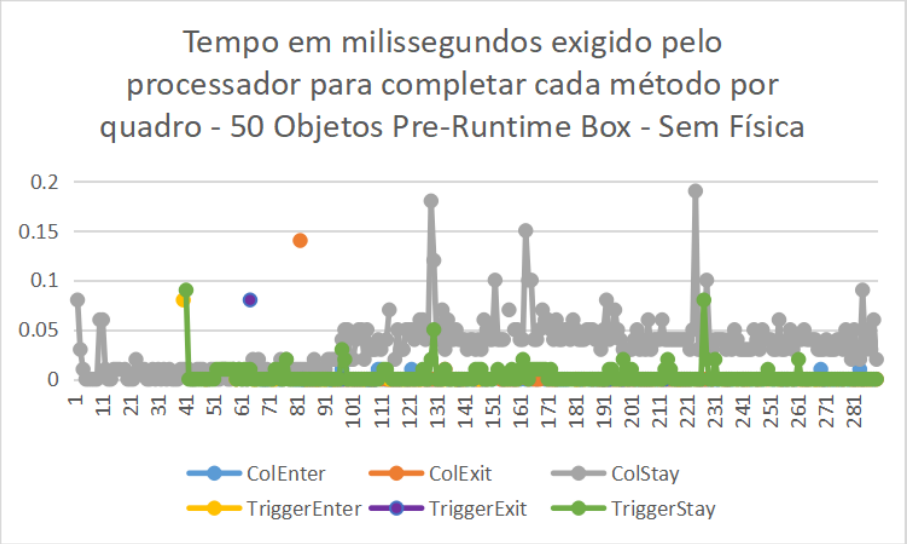

Figura 46: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos *Box Collider* em *Pre-Runtime*.

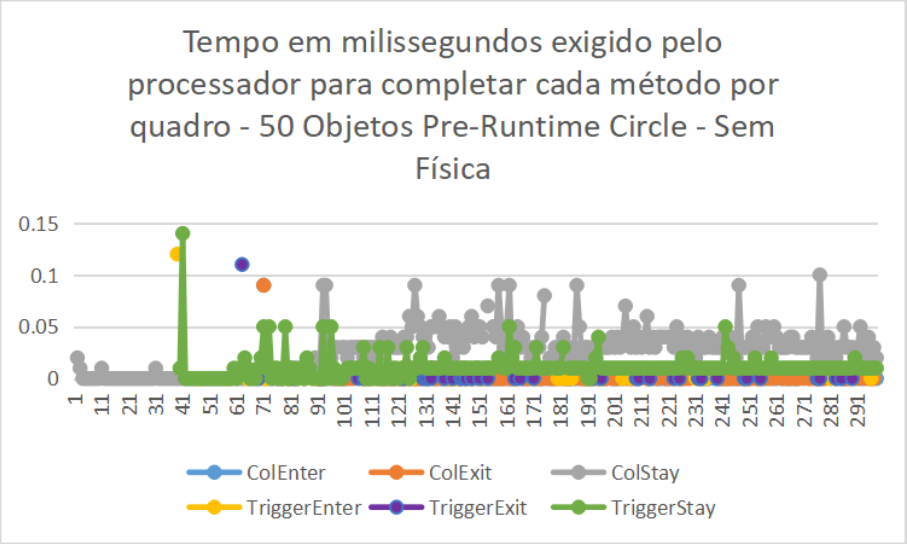

Figura 47: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos *Circle Collider* em *Pre-Runtime*.

5.6.2 COMPARAÇÕES EM *RUNTIME*

Nas comparações a seguir foi adicionado o atributo *Spawn*, referente ao método que invoca os objetos em tempo de execução, para ser medido.

Com os gráficos de porcentagem de uso total da CPU mostrados nas Figuras 53, 54 e 55, é possível notar que o *Circle collider* se mostrou mais estável e mais performático que o *Box collider*. O *Polygon collider* apresenta pouca amostra de dados por ser o resultado que começou a apresentar travamentos no programa, o que justifica o limite baixo.

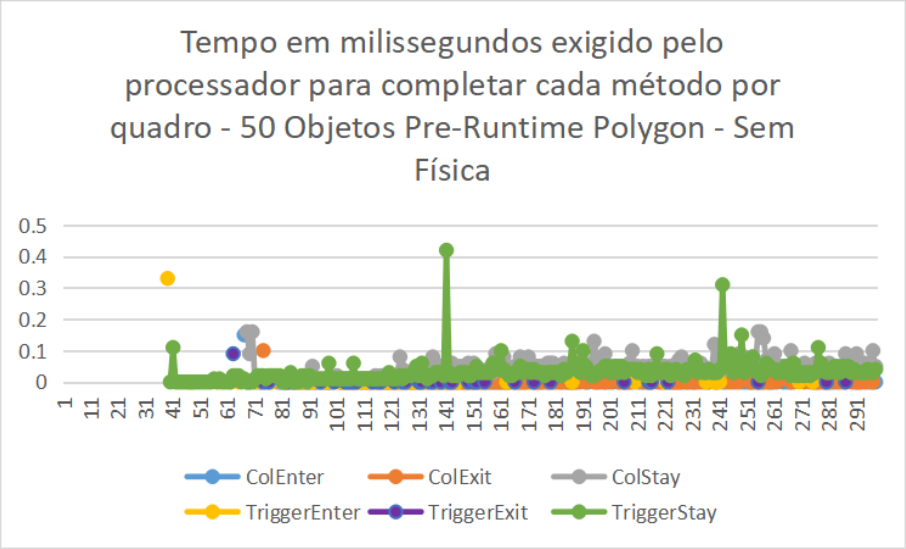

Figura 48: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos *Polygon Collider* em *Pre-Runtime*.

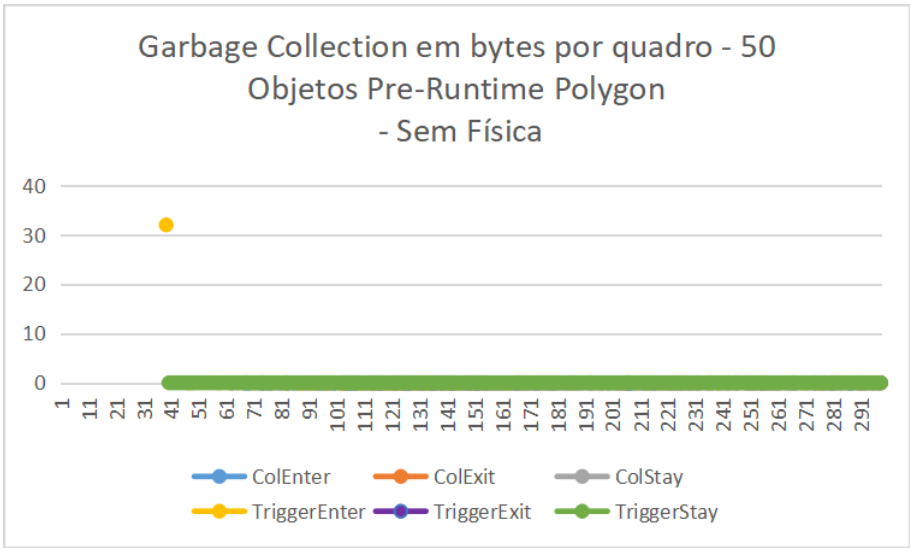

Figura 49: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos *Polygon Collider* em *Pre-Runtime*.

Em relação ao número de vezes que os métodos que foram chamados durante a execução, são mostrados nos gráficos das Figuras 56, 57 e 58. É possível notar que o *Circle collider* demonstrou ser bem mais performático e estável nesse quesito em relação ao *Box collider*. O *Polygon collider* mostrou um aumento conforme o passar do tempo, ultrapassando o total do *Circle* e *Box collider* em menos tempo que os mesmos.

O tempo em milissegundos exigido para a execução de cada método é exibido nos gráficos das Figuras 59, 60 e 61. E a partir deles é possível observar que o *Circle* e *Box collider* mostraram-se estáveis, com o método *Spawn* exigindo mais tempo em ambos, e o *CollisionStay2D* para o *Box collider*.

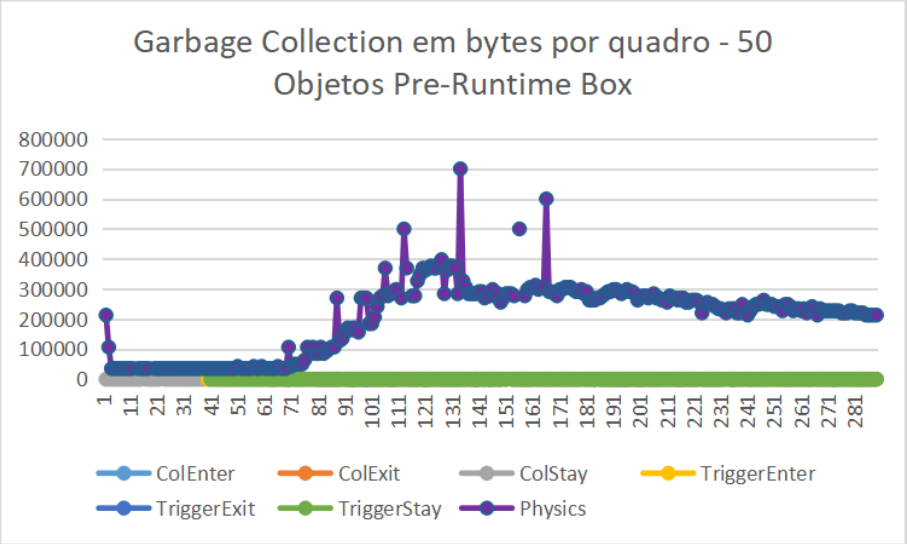

Figura 50: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos *Box Collider* em *Pre-Runtime* com física.

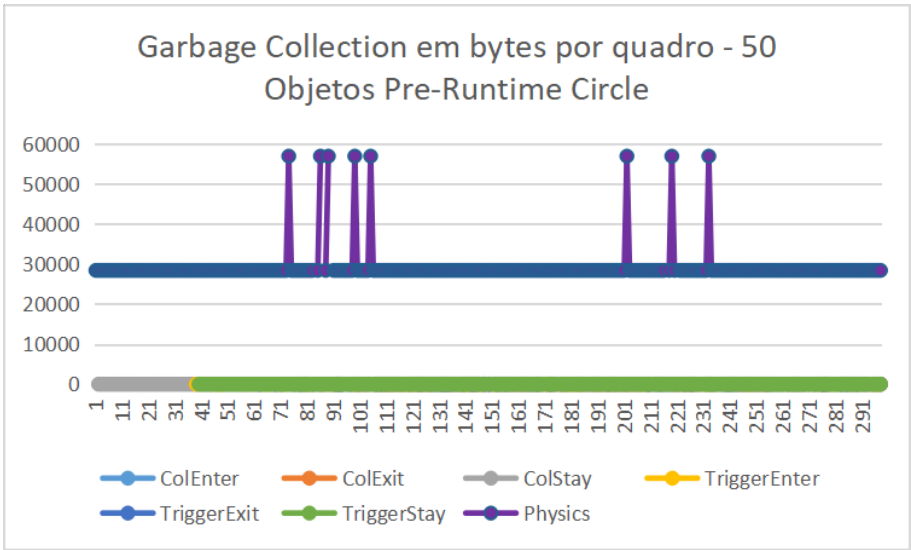

Figura 51: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos *Circle Collider* em *Pre-Runtime* com física.

A seguir é exibido nos gráficos das Figuras 62, 63 e 64, o *Garbage Collection* dos métodos. Em todos os gráficos é possível notar que o método spawn foi o mais usado no *Garbage Collection*, sendo o *Circle collider* o mais estável deles.

Os gráficos a seguir apresentados pelas Figuras 65, 66 e 67, o *Garbage Collection* dos métodos, porém com o atributo física sendo visualizado. Neles é possível observar que o *Box collider* apresenta aumento de coleta de lixo conforme os quadros são renderizados e o *Circle collider* apresenta pouca estabilidade, porém tem menos coleta de lixo. Já o *Polygon collider* é o mais instável, chegando ao dobro da coleta de lixo em comparação aos outros, mesmo com uma quantidade menor de quadros.

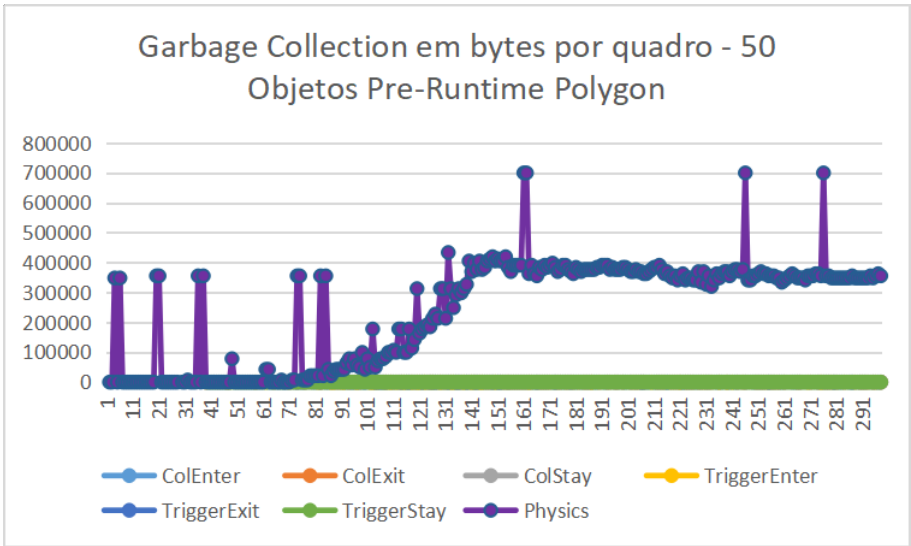

Figura 52: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos *Polygon Collider* em *Pre-Runtime* com física.

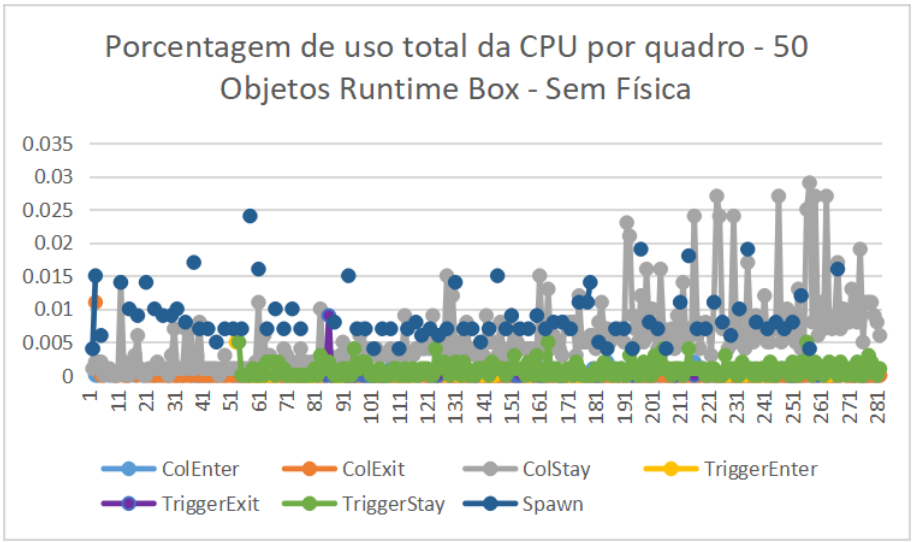

Figura 53: Gráfico de porcentagem de uso da CPU com 50 objetos por segundo *Box Collider* em *Runtime*.

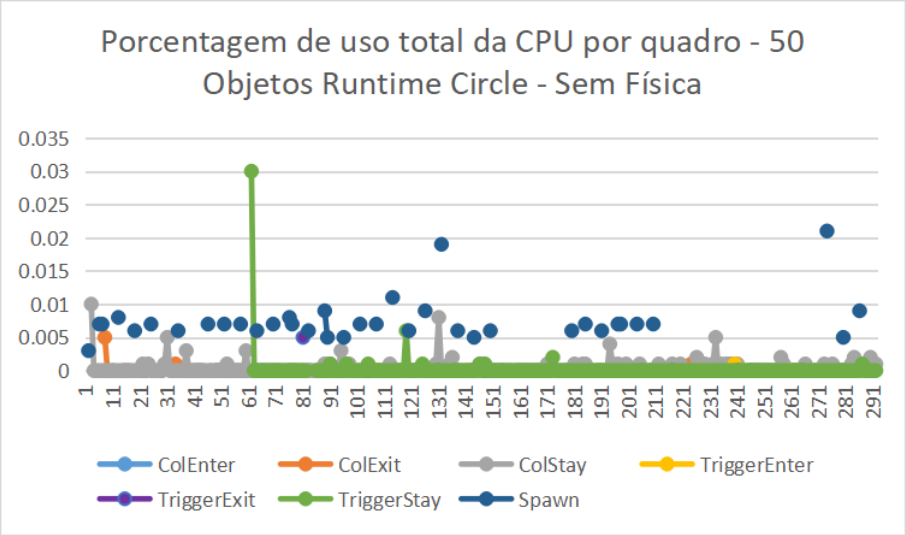

Figura 54: Gráfico de porcentagem de uso da CPU com 50 objetos *Circle Collider* em *Runtime*.

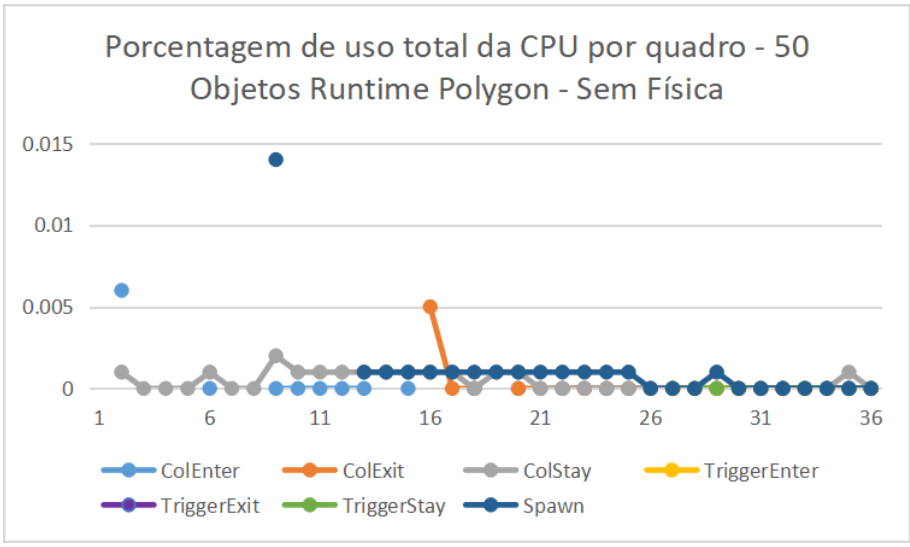

Figura 55: Gráfico de porcentagem de uso da CPU com 50 objetos por segundo *Polygon Collider* em *Runtime*.

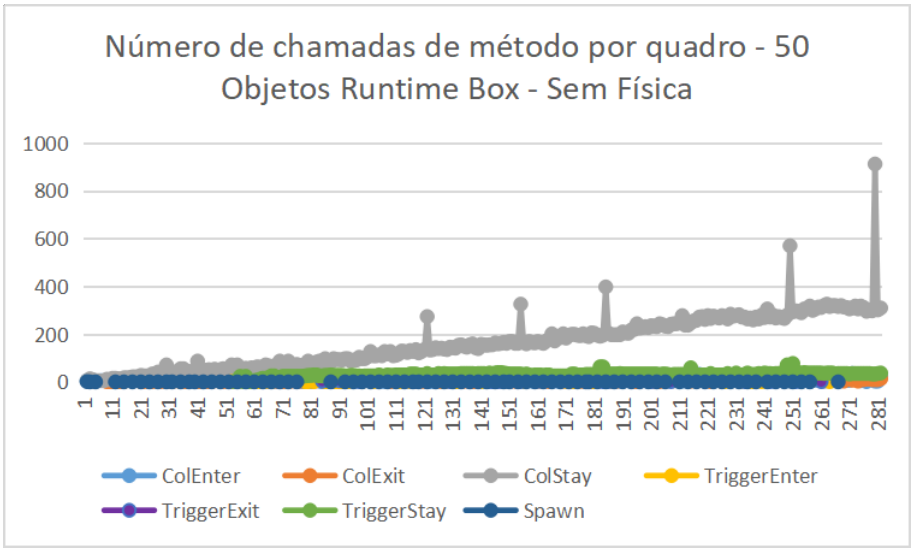

Figura 56: Gráfico de número de vezes que o método foi chamado com 50 objetos por segundo *Box Collider* em *Runtime*.

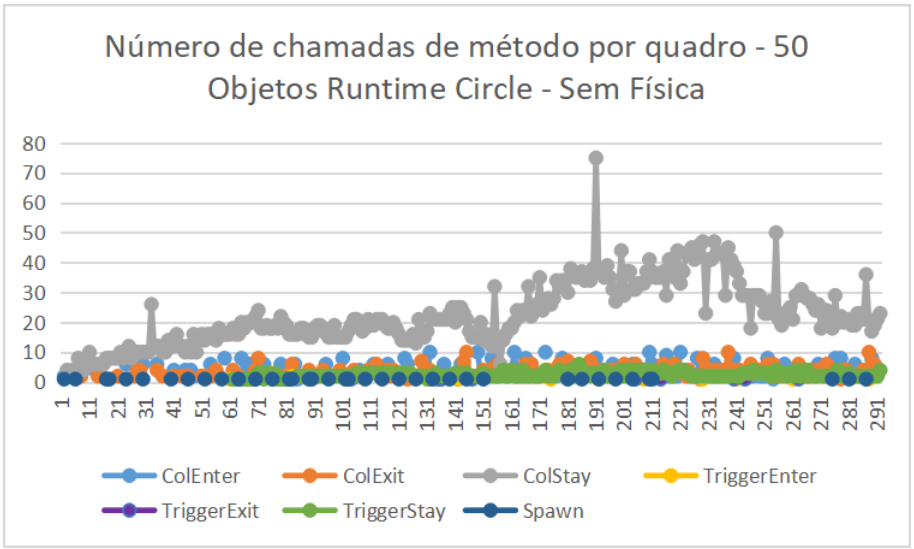

Figura 57: Gráfico de número de vezes que o método foi chamado com 50 objetos *Circle Collider* em *Runtime*.

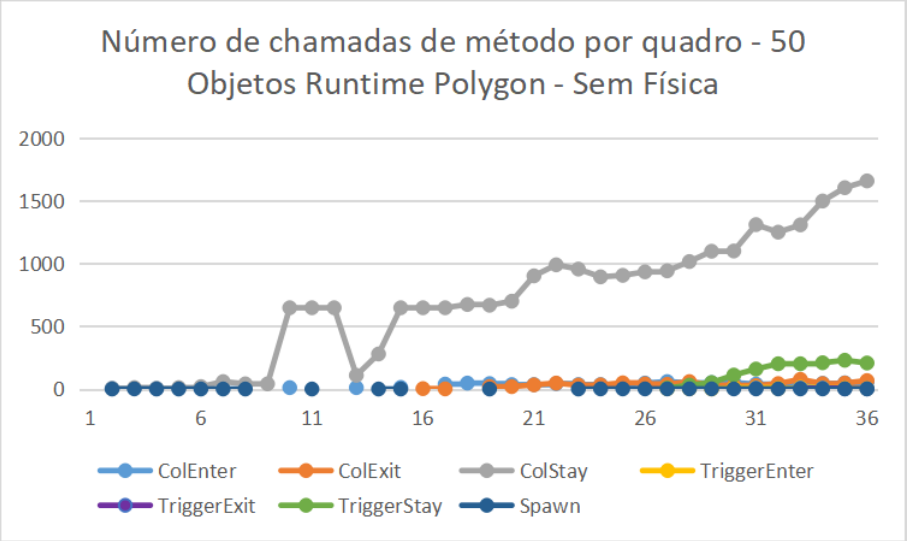

Figura 58: Gráfico de número de vezes que o método foi chamado com 50 objetos *Polygon Collider* em *Runtime*.

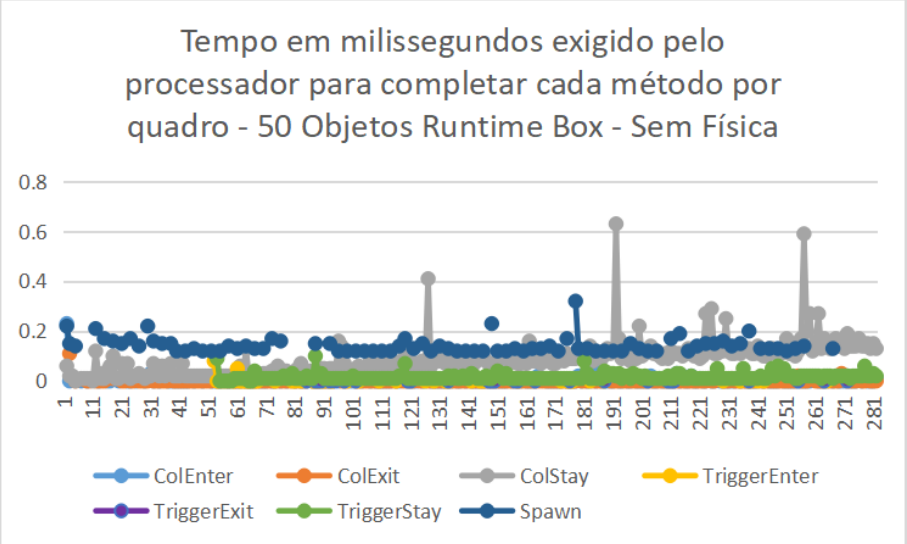

Figura 59: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo *Box Collider* em *Runtime*.

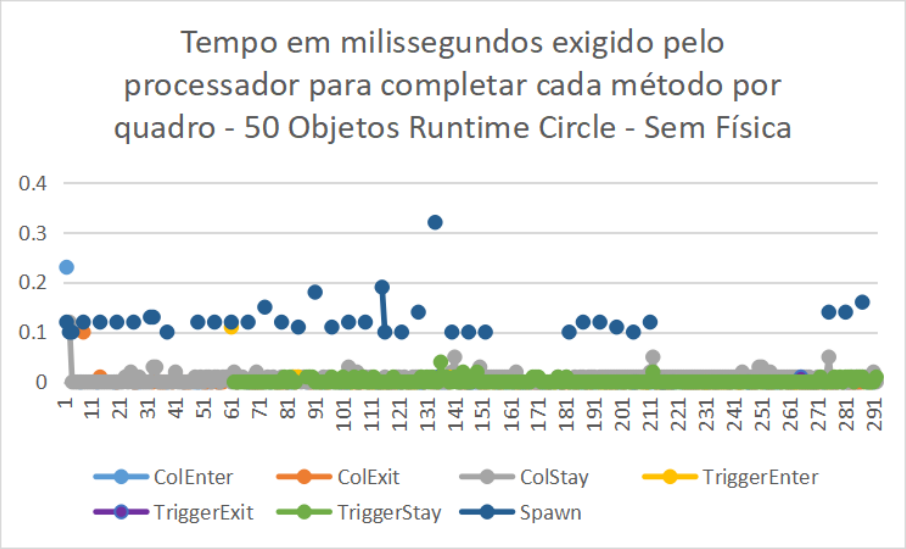

Figura 60: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo *Circle Collider* em *Runtime*.

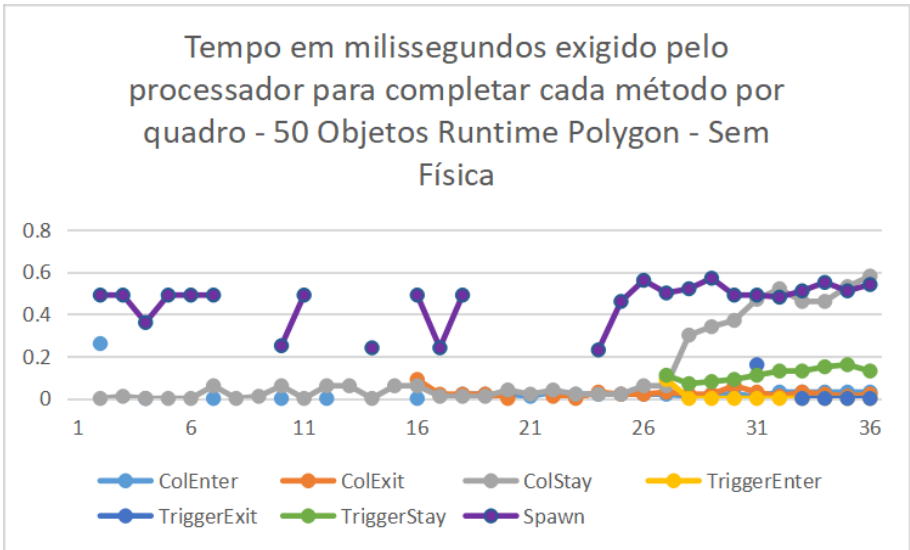

Figura 61: Gráfico de tempo em milissegundos exigido pelo processador para executar o método com 50 objetos por segundo *Polygon Collider* em *Runtime*.

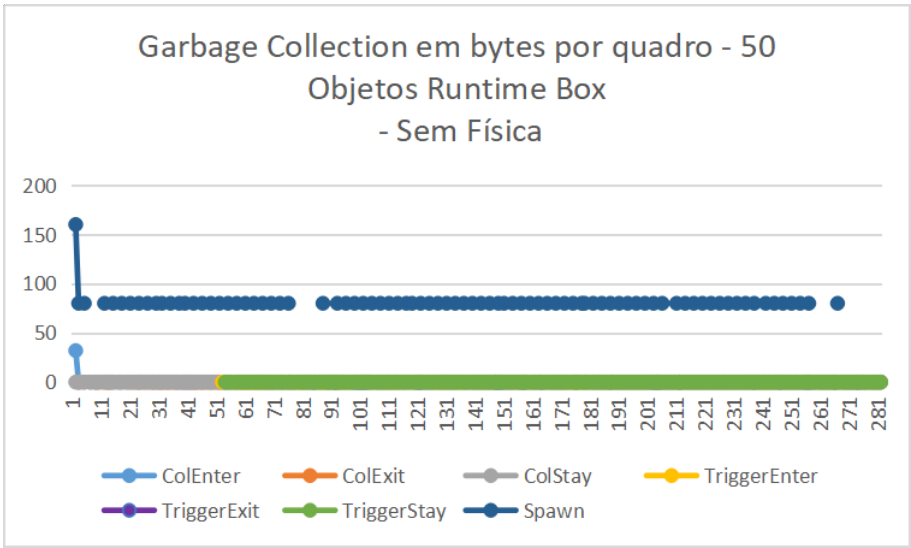

Figura 62: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Box Collider* em *Runtime*.

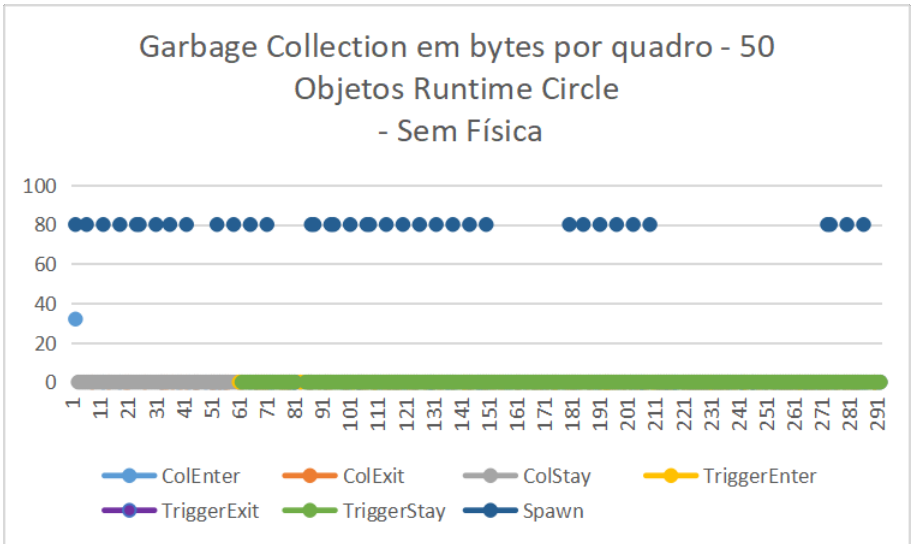

Figura 63: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Circle Collider* em *Runtime*.

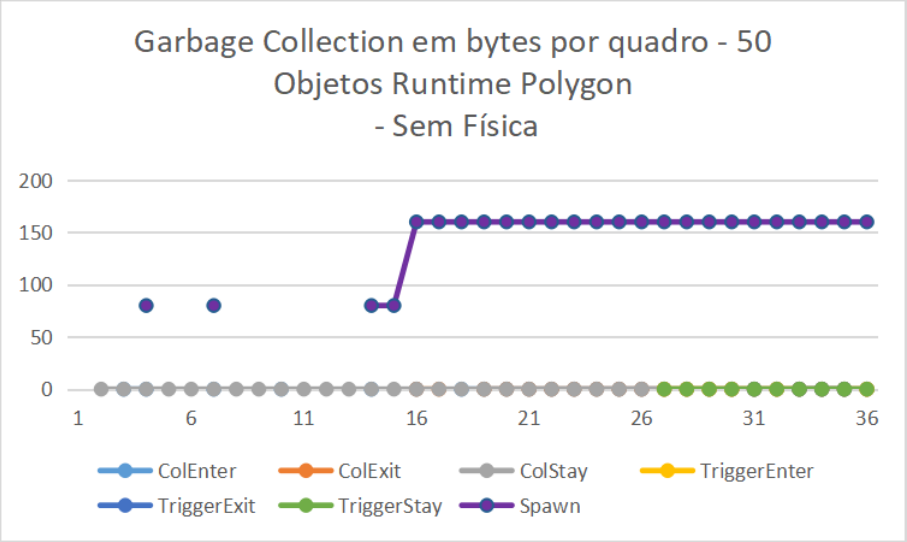

Figura 64: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Polygon Collider* em *Runtime*.

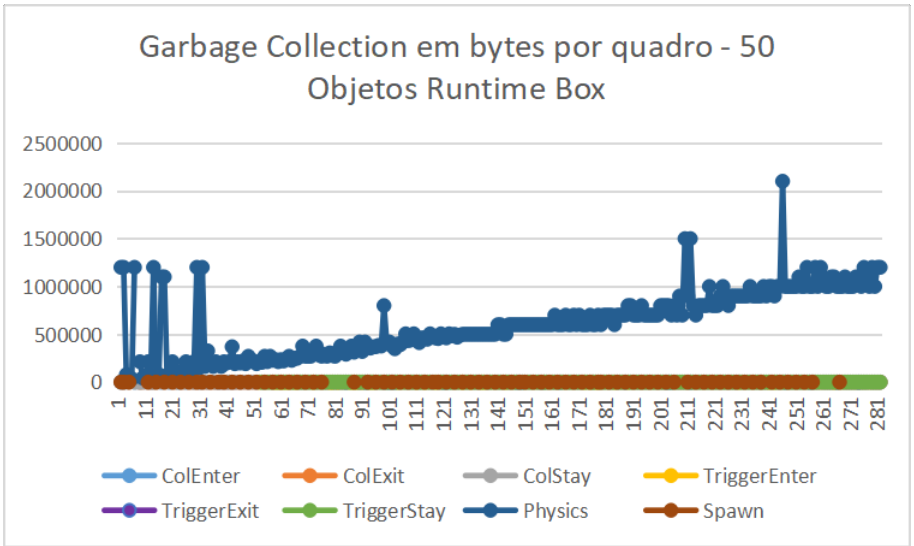

Figura 65: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Box Collider* em *Runtime* com física.

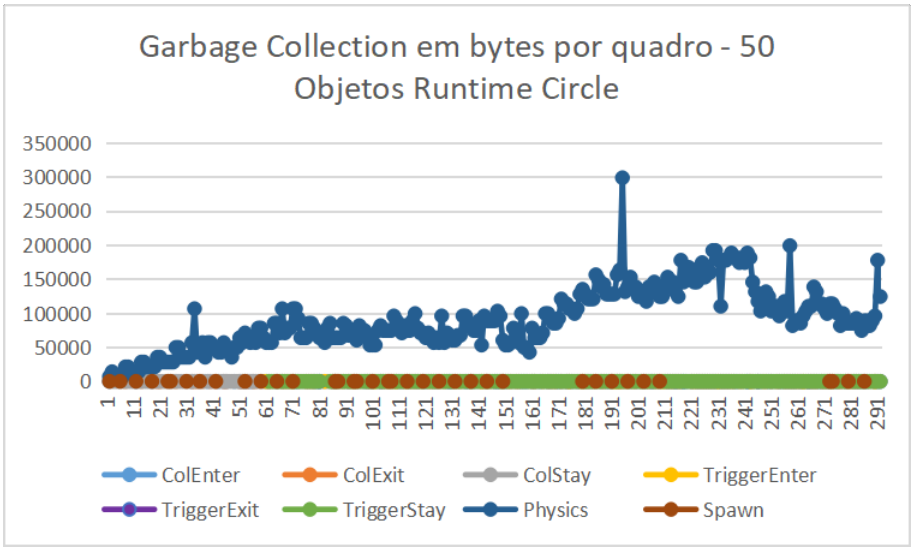

Figura 66: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Circle Collider* em *Runtime* com física.

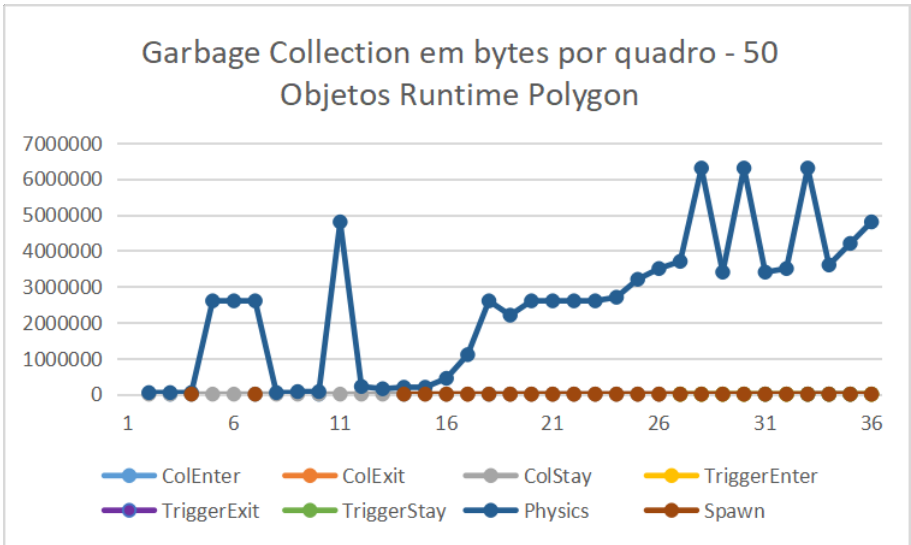

Figura 67: Gráfico de quantidade de *Garbage Collection* em *bytes* com 50 objetos por segundo *Polygon Collider* em *Runtime* com física.

6 CONSIDERAÇÕES FINAIS

6.1 IMPORTÂNCIA

As empresas de desenvolvimento de jogos independentes vem crescendo cada vez mais, e a requisição por jogos com melhores performances e desempenho também, já que na maioria das vezes essas empresas independentes são constituídas de pessoas com pouca experiência no desenvolvimento.

A *Unity* tem contribuído para ajudar no crescimento dessas empresas por ser uma *engine* de fácil burocracia e uso de desenvolvimento. Uma das características do desenvolvimento é uso de colisões de vários tipos. Apesar da documentação sobre colisões da *Unity* ser bem detalhada, ela não é completa em relação a performance, que é o intuito desse Trabalho de Conclusão de Curso.

6.2 RESULTADOS

É possível notar nos resultados que conforme o número de objetos ou número de objetos por segundo aumenta, há um aumento linear no consumo de recursos na maioria dos casos. Com poucos objetos (de um a cinquenta) já inseridos ou inseridos em tempo de execução, os resultados se mostraram semelhantes em cada tipo de *collider*. E algumas observações podem ser feitas.

O tipo de *Collider Edge* é melhor usado com elementos de terreno ou plataformas simples, pelo fato de ser apenas uma linha, e não um objeto com mais dimensão. O *Polygon Collider* é o que oferece a melhor precisão em termos de simulação, no entanto, por ser mais preciso, é o que mais exige recursos, que se usado sem cuidado pode afetar negativamente a performance do jogo.

Os *colliders* do tipo *box* e *circle* são os mais utilizados de acordo com a pesquisa de campo realizada e foram os que se mostraram mais estáveis, ambos em *Pre-Runtime* e *Runtime*. O *circle collider*, no entanto apresentou uma melhor performance gral, porém também foi o que

apresentou a maior coleta de lixo (*Garbage Collection*), o que pode indicar alguma falha de otimização desse *collider*, ou uma consequência de sua performance estável.

6.3 TRABALHOS FUTUROS

Melhorias podem ser realizadas nas comparações para obter ainda mais precisão para ser documentada, como:

- Mais de um tipo de *collider* ao mesmo tempo.
- Usar os outros tipos de *colliders* mencionados.
- Mais funções em execução.
- Usar objetos com a propriedade *kinematic* ligada.
- Comparações com *raycast*, *linecast* e *overlap*.
- Comparações com *ragdolls*.
- Comparações com limitações (*constraints*) na movimentação dos objetos.

Também podem ser feitas melhorias na aquisição dos resultados, como adquirir quantidade de memória RAM para cada quadro e a quantidade de consumo de GPU.

As comparações também podem ser realizadas com objetos e colisões usando a interface 3D da *Unity*.

REFERÊNCIAS

ARMSTRONG, W. **Spotlight Team Best Practices: Collision Performance Optimization**. 2017. Acessado em 03 de outubro de 2019. Disponível em: <https://blogs.unity3d.com/pt/2017/07/26/spotlight-team-best-practices-collision-performance-optimization/>.

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>.

PALMA, S. D. **Unity profiler data exporter**. 2017. Acessado em 03 de outubro de 2019. Disponível em: <https://github.com/steve3003/unity-profiler-data-exporter>.

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>.

APÊNDICE A – QUESTIONÁRIO - PESQUISA DE CAMPO

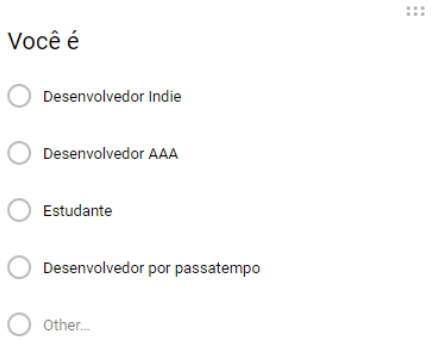

Figura 68: Pergunta 1 do questionário.

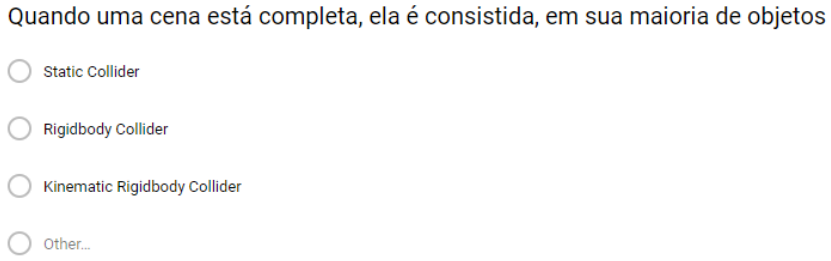

Figura 69: Pergunta 2 do questionário.

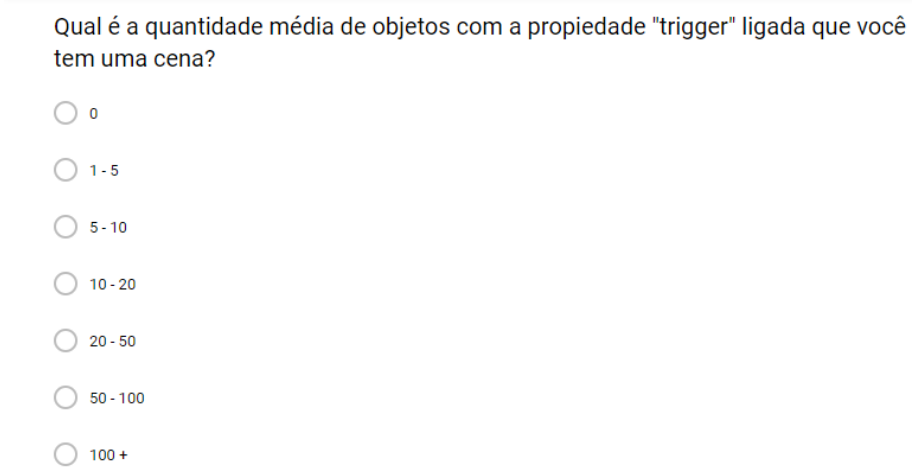

Figura 70: Pergunta 3 do questionário.

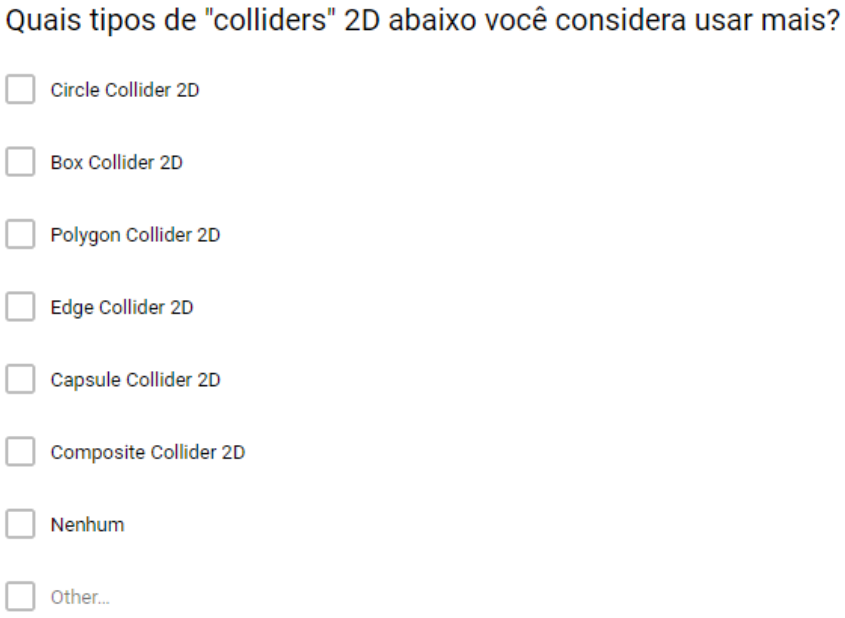

Figura 71: Pergunta 4 do questionário.

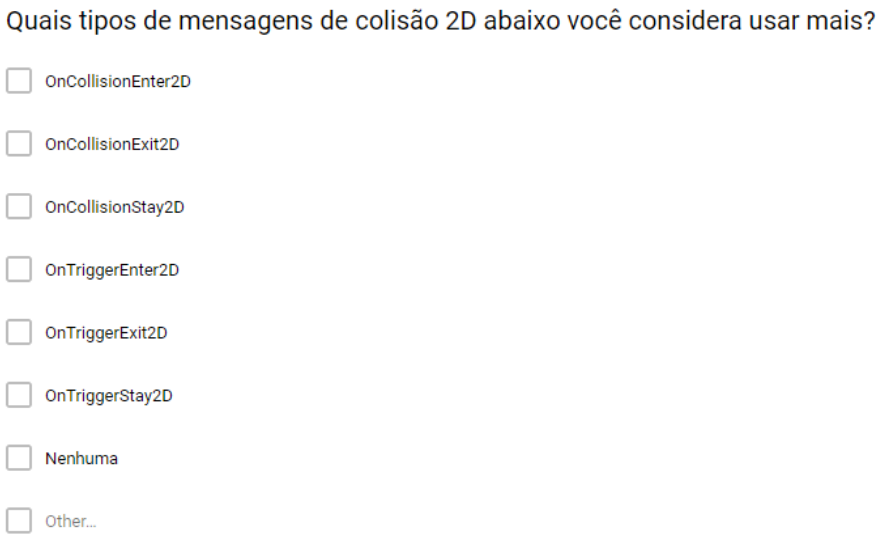

Figura 72: Pergunta 5 do questionário.

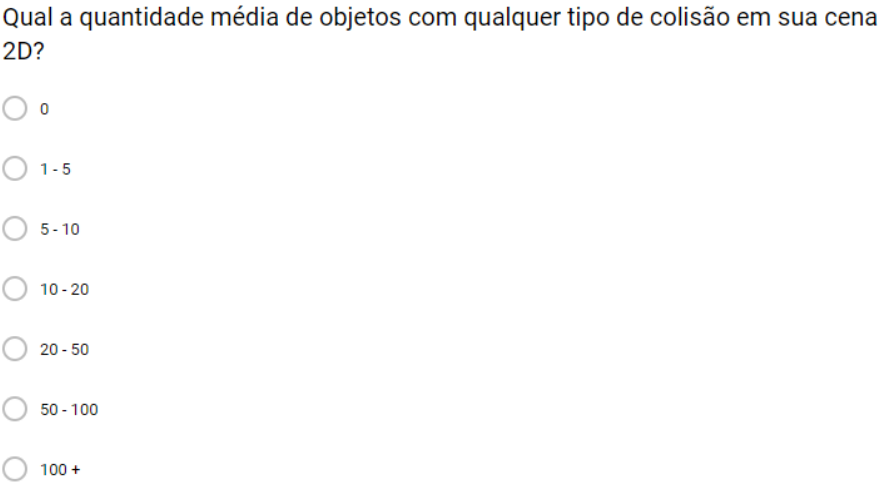

Figura 73: Pergunta 6 do questionário.

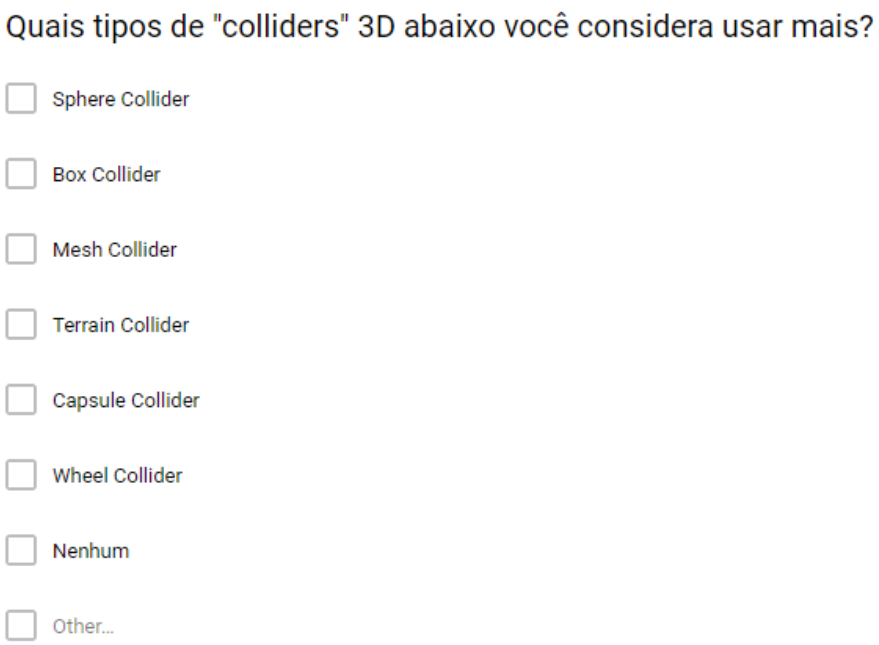

Figura 74: Pergunta 7 do questionário.

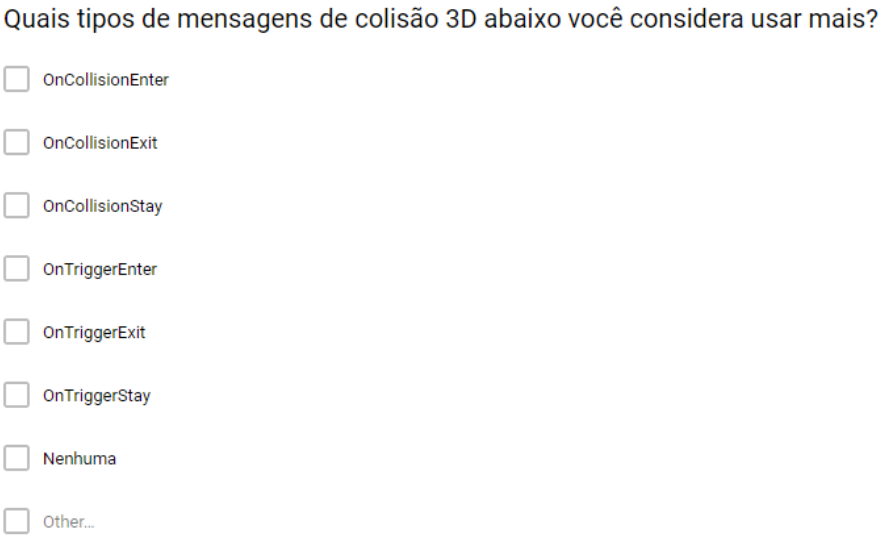

Figura 75: Pergunta 8 do questionário.

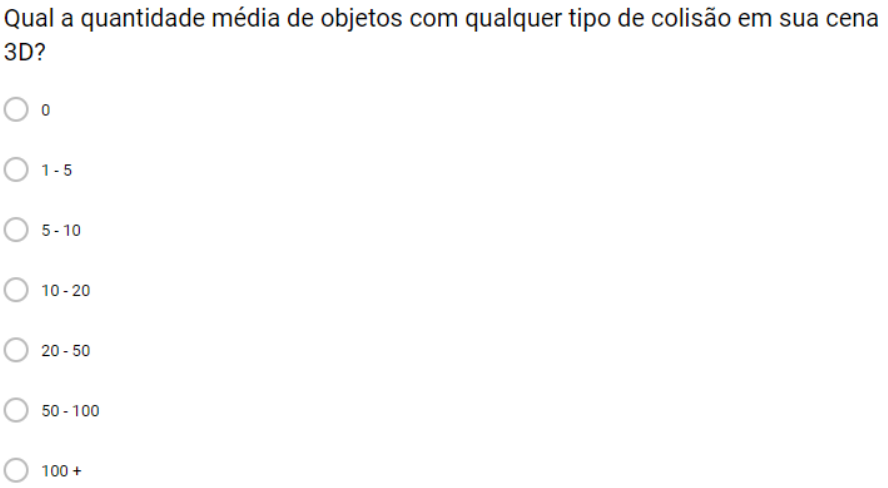

Figura 76: Pergunta 9 do questionário.

APÊNDICE B – RESULTADOS DO QUESTIONÁRIO

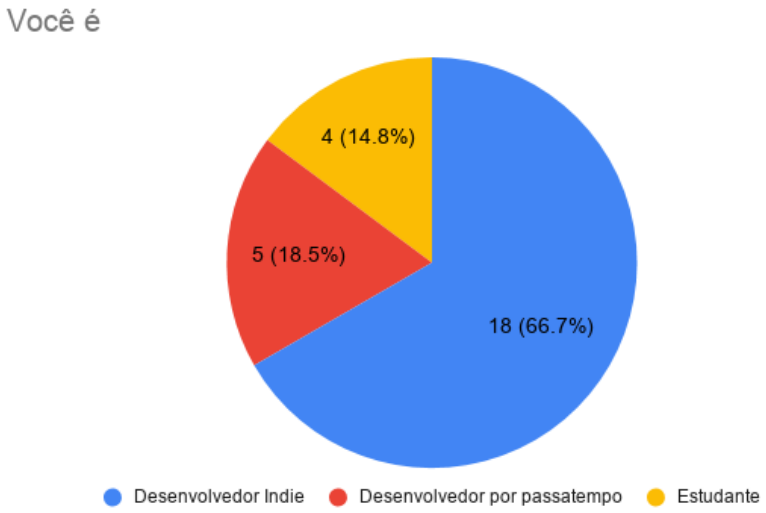

Figura 77: Respostas Pergunta 1 do questionário.

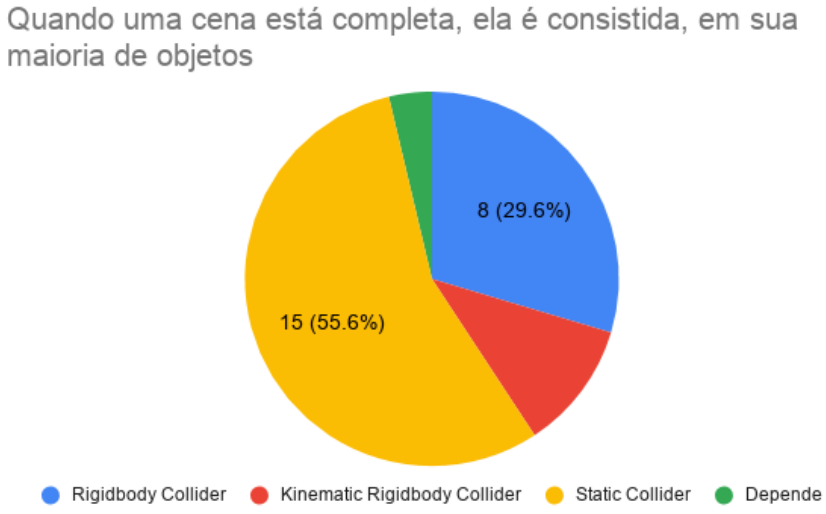

Figura 78: Respostas Pergunta 2 do questionário.

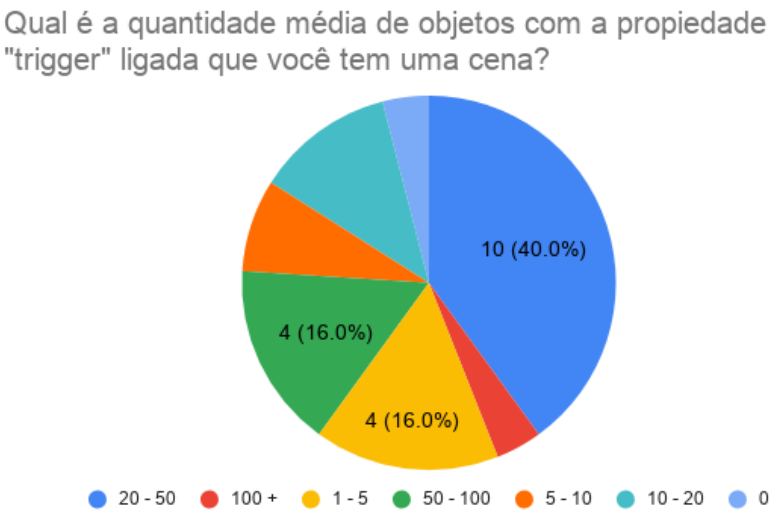

Figura 79: Respostas Pergunta 3 do questionário.

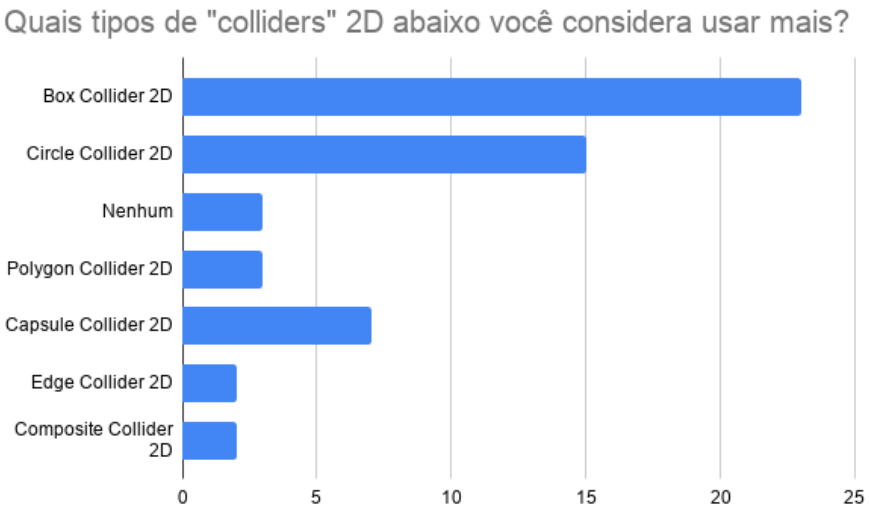

Figura 80: Respostas Pergunta 4 do questionário.

Figura 81: Respostas Pergunta 5 do questionário.

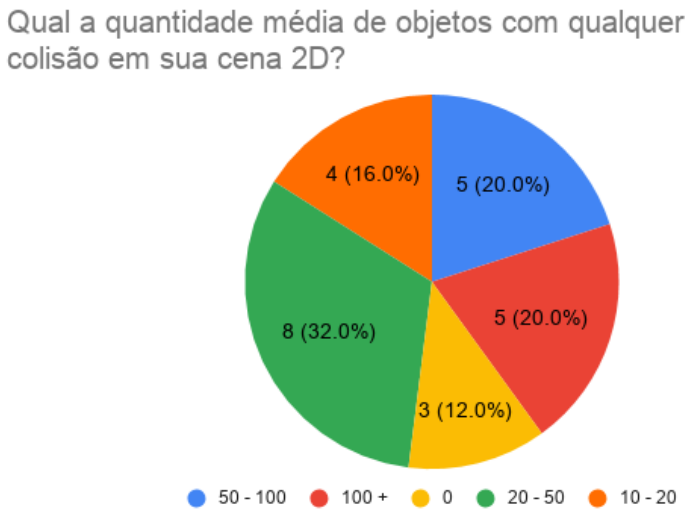

Figura 82: Respostas Pergunta 6 do questionário.

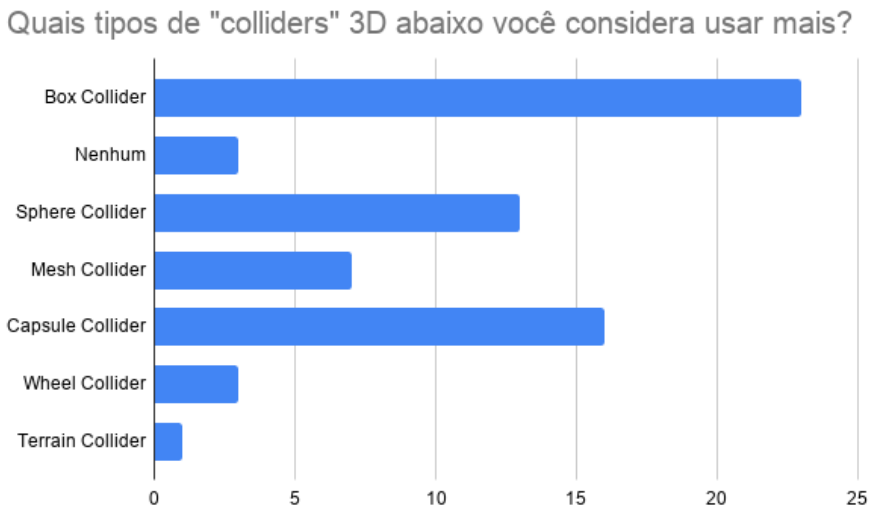

Figura 83: Respostas Pergunta 7 do questionário.

Figura 84: Respostas Pergunta 8 do questionário.

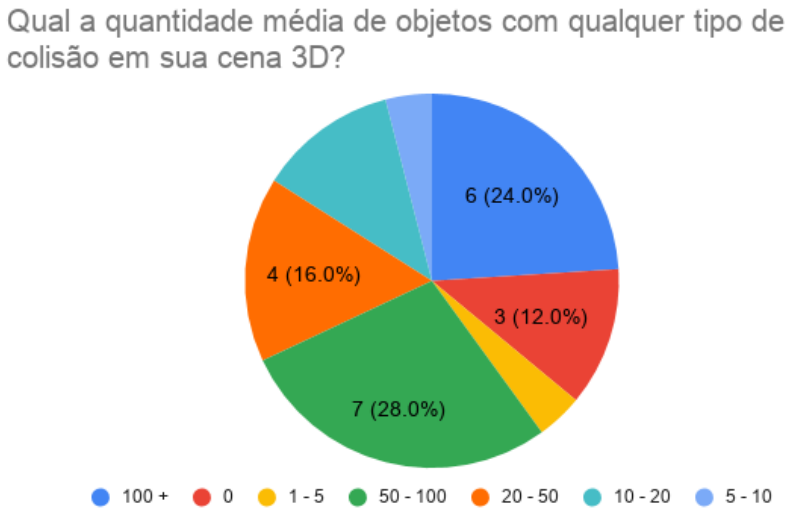

Figura 85: Respostas Pergunta 9 do questionário.

APÊNDICE C – TABELAS DE RESULTADOS DAS COMPARAÇÕES

Devido à grande quantidade de dados gerados nos resultados, as tabelas estarão disponíveis na nuvem por meio do link:

Uma breve explicação das tabelas que estão presentes no link com uma tabela exemplo representada pela Tabela 5:

Os arquivos estão separados pelo tipo de *collider* que foi testado.

A legenda das tabelas ficará em cima para melhorar a visualização e entendimento do conteúdo.

Cada célula representa um quadro chamado, e dentro dele, as medidas dos atributos adquiridos.

Células que não conterem valores (com exceção às referentes aos quadros) indicam que naquele quadro não houve medição para o atributo.

Os quadros estão separados de dez em dez por padrão e representados pelo nome de ”*quadrosTotal*”. Quando há pulo de quadros, significa que não houve medição alguma do método nos quadros apagados, Isso foi feito também para reduzir a carga visual de tabelas que eram pouco preenchidas.

O atributo ”*totalPercent*”representa a porcentagem do uso total da CPU usada para executar o método durante o quadro. Se ele estiver como zero, significa que foi feita uma medida desse atributo nesse quadro, porém com um valor muito baixo que o *Profiler* não conseguiu registrar.

O atributo ”*calls*”representa o número de vezes que o método foi chamado durante o quadro.

O atributo ”*totalTime*”representa o tempo total em milissegundos exigido pela CPU para completar a chamada do método durante o quadro. Se ele estiver como zero, significa que foi feita uma medida desse atributo nesse quadro, porém com um valor muito baixo que o *Profiler* não conseguiu registrar.

O atributo *gcMemory* representa a quantidade de memória em *bytes* que foi armazenada pelo método no *Garbage Collector* durante o quadro.

Tabela 5: Exemplo das tabelas presentes no apêndice remoto

quadrosTotal	totalPercent	calls	totalTime	gcMemory
60				
	0.8%	1	0.13	0
70				
170				

A seguir estão os links para os arquivos separadamente:

C.1 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *BOX COLLIDER*

C.2 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *CIRCLE COLLIDER*

C.3 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *EDGE COLLIDER*

C.4 TABELAS DE RESULTADOS DAS COMPARAÇÕES - *POLYGON COLLIDER*